UML Assessment Questions Set 1

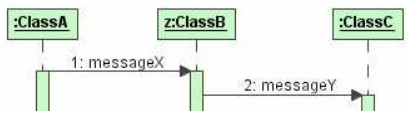
1	Divide a complex system into small, self-contained pieces that can be managed independently. How is it called?	
	a	Abstraction
	b	Modularity
	c	Encapsulation
	d	Hierarchy
	Answer: B	

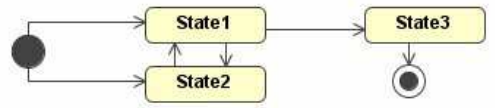
2	In order to model the relationship "a course is composed of 5 to 20 students and one or more instructors", you could use:	
	a	Aggregation
	b	Association
	c	Composition
	d	Realization
	Answer: A	

3	Which of the following statements are true?	
	a	All operations defined in a sub-class are inherited by the super-class
	b	Generalization allows abstracting common features and defining them in a super-class
	c	A super-class is a class that must not have associations
	d	Association is a "part-of" relationship
	Answer: B	

4	What is the relationship between these two use cases?	
	a	Generalization
	b	Extend
	c	Include
	d	Association
	Answer: C	

5	The following diagram shows that there is an interaction between:	
	a	An object of ClassA and an object of ClassB
	b	An object of ClassA and object z of ClassB
	c	Object z of ClassB and an object of ClassB
	d	An object of ClassC and an object of ClassA
	Answer: B	



6	Which of the following statements are true for the following diagram?			
	a	State1 is always the first state of the object after the initial state	c	State3 is always the last state of the object before the final state
	b	State2 is always the last state of the object before the final state	d	During its life, the object is in at least five states
	Answer: C			

7	How many *File*s objects for each *Directory* object?	
	a	0 or 1
	b	2
	c	1 or 2
	d	Many
	Answer: C	

8	Which of the following statement is true for the following diagram	
	a	A is a kind of B
	b	B is a kind of B
	c	A is part of B
	d	B depends on A
	Answer: None	

9	Which of these diagrams shows interactions between objects?
	a Activity diagram
	b Class diagram
	c Sequence diagram
	d Component diagram
	Answer: C

10	A statechart diagram describes:
	a Attributes of objects
	b Nodes of the system
	c Operations executed on a thread
	d Events triggered by an object
	Answer: D

11	An interface is:
	a A set of objects used to provide a specific behaviour
	b A set of classes used on a collaboration
	c A set of attributes used on an operation
	d A set of operations used to specify a service of a class or component
	Answer: D

12	The sequence diagram models:	
	a	The order in which the class diagram is constructed
	b	The way in which objects communicate
	c	The relationship between states
	d	The components of the system
	Answer: B	

13	The activity diagram:	
	a	Focuses on flows driven by internal processing
	b	Models the external events stimulating one object
	c	Focuses on the transitions between states of a particular object
	d	Models the interaction between objects
	Answer: A	

14	The deployment diagram shows:	
	a	Objects of a system
	b	Distribution of components on the nodes in a system
	c	Functions of a system
	d	Distribution of nodes
	Answer: B	

15	Unified Process is a software development methodology which is:	
	a	Use-case driven
	b	Component-driven
	c	Related to Extreme Programming
	d	None in only one iteration
	Answer: A	

UML Assessment Questions Set 2

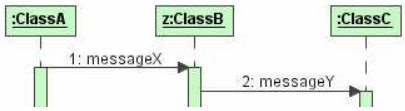
1	Ordering abstractions into a tree-like structure. How is it called?	
	a	Abstraction
	b	Modularity
	c	Encapsulation
	d	Hierarchy
	Answer: D	

2	In order to model the relationship "a hotel has rooms", between Hotel and Room, you could use:	
	a	Aggregation
	b	Association
	c	Composition
	d	Realization
	Answer: C	

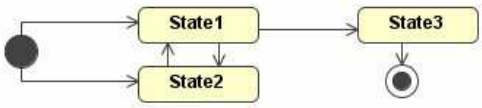
3	Which of the following statements are true?	
	a	All operations defined in a super-class are inherited by the sub-class
	b	Generalization allows abstracting common features and defining them in a sub-class
	c	A super-class is a class that must not have associations
	d	Association is a "kind-of" relationship
	Answer: A	

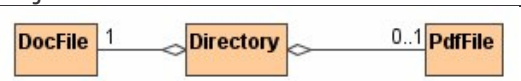
4	What is the relationship between these two use cases?	
	a	Include
	b	Extension
	c	Generalization
	d	Association
	Answer: C	

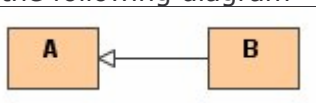
5	The following diagram shows that there is an interaction between:	
	a	An object of ClassC and an object of ClassA
	b	Object z of ClassB and an object of ClassB
	c	An object of ClassA and object z of ClassB
	d	An object of ClassA and an object of ClassC
	Answer: C	



6	Which of the following statements are true for the following diagram?			
	a	State2 is always the first state of the object after the initial state	c	State3 is always the last state of the object before the final state
	b	State1 is always the last state of the object before the final state	d	During its life, the object is in at least five states
	Answer: C			



7	How many *File*s objects for each *Directory* object?	
	a 0 or 2	
	b 0 or 1	
	c 1 or 2	
	d Many	
	Answer: C	

8	Which of the following statement is true for the following diagram	
	a B is a kind of A	
	b A is part of B	
	c A is a kind of B	
	d B depends on A	
	Answer: A	

9	Which of these diagrams shows interactions between objects?
	a Sequence diagram
	b Class diagram
	c Activity diagram
	d Component diagram
	Answer: A

10	A statechart diagram describes:
	a Operations executed on a thread
	b Nodes of the system
	c Attributes and operations of an object
	d Events triggered by an object
	Answer: D

11	An interface is:
	a A set of classes used on a collaboration
	b A set of operations used to specify a service of a class or component
	c A set of attributes used on an operation
	d A set of objects used to provide a specific behaviour
	Answer: B

12	The sequence diagram models:
	a The order in which the class diagram is constructed
	b The relationship between objects
	c The way in which objects communicate
	d The components of the system
	Answer: C

13	The activity diagram:
	a Models the interaction between objects
	b Models the external events stimulating one object
	c Focuses on the transitions between states of a particular object
	d Focuses on flows driven by internal processing
	Answer: D

14	The deployment diagram shows:	
	a	Objects of a system
	b	Functions of a system
	c	Distribution of components on the nodes in a system
	d	Distribution of nodes
	Answer: C	

15	Unified Process is a software development methodology which is:	
	a	Component-driven
	b	Iterative and incremental
	c	Related to Extreme Programming
	d	Done in only one iteration
	Answer: B	

Object-Oriented Analysis and Design with UML

Training Course

Adegboyega Ojo
Elsa Estevez

e-Macao Report 19

Version 1.0, October 2005

Table of Contents

1. Overview

The Unified Modelling Language (UML) is a graphical language for visualising, specifying, constructing, and documenting artefacts of software intensive systems [6]. UML represents the unification of a number of efforts to build notations for expressing models of Object Oriented Analysis and Design (OOAD) under the auspices of the Object Management Group (OMG). At present, UML is the de-facto standard for Object Oriented modelling.

This document describes the course "Object Oriented Analysis and Design with UML" taught to the Core and Extended Teams in the context of the e-Macao Project. The course presents the method of Object Oriented Analysis and Design (OOAD) using the UML notation.

Following the introduction of basic concepts and principles of object orientation, the course shows how informal requirements can be described in details. Considering four views of a typical system - user, static, dynamic and implementation, each of the nine UML diagrams are discussed extensively for modelling these views. For instance, the course describes: (i) use cases for modelling requirements, (ii) class and object diagrams for obtaining good understanding of an application domain and (iii) sequence, collaboration and statechart diagrams for analyzing requirements and specifying architectural and design decisions. In addition, the course teaches the best practices in OOAD based on architectural and design patterns and how UML can be used in the context of the Unified Process (UP). It concludes with a comparative analysis of some popular UML tools.

The rest of this document is as follows. Sections 2, 3 and 4 explain respectively the objectives, prerequisites and methodology for teaching the course. The content of the course is introduced in Section 5. The assessment and organization of the course are explained in Sections 6 and 7. Following references, Appendix A includes the complete set of slides and Appendix B contains two sets of assessment questions with answers.

2. Objectives

The objectives of the course are as follows:

1) Teaching the basic concepts and principles of Object Orientation including Object Oriented Analysis and Design.
2) Introducing the Syntax, Semantics and Pragmatics of the Unified Modeling Language.
3) Showing how requirements can be described informally and how they are modeled using Use Case Diagrams.
4) Teaching how the structural and behavioral aspects of a system can be analyzed, specified and designed using Class, Interaction and Statechart Diagrams.
5) Showing how implementation and deployment details of a system can be modeled using the Implementation and Deployment Diagrams respectively.
6) Teaching best practices in OOAD based on Architecture and Design patterns.
7) Introducing the Unified Process and showing how UML can be used within the process.
8) Presenting a comparison of the major UML tools for industrial-strength development.

3. Prerequisites

This course assumes that the learner possesses the basic knowledge of application development. No prior knowledge of Object Orientation is required.

4. Methodology

The course has been designed based on the following didactic principles:

- *Depth versus Breadth* - As a foundation, an attempt has been made to cover the various aspects of UML as required in practical applications, without much loss of depth.
- *Academic Orientation* – A body of concepts is defined rigorously and incrementally to establish a foundation for proper understanding and use of technology.
- *From Definitions to Demonstrations* – All major concepts introduced during the course are illustrated with small-size examples which are also demonstrated on the computer, whenever possible.
- *From Demonstrations to Assignments* – On the basis of demonstrations, students are asked to perform different tasks with increasing level of difficulty and independence.

Generally, instructors are expected to administer the course material in a tutorial style. Unlike most courses or materials available on UML, this course teaches UML from an application perspective rather than the usual notation perspective.

5. Content

The course is organized into six major sections: Object-Orientation, UML Basics, UML Modelling, Unified Process, UML Tools and Summary. The synopses for these sections are presented below. The course starts with an Introduction section, which describes the contents presented in subsequent sections, including the organizational aspect of the course. The course consists of 618 slides divided into six major sections: Introduction, Object Orientation, UML Basics, UML Modelling, Unified Process and UML Tools.

5.1. Introduction

The Introduction section provides information on the aims of the course, the outline of each section, and the schedule for the course as delivered within the context of the e-Macao Project. This part is covered in the slides 1 through 16.

5.2. Object-Orientation

This section is covered in the slides 17 through 83. It presents the background, principles and fundamental concepts of Object Orientation: Object, Class, Abstraction, Interface, Implementation, Aggregation, Composition, Generalization, Sub-Class and Polymorphism. The section concludes with the presentation of Object Oriented Analysis and Design.

5.3. UML Basics

The section provides a brief presentation of UML. It starts by explaining what modelling entails and why it is important for software and system engineering. Next, it outlines the basic building blocks of UML: Elements, Relationships and Diagrams. Finally, the section presents five basic modelling views supported by UML: Use Case, Design, Process, Implementation and Deployment. This section is covered in the slides 84 through 151.

5.4. UML Modelling

The section constitutes the major part of the course. It presents the nine UML diagrams and how they are applied through the five modelling activities: Requirements, Architecture, Design, Implementation and Deployment. The following sections are devoted to each of these modelling activities.

5.4.1. Requirements

This section teaches about requirements in general. It shows how requirements can be described informally and how they can be modelled as Use Cases. In addition, the section presents how the basic concepts and entities of an application domain can be described using Class and Object Diagrams. Furthermore it teaches how Interaction and Statechart Diagrams can be used to specify class and object interactions and their internal behaviours respectively. This Section is covered in the slides 152 through 299.

5.4.2. Architecture

The section starts by introducing the various concepts related to Architectural Design: Sub-Systems, Services, Coupling, Cohesion and Layering. Next, it presents Collaborations and how they can be used to model the functional units of a system. Static and dynamic aspects of collaborations are covered as well. Collaboration Diagrams are presented as the major diagram for architecture modelling. Packages are introduced for organizing or modularizing the functional units of a system. The section ends with a discussion of the set of major architectural patterns: Repository, Model-View-Controller, Client-Server, Peer-to-Peer and Pipe-and-Filter. This section is presented in the slides 300 through 403.

5.4.3. Design

The section, covered in the slides 404 through 526, starts with an introduction to System and Object Design activities. Next, it presents Design Classes as foundation of the whole development. The steps involved in building Design Class Diagrams are described, from reviewing conceptual classes, to decorating design class attributes with data types, to providing default values and constraints. Furthermore, Activity Diagrams are presented for describing business processes, procedures or algorithms. Sequence and Statechart Diagrams are revisited in the discussion of detailed design of both internal and external behaviours of system objects. The section is concluded with a discussion on Design Patterns.

5.4.4. Implementation

This section presents Components Diagrams and how they can be used to specify implementation artefacts: Source Code, Executable Releases and Physical Databases. Implementation modelling is covered in the slides 527 through 540.

5.4.5. Deployment

This section teaches how Component Diagrams and Nodes can be used to describe the deployment environment of a system. Slides 541 to 554 contain this Section.

5.5. Unified Process

The Unified Process (UP) is taught in this section as a process supporting Object Oriented Analysis and Design with UML. The section explains the details of the Unified Process: Core Workflows and Life Cycle Phases. Finally, it shows which UML models are relevant to specific Workflows and Phases of the UP. Slides 555 through 579 contain this Section.

5.6. Tools

This section presents some of the tools for UML modelling: Enterprise Architect, Magic Draw, Poseidon and Rational Rose. The basic features of each tool are presented, as is the comparative analysis of the tools. This Section is presented in the slides 580 through 591.

5.7. Summary

This section provides a summary of the entire course highlighting the major points in each of the sections. The summary section is given from the slide 592 to 618.

6. Assessment

Assessment comprises 15 multiple-choice questions provided at the end of the course to test the students' understanding of the various topics taught. Two sets of assessment questions are given in Appendices B.1 and B.2, with only slight differences between them. The assessment compliments the 55 tasks or assignments provided in the various sections.

7. Organization

The course consists of lectures, demonstrations and assignments:

- *lectures* – The lectures aim to incrementally build a body of concepts to establish the foundation for proper understanding and use of technology.
- *demonstrations* - Demonstrations illustrate the concepts introduced during the lectures with running code and small-size examples.

- *assignments* - During assignments, students are asked to perform a range of tasks with increasing level of difficulty and independence. They are encouraged to reuse the demonstrated code and examples in their assignments.

The course is designed to be taught for about 42 hours. In the context of the e-Macao Core Team Training, the course was taught over seven days, as follows: (1) Introduction and Object Orientation and UML Basics (2) UML Basics and Requirements, (3) Requirements, (4) Architecture, (5) Design, (6) Implementation and Deployment, and (7) Unified Process, Tools and Summary. For e-Macao Extended Team Training, a shorter version of the same course was taught over four days: (1) Introduction, Object Orientation and UML Basics, (2) Requirements, (3) Architecture and Design, and (4) Design, Implementation, Deployment, Unified Process, UML Tools and Summary.

References

1) OMG Unified Modeling Language Specification, Object Management Group.
2) UML Bible, Tom Pender, John Wiley and Sons, 2003.
3) Object-Oriented Analysis and Design using UML, Simon Bennet, Steve McRobb and Ray Farmer, McGraw-Hill, 2002.
4) Guide to Applying the UML, Sinan Si Alhir, Springer, 2002.
5) Object-Oriented Software Engineering, Bernd Bruegge, Allen H Dutoit, Prentice Hall, 2000.
6) The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, 1999.
7) OO Software Development Using UML, UNU-IIST Tech Report 229.

Appendix

A. Slides

A.1. Introduction

Object-Oriented Analysis and Design with UML

Adegboyega Ojo and Elsa Estevez

UNU-IIST

The Course

- objectives - what do we intend to achieve?
- outline - what content will be taught?
- resources - what teaching resources will be available?
- organization - duration, major activities, daily schedule

e-Macao-18-1-3

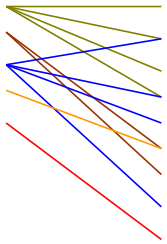
Course Objectives

1) present the concepts of Object Orientation, as well as Object Oriented Analysis (OOA) and Design (OOD)
2) introduce the syntax, semantics and pragmatics of UML, and how to integrate it with the Unified Process
3) show how to articulate requirements using use cases
4) present the development of structural and behavioural diagrams for different views supported by UML
5) introduce patterns and frameworks and their applications in architecture and design tasks
6) present a comparative analysis of the major UML tools suitable for industrial-strength development

e-Macao-18-1-4

Course Outline

1) Object Orientation
2) UML Basics
3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment
4) Unified Process
5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-5

Overview: Object Orientation

1) Background
2) Principles
3) Concepts
4) OO Analysis
5) OO Design

e-Macao-18-1-6

Overview: UML Basics

1) UML outline
2) UML building blocks
   a) elements
   b) relationships
   c) diagrams
3) modelling views

e-Macao-18-1-7

Overview: Requirements

1. software requirements

2. use case modelling
   a) use case diagrams
   b) templates

3. conceptual modelling
   a) class diagrams
   b) object diagrams

4. behavioural modelling
   a) sequence diagrams
   b) statechart diagrams

e-Macao-18-1-8

Overview: Architecture

1) software architecture

2) models
   a) collaboration diagrams
   b) component diagrams

3) frameworks and patterns

e-Macao-18-1-9

Overview: Design

1) software design

2) design models
   a) class diagrams
   b) sequence diagrams
   c) activity diagrams
   d) statechart diagrams

3) design patterns

e-Macao-18-1-10

Overview: Implementation

1) packages

2) components

3) component diagrams

e-Macao-18-1-11

Overview: Deployment

1) nodes and components
2) deployment diagrams
3) use of deployment diagrams

e-Macao-18-1-12

Overview: Unified Process

1) Unified Process
2) life cycle phases
3) workflows
4) workflows and UML diagrams

e-Macao-18-1-13

Overview: UML Tools

1) CASE tools
2) Selected UML tools:
   a) Magic Draw
   b) Enterprise Architect
   c) Poseidon
   d) Rational Rose

e-Macao-18-1-14

Course Resources

1) OMG Unified Modeling Language Specification, Object Management Group.
2) UML Bible, Tom Pender, John Wiley and Sons, 2003.
3) Object-Oriented Analysis and Design using UML, Simon Bennet, Steve McRobb and Ray Farmer, McGraw-Hill, 2002.
4) Guide to Applying the UML, Sinan Si Alhir, Springer, 2002.
5) Object-Oriented Software Engineering, Bernd Bruegge, Allen H Dutoit, Prentice Hall, 2000.
6) The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, 1999.
7) OO Software Development Using UML, UNU-IIST Tech Report 229.
8) Course materials

e-Macao-18-1-15

Course Organization

Monday to Thursday:

09:00 – 10:20 lecture/lab
10:20 – 10:30 break
10:30 – 11:20 lecture/lab
11:20 – 11:30 break
11:30 – 13:00 lecture/lab

13:00 – 14:30 lunch time

14:30 – 15:30 lecture/lab
15:30 – 15:40 break
15:40 – 16:30 lecture/lab
16:30 – 16:40 break
16:40 – 17:45 lecture/lab (except Thursday)

e-Macao-18-1-16

Course Assessment

Thursday, 16:45 – 17:45

15 multiple-choice questions

30 minutes allowed

maximum obtainable points – 30

grades	points
A	26-30
B	21-25
C	16-20
D	11-15
E	00-10

A.2. Object Orientation

A.2.1. Background

Object Orientation

e-Macao-18-1-18

Course Outline

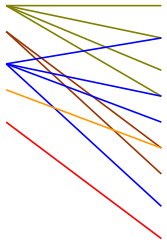
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

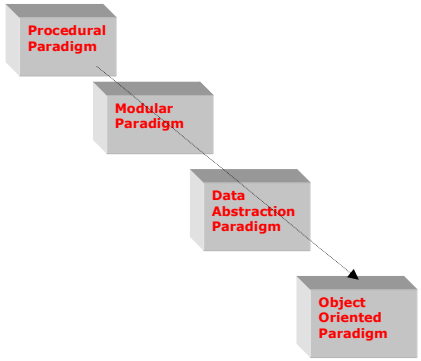
5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-19

Towards Object Orientation



e-Macao-18-1-20

What is Object Orientation?

Definition
Object Orientation is about viewing and modelling the world/system as a set of interacting and interrelated *objects.*

Features of the OO approach:

1) the universe consists of interacting objects

2) describes and builds systems consisting of objects

A.2.2. Principles

e-Macao-18-1-21

OO Principles

1) abstraction

2) encapsulation

3) modularity

4) hierarchy

e-Macao-18-1-22

Principle 1: Abstraction

A process allowing to focus on most important aspects while ignoring less important details.

Abstraction allows us to manage complexity by concentrating on essential aspects making an entity different from others.

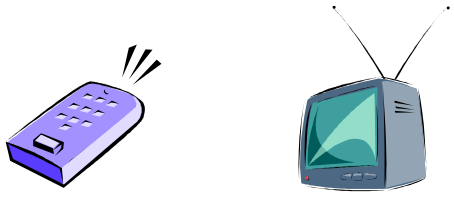
An example of an order processing abstraction

e-Macao-18-1-23

Principle 2: Encapsulation

Encapsulation separates implementation from users/clients.

A client depends on the interface.

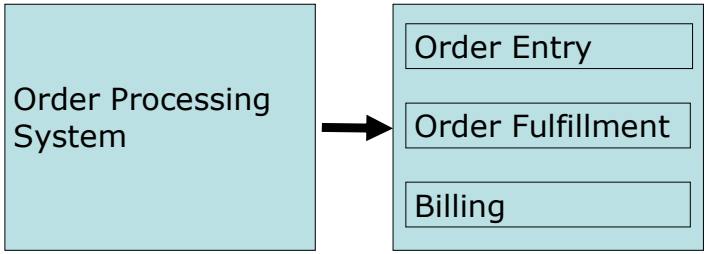


Courtesy Rational Software

e-Macao-18-1-24

Principle 3: Modularity

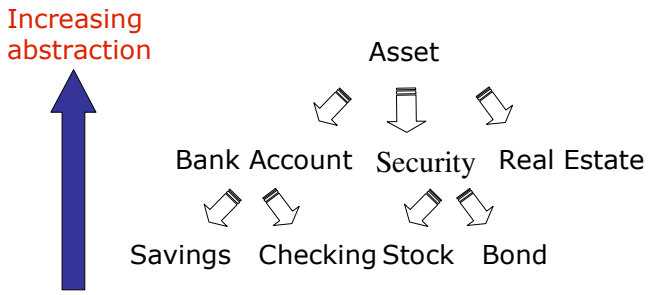
Modularity breaks up complex systems into small, self-contained pieces that can be managed independently.



e-Macao-18-1-25

Principle 4: Hierarchy

Ordering of abstractions into a tree-like structure.



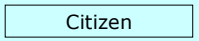
e-Macao-18-1-26

Task 1

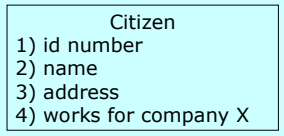
Which of the following models A and B is more abstract?

Justify your answer.

Model A:



Model B:



e-Macao-18-1-27

Task 2

Provide an example of encapsulation:

a) identify the interface

b) identify the implementation

Tip:
interface: computer case
implementation: computer mother board

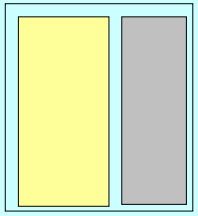
e-Macao-18-1-28

Task 3

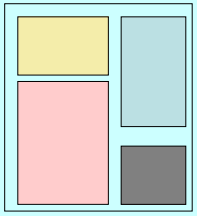
Suppose A and B are models of the same system.

Which of them is more modular?

Model A:



Model B:



e-Macao-18-1-29

Task 4

Describe the structure of your agency as a hierarchy.

Include at least three levels.

A.2.3. Concepts

e-Macao-18-1-30

OO Concepts

1. Object
2. Class
3. Attribute
4. Operation
5. Interface
6. Implementation
7. Association
8. Aggregation
9. Composition
10. Generalization
11. Super-Class
12. Sub-Class
13. Abstract Class
14. Concrete Class
15. Discriminator
16. Polymorphism
17. Realization

e-Macao-18-1-31

Concept 1: Objects

What is an object?

1) any abstraction that models a single thing

2) a representation of a specific entity in the real world

3) may be tangible (physical entity) or intangible

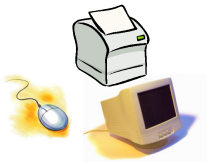
Examples: specific citizen, agency, job, location, order, etc.

e-Macao-18-1-32

Examples of Objects

chairs

persons



computer devices

e-Macao-18-1-33

Object Definition

Two aspects:

- Information:
  1) has a unique identity
  2) has a description of its structure
  3) has a state representing its current condition
- Behaviour:
  1) what can an object do?
  2) what can be done to it?

e-Macao-18-1-34

Object Definition Example

1) information:
   a) serial number
   b) model
   c) speed
   d) memory
   e) status

2) behaviour:
   a) print file
   b) stop printing
   c) empty the queue

e-Macao-18-1-35

Concept 2: Class

What is a class?

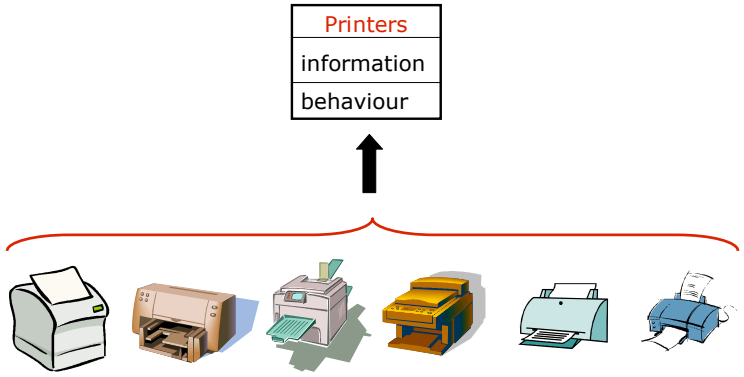
1) any uniquely identified abstraction of a set of logically related instances that share similar characteristics
2) rules that define objects
3) a definition or template that describes how to build an accurate representation of a specific type of objects

Examples: agency, citizen, car, etc.

Objects are created using class definitions as templates.

e-Macao-18-1-36

Class Example



e-Macao-18-1-37

Concept 3: Attribute

Definition
Attribute is a named property of a class describing a range of values that instances of the class may hold for that property.

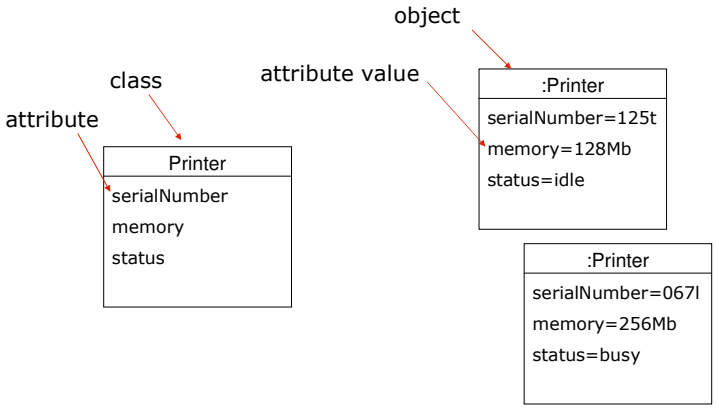
An attribute has a type and defines the type of its instances.

Only the object is able to change the values of its own attributes.

The set of attribute values defines the state of the object.

e-Macao-18-1-38

Attribute Examples



e-Macao-18-1-39

Concept 4: Operation

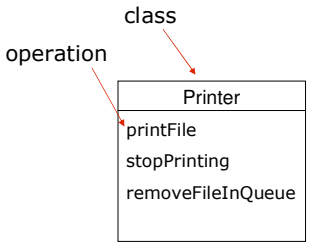
Definition
Operation is the implementation of a service that can be requested from any object of a given class.

An operation could be:

1. a question - does not change the values of the attributes
2. a command – may change the values of the attributes

e-Macao-18-1-40

Operation Example



e-Macao-18-1-41

Task 5

Which of the following statements are true?

For those that are false, explain why.

1) Trainee is an example of a class.

2) Dora Cheong is an example of a class.

3) createJob is an example of an object.

4) deleteFile is an example of an operation

5) name is an attribute

e-Macao-18-1-42

Task 6

Suppose we wish to model an application for issuing business registration licenses.

Identify:

1) three classes for the model

2) at least three attributes for each class

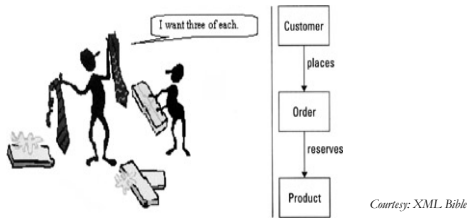
e-Macao-18-1-43

Applying Abstraction

Abstraction in Object Orientation:

1) use of objects and classes to represent reality

2) software manages abstractions based on the changes occurring to the real-world objects



e-Macao-18-1-44

Concept 5: Interface

1) minimum information required to use an object

2) allows users to access the object's knowledge

3) must be exposed

4) provides no direct access to object internals

e-Macao-18-1-45

Interface Example

```
package edu.unu.iist;

import java.util.Date;
import java.util.Properties;

/**
 * @author elsa
 *
 */
public interface BusinessCalendar {

    /**
     * Get information on the properties of the business calendar
     * @return Properties of the business calendar
     */
    public Properties getBusinessCalendarProperties();

    /**
     * Add an appointment to the calendar
     * @param date the appointment date
     * @return a confirming date
     */
    public Date add(Date date);
}
```

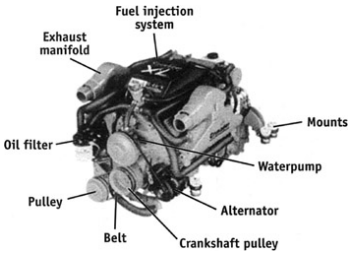
e-Macao-18-1-46

Concept 6: Implementation

1) information required to make an object work properly

2) a combination of the behaviour and the resources required to satisfy the goal of the behaviour

3) ensures the integrity of the information upon which the behaviour depends



e-Macao-18-1-47

Implementation Example

```
public class BusinessCalendar {

  Day[] weekDays = null;
  List holidays = null;

  private static Properties businessCalendarProperties = null;
  public static Properties getBusinessCalendarProperties() {
    if (businessCalendarProperties==null) {
      businessCalendarProperties = new Properties();
      InputStream is = ClassLoaderUtil.getStream("jbpm.business.calendar.properties", "org/jbpm/calendar");
      try {
        businessCalendarProperties.load(is);
      } catch (IOException e) {
        throw new RuntimeException("couldn't load business.calendar.properties", e);
      }
    }
    return businessCalendarProperties;
  }

  public Date add(Date date, Duration duration) {
    Date end = null;
    if (duration.isBusinessTime) {
      DayPart dayPart = findDayPart(date);
      boolean isInbusinessHours = (dayPart!=null);
      if (! isInbusinessHours) {
        Object[] result = new Object[2];
        findDay(date).findNextDayPartStart(0, date, result);
        date = (Date) result[0];
        dayPart = (DayPart) result[1];
      }
      end = dayPart.add(date, duration);
    } else {
      end = new Date(date.getTime()+duration.milliseconds);
    }
    return end;
  }
```

e-Macao-18-1-48

Relations and Links

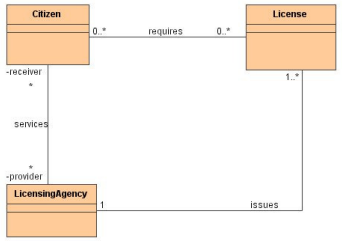
Relationships:
- between classes (relations)
- between objects (links)

Three kinds of relations between classes:
1) association
2) aggregation
3) composition

e-Macao-18-1-50

Concept 7: Association

1. the simplest form of relation between classes
2. peer-to-peer relations
3. one object is aware of the existence of another object
4. implemented in objects as references



e-Macao-18-1-51

Association Examples

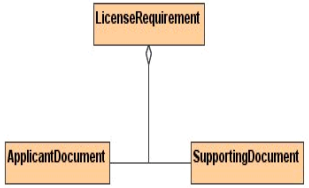
Associations between classes A and B:

1) A is a physical or logical part of B
2) A is a kind of B
3) A is contained in B
4) A is a description of B
5) A is a member of B
6) A is an organization subunit of B
7) A uses or manages B
8) A communicates with B
9) A follows B
10)A is owned by B

e-Macao-18-1-52

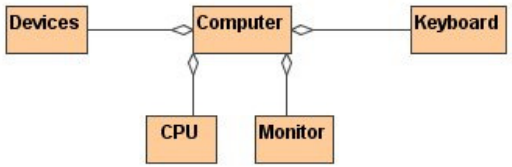
Concept 8: Aggregation

1. a restrictive form of "part-of" association
2. objects are assembled to create a more complex object
3. assembly may be physical or logical
4. defines a single point of control for participating objects
5. the aggregate object coordinates its parts



e-Macao-18-1-53

Aggregation Example



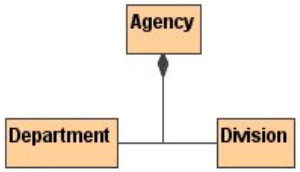
A CPU is part of a computer.

CPU, devices, monitor and keyboard are assembled to create a computer.

e-Macao-18-1-54

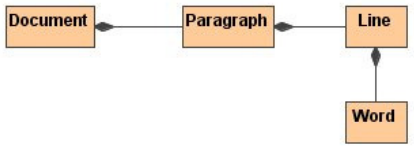
Concept 9: Composition

1. a stricter form of aggregation
2. lifespan of individual objects depend on the on lifespan of the aggregate object
3. parts cannot exist on their own
4. there is a create-delete dependency of the parts to the whole



e-Macao-18-1-55

Composition Example



A word cannot exist if it is not part of a line.

If a paragraph is removed, all lines of the paragraph are removed, and all words belonging to that lines are removed.

e-Macao-18-1-56

Task 7

Identify which of the following statements are true.

For those that are false, explain why.

1) there is an association between Trainee and Course

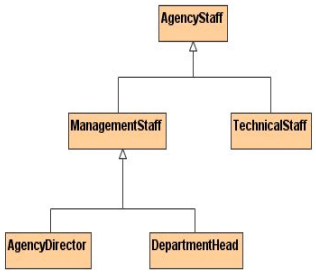
2) there is a composition between Course and Professor

3) there is an aggregation between Course and Venue

e-Macao-18-1-58

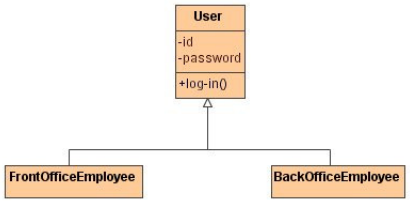
Concept 10: Generalization

1) a process of organizing the features of different kinds of objects that share the same purpose
2) equivalent to "kind-of" or "type-of" relationship
3) generalization enables inheritance
4) specialization is the opposite of generalization
5) not an association



e-Macao-18-1-59

Generalization Example



Common features are defined in User.

FrontOfficeEmployee and BackOfficeEmployee inherit them.

e-Macao-18-1-60

Concept 11: Super-Class

Definition
Super-Class is a class that contains the features common to two or more classes.

A super-class is similar to a superset, e.g. agency-staff.

e-Macao-18-1-61

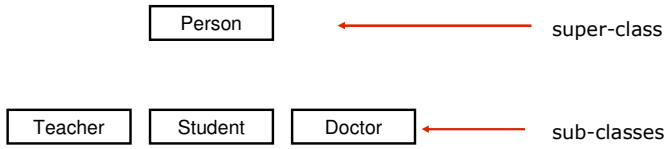
Concept 12: Sub-Class

Definition
Sub-Class is a class that contains at least the features of its super-class(es).

A class may be a sub-class and a super-class at the same time, e.g. management-staff.

e-Macao-18-1-62

Super/Sub-Class Example



e-Macao-18-1-63

Concept 13: Abstract Class

1. a class that lacks a complete implementation - provides operations without implementing some methods
2. cannot be used to create objects; cannot be instantiated
3. a concrete sub-class must provide methods for unimplemented operations

e-Macao-18-1-64

Concept 14: Concrete Class

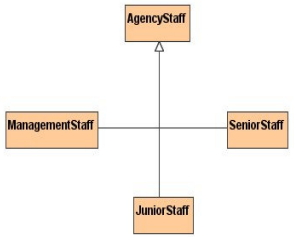
1. has methods for all operations
2. can be instantiated
3. methods may be:
   a) defined in the class or
   b) inherited from a super-class

e-Macao-18-1-65

Concept 15: Discriminator

Discriminator – an attribute that defines sub-classes

Example: "status" of agency staff is a possible discriminator to derive "management", "senior" and "junior" sub-classes.

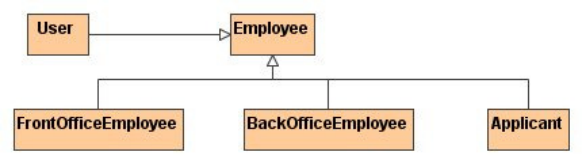


e-Macao-18-1-66

Task 9

Suppose we wish to model an e-service application for a government agency. Does this diagram reasonably model the relationship between the entities User, Employee, Front-Office Employee, Back-Office Employee and Applicant?

If not, provide a more appropriate model.



e-Macao-18-1-67

Task 10

- Add a generalization for the classes defined in Task 6.
- Identify the super-class and one sub-class.
- Do you have any abstract class? Justify.
- Which discriminator is used in your generalization hierarchy?

e-Macao-18-1-68

Concept 16: Polymorphism

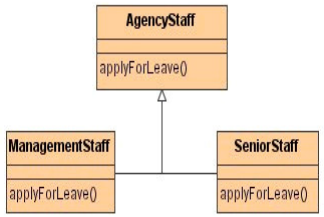
1. ability to dynamically choose the method for an operation at run-time or service-time
2. facilitated by encapsulation and generalization:
   a) encapsulation –
   separation of interface from implementation
   a) generalization –
   organizing information such that the shared features reside in one class and unique features in another
3. Operations could be defined and implemented in the super-class, but re-implemented methods are in unique sub-classes.

e-Macao-18-1-69

Polymorphism Example

Many ways of doing the same thing!

Example: management-staff and agency-staff can apply for leave, but possibly in different ways.



e-Macao-18-1-70

Task 11

Consider the generalization hierarchy provided in Task 10.

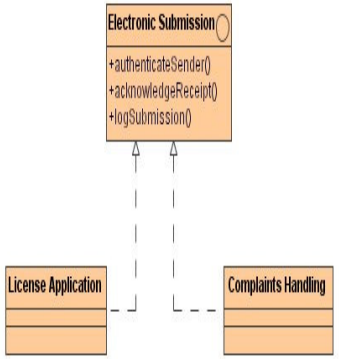
Introduce an operation for the super-class which could be implemented in different ways by its sub-classes.

Provide an example of polymorphism. Explain.

e-Macao-18-1-71

Concept 17: Realization

1. allows a class to inherit from an interface class without being a sub-class of the interface class
2. only inherits operations
3. cannot inherit methods, attributes or associations



e-Macao-18-1-72

Task 12

Using realization model the fact that vehicles and aircrafts must be to accelerate, decelerate, brake and park.

A.2.4. Object Oriented Analysis

e-Macao-18-1-73

OO Analysis and Design

A methodology for system/software modelling and design using Object-Oriented concepts.

Consists of two parts:

1) Object Oriented Analysis

2) Object Oriented Design

e-Macao-18-1-74

Object Oriented Analysis

1) a discovery process

2) clarifies and documents the requirements of a system

3) focuses on understanding the problem domain

4) discovers and documents the key problem domain classes

5) concerned with developing an object-oriented model of the problem domain

6) identified objects reflect the entities that are associated with the problem to be solved

e-Macao-18-1-75

OOA Definition

Definition
Object Oriented Analysis (OOA) is concerned with developing requirements and specifications expressed as an object model (population of interacting objects) of a system, as opposed to the traditional data or functional views.

[Software Engineering Institute]

e-Macao-18-1-76

Benefits

1) maintainability: simplified mapping to the real world
   a) less analysis effort
   b) less complexity in system design
   c) easier verification by the user

2) reusability: reuse of the artifacts that are independent of the analysis method or programming language

3) productivity: direct mapping to the features implemented in Object Oriented Programming Languages

A.2.5. Object Oriented Design

e-Macao-18-1-77

Object Oriented Design

- process of invention and adaptation
- creates abstractions and mechanisms necessary to meet behavioural requirements determined during analysis
- language-independent
- provides an object-oriented model of a software system to implement the identified requirements

e-Macao-18-1-78

OOD Design

Definition [SEI]
Object Oriented Design (OOD) is concerned with developing object-oriented models of a software/system to implement the requirements identified during OOA.

The same set of benefits as those in OOA.

e-Macao-18-1-79

OOD Process

Stages in OOD:

1) define the context and modes of use of the system
2) design the system architecture
3) identify the principal objects in the system
4) develop design models
5) specify object interfaces

Notes on the OOD activities:

1) activities are not strictly linear but interleaved
2) back-tracking may be done a number of times due to refinement or availability of more information

e-Macao-18-1-80

Task 13

List two basic differences between traditional systems analysis and object-oriented analysis.

e-Macao-18-1-81

Summary 1

Object is any abstraction that models a single thing in a universe with some defined properties and behaviour.

A class is any uniquely identified abstraction of a set of logically related objects that share similar characteristics.

Classes may be related by three types of relations:
1) association
2) aggregation
3) composition

e-Macao-18-1-82

Summary 2

Object Orientation is characterized by:

1) encapsulation – combination of data and behaviour, separation of an interface from implementation

2) inheritance – generalization and specialization of classes, forms of hierarchy

3) polymorphism – different implementations for the shared operations depending on the position of the involving object in the inheritance hierarchy.

e-Macao-18-1-83

Summary 3

Object Oriented Analysis is concerned with specifying system requirements and analysing the application domain.

Object Oriented Design is concerned with implementing the requirements identified during OOA in the application domain.

A.3. UML Basics

A.3.1. UML Background

UML Basics

e-Macao-18-1-85

Course Outline

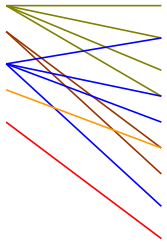
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-86

What Is Modelling?

Modelling involves:

1) representation or simplification of reality

2) providing a blueprint of a system

e-Macao-18-1-87

Why Model?

1) better understand the system we are developing

2) describe the structure or behaviour of the system

3) experiment by exploring multiple solutions

4) furnish abstraction for managing complexity

5) document the design decisions

6) visualize the system "as-is" and "to-be"

7) provide a template for constructing a system

e-Macao-18-1-88

Modelling Principles

1) the choice of models affects how a problem is tackled

2) every model may be expressed at different levels of abstraction

3) effective models are connected to reality

4) no single model is sufficient to describe non-trivial systems

e-Macao-18-1-89

What is UML?

UML = Unified Modelling Language

A language for visualizing, specifying, constructing and documenting artifacts of software-intensive systems.

Examples of artifacts: requirements, architecture, design, source code, test cases, prototypes, etc.

UML is suitable for modelling various kinds of systems:

1. enterprise information systems,
2. distributed web-based,
3. real-time embedded system, etc.

e-Macao-18-1-90

UML – Specification Language

Provides views for development and deployment.

UML is process-independent.

UML is recommended for use with the processes that are:

1) use-case driven
2) architecture-centric
3) iterative
4) incremental

e-Macao-18-1-90

UML – Specification Language

Provides views for development and deployment.

UML is process-independent.

UML is recommended for use with the processes that are:

1) use-case driven
2) architecture-centric
3) iterative
4) incremental

e-Macao-18-1-91

Goals of UML

1) provide modelers with an expressive, visual modelling language to develop and exchange meaningful models
2) provide extensibility and specialization mechanisms to extend core concepts
3) support specifications that are independent of particular programming languages and development processes
4) provide a basis for understanding specification languages
5) encourage the growth of the object tools market
6) supports higher level of development with concepts such as components frameworks or patterns

e-Macao-18-1-92

History of UML

1) started as a unification of the Booch and Rumbaugh methods - *Unified Method v. 0.8* (1995)
2) Jacobson contributes to produce UML 0.9, 1996
3) UML Partners work with three Amigos to propose UML as a standard modelling language to OMG in 1996.
4) UML partners tender their proposal (UML 1.0) to OMG in 1997, and 9 months later submit final UML 1.1.
5) Minor revision is UML 1.4 adopted in May 2001, and the most recent revision is UML 1.5, March 2003.
6) UML 2.0 released by the end 2004.

A.3.2. UML Building Blocks

e-Macao-18-1-93

UML Building Blocks

Three basic building blocks:

1) elements: main "citizens" of the model
2) relationships: relationships that tie elements together
3) diagrams: mechanisms to group interesting collections of elements and relationships

They will be used to represent complex structures.

e-Macao-18-1-94

Elements

Four basic types of elements:

1) structural
2) behavioural
3) grouping
4) annotation

They will be used to specify well-formed models.

e-Macao-18-1-95

Structural Elements

a) static part of the model to represent conceptual elements

b) "nouns" of the model

c) seven kinds of structural elements:

1) class
2) interface
3) collaboration
4) use case
5) active class
6) component
7) node

e-Macao-18-1-96

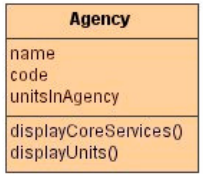
Element 1: Class

1) description of a set of objects that share the same attributes, operations, relationships and semantics

2) implements one or more interfaces

3) graphically rendered as a rectangle usually including a name, attributes and operations

4) can be also used to represent actors, signals and utilities



e-Macao-18-1-97

Element 2: Interface

1) collection of operations that specifies a service of a class

2) describes the externally visible behaviour, partial or complete, of a class

3) defines a set of operation signatures but not their implementations

4) rendered as a circle with a name

e-Macao-18-1-98

Element 3: Collaboration

1) defines an interaction between elements

2) several elements cooperating to deliver a behaviour, rather than individual behaviour

3) includes structural and behavioural dimensions

4) represents implementations of patterns of cooperation that make up a system

5) represented as a named ellipse drawn with a dashed line

e-Macao-18-1-99

Element 4: Use Case

1) description of a sequence of actions that a system performs to deliver an observable result to a particular actor

2) used to structure the behavioural elements in a model

3) realized by collaboration

4) graphically rendered as an ellipse drawn with a solid line

e-Macao-18-1-100

Element 5: Active Class

1) a class whose objects own one or more processes or threads and therefore can initiate an action

2) a class whose objects have concurrent behaviour with other objects

3) it also can be used to represent processes and threads

4) graphically, an active class is rendered just like a class drawn with a thick line

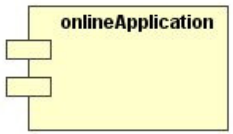
e-Macao-18-1-101

Element 6: Component

1) physical, replaceable part of a system that conforms to and provides the realization of a set of interfaces

2) represents deployment components such as COM+ or Java Beans components

3) represents a physical packaging of logical elements such as classes, interfaces and collaborations



4) it can also be used to represent applications, files, libraries, pages and tables

e-Macao-18-1-102

Element 7: Node

1) a physical element, exists at run time

2) represents a computational resource with memory and processing capacity

3) a set of components may reside in a node

4) components may also migrate from one node to another

5) graphically modelled as a cube

e-Macao-18-1-103

Behavioural Elements

a) represent behaviour over time and space

b) "verbs" of the model

c) two kinds of behavioural elements:

8) interaction
9) state machine

e-Macao-18-1-104

Element 8: Interaction

1) a set of messages exchanged among a set of objects within a particular context to accomplish a specific goal

2) specifies the behaviour of a set of objects

3) involves a number of other elements:
- messages
- action sequences (behaviour invoked by a message)
- links (connection between objects)

4) graphically rendered as an arrow

e-Macao-18-1-105

Element 9: State Machine

1) a sequence of states an object or interaction goes through during its lifetime, and its response to external events

2) may specify the behaviour of an individual class or a collaboration of classes

3) includes a number of elements: states, transition, events and activities

4) presented as a rounded rectangle with a name and sub-states

Interactions and state machines are associated with structural elements such as classes, collaborations and objects.

e-Macao-18-1-106

Element 10: Package

Grouping element of UML:

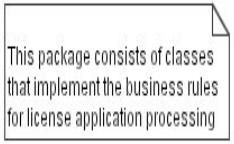
- organizes diagrams
- primary kind of grouping and decomposition
- conceptual, only available at development time
- graphically represented as a tabbed folder

e-Macao-18-1-107

Element 11: Note

Annotation element:

- comments added to models for better explanation or illumination on specific elements
- used primarily for annotation e.g. rendering constraints and comments attached to elements or collections of elements
- presented as a rectangle with a dog-eared corner
- may include both textual and graphical comments



e-Macao-18-1-108

Task 14

Identify the UML element which best models each of the following entities:

1) Jackie Chan
2) a set of operations for validating dates
3) a module capable of validating dates based on the operations in 2
4) a set of trainees
5) "withdraw money" request from an ATM system
6) a computer server that hosts the payroll system

e-Macao-18-1-109

Relationships

Four basic types of relationships:

1) dependency
2) associations
3) generalization
4) realization

Meanings are consistent with the basic Object Oriented relationship types described earlier.

e-Macao-18-1-110

Relationship 1: Dependency

A semantic relationship between two elements in which a change to one element (independent element) may affect the meaning of another (dependent element).

Given as a directed dashed line possibly with a label:

dependent element independent element

e-Macao-18-1-111

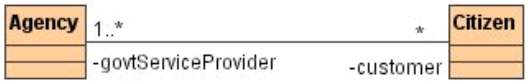
Relationship 2: Association

1) a structural relationship describing a set of links

2) links are connections between objects

3) aggregation is a special type of association depicting the whole-part relationship

4) association is presented as a solid line, possibly directed, labelled and with adornments (multiplicity and role names)



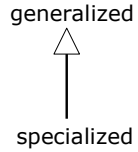
e-Macao-18-1-112

Relationship 3: Generalization

1) a relationship in which objects of a specialized element (child) are substitutable for objects of a generalized element (parent)

2) child elements share the structure/behaviour of the parent

3) rendered graphically as a solid line with hollow arrowhead pointing to the parent



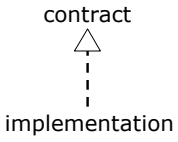
e-Macao-18-1-113

Relationship 4: Realization

1) a semantic relationship between elements, wherein one element specifies a contract and another guarantees to carry out this contract

2) relevant in two basic scenarios:
   a) interfaces versus realizing classes/components
   b) uses cases versus realizing collaborations

3) graphically depicted as a dashed arrow with hollow head, a cross between dependency and generalization



e-Macao-18-1-114

Variants of Relationships

Variations of these four relationship types include:

1) refinement

2) trace

3) include

4) extend

e-Macao-18-1-115

Task 15

Define the relationship type which best relates the following classes:

1) Employee – Administrative officers
2) Employee – Office
3) Application Software – Operating system
4) PC - Device

e-Macao-18-1-116

Diagrams

A diagram is a graph presentation of a set of elements and relationships where:

a) nodes are elements
b) edges are relationships

Can visualize a system from various perspective, thus is a projection of a system.

e-Macao-18-1-117

Diagram Types

UML is characterized by nine major diagrams:

1) class
2) object
3) use case
4) sequence
5) collaboration
6) statechart
7) activity
8) component
9) deployment

e-Macao-18-1-118

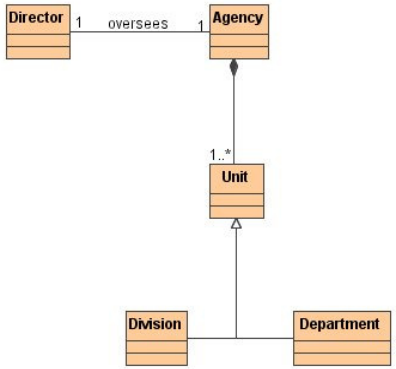
Diagram 1: Class

Class Diagrams:

1) show a set of classes, interfaces and collaborations, and their relationships
2) address static design view of a system

e-Macao-18-1-119

Class Diagram Example



e-Macao-18-1-120

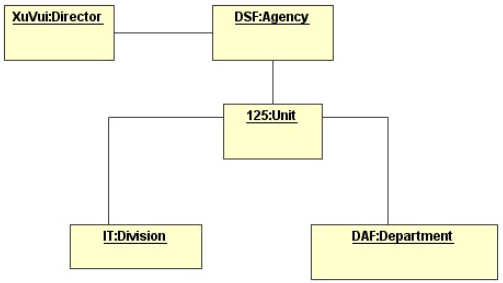
Diagram 2: Object

Object Diagrams:

1) show a set of objects and their relationships
2) static snapshots of element instances found in class diagrams

e-Macao-18-1-121

Object Diagram Example



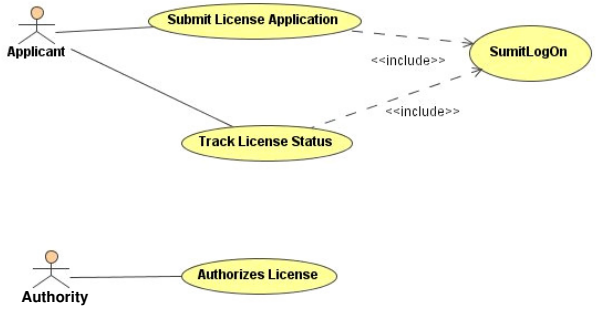
e-Macao-18-1-122

Diagram 3: Use Case

1) shows a set of actors and use cases, and their relationships
2) addresses static use case view of the system
3) is important for organizing and modelling the external behaviour of the system

e-Macao-18-1-123

Use Case Diagram Example



e-Macao-18-1-124

Diagram 4: Sequence

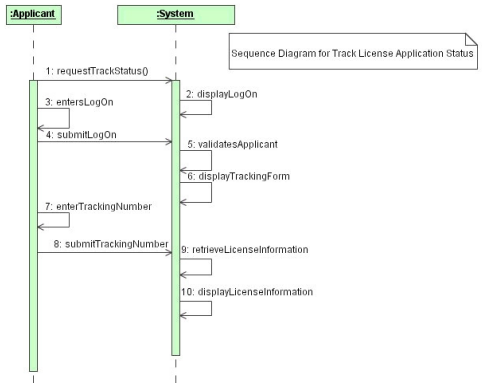
Sequence Diagrams:

1) shows interactions consisting of a set of objects and the messages sent and received by those objects

2) addresses the dynamic behaviour of a system with special emphasis on the chronological ordering of messages

e-Macao-18-1-125

Sequence Diagram Example



e-Macao-18-1-126

Diagram 5: Collaboration

Collaboration Diagram:

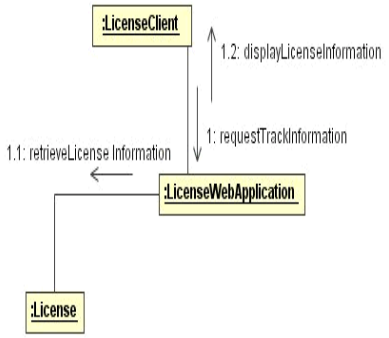
1) shows the structural organization of objects that send and receive messages

Sequence and collaboration diagrams:

- are jointly called interaction diagrams
- can be transformed one into another

e-Macao-18-1-127

Collaboration Diagram Example



e-Macao-18-1-128

Diagram 6: Statechart

1) shows a state machine consisting of states, transitions, events, and activities

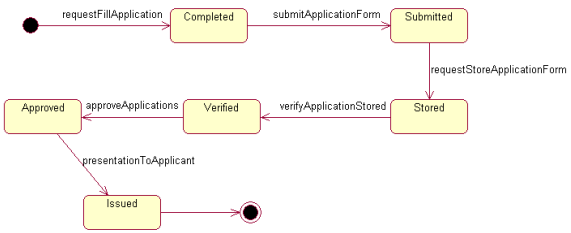
2) addresses the dynamic view of a system

3) is important in modelling the behaviour of an interface, class or collaboration

4) emphasizes the event-driven ordering

e-Macao-18-1-129

Statechart Diagram Example



e-Macao-18-1-130

Diagram 7: Activity

1) shows control/data flows from one activity to another

2) addresses the dynamic view of a system, useful for modelling its functions

3) emphasizes the flow of control among objects

e-Macao-18-1-130

Diagram 7: Activity

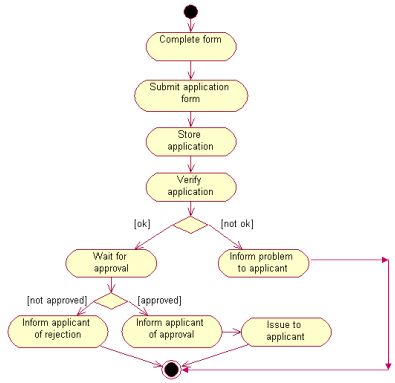
1) shows control/data flows from one activity to another

2) addresses the dynamic view of a system, useful for modelling its functions

3) emphasizes the flow of control among objects

e-Macao-18-1-131

Activity Diagram Example



e-Macao-18-1-132

Diagram 8: Component

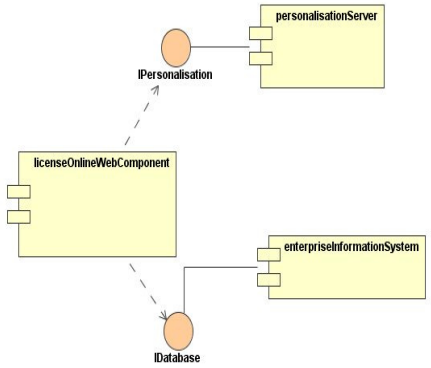
1) shows the organization and dependencies amongst a set of components

2) addresses static implementation view of a system

3) shows how components map to one or more classes, interfaces and collaborations

e-Macao-18-1-133

Component Diagram Example



e-Macao-18-1-134

Diagram 9: Deployment

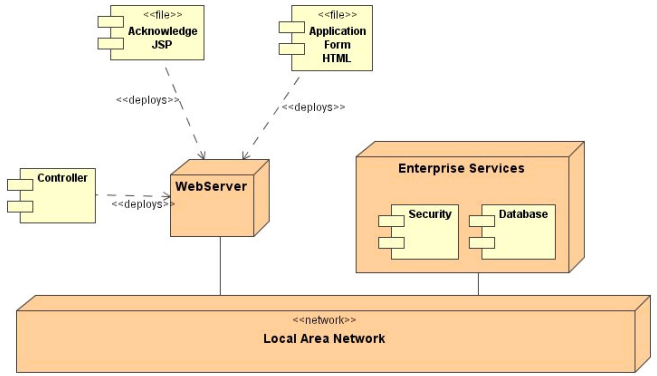
1) shows configuration of run-time processing nodes and the components hosted on them

2) addresses the static deployment view of an architecture

3) is related to component diagram with nodes hosting one or more components

e-Macao-18-1-135

Deployment Diagram Example



e-Macao-18-1-136

Task 16

Consider a software process consisting of the following activities: requirement gathering, object oriented analysis, object design, implementation and deployment.

List the diagrams that are essential for each of these activities.

Provide justifications for your choice of diagrams.

e-Macao-18-1-137

Task 17

Select the best answer:

The class diagram:

a) is the first model created in the project

b) is created after the other models

c) is used to specify objects and generate code

d) is used to create the sequence and the collaboration diagrams.

e-Macao-18-1-138

Task 18

Select the best answer:

The sequence diagram models:

a) the sequence of activities to implement the model

b) the way that objects communicate

c) the relationships among objects

d) the order in which the class diagram is constructed

e-Macao-18-1-139

Task 19

Select the best answer:

The collaboration diagram:

a) is a unique view of object behaviour

b) models the connections between different views

c) models the relationships between software and hardware components

d) models the way objects communicate

A.3.3. Modelling Views

e-Macao-18-1-140

Modelling Views

Modelling views – different perspectives on the system:

- Use Case View
- Design View
- Process View
- Implementation View
- Deployment View

A view is essentially a set of diagrams.

e-Macao-18-1-141

View 1: Use Case

Use Case View describes the behaviour of the system as seen by its end users, analysts and testers.

This view shapes the system architecture.

e-Macao-18-1-142

View 2: Design

Design View encompasses:

1) classes,
2) interfaces and
3) collaborations

that form the vocabulary of the problem and its solution.

e-Macao-18-1-143

View 3: Process

Process View encompasses threads and processes that form the system's concurrency and synchronization mechanisms.

This view addresses:

1) performance,
2) scalability and
3) throughput

of the system.

e-Macao-18-1-144

View 4: Implementation

Implementation View encompasses the components and files that are used to assemble and release the physical system.

This view addresses the configuration management of the system's releases.

e-Macao-18-1-145

View 5: Deployment

Deployment View encompasses the nodes that form the system's hardware topology on which the system executes.

e-Macao-18-1-146

Selecting Views

Different modelling views for different system types:

1) monolithic systems
2) distributed systems
3) etc.

e-Macao-18-1-147

Monolithic Systems

Two views are relevant:

1) Use case View: use case diagrams
2) Design View:
   a) class diagrams (structural modelling)
   b) interaction diagrams (behavioural modelling)
3) Process View: none
4) Implementation View: none
5) Deployment View: none

e-Macao-18-1-148

Distributed Systems

All five views are relevant:

1) Use case View:
   a) use case diagrams
   b) activity diagrams
2) Design View:
   a) class diagrams
   b) interaction diagrams
   c) statechart diagram
3) Process View:
   a) class diagram
   b) interaction diagrams
4) Implementation View: component diagrams
5) Deployment View: deployment diagrams

e-Macao-18-1-149

Summary 1

A model provides a blueprint of a system.

UML is a language for visualizing, specifying, constructing and documenting artifacts of software-intensive systems.

UML is process-independent but recommended for use with processes that are: use case driven, architecture-centric, iterative and incremental.

e-Macao-18-1-150

Summary 2

There are three building blocks which characterize UML: elements, relationships and diagrams.

Categories of elements in UML include:
1. structural
2. behavioural
3. grouping
4. annotation

There are four basic types of relationships in UML:
1. dependency
2. association
3. generalization
4. realization

e-Macao-18-1-151

Summary 3

UML provides 9 diagrams for modelling:

1. class diagrams
2. object diagrams
3. use case diagrams
4. sequence diagrams
5. collaboration diagrams
6. statechart diagrams
7. activity diagrams
8. component diagrams
9. deployment diagrams

There are five different modelling views in UML: use case, design, process, implementation and deployment.

A.4. Requirements

A.4.1. Software Requirements

Requirements Modelling

e-Macao-18-1-153

Course Outline

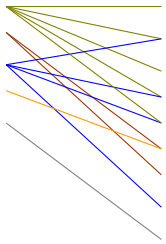
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-154

Requirements

Definition
A requirement is:

1) a function that a system must perform

2) a desired characteristic of a system

3) a statement about the proposed system that all stakeholders agree that must be true in order for the customer's problem to be adequately solved.

e-Macao-18-1-155

Requirements Process

Typically includes:

1) elicitation of requirements

2) modelling and analysis of requirements

3) specification of requirements

4) validation of requirements

5) requirements management

The process is not linear!

e-Macao-18-1-156

Functional or Non-Functional?

Functional requirements:

1) describe interaction between a system and its environment
2) describe how a system should behave under certain stimuli

Non-functional requirements:

1) describe restrictions on a system that limit the choices for its construction as a solution to a given problem

e-Macao-18-1-157

Requirements Types

1) functional
2) interface
3) data
4) human engineering
5) qualification
6) operational
7) design constraints
8) safety
9) security, etc.

e-Macao-18-1-158

Requirements Specification

Requirements must be expressed precisely.

Requirements specification should be:

1) correct
2) consistent
3) feasible
4) verifiable
5) complete
6) traceable

e-Macao-18-1-159

Requirements Standards

Various standards are available:

1) IEEE P1233/D3 Guide
2) IEEE Std. 1233 Guide
3) IEEE Std. 830-1998
4) ISO/IEC 12119-1994
5) IEEE std 1362-1998 (ConOps)

e-Macao-18-1-159

Requirements Standards

Various standards are available:

1) IEEE P1233/D3 Guide
2) IEEE Std. 1233 Guide
3) IEEE Std. 830-1998
4) ISO/IEC 12119-1994
5) IEEE std 1362-1998 (ConOps)

e-Macao-18-1-160

Requirements Template

req. id	F44
category	Functional
description	The applicants of social benefits, processing staff and approving authority shall be able to register in the system. Registration shall involve creating an account for the user for the purpose of authentication and personalisation of the system services. The agency staff users will be created by the system administrator. Applicants can register themselves by an option provided on the website.
terms	Account, Applicant, Authentication, Registration, User
justification	Registration is essential for subsequent authentication and personalisation functions.
priority	High
dependencies	F43
documents	
feasibility argument	There are mature solutions (e.g. directory services) for implementing the requirement.
verification method	When the system registers a new applicant who does not have an account, an account for the user is created by the system.

e-Macao-18-1-161

Task 20

1) Enumerate three functional requirements for the application issuing business registration licenses.

2) Enumerate two non-functional requirements for the same application.

3) Using the template provided earlier specify one of the functional requirements described in the first point.

e-Macao-18-1-162

Glossary 1

1) a set of terms that are defined and understood to form the basis for communication

2) a dictionary to carry out modelling

3) its purpose is to clarify the meaning of terms or to have the shared understanding of the terms amongst team members

4) is created during requirements definition, use case identification and conceptual modelling

5) is maintained throughout development

e-Macao-18-1-163

Glossary 2

It is usually the central place for:

1) definitions of key concepts
2) clarification of ambiguous terms and concepts
3) explanations of jargons
4) description of business events
5) description of software actions

There is no specific format for glossaries:

1) reference identification for terms
2) definitions
3) categories
4) cross references

e-Macao-18-1-164

Glossary Example

Id	Term	Definition
1	Agency	A government organization providing services to citizens, businesses and other government agencies.
2	Authentication	Proving a user's identity. To be able to access a Website or resource, a user must provide authentication via a password or some combination of tokens, biometrics and passwords.
3	Authorization	The act of granting approval. Authorization to resources or information within an application can be based on simple or complex access control methods.
4	Booking	A process of arranging a date and time when a citizen can visit a social welfare centre.
5	Bulletin	A message created by an agency that contains information about the social benefit service they want to post for advertisement.
6	Bulletin Board	A place on a computer system where citizens can read messages. All agencies can add their own messages.

e-Macao-18-1-165

Task 21

- Define a glossary containing two definitions for the business license issuing application.

A.4.2. Use Case Modelling

Requirements Modelling

Use Case Diagrams

e-Macao-18-1-167

Course Outline

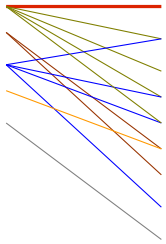
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-168

Use Cases

1) describe or capture functional requirements

2) represent the desired behaviour of the system

3) identify users (actors) of the system and associated processes

4) are the basic building blocks of use case diagrams

5) tie requirements phase to other development phases

e-Macao-18-1-169

Use Case Definition 1

A use case:

1. is a collection of task-related activities describing a discrete chunk of a system

2. describes a set of actions sequences that a system performs to present an observable result to an actor

3. describes a system from an external usage viewpoint

e-Macao-18-1-170

Use Case Definition 2

Key attributes of a use case:

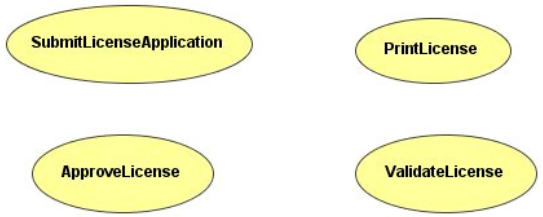
1) description
2) action sequence
3) includes variants
4) produces observable results

A use case does not describe:

1) user interfaces
2) performance goals
3) non-functional requirements

e-Macao-18-1-171

Use Case Examples



e-Macao-18-1-172

Use Case Relationships

Use cases are organized by relationships.

Three kinds:

1. generalization
2. include
3. extend

e-Macao-18-1-173

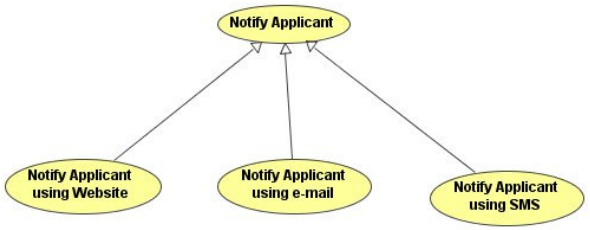
Relationship 1: Generalization

Generalization of use cases:

- the same meaning as before
- a more specialized use case is related to a more general use case

e-Macao-18-1-174

Generalization Example



e-Macao-18-1-175

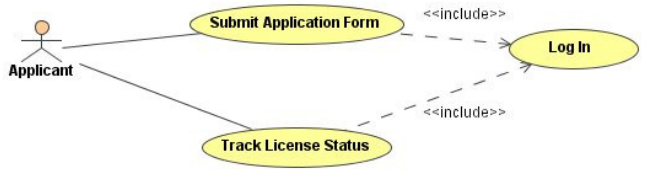
Relationship 2: Include

Include relationship between use cases:

- the base use case explicitly incorporates the behaviour of another use case at a location specified in the base
- the include relationship never stands alone, but is instantiated as part of some larger base of use cases
- rendered using the "include" stereotype

e-Macao-18-1-176

Include Example



e-Macao-18-1-177

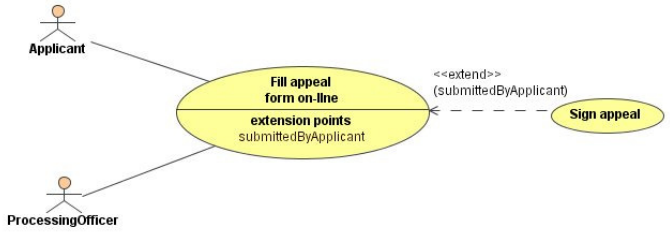
Relationship 3: Extend

Extend relationship between use cases:

a) the base use case implicitly incorporates the behaviour of another use case at a location specified by the extending use case (extension point)
b) the base use case may stand alone and usually executes without regards to extension points
c) depending on the system behaviour, the extension use case will be executed or not
d) rendered using the "extend" stereotype

e-Macao-18-1-178

Extend Example



e-Macao-18-1-179

Actor

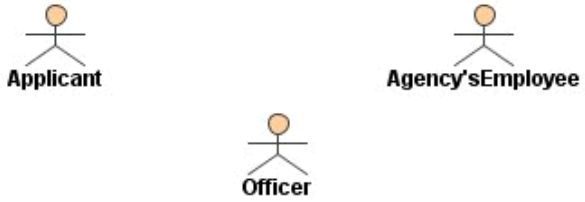
Definition
Actor is anyone or anything that interacts with the system causing it to respond to events.

An actor:

1) is something or somebody that stimulates the system to react or respond to its request
2) is something we do not have control over
3) represents a coherent set of roles that the external entities to the system can play
4) represents any type of a system's user

e-Macao-18-1-180

Actor Examples



e-Macao-18-1-181

Notes about Actors

1) actors stimulate a system providing input events and/or receive something output from the system

2) actors communicate with a system by sending and receiving messages to/from it, while performing use cases

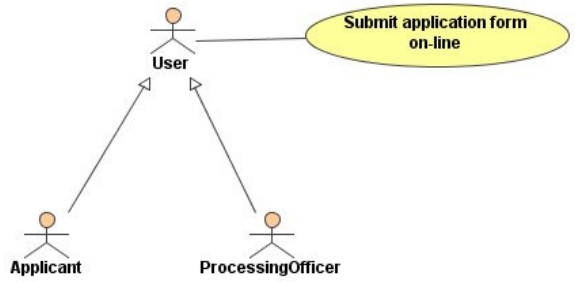
3) actors model anything that needs to interact with the system to exchange information – human users, computer systems and others

4) a user may act as one or several actors as it interacts with the system, while several individual users may act as different instances of one and the same actor

e-Macao-18-1-182

Relationship between Actors

It is also possible to relate actors using generalizations.



e-Macao-18-1-183

Types of Actors

Initiator versus participant:

– When there is more than one actor in a use case, the one that generates the stimulus is called the initiator and the others are participants.

Primary versus secondary:

– The actor that directly interacts with the system is called the primary actor, others are called secondary actors.

e-Macao-18-1-184

Task 22

- Based on the application for issuing business registration licenses define three use cases using a UML tool.
- Define a use case that extends one of the former use cases.
- Define a use case that can be included in one of the former use cases.

e-Macao-18-1-185

Task 23

Select the best answer:

An actor is:

a) a person

b) a job title

c) a role

d) a system

e-Macao-18-1-186

Task 24

- Add actors to the diagram produced in the previous Task.
- For each of the use cases in the previous task, enumerate all actors involved and indicate who is the initiator.

e-Macao-18-1-187

Use Case Diagrams

1) a diagram that shows a set of use cases and actors, and their relationships

2) is central to modelling the functions of the system

3) is used to visualize the functions of a system, so that:

   a) users can comprehend how to use the system and
   b) developers can understand how to implement it

4) puts everything together

e-Macao-18-1-188

Use Case Diagram Parts

A use case diagram commonly contains:

1) use cases
2) actors
3) relationships:
   a) dependency - between use cases
   b) generalization - between use cases or actors
   c) association - between use cases and actors

e-Macao-18-1-189

Identifying Use Cases

Use cases describe:

1. the functions that the user wants a system to accomplish
2. the operations that create, read, update, delete information
3. how actors are notified of the changes to the internal state and how they notify the system about external events

e-Macao-18-1-190

Identifying Actors

To determine who are the actors,
we try to answer the following questions:

1) who uses the system?
2) who gets information from the system?
3) who provides information to the system?
4) who installs, starts up or maintains the system?

e-Macao-18-1-191

Naming Use Cases

Use concrete verb-noun phrases:

1) a weak verb may indicate uncertainty, a strong verb may clearly identify the action taken:
   a) strong verbs: create, calculate, migrate, activate, etc.
   b) weak verbs: make, report, use, organize, record, etc.

2) a weak noun may refer to several objects, a strong noun clearly identifies only one object
   a) strong nouns: property, payment, transcript, etc.
   b) weak nouns: data, paper, report, system, etc.

e-Macao-18-1-192

Naming Actors

1) group individuals according to how they use the system by identifying the roles they adopt while using the system

2) each role is a potential actor

3) name each role and define its distinguishing characteristics

4) do not equate job titles with roles; roles cut across jobs

5) use common names for existing system; avoid inventing new

e-Macao-18-1-193

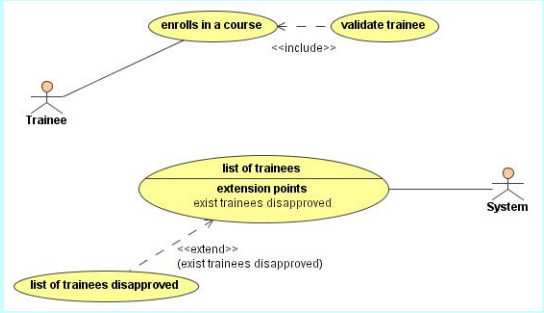
Use Case Template

Fields	Description
Use Case Name	Name of the use case
Actors	Role names of people or external entities initiating the use case
Purpose	The intention of the use case
Overview	A brief description of the usage of the process
Precondition	A condition that must hold before a use case can begin
Variation	Different ways to accomplish use case actions
Exceptions	What might go wrong during the execution of the use case
Policies	Specific rules that must be enforced by the use case
Post-conditions	Condition that must prevail after executing the use case
Priority	How important is the use case?
Frequency	How often is the use case performed?
Cross reference	Relate use cases and functional requirements

e-Macao-18-1-194

Task 25

Is this diagram correct ? If not, explain.



e-Macao-18-1-195

Task 26

In the following bank ATM-related use cases, what would best describe a use case of type *UseCaseA*?

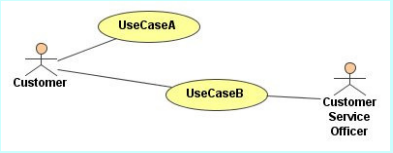
a) Open Account

b) Withdraw from Account

c) Close Account

d) Reopen Account

e) None of the above



A.4.3. Conceptual Modelling

Requirements Modelling

Class and Object Diagrams

e-Macao-18-1-197

Course Outline

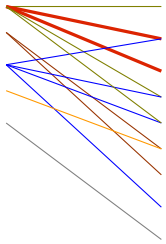
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-198

Concept Definition

What is a concept? An idea, thing or object.

An object can be:

1) represented symbolically
2) defined or described
3) exemplified

What is an instance? Each concrete application of the concept.

e-Macao-18-1-199

Concept Examples

1) agency
2) license
3) officer
4) applicant
5) internal applicant
6) external applicant

e-Macao-18-1-200

Concept Identification

Concepts can be:

1) physical or tangible objects
2) places
3) documents, specifications, design or descriptions
4) roles of people
5) container of other things
6) organizations
7) processes
8) catalogs, etc.

Concepts are identified through requirements and use cases.

e-Macao-18-1-201

Conceptual Model and Classes

Conceptual model:

1) captures the concepts in a domain in an abstract way
2) important part of OO requirements analysis

Classes:

1) equivalent to concepts in UML
2) an abstraction of a set of objects
3) objects are concrete entities existing in space and time

e-Macao-18-1-202

Class Diagrams

1) the most widely used diagram of UML

2) models the static design view of a system

3) also useful in modelling business objects

4) used to specify the structure, interfaces and relationships between classes that underlie the system architecture.

5) primary diagram for generating codes from UML models

e-Macao-18-1-203

A Class

A class is:

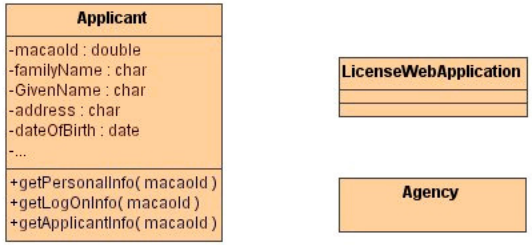
1) a description of a set of objects that share the same

   a) attributes,
   b) operations,
   c) relationships and
   d) semantics

2) a software unit that implements one or more interfaces

e-Macao-18-1-204

Class Examples



e-Macao-18-1-205

Class Notation

Basic notation: a solid-outline rectangle with three compartments separated by horizontal lines.

Three compartments:

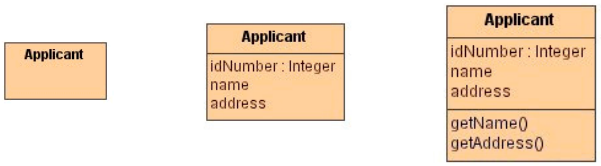
1) top compartment holds the class name and other general properties of the class
2) middle compartment holds a list of attributes
3) bottom compartment holds a list of operations

Alternative styles:

1) suppress the attributes compartment
2) suppress the operation compartment

e-Macao-18-1-206

Class Notation Example



e-Macao-18-1-207

Stereotypes

1) a mechanism allowing to extend the semantics of UML

2) used to present more information about an artifact

3) notation – the name of a new element within the matched guillemets, e.g. <<thread>>

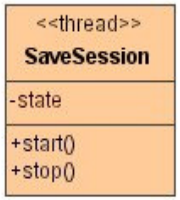
e-Macao-18-1-208

Stereotypes for Classes

Attribute or operation lists in a class may be organized into groups with stereotypes.

A number of stereotypes of classes and data types:

1) thread
2) event
3) entity
4) process
5) utility
6) metaclass
7) powerclass
8) enumeration, etc.



e-Macao-18-1-209

Stereotype Descriptions

Stereotype	Description
Thread	An active class which specifies a lightweight flow that can execute concurrently with other threads within the same processes.
Process	An active class which specifies a heavyweight flow that can execute concurrently with other processes.
Control	A class which owns almost no information about itself. It represents a behaviour rather than resources and directs the behaviour of other objects almost having no behaviour of its own.
Entity	A class which represents a resource in the real world. It describes its features and their current condition (their state) and preserves its own integrity regardless of where and when it is used.
Utility	A class whose attributes and operations are all class scoped. That is a class which no instance.
Metaclass	A classifier whose objects are all classes.
Powerclass	A classifier whose objects are the children of a given parent.
Enumeration	A user defined data type that defines a set of values that do not change.

e-Macao-18-1-210

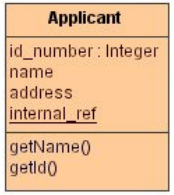
Attribute/Operation Scope

Different scopes can be specified for the attributes and operations (features) of a class:

1) instance scope - a feature appears in each instance of the class (classifier)

2) class scope - there is just a single instance of the feature for all instances of a class (classifier)

Underlining the feature's name indicates the classifier scope.



e-Macao-18-1-211

Types of Classes

1) abstract class
   a) cannot have direct instances
   b) the name is written in italics

2) root class
   a) cannot be a sub-class
   b) written with *root* stereotype

3) leaf class
   a) cannot be a super-class
   b) written with *leaf* stereotype

e-Macao-18-1-212

Class Relationships

Four relationships between classes:

1) association
2) aggregation
3) generalization
4) dependency

e-Macao-18-1-213

Relationship 1: Association

Association - a relationship between two or more classifiers that involves connection among instances.

Person ———— Company

Example: Person works for the Company.

e-Macao-18-1-214

Relationship 2: Aggregation

Aggregation - A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.

Car ◇——— Engine

Example: Car has an Engine.

e-Macao-18-1-215

Relationship 3: Generalization

Generalization - A taxonomic relationship between a more general and a more specific element.

Truck ———▷ Vehicle

Example: Truck is a Vehicle.

e-Macao-18-1-216

Relationship 4: Dependency

Dependency - A relationship between two elements, in which a change to one element (source) will affect another (target).

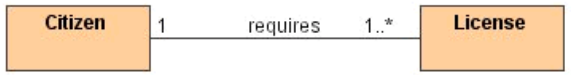
Policy - - - - - - - - -> Procedure

Example: Procedure depends on the Policy.

e-Macao-18-1-217

Multiplicities for Classes

Shows how many objects of one class can be associated with one object of another class



Example: a citizen can apply for one or more licenses, and a license is required by one citizen.

e-Macao-18-1-218

Multiplicities Syntax

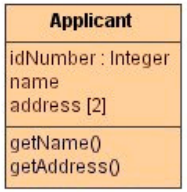
Value	Description
0..0	Zero
0..1	Zero or one
0..*	Zero or more
1..1	One
1..*	One or more
*	Unlimited number
<literal>	Exact number (Example: 4)
<literal>..*	Exact number or more (Example: 4..* indicating 4 or more)
<literal>..<literal>	Specified range (Example: 4..13)
<literal>..<literal>, <literal>	Specified range or exact number (Example: 4..13,31 indicating 4 through 13 and 31)
<literal>..<literal>,<literal>..<literal>	Multiple specified ranges (Example: 4..13, 31-41)

e-Macao-18-1-219

Multiplicities for Attributes

Can specify how may instances of an attribute can be associated with one instance of the class.

Example: an applicant can have two addresses.



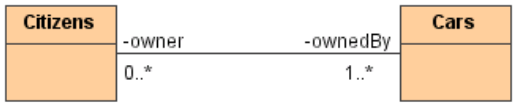
e-Macao-18-1-220

Roles

A role names a behaviour of an entity participating in a particular relationship.

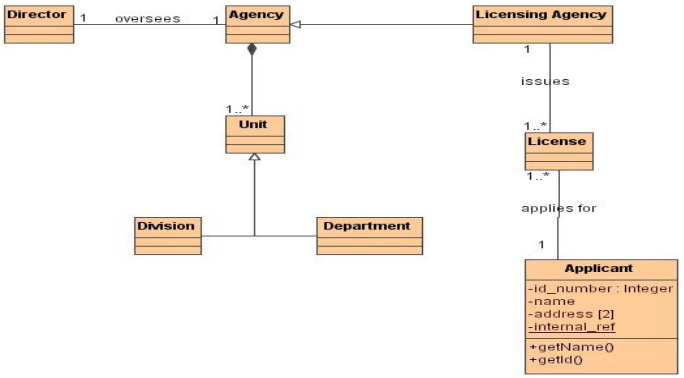
Notation: add the name of the role at the end of the association line, next to the class icon.

Example: a citizen own a car, sell cars, be an employee, etc.



e-Macao-18-1-221

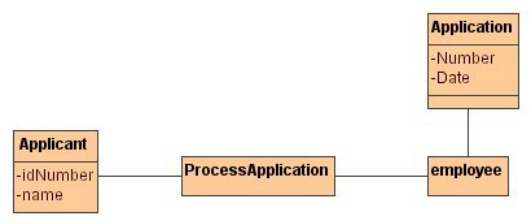
Class Diagram Example



e-Macao-18-1-222

Task 27

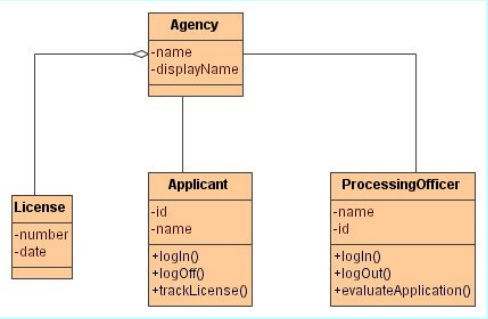
Is this diagram correct ? If not, explain.



e-Macao-18-1-223

Task 28

Discover and correct all mistakes in the following diagram :



e-Macao-18-1-224

Task 29

Using a UML tool produce a class diagram with all the classes and relationships defined in the Tasks 6, 8 and 10.

e-Macao-18-1-225

Object Diagram

1) models the instances of classes contained in class diagrams
2) shows a set of objects and their relationships at one time
3) modelling object structures involves taking a snapshot of a system at a given moment in time
4) is an instance of a class diagram or the static part of an interaction diagram
5) it contains objects and links

e-Macao-18-1-226

Object Diagram Usage

Object diagrams are used to:

1) visualize
2) specify
3) construct
4) document

The existence of certain instances in a system, together with their relationships.

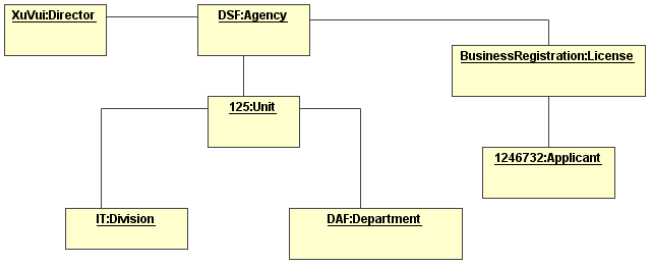
e-Macao-18-1-227

Creating an Object Diagram

1) identify the function/behaviour of interest that results from interaction of classes, interfaces and other artifacts
2) for each function/behaviour, identify the artifacts that participate in the collaboration as well as their relationships
3) consider one scenario that invokes the function/behaviour, freeze the scenario and render each participating object
4) expose the state and attribute values of each object, as necessary to understand the scenario
5) expose the links among these objects

e-Macao-18-1-228

Object Diagram Example



e-Macao-18-1-229

Task 30

Using a UML tool produce an object diagram illustrating the class diagram shown in the previous task.

e-Macao-18-1-230

Task 31

Select the best answer:

An object is named:

a) with a noun

b) with a noun, a colon and then a class name

c) with a number and a colon followed by a class name

d) just like a class

e-Macao-18-1-231

Task 32

Select the best answer:

The object diagram is used for:

a) testing and verifying the use cases

b) modelling scenarios

c) analyzing and testing class diagrams

A.4.4. Behavioural Modelling

Requirements Modelling

Sequence and State Diagrams

e-Macao-18-1-233

Course Outline

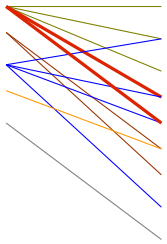
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-234

Behavioral Diagrams 1

1) represent how objects behave when you put them to work using the structure already defined in structural diagrams

2) model how the objects communicate in order to accomplish tasks within the operation of the system

3) describe how the system:

   a) responds to actions from the users
   b) maintains internal integrity
   c) moves data
   d) creates and manipulates objects, etc.

e-Macao-18-1-235

Behavioral Diagrams 2

4) describe discrete pieces of the system, such as individual scenarios or operations

Note: not all system behaviour have to be specified - simple behaviours may not need a visual explanation of the communication required to accomplish them.

e-Macao-18-1-236

Scenario

Definition
Scenario is a textual description of how a system behaves under a specific set of circumstances.

Behaviour described in use cases is the basis for scenarios.

Scenarios also provide a basis for developing test cases and acceptance-level test plans.

e-Macao-18-1-237

Scenario Example

Consider the use case "Track License Application"

There are at least two possible scenarios:

a) the applicant enters the license application number; the system retrieves the information related to it; the system displays this information

b) the applicant enters the license application number; the number does not exist in the agency's database; the system displays an error message

e-Macao-18-1-238

Scenario Example

Scenario: the applicant enters the license application number; the system retrieves the information related to it; the system displays this information.

Steps:
1) Applicant requests to track status of a license application
2) System displays the logon form
3) Applicant enters the logon information
4) Applicant submits the logon information
5) System validates the applicant
6) System displays the form to enter the tracking number
7) Applicant enters the tracking number
8) Applicant submits the license number
9) System retrieves the license information
10) System displays the license information

e-Macao-18-1-239

Task 33

Select a use case among those defined on Task 22 and define a possible scenario for it.

e-Macao-18-1-240

Task 34

Select the best answer:

Scenarios are:

a) the same as use cases

b) the same as test cases

c) used to derive test cases

d) the same as object diagrams

e-Macao-18-1-241

Sequence Diagram

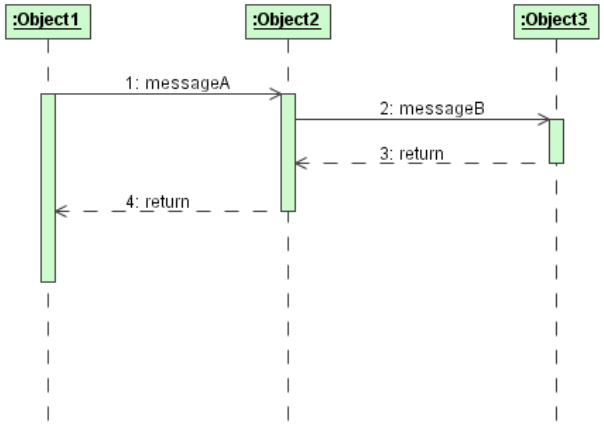
A sequence diagram shows interactions between objects.

Components of sequence diagrams:

1) object lifelines
   a) object
   b) timeline
2) messages
   a) message, stimulus
   b) signal, exception
   c) operations, returns
   d) identification
3) message syntax

e-Macao-18-1-242

Sequence Diagram Example



e-Macao-18-1-243

Timeline

The *timeline* is a line that runs:

1. from the beginning of a scenario at the top of the diagram
2. to the end of the scenario at the bottom of the diagram.

e-Macao-18-1-244

Object Lifeline

An object lifeline consists of:

1) an object
2) a timeline

If an object is created and destroyed
during the message sequence,
the lifeline represents its whole lifecycle.

e-Macao-18-1-245

Message

Definition
Message is a description of some type of communication between objects.

A unit of communication between objects.

The sender object may:

1) invoke an operation,
2) raise a signal or
3) cause the creation or destruction of the target object.

e-Macao-18-1-246

Message Notation

Notation: modelled as an arrow where the tail of the arrow identifies the sender and the head points to the receiver.

sender ⟶ receiver

For each message we need to define:

1) the name of the invoked operation and its parameters
2) the information returned from the message

e-Macao-18-1-247

Stimulus

Definition
Stimulus is an item of communication between two objects:

1) it is associated with both sending and receiving objects

2) it travels across a link

3) it may:
   a) invoke an operation,
   b) raise a signal (asynchronous message),
   c) create or destroy an object

4) it may include parameters/arguments in the form of primitive values or object references

5) it is associated with a procedure that causes it to be sent

e-Macao-18-1-248

Stimulus Example

1) Example 1 – invoke an operation

   The citizen object submits the application form by sending a message to the system. The system receives the message and executes the associated procedure.

1) Example 2 – create an object

   When the System receives the message from the applicant submitting the application form, it creates an object to support the new session.

e-Macao-18-1-249

Messages and Stimuli

A message is the specification of a stimulus.

The specification includes:

1) the roles the sender/receiver objects play in the interaction
2) the procedure that dispatches the stimulus

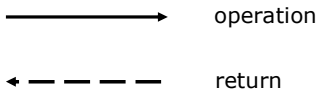
A stimulus is an instance of a message.

e-Macao-18-1-250

Operations and Returns

The operation specifies the procedure that the message invokes on the receiving object.

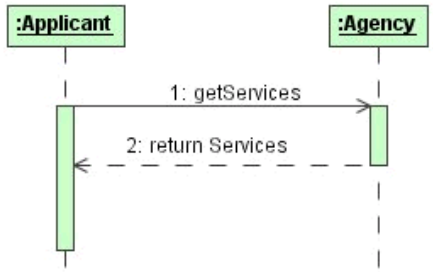
The return contains the information passed back from the receiver to the sender. An empty return is valid.



e-Macao-18-1-251

Operations/Returns Example

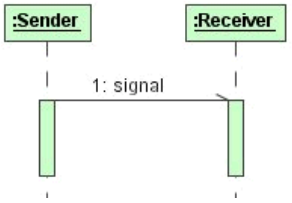
Example: The *Applicant* object sends a message to the *Agency* object to get the list of services it provides. The *Agency* object returns this information.



e-Macao-18-1-252

Signal

1) an object may raise a signal through a message
2) a signal is a special type of a class associated with an event that can trigger a procedure within the receiving object
3) a signal does not require a return from the receiving object



e-Macao-18-1-253

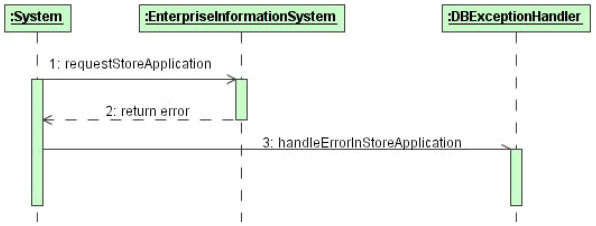
Exception

1) an exception is a special type of a signal
2) throwing an exception means sending out a message containing an object that describes the error condition

e-Macao-18-1-254

Signal Example

Each time System receives a message indicating that some abnormal situation occurred with the database, it raises a signal to DBExceptionHandler to handle the error.

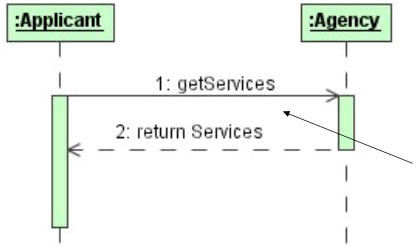


e-Macao-18-1-255

Identification of Messages

A message number or name is used to identify messages.

Example:



We identify the message getServices with 1.

e-Macao-18-1-256

Message Syntax 1

All former concepts – messages, signals, operations, returns, etc. are part of the message syntax.

predecessors '/' sequence-term
iteration [condition] return ':=' operation

where:

1) predecessors is a comma-separated list of sequence numbers of all messages that must come before the current message

2) sequence-term may be either a number or a name that identifies the message

e-Macao-18-1-257

Message Syntax 2

3) iteration determines if a message should be sent once or several times in a sequence:
   a) one message - add an iteration symbol (*) and a condition to control the number of iterations
   b) many messages - enclose the set of messages in a box

4) condition specifies the control of the iteration; expressed as a text enclosed within square brackets

5) return may include a list of values sent back to the sender

6) operation defines the name of the operation and optionally its parameters and a return value

e-Macao-18-1-258

Message Syntax Example

Example:

6/8:getAddress * [foreach ApplicationForm]
return text*:*= getAddress(Citizen.Id:Integer)

1. Specifies the message number 8 called getAddress.
2. The message will be executed more than once (*), one time for each ApplicationForm.
3. Each message calls the operation getAddress of the receiving object, sending CitizenId parameter of type Integer, and returns a value of type text.
4. For the execution of this message, it is required that the message 6 has already been executed.

e-Macao-18-1-259

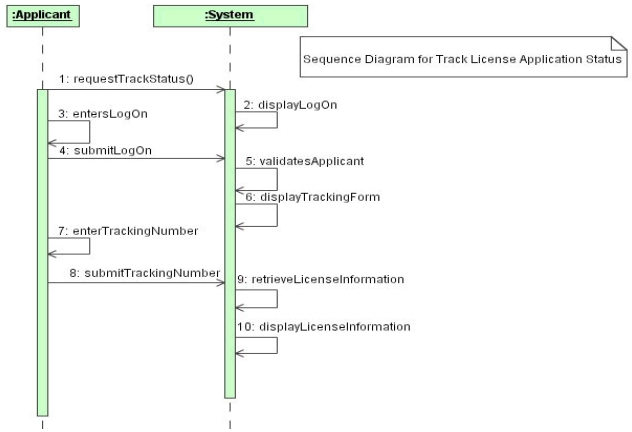
Example Scenario

Recall the scenario: an applicant tracks the status of a license application and the system displays the license information.

Procedure:
1. Applicant requests to track the status of a license application
2. System displays the logon form
3. Applicant enters the logon information
4. Applicant submits the logon information
5. System validates the applicant
6. System displays the form to enter the tracking number
7. Applicant enters the tracking number
8. Applicant submits the tracking number
9. System retrieves the license information
10. System displays the license information

e-Macao-18-1-260

Example Sequence Diagram



e-Macao-18-1-261

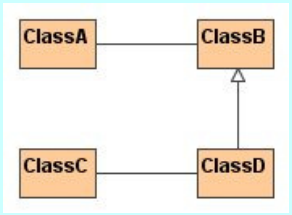
Task 35

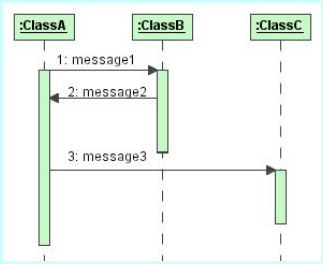
- Based on the scenario defined in the previous Task, produce the sequence diagram using the UML tool.

e-Macao-18-1-262

Task 36

Is there any problem in the following sequence diagram? Explain.





e-Macao-18-1-263

Task 37

Select the best answer:

A timeline is:

a) an event signaling the termination of a timed process, much the same as an alarm on a timer

b) another name for a sequence

c) used in a sequence diagram as an alternative to numbering events

d) the amount of time it takes to complete a set of iterations

e-Macao-18-1-264

Statechart Diagrams

A statechart diagram defines the behaviour of a single object or of a set of objects related by a collaboration.

It captures the changes in an object throughout its lifecycle as they occur in response to internal and external events.

The scope of a statechart is the entire life of one object.

e-Macao-18-1-265

Statechart Components

Statecharts are composed of:

1) states
   a) initial state
   b) final state
2) events
   a) guard conditions
   b) actions
   c) event syntax
3) complex states
   a) activities
   b) entry actions
   c) exit actions

e-Macao-18-1-266

State

Definition
State is the current condition of an object reflected by the values of its attributes and its links to other objects.

Notation:

Particular states are initial and final states.

e-Macao-18-1-267

Initial State

The initial state identifies the state in which an object is created or constructed.

The initial state is called a pseudo-state because it does not really have the features of an actual state, but it helps clarify the purpose of another state of the diagram.

Notation:

e-Macao-18-1-268

Final State

The final state is the state in which once reached, an object can never do a transition to another state.

The final state may also mean that the object has actually been destroyed and can no longer be accessed.

Notation:

e-Macao-18-1-269

Event

Definition
Event is an occurrence of a stimulus that can trigger a state transition.

An event may be:

1) the receipt of a signal, e.g. the reception of an exception

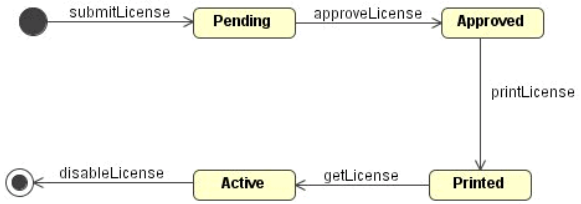
2) the receipt of a call, that is the invocation of an operation, e.g. for changing the expiration date of a license

An event on a statechart diagram corresponds to a message on a sequence diagram.

Notation: 

e-Macao-18-1-270

Statechart Diagram Example

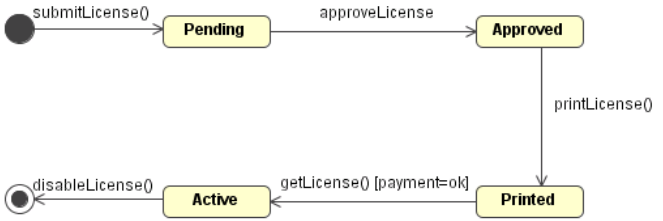


e-Macao-18-1-271

Guard Condition

Typically, an event is received and responded unconditionally.

When the receipt of an event is conditional, the test needed is called the guard condition.



e-Macao-18-1-272

Event Actions

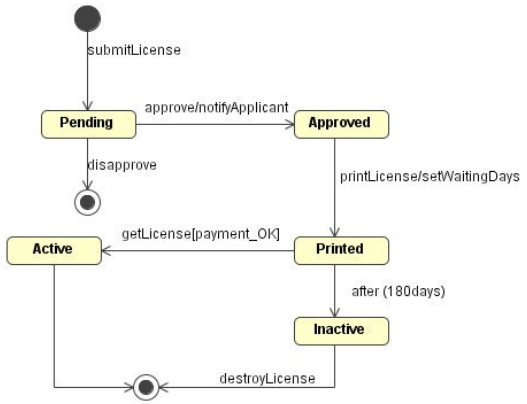
The response to an event has to explain how to change the attribute values that define the object's state.

The behaviour associated with an event is called action expression:

1. Part of a transition event – specifying the change from one state to another.
2. An atomic model of execution, referred to as run-to-completion semantics.

e-Macao-18-1-273

Event Actions Example



e-Macao-18-1-274

Event Syntax

event-name ‘(‘ [comma-separated-parameters-list] ‘) ‘
[‘[‘guard-condition’]’] / [action-expression]

where:

1) event-name - identifies the event
2) parameters-list - data values passed with the event for use by the receiving object in its response to the event
3) guard-condition - determines whether the receiving object should respond to the event
4) action-expression - defines how the receiving object must respond to the event

e-Macao-18-1-275

Event Syntax Example

Example:

approveLicense(License.Id) [req=ok]
/setExistLicense(true)

where:

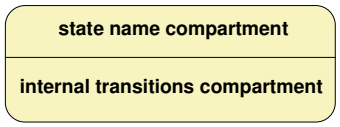
1) approveLicense is the event name
2) License.Id is the event parameter
3) the guard condition specifies that the req attribute must be OK for the receiving object to respond to the event.
4) the action executed in the receiving object is a call to the method setExistLicense which sends true as its parameter

e-Macao-18-1-276

Complex States

The state icon can be expanded to model what the object can do while it is in a given state.

The notation splits the state icon into two compartments:
1. name compartment and
2. internal transitions compartment.



Internal transitions compartments contains information about actions, activities and internal transitions specific to that state.

e-Macao-18-1-277

Entry Actions

More than one event can trigger a transition of an object into the same state.

When the same action takes place in all events that goes into a state, the action may be written once as entry action.

Notation:

1) use the keyword entry followed by a slash and the actions to be performed every time the state is entered
2) entry actions are part of internal transitions compartment

e-Macao-18-1-278

Exit Actions

The same simplification can be used for actions associated with events that trigger a transition out of a state.

They are called exit actions.

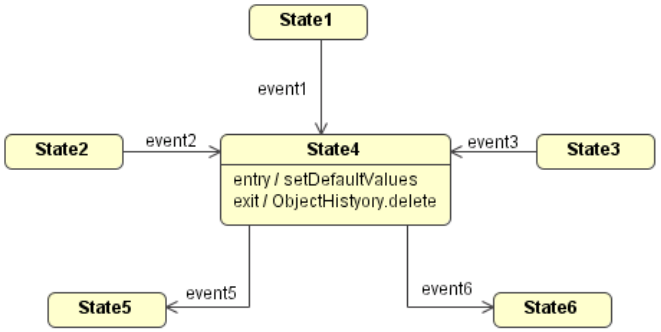
Notation:

1) use the keyword exit followed by a slash and the actions performed every time the state is exited
2) exit actions are part of the internal transitions compartment

Only when the action takes places every time the state is exited.

e-Macao-18-1-279

Entry/Exit Actions Example



e-Macao-18-1-280

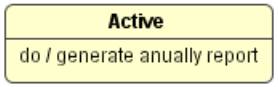
Activities 1

Activities are the processes performed within a state.

Activities may be interrupted because they do not affect the state of the object.

Notation:

1) use the do keyword followed by a slash and activities
2) activities are placed in the internal transitions compartment.



e-Macao-18-1-281

Activities 2

An activity should be performed:

1) from the time the object enters the state
2) until
   a) either the object leaves the state or
   b) the activity finishes.

If an event produces a transition out of the activity state, the object must shut down properly and exit the state.

e-Macao-18-1-282

Internal Transitions

An event that can be handled completely without a change in state is called an internal transition.

It also can specify guard conditions and actions.

Notation:

1) uses the keyword internal transition followed by a slash and one event action
2) they are placed in the internal transitions compartment

e-Macao-18-1-283

Order of Events

1) if an activity is processed in the current state, interrupt it and finish it properly
2) execute the exit actions
3) execute the actions associated with the event that triggered the transition
4) execute the entry actions of the new state
5) begin executing the activities of the new state

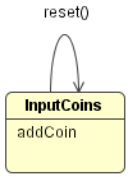
e-Macao-18-1-284

Self Transition

A self-transition is an event that is sufficiently significant to interrupt what the object is doing.

It forces the object to exit the current state and return to the same state.

The result is to stop any activity within the object, exit the current state and re-enter the state.



e-Macao-18-1-285

Important Features

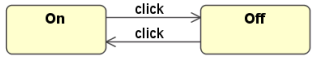
Within the statechart diagram:

1) an object need not know who sent the message
2) an object is only responsible for how it responds to the event

Focusing on the condition of the object and how it responds to the events, which object sends the message becomes irrelevant and the model is simplified

The state of the object when it receives an event can affect the object's response

event + state = response



e-Macao-18-1-286

Defining Send Events

Sometimes the object modelled by the statechart needs to send a message to another object, in this case the outgoing event must define which is the receiving object. Also, this event must be caught by the receiving object.

A message send to another object is called a send event.

Notation: provide the object name before the action expression with a period separating both.

e-Macao-18-1-287

Relating Diagrams 1

1) the sequence diagram models the interactions between objects

2) the statechart diagram models the effect that these interactions have on the internal structure of each object

3) the messages modelled in the sequence diagrams are the external events that place demands on objects

e-Macao-18-1-288

Relating Diagrams 2

4) the objects internal responses to those events that cause changes to the objects' states are represented in the statechart diagram

5) not all objects need to be modelled with a statechart diagram

6) the objects that appear in many interactions and are target of many events are good candidates to be modelled with a statechart diagram

e-Macao-18-1-289

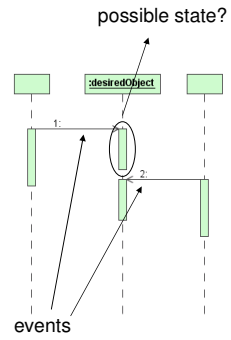
Sequence to Statechart

1) identify the events directed at the lifeline of the desired object

2) identify candidate states by isolating the portions of the lifeline between the incoming events

3) name the candidate states using adjectives that describe the condition of the object during the period of time represented by the gap

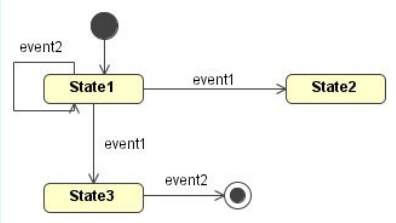
4) add the new state and events to the statechart diagram



e-Macao-18-1-290

Task 38

Discover and correct all mistakes in the following diagram:



e-Macao-18-1-291

Task 39

Select the best answer:

A state is the condition of an object

a) upon construction

b) that governs whether an event will trigger a transition

c) at the beginning and at the end of a scenario

d) at a point in time

e-Macao-18-1-292

Task 40

Select the best answer:

An event action is an action that:

a) must take place when an object enters the state

b) must take place when triggered by the associated event

c) must always take place when an object leaves the state

d) causes no change in an object's state

e-Macao-18-1-293

Task 41

- Using a UML tool define a statechart diagram for the license class. The diagram should contain at least three states.
- What is the relation between this diagram and the class diagram defined in the conceptual modelling?
- What is the relation between this diagram and the set of sequence diagrams?

e-Macao-18-1-294

Summary 1

A requirement is a function that a system must perform or a desirable characteristic of a system.

There are different kinds of requirements such as functional and non-functional requirements.

Most project failures can be traced back to errors made in the requirements gathering and specification.

e-Macao-18-1-295

Summary 2

Use cases are descriptions of sets of action sequences that a system performs to deliver observable results.

Use cases may be related using generalization, include and extend relationships.

Actors are the entities that interact with the system, causing it to respond to events.

Use case diagrams show a set of:
1. use cases,
2. actors and
3. their relationships.

e-Macao-18-1-296

Summary 3

Conceptual modelling helps to understand application domains.

Conceptual class diagrams describe and relate the concepts of a domain. They show attributes but not methods.

An object diagram models the instances of the classes contained in a given class diagram.

e-Macao-18-1-297

Summary 4

Behavioral modelling specifies how objects work together to provide a specific behaviour.

Sequence diagrams show how objects interact in order to deliver a discrete piece of the system functionality.

Statechart diagrams capture the changes in an object or a set of related objects as they occur in response to events.

e-Macao-18-1-298

Summary 5

Sequence diagrams:

1) show interactions between objects.

2) model the dynamic behaviour of a system.

3) emphasize the chronological ordering of messages.

e-Macao-18-1-299

Summary 6

Statechart diagrams:

1) show a state machine consisting of states, transitions, events and activities.

2) model the dynamic behaviour of a system.

3) emphasize the event-driven ordering.

A.5. Architecture

A.5.1. Software Architecture

Architecture Modelling

e-Macao-18-1-301

Course Outline

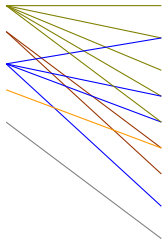
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-302

Architecture

Architecture involves defining:

a) what are the main components of the system
b) how this components are related.

The architecture shows:

a) the structural organization of a system from its components
b) how elements interact to provide the system's overall behaviour or required functionality

Architectural concerns:

a) structural or static
b) behavioural

e-Macao-18-1-303

Architecture Definition

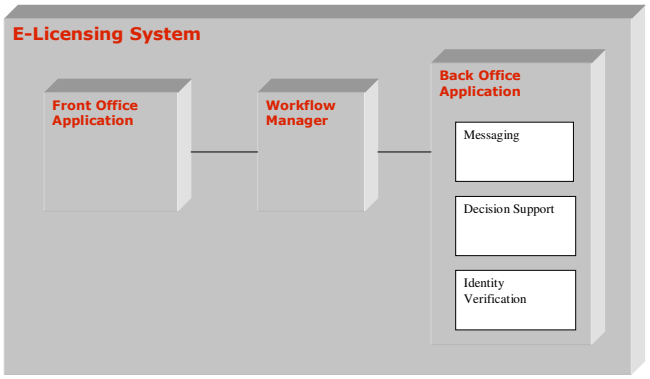
Definition

Architecture is a set of significant decisions about the organization of a software system. Such decisions include:

1. the selection of structural elements and their interfaces
2. the composition of these structural and behavioural element into progressively larger subsystem
3. the architectural style that guides this organization – the elements and their interfaces, their collaboration and composition

e-Macao-18-1-304

Architecture Example



e-Macao-18-1-305

Architecture Concepts

Some concepts related to architecture:

1) subsystems
   a) classes
   b) services

2) design principles for defining subsystems:
   1) coupling
   2) cohesion

3) layering strategy for defining subsystems:
   1) responsibility driven
   2) reuse driven

e-Macao-18-1-306

Subsystems: Classes

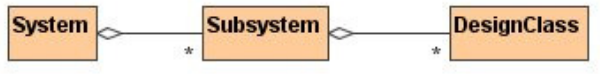
A solution domain may be decomposed into smaller parts called subsystems.

Subsystems may be recursively decomposed into simpler subsystems.

Subsystems are composed of solution domain classes (design classes).

e-Macao-18-1-307

Subsystems: Classes Example



e-Macao-18-1-308

Subsystems: Services

A subsystem is characterized by the services it provides to other subsystems.

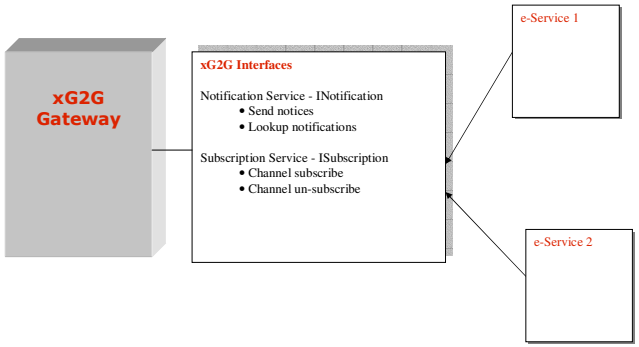
A service is:

1) a set of related operations that share a common purpose

2) a set of operations of a subsystem that are available to other subsystem through the subsystem's interface

e-Macao-18-1-309

Subsystem: Services Example



e-Macao-18-1-310

Coupling

Definition
Coupling is the strength of dependencies between two sub-systems.

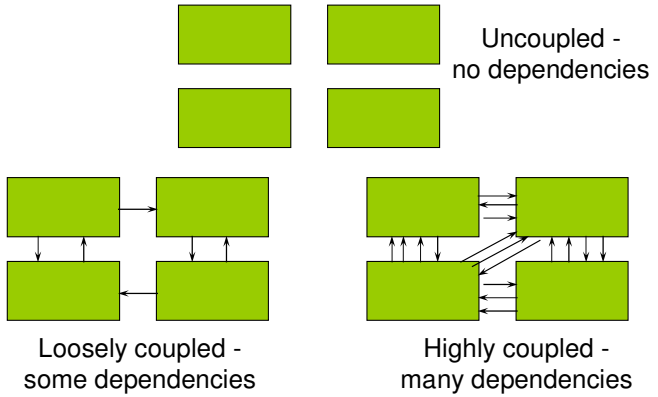
Loose coupling results in:

1) sub-system independence
2) better understanding of sub-systems
3) easier modification and maintenance

High coupling is generally undesirable.

e-Macao-18-1-311

Coupling Example



e-Macao-18-1-312

Cohesion

Definition
Cohesion or Coherence is the strength of dependencies within a subsystem.

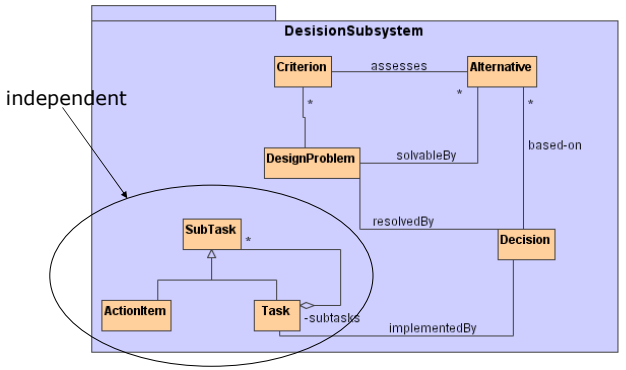
In a highly cohesive subsystem:

- subsystem contains related objects
- all elements are directed toward and essential for performing the same task.

Low cohesion is generally undesirable

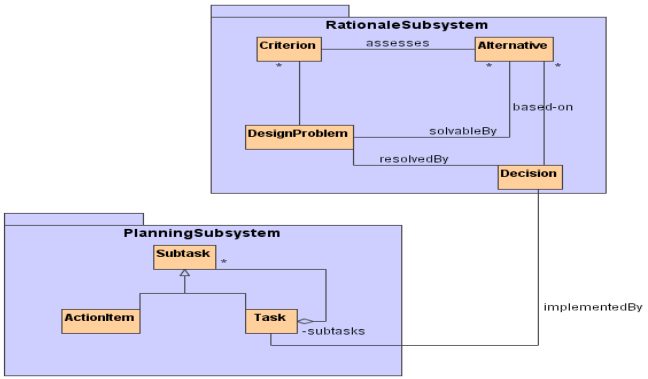
e-Macao-18-1-313

Low Cohesion Example



e-Macao-18-1-314

High Cohesion Example



e-Macao-18-1-315

Layering

Layering is a strategy for dividing system into subsystems.

Layering:

1) divides a system into a hierarchy of subsystems
2) follows two common approaches:
    a) responsibility driven
    b) reuse driven

e-Macao-18-1-316

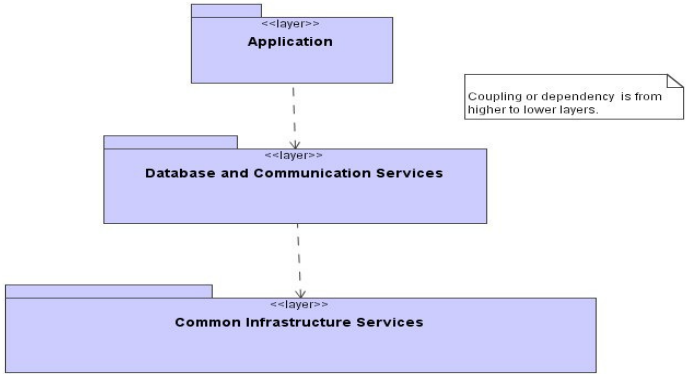
Layering styles

1) responsibility driven:
    a) layers have well-defined responsibilities
    b) layers fulfill specific roles

2) reuse driven:
    a) layers are designed to allow maximum reuse of system elements
    b) higher level layers use services of lower level layers

e-Macao-18-1-317

Layered Architecture Example



e-Macao-18-1-318

Task 42

1) Consider a typical client-server application. Assume that the client subsystem is a layer and the server is the other layer. Which of the two layering styles best describes this architecture?

2) Justify your answer in question 1.

A.5.2. Collaboration Diagrams

Architecture Modelling

Collaboration Diagrams

e-Macao-18-1-320

Course Outline

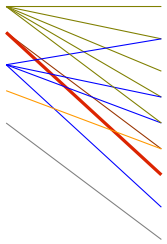
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-321

Collaboration

Definition

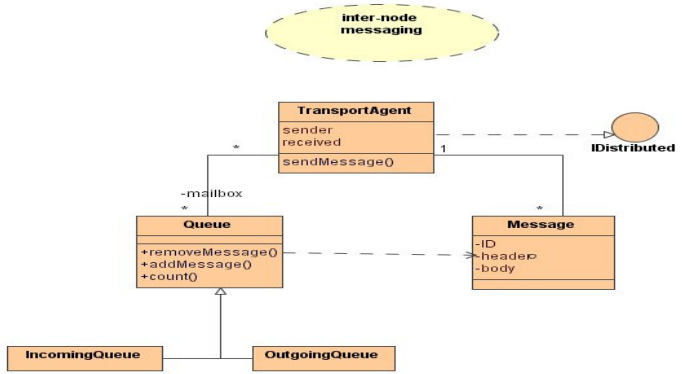
A collaboration is a society of classes, interfaces, and other elements that work together to deliver or provide some cooperative behaviour that is bigger than the sum of all its parts.

A collaboration:

1) names a conceptual chunk that encompasses both static and dynamic aspects
2) specifies the realization of a use case

e-Macao-18-1-322

Collaboration Example



Inter-node Messaging Collaboration and its details

e-Macao-18-1-323

Collaboration Names

1) every collaboration must have a name
2) collaboration names are nouns or short noun phrases, typically first letter of the letter is capitalized.

Example: *"Inter-node Messaging"* or *"Application Submission"*

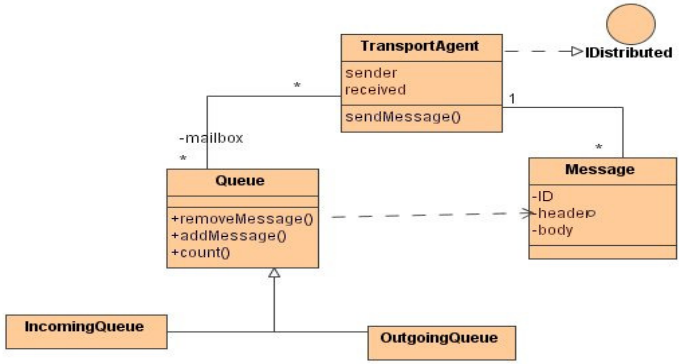
e-Macao-18-1-324

Collaboration – Structural

1) specifies the classifiers, such as classes, interfaces, components and nodes that are required to interact
2) does not own any of its structural elements
3) only references the classifiers declared elsewhere

e-Macao-18-1-325

Collaboration Structural View



Structural View of Collaboration which implements a "send and receive message" use case

e-Macao-18-1-326

Task 43

Select one of the use cases already defined restaurant licensing application:

1) identify the classes that are essential in realizing this use case
2) show the relationship between these classes
3) name the collaboration defined by these classes

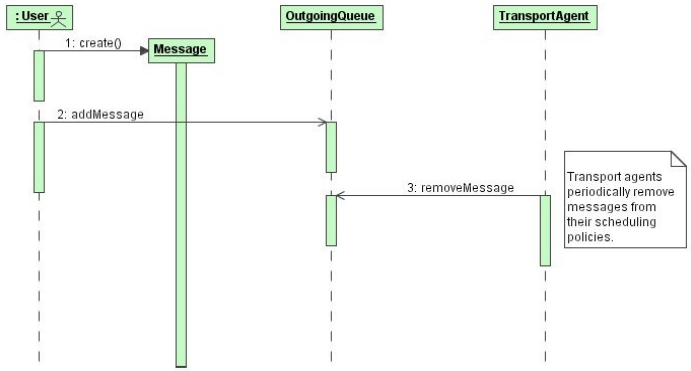
e-Macao-18-1-327

Collaboration – Behavioural

1) rendered using interaction diagram
2) specifies a set of messages that are exchanged among a set of objects to accomplish a specific purpose – a use case or operation
3) defines an interaction context

e-Macao-18-1-328

Collaboration Behavioral View



e-Macao-18-1-329

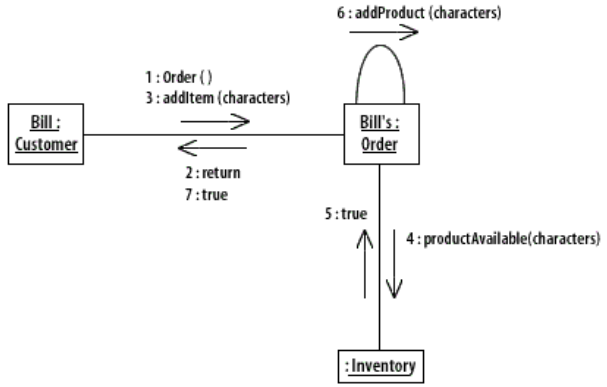
Collaboration Diagram

Definition
A collaboration diagram shows the interactions organized around the structure of a model, using either:
a) classifiers (e.g. classes) and associations, or
b) instances (e.g. objects) and links.

1) is an interaction diagram
2) is similar to the sequence diagram
3) reveals both structural and dynamic aspects of a collaboration
4) reveals the need for the associations in the class diagram

e-Macao-18-1-330

Collaboration Diagram Example



e-Macao-18-1-331

Notation 1

1) A collaboration diagram shows a graph of either instances linked to each other or classifiers and associations.

2) Navigability is shown using arrow heads on the lines representing links.

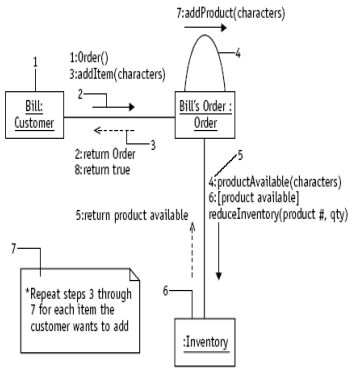
3) An arrow next to a line indicates a stimuli or message flowing in the given direction.

4) The order of interaction is given with a number.

e-Macao-18-1-332

Notation 2

1. object
2. synchronous event or procedure call
3. simple return
4. self-reference
5. sequence number
6. anonymous object
7. iteration comment



e-Macao-18-1-333

Collaboration Objects

1) the backbone of the collaborating diagram
2) may have fully qualified name e.g. Bill of class Customer
3) may be anonymous
4) same as in the sequence diagram

e-Macao-18-1-334

Collaboration Messages

1) involves sending of messages between class roles over associations
2) messages could be:
   a) synchronous events: requires a reply
   b) return: a reply message
   c) asynchronous: does not require a reply
3) same as in sequence diagram

e-Macao-18-1-335

Self Reference

1) a message from an object to itself
2) sender and receiver of the message is the same object
3) self invocation

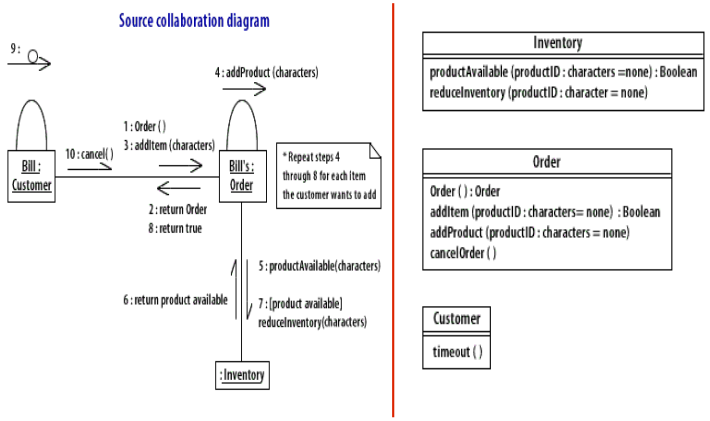
e-Macao-18-1-336

Sequence Number

1) collaboration diagram does show explicitly passage of time
2) sequence numbers to show order of execution for the messages
3) no particular standard for numbering

e-Macao-18-1-337

Collaboration and Object Diagram



e-Macao-18-1-338

Building Collaboration Diagram

1) place participating objects on the diagram
2) draw the links between the objects using the class diagram as your guide
3) add each event by placing the message arrow parallel between the two objects
4) position the arrow to point from the sender to the receiver
5) number the messages in order of execution
6) repeat steps 3 and 4 until the entire scenario has been modeled

e-Macao-18-1-339

Task 44

1) Consider the use case selected in Task 22. List an instance for each of the classes involved in the collaboration for this use case.
2) Provide a collaboration diagram for involving the objects listed in 1.

A.5.3. Component Diagrams

Architecture Modelling

Component Diagrams

e-Macao-18-1-341

Course Outline

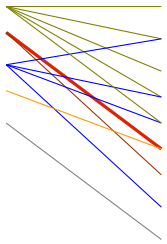
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-342

Component

Definition

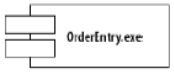
A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.

A component:

1) encapsulates the implementation of classifiers residing in it
2) does not have its own features, but serves as a mere container for its elements
3) are replaceable or substitutable parts of a system

e-Macao-18-1-343

Component Example



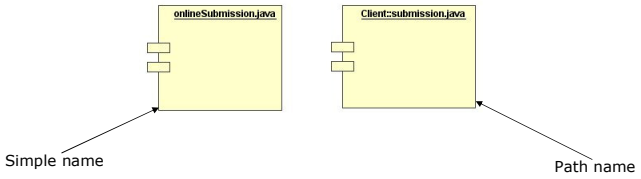
A component named OrderEntry.exe

e-Macao-18-1-344

Component Names

1) component must have a unique name

2) name is a textual string which may be written as a simple name or with a path name.



e-Macao-18-1-345

Component and Classes 1

similarities

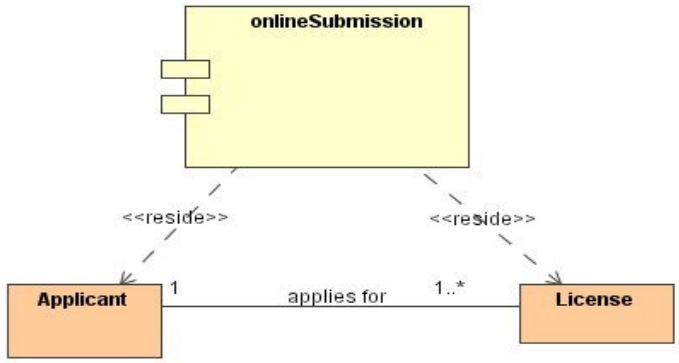
1) both may realize a set of interfaces
2) both may participate in dependencies, generalizations and associations
3) both may be nested
4) both may have instances
5) both may participate in an interaction

differences

1) classes represent logical abstraction while components represent physical things
2) components represent the physical packaging of logical components and are at a different level of abstraction
3) classes may have attributes and operations whereas components only have operations reachable only through the their interfaces

e-Macao-18-1-346

Component and Classes 2



e-Macao-18-1-347

Components and Interfaces

Definition

An interface is a collection of operations that are used to specify a service of class or components.

Interfaces:

1) represent the major seam of the system
2) are realized by components in implementation
3) promotes the deployment of systems whose services are location independent and replaceable

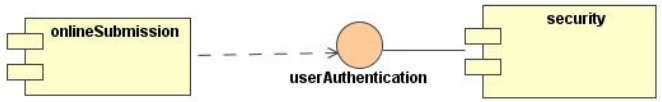
e-Macao-18-1-348

Interface Notation 1

Relationship between a component and its interface may be shown in two ways:

Style 1

a) interface in elided iconic form
b) component that realize the interface is connected to the interface using an elided realization relationship

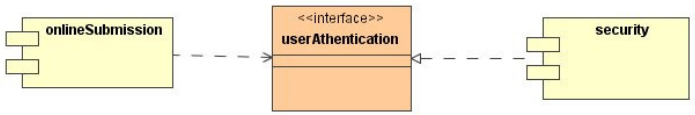


e-Macao-18-1-349

Interface Notation 2

Style 2:

a) interface is presented in an enlarged form, possibly revealing operations elided iconic form
b) realizing component is connected to it using a full realization relationship



e-Macao-18-1-350

Export Interface

Definition
An interface that a component realizes is called an export interface, meaning an interface that the component provides as a service to other components.

Components may export more than one interface.

e-Macao-18-1-351

Import Interface

Definition
An interface that a component uses is called the import interface, meaning the interface that the component relies upon to implement its own behavior.

Components may import more than one interface.

They may also import and export interfaces at the same time.

e-Macao-18-1-352

Component Replaceability

1) The key intent of any component-based system is to permit the assembly of systems from replaceable parts.

2) A system can be:
   a) created out of existing components
   b) evolved by adding new components and replacing old ones without rebuilding the system

3) Interfaces allow easy reconfiguration of component-based systems.

e-Macao-18-1-353

Task 45

1) Identify at least three major components for the restaurant license application.

2) Describe the interface for each of the components.

e-Macao-18-1-354

Types of Components

Three types: deployment, work product and execution.

1) deployment:
   component necessary and sufficient to form an executable system such as DLLs and EXEs

2) work product component:
   residue of development process consisting of things like source code files and data files from which deployment components are created

3) execution component:
   created as a consequence of an executing system

e-Macao-18-1-355

Component Stereotypes 1

1) executable: specifies that a component may be executable on a node

2) library: specifies a static or dynamic object library

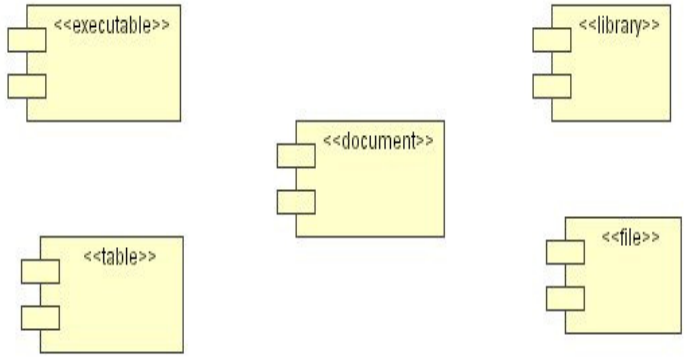
3) table: specifies a component that represents a database table

4) file: specifies a component that represents a document containing source code or data

5) document: specifies a component that represents a document

e-Macao-18-1-356

Component Stereotypes 2



e-Macao-18-1-362

Task 46

1) Add possible implementing artifacts for components listed in Task 45.

2) Using a UML tool, produce a component diagram containing the components in Task 45 and the implementing artifacts listed in question 1.

A.5.4. Packages

Architecture Modelling

Packages

e-Macao-18-1-364

Packages

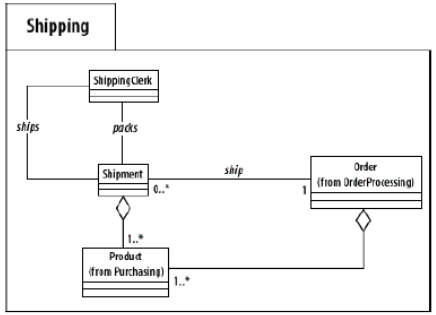
Packages:

1) are general purpose mechanism for organizing modelling elements into groups
2) group elements that are semantically close and that tend to change together

Packages should be loosely coupled, highly cohesive, with controlled access to its contents.

e-Macao-18-1-365

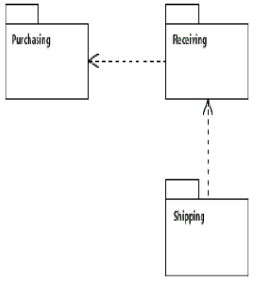
Package Example 1



e-Macao-18-1-366

Package Notation

- drawn as a tabbed folder
- packages references one another using standard dependency notation
- for instance: Purchasing package depends on Receiving package
- packages and their dependencies may be stereotyped



e-Macao-18-1-367

Owned Elements

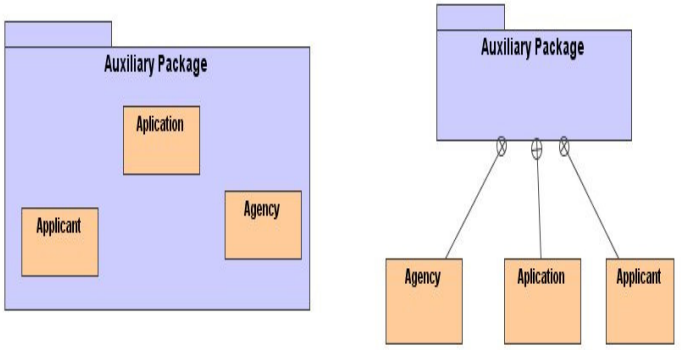
1) A package may own other elements for instance: classes, interfaces, components, nodes, collaborations, use cases, diagrams and other packages.

2) "Owning" is a composite relationship.

3) A package forms a namespace, thus elements of the same kind must be named uniquely. For example you cannot have two classes in the same package with the same name.

e-Macao-18-1-368

Owned Elements Example



e-Macao-18-1-369

Visibility

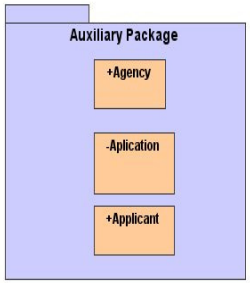
1) visibility of owned elements is similar to visibility of attributes and operations of classes
2) owned elements are visible to importing or enclosing package contents
3) protected elements can also be seen by children packages
4) private elements cannot be seen outside their owning package

e-Macao-18-1-370

Visibility Example

1) "Auxiliary Package" owns:
   a) Agency
   b) Application
   c) Applicant

2) package importing "Auxiliary Package" will have access to the "Agency" element but not "Application"



e-Macao-18-1-371

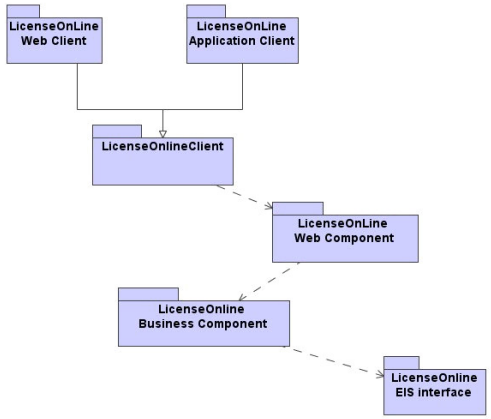
Package Stereotypes

UML provides five stereotype that apply to packages:

1) Façade: a package that is only a view on some other packages
2) Framework: a package that consists mainly of patterns
3) Stub: specifies a package that serves as a proxy for the public contents of another package
4) Subsystem: a package representing an independent part of the entire system being modeled
5) System : specifies a package representing the entire system being modeled

e-Macao-18-1-372

Package Example 1



e-Macao-18-1-373

Task 47

1) Consider an application currently running in your organization. List the major subsystems of the application

2) Using a UML tool, provide an architectural model in terms of these subsystems using packages

A.5.5. Frameworks and Patterns

Architecture Modelling

Patterns

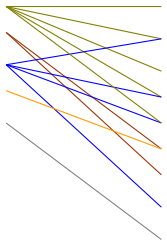
e-Macao-18-1-375

Course Outline

1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

4) Unified Process

5) UML Tools

e-Macao-18-1-376

Patterns – What are they?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way the you can use this solution a million times over, without doing it the same way twice."

– Christopher Alexander (Patterns in buildings and towns)

e-Macao-18-1-377

Patterns Definition

Definition
A generalized solution to a problem in a given context where each pattern has a description of the problem, solution context in which it applies, and heuristics including use advantages, disadvantages and trade-offs.

Patterns:

- identify, document, and classify best practices in OOD
- generalize the use and application of a society of elements

e-Macao-18-1-378

Pattern Elements

1) name: a handle which describes the design problem, its solution, and consequences in a word or two

2) problem: describes when to apply the pattern and explains the problem and its context

3) solution: elements that must make up the design, their relationships, responsibilities and collaborations.

4) consequences: associated trade-offs in using the pattern.

e-Macao-18-1-379

Pattern Example 1

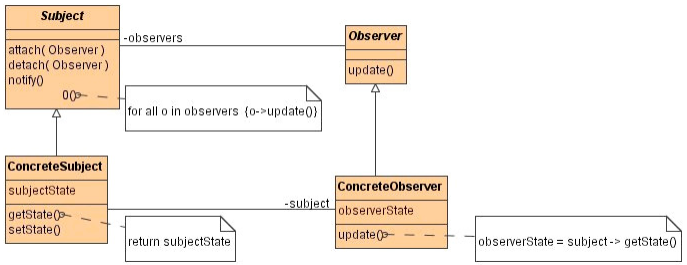
1) Name: Observer

2) Problem:
   a) when an abstraction has two aspects, one dependent on the other
   b) when a change to one object requires changing others, and you don't know how many objects need to be changed
   c) when an object should be able to notify other object without making assumptions about who these objects are

e-Macao-18-1-380

Pattern Example 2

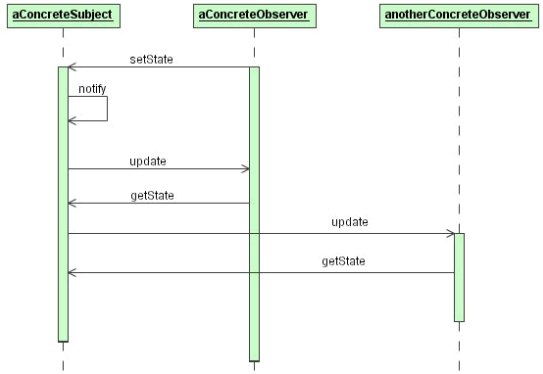
3) Solution:



e-Macao-18-1-381

Pattern Example 3

3) Solution:



e-Macao-18-1-382

Pattern Example 4

4) Consequences:

a) abstract coupling between Subject and Observer

b) support for broadcast communication

c) unexpected updates

e-Macao-18-1-383

Types of Patterns

1) Creational Patterns
   a) instantiation of objects
   b) decoupling the type of objects from the process of constructing that object

2) Structural and Architectural patterns
   a) organization of a system
   b) larger structures composed from smaller structures.

3) Behavioural Patterns
   a) assigning responsibilities among a collection of objects.

e-Macao-18-1-384

Framework 1

Definition

A Framework is a reusable software architecture that provides the generic structure and behaviour for a family of software applications, along with a context that specifies their collaboration and use.

Framework:

1) collection of patterns defined as template collaborations with supporting elements.

2) skeletal solution in which specific element must be plugged in order to establish an actual solution.

3) depicted as packages stereotyped with the "framework" keyword

e-Macao-18-1-385

Framework 2

1) made up of a set of related classes that can be specialized or instantiated to implement an application

2) lack the necessary application specific functionality and therefore not immediately useful

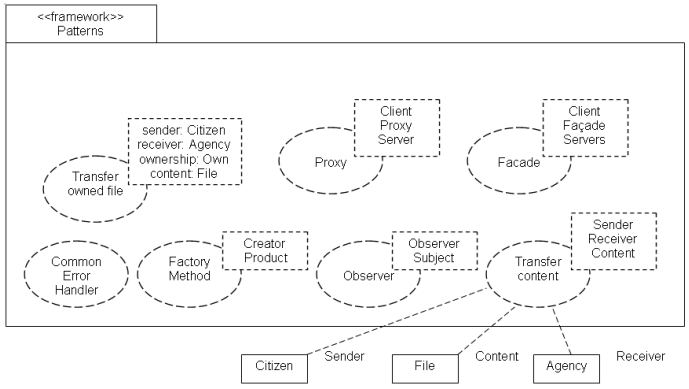
3) prefabricated structure or template of a working application in which "plug-points" or "hot spots" are not implemented or are given overidable implementations

4) patterns can be used to document frameworks

5) they are physical realization of one or more patterns

e-Macao-18-1-386

Framework Example



e-Macao-18-1-387

Architecture Patterns

1) Repository

2) Model-View-Controller (MVC)

3) Client-Server

4) Peer-to-Peer

5) Pipe-and-Filter

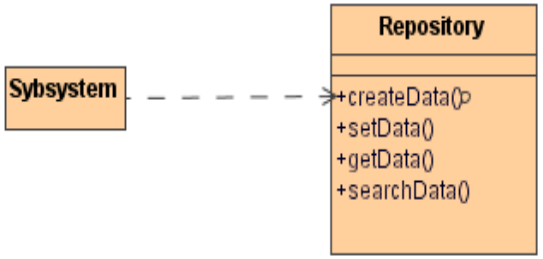
e-Macao-18-1-388

Repository Architecture 1

1) subsystems access and modify data from a single data structure called the repository
2) subsystems are relatively independent and interact through the central data structure
3) control flow can be dictated either by the central repository or by the subsystem
4) it is usually employed in database applications, compilers and software development environment

e-Macao-18-1-389

Repository Architecture 2



UML Class Diagram Describing Repository Architecture

e-Macao-18-1-390

MVC Architecture 1

Subsystems are classified into three different types:

a) model subsystems – maintains domain knowledge
b) view subsystems – displays information to users
c) controller subsystem – controls sequence of interaction

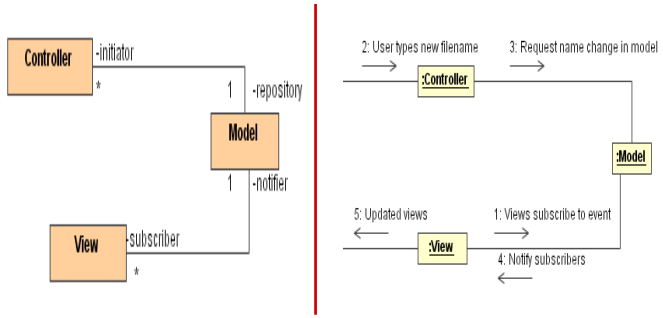
e-Macao-18-1-391

MVC Architecture 2

1) model subsystems are written independently of view or controller subsystems
2) changes in model's state are propagated to the view subsystem through the subscribe/notify protocol
3) MVC architecture is a special case of repository architecture; model is the central repository and the controller subsystem dictates control flow

e-Macao-18-1-392

MVC Architecture 3



Structural and Behavioural Description of MVC Architecture

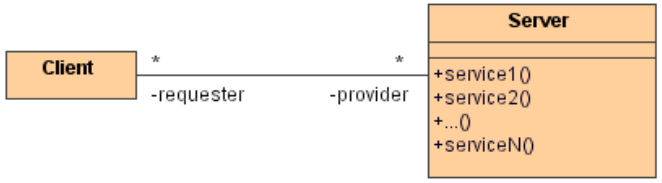
e-Macao-18-1-393

Client – Server Architecture 1

1) two kinds of subsystems – the Server and Client
2) server subsystem provides services to instances of the other subsystems called clients; which interacts with the users
3) service requests are usually through remote procedure call or some other distributed programming techniques
4) control flow in clients and servers is independent except for synchronization to manage requests and receive results
5) there may be multiple servers subsystems like in the case of the world-wide web

e-Macao-18-1-394

Client – Server Architecture 2



Client – Server Architecture - Structural Description

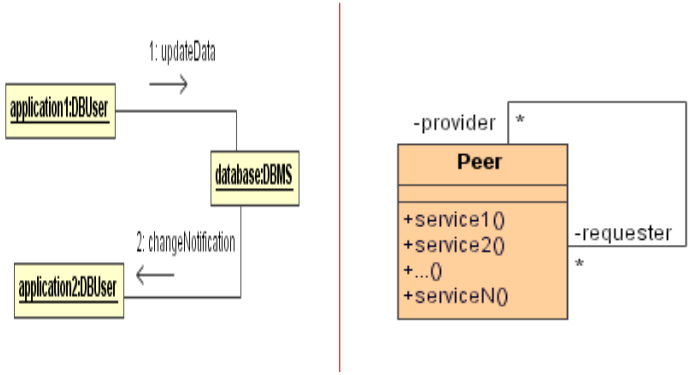
e-Macao-18-1-395

Peer-to-Peer Architecture 1

1) generalization of the client-server architecture in which subsystems can be clients or servers dynamically
2) control flow within subsystems is independent from the others except for synchronization on requests
3) more difficult to design than client server systems as there are possibilities of deadlocks

e-Macao-18-1-396

Peer-to-Peer Architecture 2



Peer-to-Peer Architecture – Behavioural and Structural Descriptions

e-Macao-18-1-397

Pipe-and-Filter Architecture 1

1) subsystems process data received from a set of input and send results to other subsystems via set of outputs
2) subsystems are called filters and the association between filters are called pipes
3) filters only know about the content and format of the data received on the input pipes and not the input filters
4) each filter is executed concurrently and synchronization is done via the pipes
5) pipes and filters can be reconfigured as required

e-Macao-18-1-398

Pipe-and-Filter Architecture 2

e-Macao-18-1-399

Task 48

1. The MVC architecture is better than the Client-Server architecture. Do you agree with this statement?
2. List two reasons for your agreement or disagreement with the statement in question 1.

e-Macao-18-1-400

Summary 1

1) Architecture is concerned with the structural organization of system components as well as how they interact to provide the system's overall behaviour or functionality.
2) A collaboration is a society of classes which provides some cooperative behavior that is more than the sum of all its parts.
3) Collaborations have both structural and behavioral aspects.
4) Collaborations may realize a use case or an operation.

e-Macao-18-1-401

Summary 2

5) A collaboration diagram shows interactions organized around the structure of a model, using either classes and associations or instances and links.
6) A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.
7) An interface is a collection of operation that are used to specify a service of class or components.

e-Macao-18-1-402

Summary 3

8) There are three categories of components: deployment, work product and execution.
9) Components may be stereotyped as executable, library, table, file or document.
10) Component diagram consists of specifying classes, implementing artifacts, components, interfaces and relationships between these model elements.

e-Macao-18-1-403

Summary 4

11) Packages are general purpose mechanisms for organizing modeling elements into groups.
12) Well structured packages must be loosely coupled and very cohesive.
13) Basic architectural patterns or styles include: repository, MVC and client-server.

A.6. Design

A.6.1. Software Design

Design Modelling

e-Macao-18-1-405

Course Outline

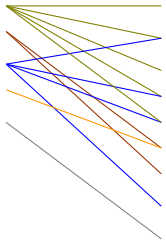
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-406

System Design

"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies"

C.A.R. Hoare

e-Macao-18-1-407

What is System Design

Definition

System Design is the transformation of analysis models of the problem space into design models (based on the solution space).

It involves selecting strategies for building the system e.g. software/hardware platform on which the system will run and the persistent data strategy.

e-Macao-18-1-408

System Design – Overview 1

Analysis produces:

a) a set of non-functional requirements and constraints
b) a use case model describing functional requirements
c) a conceptual model, describing entities
d) a sequence diagram for each use case showing the sequence of interaction among objects
e) statechart diagrams showing possible states of objects

e-Macao-18-1-409

System Design – Overview 2

Design results in:

a) a list of design goals – qualities of the system

b) software architecture describing:
   a) the subsystem responsibilities,
   b) dependencies among subsystems,
   c) subsystem mapping to hardware,
   d) major policy decision such as control flow, access control, and data storage

e-Macao-18-1-410

System Design – Activities

1) definition of design goals
2) decomposition of system into sub-system
3) selection of off-the-shelf and legacy components
4) mapping of sub-systems to hardware
5) selection of persistent data management infrastructure
6) selection of access control policy
7) selection of a global control flow mechanism
8) handling of boundary conditions

e-Macao-18-1-411

Some Design Activities 1

1) Define goals
   a) derived from non-functional requirements
2) Hardware and software mapping
   a) what computers (nodes) will be used?
   b) which node does what?
   c) how do nodes communicate?
   d) what existing software will be used and how?
3) Data management
   a) which data needs to be persistent?
   b) where should persistent data be stored?
   c) how will the data be stored?

e-Macao-18-1-412

Some Design Activities 2

4) Access control
   a) who can access which data?
   b) can access control be changed within the system?
   c) how is access control specified and realized?
5) Control flow
   a) how does the system sequence operations?
   b) is the system event-driven?
   c) does it need to handle more than one user interaction at a time?

e-Macao-18-1-413

What is Object Design? 1

1) system design describes the system in terms of its architecture:
   a) subsystem decomposition,
   b) global control flow and
   c) persistency management
2) object design includes:
   a) service specification
   b) component selection
   c) object model restructuring
   d) object model optimization

e-Macao-18-1-414

What is Object Design? 2

1) service specification – precise description of class interface
2) component selection – identification of off-the-shelf components and solution objects
3) object model restructuring – transform the object design model to improve understandability and extensibility
4) object model optimization – transform object model to address performance (response time, memory etc.)

e-Macao-18-1-415

Object Design Activities 1

1) specification
   a) missing attributes and operations
   b) type signatures and visibility
   c) constraints
   d) exceptions
2) component selection
   a) adjusting class libs
   b) adjusting application frameworks
3. restructuring
   a) realizing associations
   b) increasing reuse
   c) removing implementation dependencies
4. optimization
   a) access paths
   b) collapsing objects
   c) caching results of expensive comp
   d) delaying expensive comp

A.6.2. Design Class Diagrams

Design Modelling

Class Diagrams

e-Macao-18-1-417

Course Outline

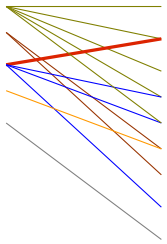
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-418

Design Classes

Class:

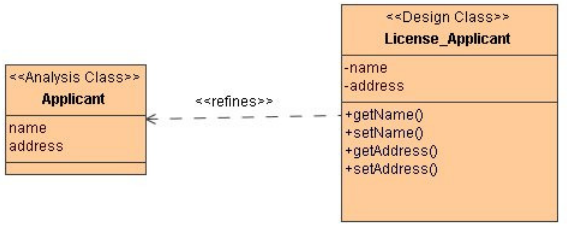
- basis for other diagrams and UML models
- describes the static view of the system

Design Classes:

- different from domain classes
- design classes express the definitions of classes as software components

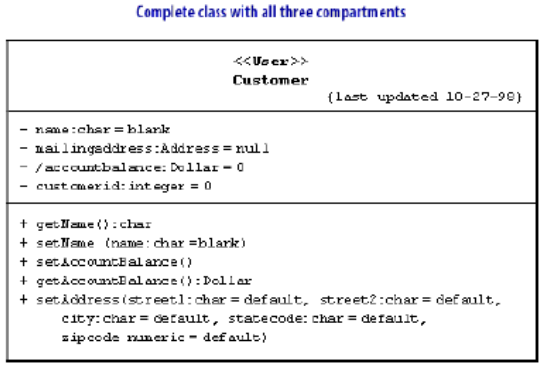
e-Macao-18-1-419

Design Class Example 1



e-Macao-18-1-420

Design Class Example 2



e-Macao-18-1-421

Design Class Diagram

1) provides the specification for software classes and interfaces in an application

2) includes:
   a) classes, associations and attributes
   b) interfaces, with their operations and constants
   c) methods
   d) attribute type information
   e) navigability
   f) dependencies

e-Macao-18-1-422

Creating Design Class Diag. 1

1) identify all classes participating in object interaction by analyzing the collaborations
2) present them in a class diagram
3) copy attributes from the associated concepts in the conceptual model
4) add methods names by analyzing the interaction diagrams
5) add type information to the attributes and methods

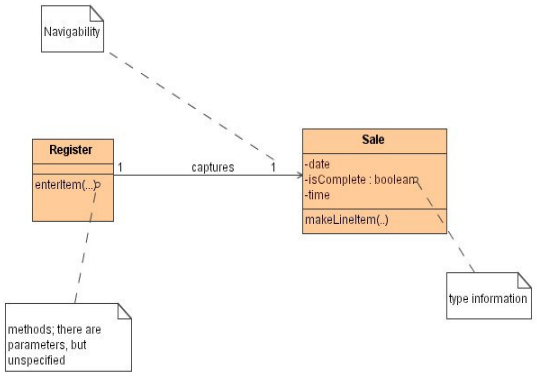
e-Macao-18-1-423

Creating Design Class Diag. 2

6) add the association necessary to support the required attribute visibility
7) add navigability arrows necessary to the associations to indicate the direction of the attribute visibility
8) add dependency relationship lines to indicate non-attribute visibility

e-Macao-18-1-424

Design Class Diagram Example



e-Macao-18-1-425

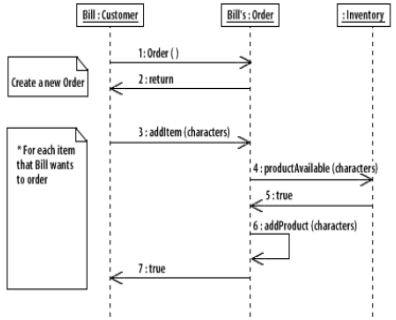
Adding Classes

1) identify those classes that participate in the software solution
2) classes can be found by scanning all the interaction diagrams and listing the participating classes

e-Macao-18-1-426

Adding Classes Example

Candidate classes:
a) Customer
b) Order
c) Inventory



e-Macao-18-1-427

Adding Attributes

1) after defining the classes, attributes must be added

2) attributes come from software requirements artifacts for example conceptual classes provide a substantial part of the attribute applicable to design classes

e-Macao-18-1-428

Attribute Syntax

```
6 7 1     2          3            4                5
| | |     |          |            |                |
+ $ / AttributeName: Data Type = Default Value {Constraints}
```

1. Derived attribute indicator: Optional
2. Attribute name: Required
3. Data type: Required
4. Assignment operator and default value: Optional
5. Constraints: Optional
6. Visibility: Required before code generation
7. Class attribute: Optional

e-Macao-18-1-429

Adding Operations

The operations of each class can be identified by analyzing the interaction diagrams:

a) messages will include calls to other objects
b) self references

e-Macao-18-1-430

Operations Syntax

```
4 5      1          6      7        2
| |      |          |      |        |
+$ OperationName(ArgName:Data Type,...):
Return Data Type {Constraints}
       |              |
       3              8
```

1. Operation name: Required
2. Any number of arguments is allowed
3. Return data type: Required for a return value, but return values themselves are optional
4. Visibility: Required before code generation
5. Class operation: Optional
6. Argument name: Required for each argument, but arguments themselves are optional
7. Argument data type: Required for each argument, but arguments themselves are optional
8. Constraints: Optional

e-Macao-18-1-431

Adding Visibility

Visibility:

1) scope of access allowed to a member of a class
2) applies to attributes and operations

UML visibility maps to OO visibilities:

1) private scope: within a class (-)
2) package scope: within a package (~)
3) public scope: within a system (+)
4) protected scope: within an inheritance tree (#)

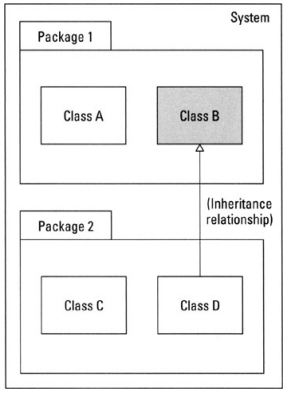
e-Macao-18-1-432

Private Visibility

1) private element is only visible inside the namespace that owns it

2) notation is "-"

3) useful in encapsulation.

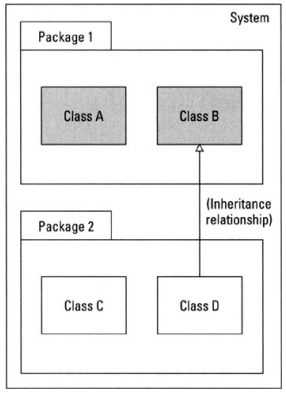


e-Macao-18-1-433

Package Visibility

1) package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace

2) notation is "~"

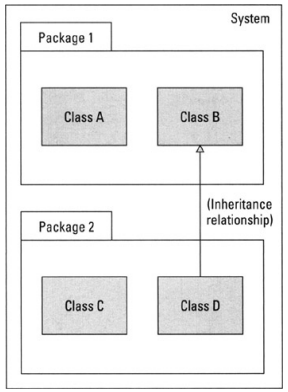


e-Macao-18-1-434

Public Visibility

1) public element is visible to all elements that can access the contents of the namespace that owns it

2) notation is "+"

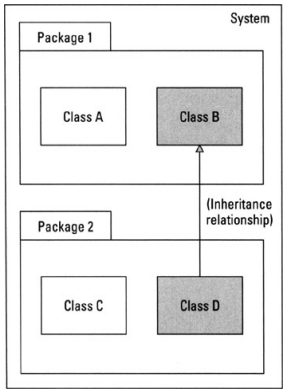


e-Macao-18-1-435

Protected Visibility

1) a protected element is visible to elements that are in the generalization relationship to the namespace that owns it

2) notation is "#"



e-Macao-18-1-436

Adding Associations

1) end of an association is called a role

2) roles may be decorated with a navigability arrow

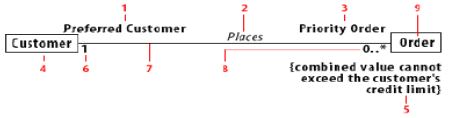
3) navigability is a property of a role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target

4) navigability indicates attributes visibility

5) implementation assumes the source class has an attribute that refers to an instance of the target class

e-Macao-18-1-437

Association Syntax



1. Role: Must have a name or roles or both
2. Association name: Must have a name or roles or both
3. Role: Must have a name or roles or both
4. Class
5. Constraint: Optional
6. Multiplicity: Required
7. Association
8. Multiplicity: Required
9. Class

e-Macao-18-1-438

Adding Dependency

1) useful to depict non-attribute visibility between classes

2) useful when there are parameters, global or locally declared visibility

e-Macao-18-1-439

Task 49

Using a UML tool, produce a design class diagram based on the conceptual classes and dynamic models already developed in previous tasks:

1) add attributes if needed
2) add operations
3) add associations with names, roles and multiplicity
4) add necessary dependencies

A.6.3. Activity Diagrams

Design Modelling

Activity Diagrams

e-Macao-18-1-441

Course Outline

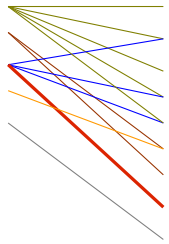
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-442

Activity Diagrams

1) variation of the state machine in which the states represent the performance of actions or sub-activities

2) transitions are triggered by the completion of the actions or sub-activities

3) one of the five diagrams in UML for modeling the dynamic aspect of a system.
   Others: sequence, collaboration, statechart, use cases

4) attached through a model to use cases, or to the implementation an operations

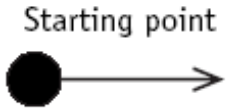
5) focuses on flows driven by internal processing (as opposed to external events)

e-Macao-18-1-443

Start State

1) signals the beginning of the activity diagram

2) indicated as a solid dot

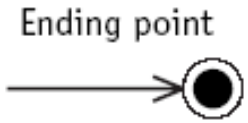


e-Macao-18-1-444

End State

1) signals the end of the activity diagram

2) indicated as a bull eye



e-Macao-18-1-445

Action State

1) shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action

2) there may be several outgoing transitions with guard conditions

3) models a step in the execution of a workflow

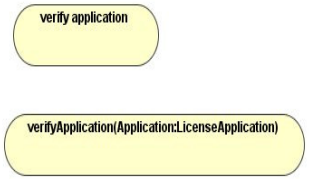
e-Macao-18-1-446

Action State – Notation

1) a shape with straight top and bottom and with convex arcs on the two sides

2) action expression is placed in the action state symbol

3) action expression may be described as natural language or pseudo code



e-Macao-18-1-447

Subactivity State

1) it invokes an activity graph

2) executes the associated activity graph

3) completes the subactivity graph is not exited until the final state of the nested graph is reached

4) a single activity graph may be invoked by many subactivity states

e-Macao-18-1-448

Subactivity State – Notation

1) shown the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram

2) subactivity name is placed in the symbol

e-Macao-18-1-449

Transitions 1

1) specify the flow of control from one action or activity state to another

2) trigger-less

3) passes control immediately on completion of the work in the source state to the next activity state

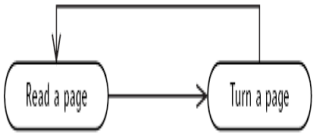
4) causes the state's exit action (if any) to be executed

e-Macao-18-1-450

Transitions 2

1) after reading a page, the page is turned

2) this is repeated continuously



e-Macao-18-1-451

Decisions

1) Decisions are expressed as guard conditions.

2) Different guards are used to indicate different possible transitions that depend on boolean conditions of the owning objects.

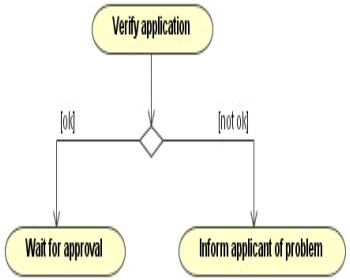
3) The predefined guard else may be defined for at most one outgoing transition which is enabled if all the guards labeling the other transitions are false.

e-Macao-18-1-452

Decision - Notation

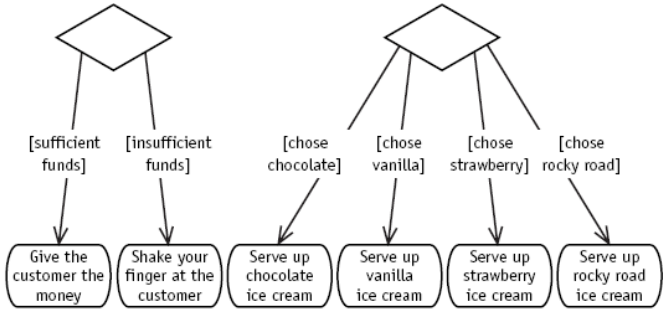
Decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each with a distinct guard condition with no event trigger.



e-Macao-18-1-453

Decision Example



e-Macao-18-1-454

Forks

1) used for modelling concurrency and synchronization in business processes

2) uses a synchronization bar

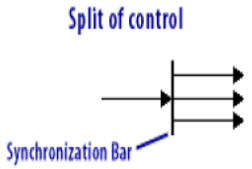
3) represents the splitting of a single flow of control into two or more concurrent flows of controls

4) indicates that activities of each of the these flows are truly concurrent when deployed across multiple nodes or sequentially interleaved if deployed on a single node

e-Macao-18-1-455

Fork Example

A single control splitting to three concurrent flows



e-Macao-18-1-456

Joins

1) represents the synchronization of two or more concurrent flows of control.

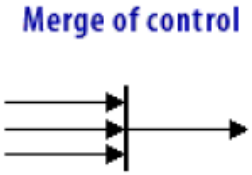
2) may have two or more incoming transitions and one outgoing transition

3) synchronizes concurrent flows, constraining each flow to waits for other incoming flows

e-Macao-18-1-457

Join Example

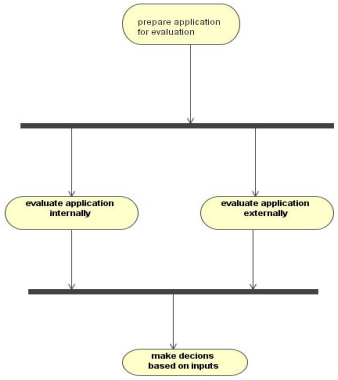
Three incoming flows synchronized and proceed as just one flow



e-Macao-18-1-458

Forks and Join Example

1) application is prepared for evaluation
2) internal and external evaluation proceeds concurrently
3) decision is made only when the internal and external evaluation are completed



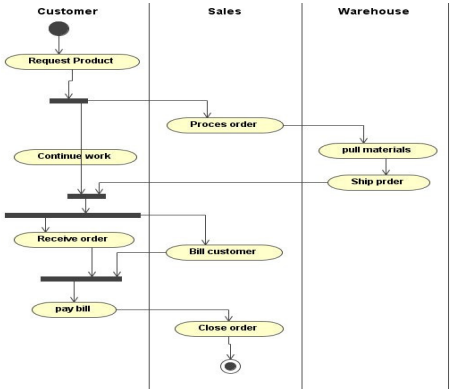
e-Macao-18-1-459

Swimlanes

1) partitions the activity states into groups
2) represent the entity within the organization responsible for those activities
3) has a unique name within a diagram

e-Macao-18-1-460

Swimlanes Example



e-Macao-18-1-461

Object Flow

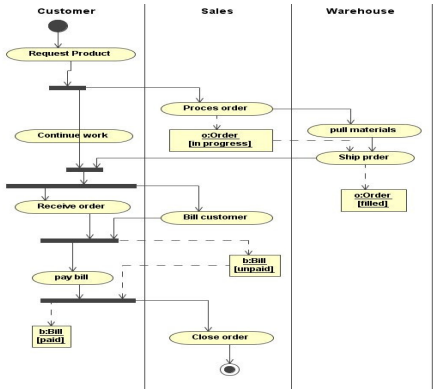
Objects maybe involved in the flow of control associated with an activity diagram.

For example in the workflow of processing an order as shown in the last example, we may wish to show how the state of the *order* and *Bill* objects change.

We can use dependency relationship to show how objects are created, modified and destroyed by transitions in the activity diagram

e-Macao-18-1-462

Object Flow Example



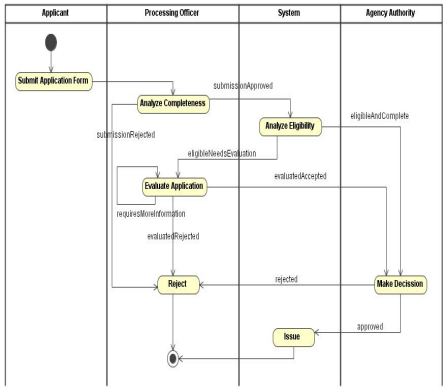
e-Macao-18-1-463

Task 50

1) Provide a narrative for the activity diagram shown.

2) Assume the procedure shown is a workflow for restaurant licensing, suggest possible changes to the model.

3) Show how the status of the application object changes along the flow.



A.6.4. Sequence Diagrams

Design Modelling

Sequence Diagrams

e-Macao-18-1-465

Course Outline

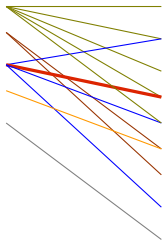
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-466

Design Sequence Diagrams

1) present the interactions between objects needed to provide a specific behavior
2) capture the sequence of events between participating objects
3) take into account temporal ordering
4) optionally shows which objects active at any time

e-Macao-18-1-467

Design Sequence Diagrams

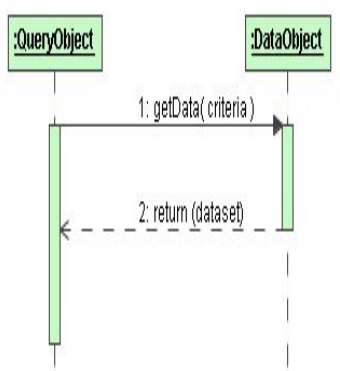
Design considerations:

a) type of message
b) timing of messages
c) recursive messages
d) object creation and destruction

e-Macao-18-1-468

Synchronous Messages

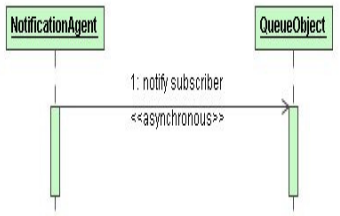
1) assumes that a return is needed
2) sender waits for the return before proceeding with any other activity
3) represented as a full arrow head
4) return messages are dashed arrows



e-Macao-18-1-469

Asynchronous Messages

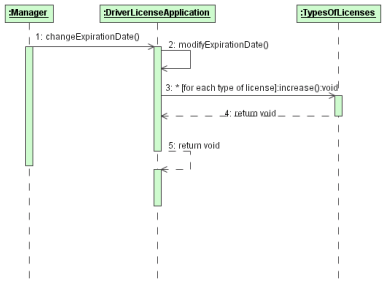
1) does not wait for a return message
2) exemplified by signals
3) sender only responsible for getting the message to the receiver
4) usually modeled using a solid line and a half arrowhead to distinguish it from the full arrowhead of the synchronous message



e-Macao-18-1-470

Self-Reference Message

A self-reference message is a message where the sender and receiver are one and the same object.



1) in a self-reference message the object refers to itself when it makes the call
2) message 2 is only the invocation of some procedure that should be executed

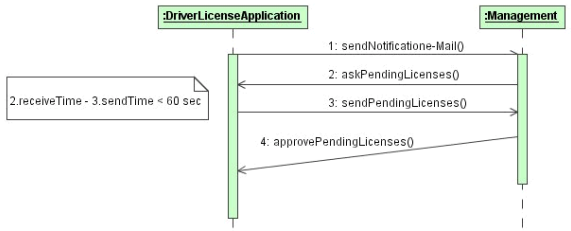
e-Macao-18-1-471

Timed Messages

1) messages may have user-defined time attributes, such as sentTime or receivedTime
2) user-defined time attributes must be associated with message numbers
3) instantaneous messages are modeled with horizontal arrows
4) messages requiring a significant amount of time, it is possible to slant the arrow from the tail down to the head

e-Macao-18-1-472

Timed Messages Example



For messages 1, 2 and 3 the time required for their execution is considered equal to zero.

Message 4 requires more time (time > 0) for its execution.

e-Macao-18-1-473

Activation and Deactivation

Sequence diagrams can show object activation and deactivation.

Activation means that an object is occupied performing a task.

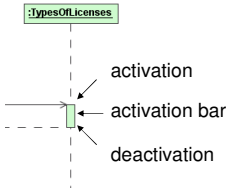
Deactivation means that the object is idle, waiting for a message.

e-Macao-18-1-474

Activation and Focus of Control

Activation is shown by widening the vertical object lifeline to a narrow rectangle, called an activation bar or focus of control.

An object becomes active at the top of the rectangle and is deactivated when control reaches the bottom of the rectangle.



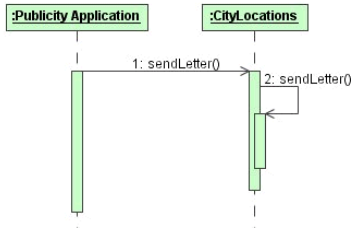
e-Macao-18-1-475

Recursion

An object might also need to call a message recursively, this means to call the same message from within the message.

1) suppose that cityLocations is defined in the class diagram as a set of one or more apartments or houses

2) A letter could be sent to all apartments in a location as shown



e-Macao-18-1-476

Creation and Destruction

Object Creation:

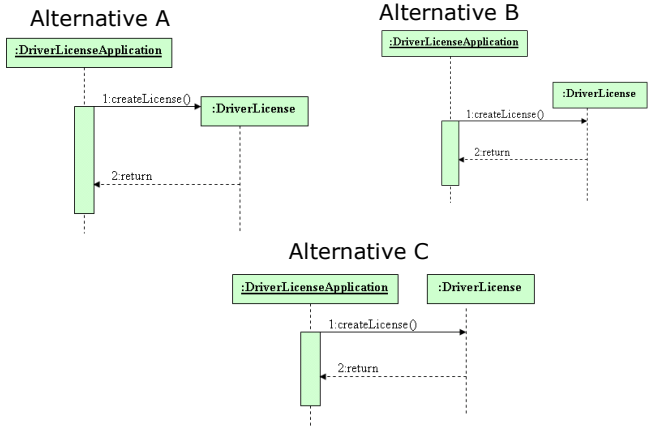
- if the object is created during the sequence execution it should appear somewhere below the top of the diagram.

Object Destruction:

- if the object is deleted during the sequence execution, place an X at the point in the object lifeline when the termination occurs.

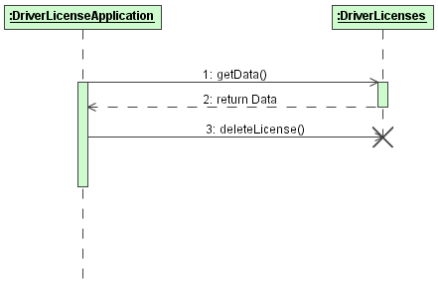
e-Macao-18-1-477

Object Creation Example



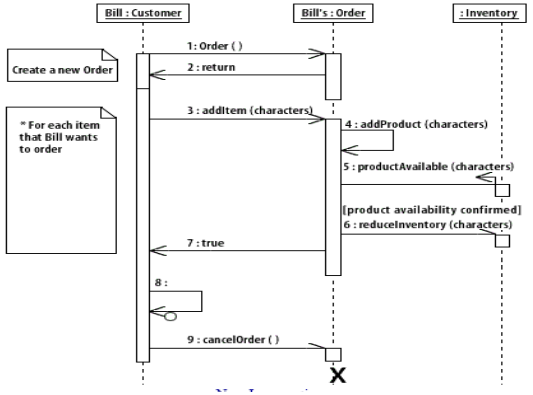
e-Macao-18-1-478

Object Destruction Example



e-Macao-18-1-479

Sequence Diagram Example



e-Macao-18-1-480

Task 51

Consider the "submit new application" use case for restaurant license application.

Based on your design classes, provide a sequence diagram to show the interaction required for this use case.

A.6.5. Statechart Diagrams

Design Modelling

State Diagrams

e-Macao-18-1-482

Course Outline

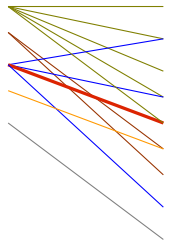
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-483

Design Statechart Diagrams

Specifies detailed state information:

1) composite states

2) dynamic and static branching

e-Macao-18-1-484

Design Statechart Diagrams

1) statechart diagrams illustrate how these objects behave internally

2) statechart diagrams relate events to state transitions and states

3) transitions change the state of the system and are triggered by events

4) design statechart diagrams provide detailed information on internal behaviors of objects

e-Macao-18-1-485

Static Branch Point

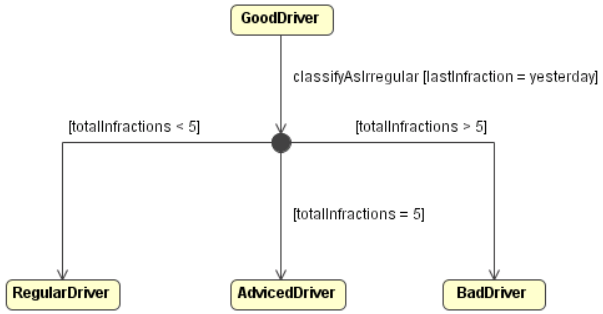
Another pseudo state provided by UML is the static branch point that allows a transition to split into two or more paths.

Notation:

It provides a means to simplify compound guard conditions by combining the like portions into a single transition segment, then branching based on the portions of the guard that are unique.

e-Macao-18-1-486

Static Branch Point Example

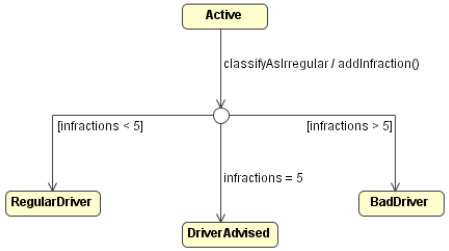


e-Macao-18-1-487

Dynamic Branch Point

In other transitions, the destination is not known until the associated action has been completed.

Notation: ○



e-Macao-18-1-488

Modelling Composite States

A composite state is simply a state that contains one or more statecharts diagrams.

Composite states may contain either:

1) a set of mutually exclusive states: literally like embedding a statechart diagram inside a state
2) a set of concurrent states: different states divided into regions, active at the same time

A composite state is also called a super-state, a generalized state that contains a set of specialized states called sub-states.

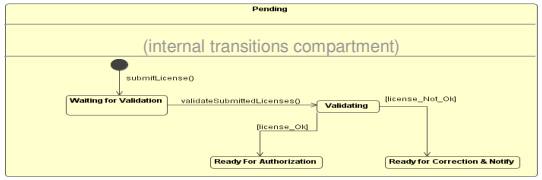
e-Macao-18-1-489

Sub-States

A composite state (super-state) may be decomposed into two or more lower-level states (sub-states).

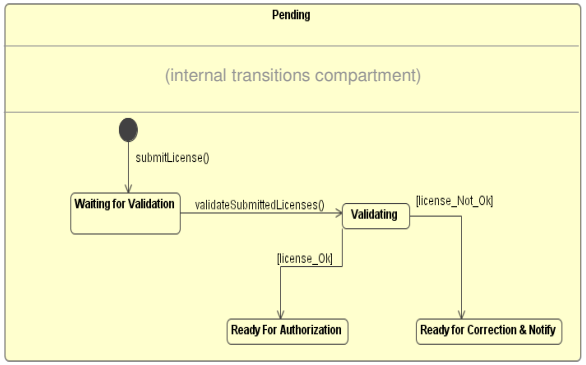
All the rules and notation are the same for the contained substates as for any statechart diagram.

Decomposition may have as many levels as needed.



e-Macao-18-1-490

Sub-States Example

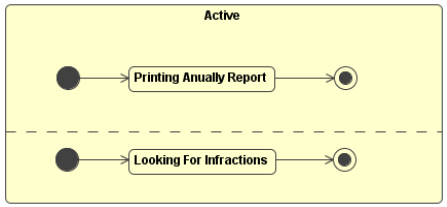


e-Macao-18-1-491

Concurrent Sub-States

Modelling concurrent substates implies that you have many things ocurring at the same time.

To isolate them, the composite state is divided into regions, and each region contains a distinct statechart diagram.



e-Macao-18-1-492

Sub-Machine States

A sub-machine state is a shorthand for referring to an existing statechart diagram.

Within a composite state, it is possible to reference to a sub-machine state in the same way that a class may call a subroutine or a function of another class.

The composite state containing the sub-machine is called the containing state machine, and the sub-machine is called the referenced state machine.

Access to sub-machines states is through entry and exit points that are specified by stub states.

e-Macao-18-1-493

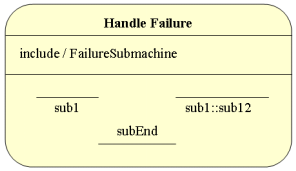
Notation for Sub-Machine

The containing state icon models the reference to the sub-machine adding the keyword include followed by a slash plus the name of the submachine state.

The stub state uses a line with the state name placed near the line.

Entry points are represented with a line and the name under the line.

Exit points are represented with a line and the name over the line.

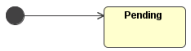


e-Macao-18-1-494

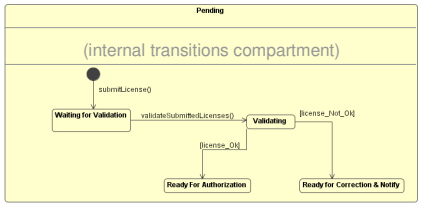
Transitions to Sub-States 1

Alternative A:

- draw a transition pointing to the edge of the composite state icon
- it means that the composite state starts in the default initial state
- the default initial state is the sub-state associated with the initial state icon in the contained statechart diagram



The initial sub-state is Waiting for Validation



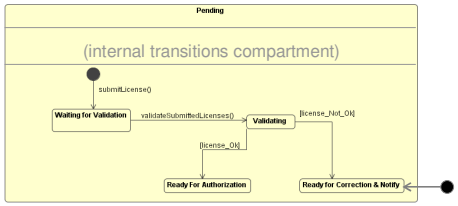
e-Macao-18-1-495

Transitions to Sub-States 2

Alternative B:

- draw a transition through the edge of the super-state to the edge of the specific sub-state
- it means that the composite state starts in that specific sub-state

The initial sub-state is Ready for Correction & Notify



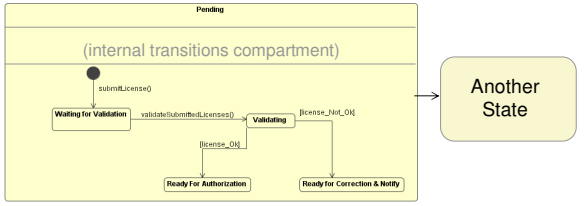
e-Macao-18-1-496

Transitions from Sub-States 1

Alternative A:

1) an event can cause the object to leave the super-state regardless of the current sub-state
2) draw the transition from the edge of the super-state to the new state.

At any sub-state the object can do a transition to AnotherState

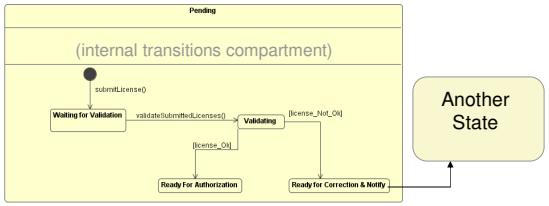


e-Macao-18-1-497

Transitions from Sub-States 2

Alternative B:

1) an event can cause the object to leave the super-state directly from a specific sub-state
2) implies that the exit event may only happen when the object is in the specific sub-state
3) draw the transition from the edge of the specific sub-state through the edge of the super-state.

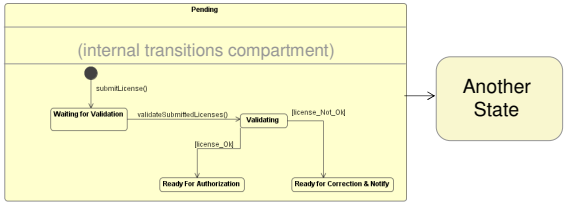


e-Macao-18-1-498

Transitions from Sub-States 3

Alternative C:

1) an object may leave a super-state because all the activities in the state and its sub-state has been completed
2) it is called an automatic transition
3) draw the transition from the edge of the super-state to the new state



e-Macao-18-1-499

History Indicator 1

The history indicator is used to represent the ability for doing backtracking to a previous composite state.

Represents a pseudo-state and is a shorthand notation to solve a complex modelling problem.

May refer to:

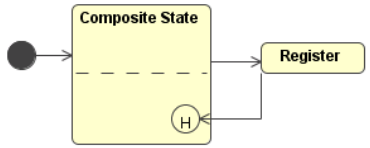
1) shallow history: the object should return to the last sub-state on the top most layer of sub-states
2) deep history: the object needs to return to the exact sub-state from it which left, no matter how many layers down that is

e-Macao-18-1-500

History Indicator 2

Notation:

1) shallow history (H)
2) deep history (H*)



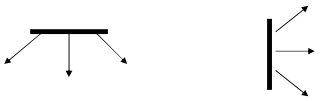
e-Macao-18-1-501

Split of Control

Split of control means that based on a single transition it is necessary to proceed with several tasks concurrently.

Notation:

1) a single transition divided into multiple arrows, each pointing to a different sub-state
2) the division is accomplished by the synchronization bar



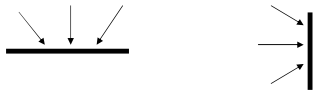
e-Macao-18-1-502

Merge of Control

Merge of control means that based on the completion of a number of transitions it is necessary to proceed with a single task.

Notation:

– multiple transitions converge to a synchronization bar and only one transition outputs from the bar

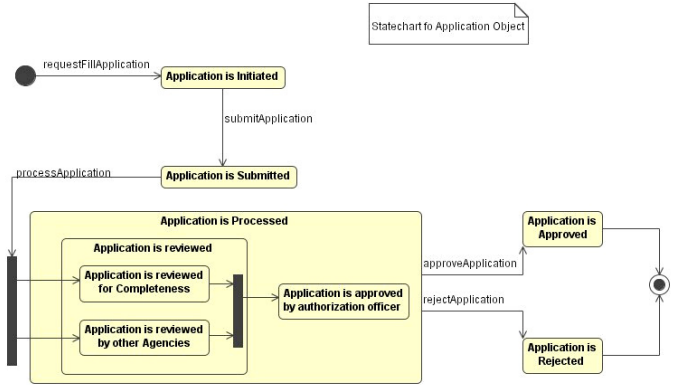


e-Macao-18-1-503

Split/Merge Control Example

e-Macao-18-1-504

Statechart Diagram Example



e-Macao-18-1-505

Task 52

Consider the restaurant license application, provide a detailed statechart diagram for the "Application" object.

A.6.6. Design Patterns

Design Modelling

Patterns

e-Macao-18-1-507

Course Outline

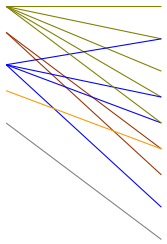
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-508

Design Patterns 1

Definition

Design patterns are partial solutions to common problems. They name, abstract, and identify the key aspects of common design structure that make it useful for creating reusable object-oriented design.

Some common problems:

a) separating interfaces from a number of alternate implementations
b) wrapping around a set of legacy classes
c) protecting a caller from changes associated with specific platforms

e-Macao-18-1-509

Design Patterns 2

1) They are related to coding idioms that exist for programming languages.

2) Design patterns capture expert knowledge and design tradeoffs and support the sharing of architectural knowledge among developers.

3) Design patterns as a shared vocabulary can clearly document the software architecture of a system.

4) Allow design engineers relate to one another a a higher level of abstraction.

e-Macao-18-1-510

Design Patterns: Description 1

Attribute	Description
Pattern Name and Classification	Conveys the essence of the pattern succinctly.
Intent	A short statement that answers the following questions: what does the design do? Its rationale and intent.
Also Known As	Other well known names for the patterns, if any
Motivation	A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
Applicability	What are the situations in which the design pattern can be applied?
Structure	Class diagram and sequence diagram for the classes involved in the pattern.
Participants	The classes and/or the object participating in the design pattern and their responsibilities.
Collaborations	How participants collaborate to carry out their responsibilities.

e-Macao-18-1-511

Design Patterns: Description 2

Attribute	Description
Consequences	How does pattern support its objectives? What are the tradeoffs?
Implementation	What pitfalls, hints, or techniques you should be aware of when implementing the pattern
Sample Code	Code fragment to show implementation
Known uses	Example of patterns found in real systems
Related Patterns	What design patterns are closely related to this one?

e-Macao-18-1-512

Design Patterns Catalog

1.	Abstract Factory (C)	13.	Interpreter (B)*
2.	Adapter (S), Adapter (S)*	14.	Iterator (B)
3.	Bridge (S)	15.	Mediator (B)
4.	Builder (C)	16.	Memento (B)
5.	Chains of Responsibility (B)	17.	Observer
6.	Command (B)	18.	Prototype (C)
7.	Composite (S)	19.	Proxy (S)
8.	Decorator (S)	20.	Singleton (C)
9.	Façade (S)	21.	State (B)
10.	Factory Method (C)*	22.	Strategy (B)
11.	Flyweight (S)	23.	Template Method (B)*
12.	Visitor (B)		

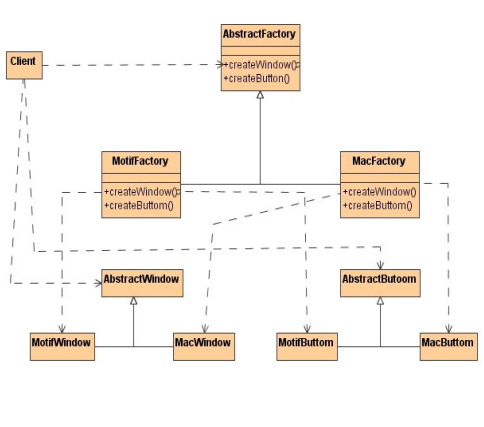
*C – creational pattern, S – structural pattern, B – Behavioural pattern, * - Class Scope*

e-Macao-18-1-513

Applying Abstract Factory

Problem:
encapsulating platforms

Solution:
Abstract Factory - shields an application from the concrete classes provided by a specific platform

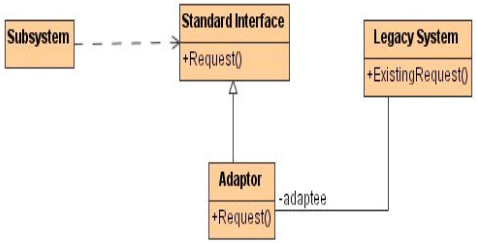


e-Macao-18-1-514

Applying Adapter

Problem:
wrapping around legacy code

Solution:
Adapter - encapsulate a piece of legacy code not designed to work with the system

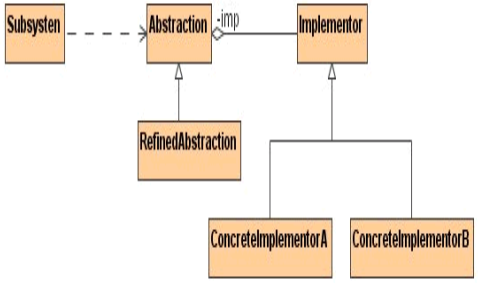


e-Macao-18-1-515

Applying Bridge

Problem:
allowing for alternate implementation

Solution:
Bridge - decouples the interface from the its implementation

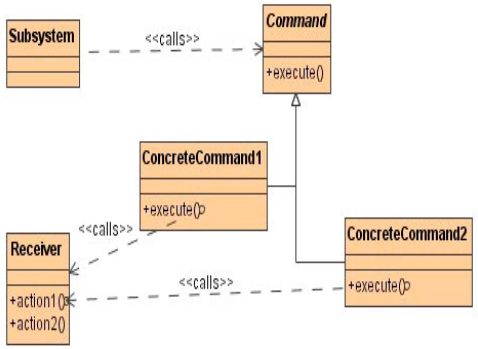


e-Macao-18-1-516

Applying Command

Problem:
encapsulating control

Solution:
Command - encapsulates a control such that user requests can be treated uniformly

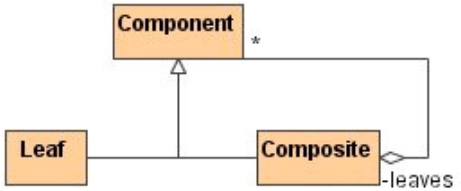


e-Macao-18-1-517

Applying Composite

Problem:
representing recursive hierarchies

Solution:
Composite - represent recursive hierarchies

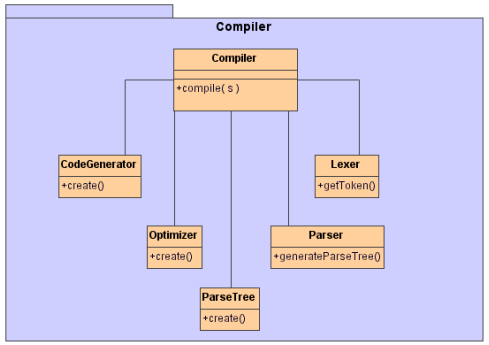


e-Macao-18-1-518

Applying Façade

Problem:
encapsulating subsystems

Solution:
Façade - reduces dependencies among classes by encapsulating subsystems from with simple unified interfaces

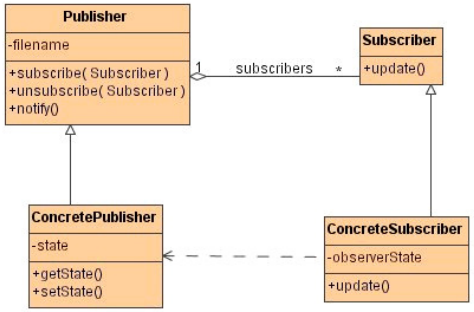


e-Macao-18-1-519

Applying Observer

Problem:
decoupling entities from view

Solution:
Observer – allows us to maintain consistency across the state of one publisher and many subscribers

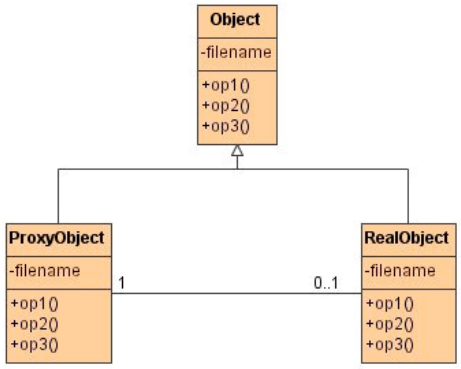


e-Macao-18-1-520

Applying Proxy

Problem:
encapsulating expensive objects

Solution:
Proxy – improves the performance or the security of a system by delaying expensive computations, … until when needed

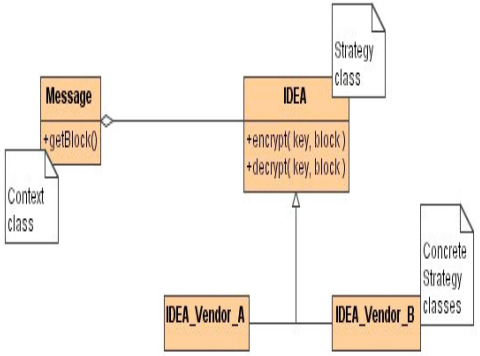


e-Macao-18-1-521

Applying Strategy

Problem:
encapsulating algorithms

Solution:
Strategy - decouples an algorithm from its implementations



e-Macao-18-1-522

Task 53

For each of the following statements, specify the most convenient pattern to apply to the following scenarios:

1) a component on the web tier requires access to business components
2) there a need to provide several buttons on a web form which executes different actions
3) messages need to be sent to citizens each time a typhoon approaches

e-Macao-18-1-523

Summary 1

1) design involves the transformation of analysis models in the problem space into design models in the solution space
2) object design includes:
   a) service specification for classes
   b) component selection
   c) object model restructuring
   d) object model optimization

e-Macao-18-1-524

Summary 2

3) design class diagrams provide specification for software classes and interfaces in an application

4) typical information contained in a Design Class Diagram include:
   a) classes
   b) associations
   c) attributes
   d) interfaces
   e) methods
   f) attribute type information
   g) navigability
   h) dependencies

e-Macao-18-1-525

Summary 3

5) activity diagrams models the dynamic aspect of a system by showing the flow from one activity to another

6) activity diagram can be used to describe a workflow or the details of an operation

7) an activity diagram may also show the flow of an object as it moves from one state to another at different points in the flow of control

e-Macao-18-1-526

Summary 4

9) design sequence diagrams specifies message types, temporal constraints, object creation and destruction

10) design statechart diagrams describes detailed the internal behavior of objects

11) design patterns are partial solutions to common problems. They name, abstract and identify the key aspects of common design structure that make them useful for creating reusable object oriented design

12) there are 23 basic patterns as provided by the GoF

A.7. Implementation

Implementation Model

e-Macao-18-1-528

Course Outline

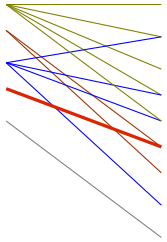
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-529

Implementation Phase

At this stage it is necessary to implement the design specified in the design models.

It also deals with non-functional requirements and the deployment of the executable modules onto nodes.

Two models are developed during this phase:

1) the implementation model describing how the design elements have been implemented in terms of software system
2) the deployment model describing how the implemented software should be deployed on the physical hardware

e-Macao-18-1-530

Implementation Model

The implementation model indicates how various aspects of the design map onto the target language.

It describes how components, interfaces, packages and files are related.

The modelling elements are:

1) packages
2) components
3) their relationships

and they are shown in the implementation component diagram.

e-Macao-18-1-531

Modelling Techniques

Component diagrams are used to model the static implementation view of a system.

To model this view, it is possible to use component diagrams in one of four ways:

1) to model source code
2) to model executable releases
3) to model physical databases
4) to model adaptable systems

e-Macao-18-1-532

Modelling Source Code

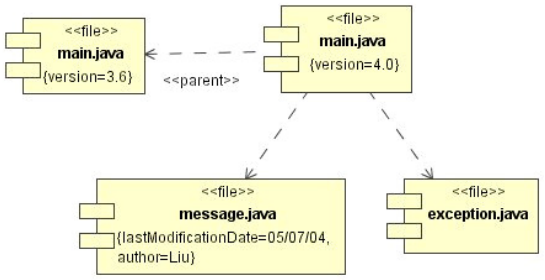
1) Component diagrams are used to model the configuration management of source files, which represent work-product components.

2) Modelling procedure:

   a) identify the set of source code files of interest and model them as components stereotyped as files
   b) for larger systems, use packages to show groups of source code files
   c) consider exposing a tagged value to show interest information such as author, version number, etc.
   d) model the compilation dependencies among these files using dependencies

e-Macao-18-1-533

Source Code Files Example



e-Macao-18-1-534

Modelling Executable Releases

1) Component diagrams are used model executable releases, including the deployment components that form each release, and the relationships among those components.

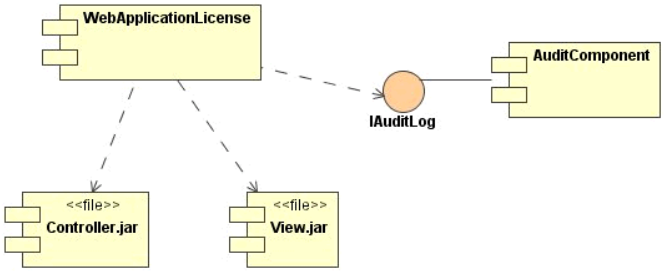
2) Each component diagram focuses on one set of components at a time, such as all components that live on one node.

3) Modelling procedure:

   a) identify the set of components to model
   b) consider the stereotype of each component in the set
   c) for each component, consider its relationship to its neighbors.

e-Macao-18-1-535

Executables Example



e-Macao-18-1-536

Modelling Physical Databases

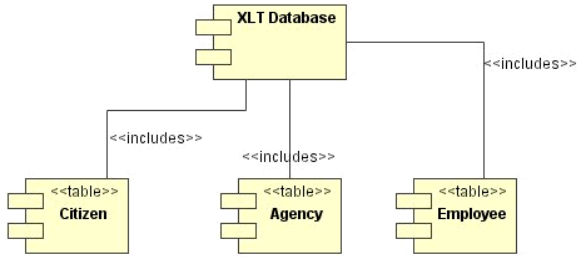
1) Component diagrams model the mapping of classes into tables of a database.

2) Modelling procedure:

   a) identify the classes in your model that represent your logical database schema
   b) select a strategy for mapping these classes to tables, possibly considering the physical distribution
   c) create a component diagram containing components stereotyped as tables to model the mapping

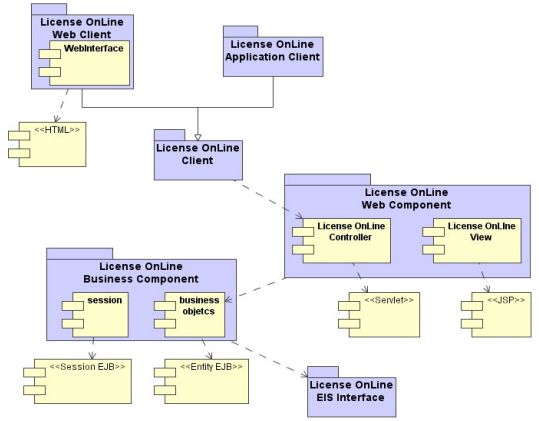
e-Macao-18-1-537

Physical Databases Example



e-Macao-18-1-538

Example



e-Macao-18-1-539

Summary

1) The implementation component diagram describes how components, interfaces, packages and files are related.

2) Implementation component diagrams may be used to model source code, executable releases, physical databases and adaptable systems.

e-Macao-18-1-540

Task 54

Provide an possible implementation model for the restaurant license application based on any implementation framework of your choice (.NET, J2EE, etc.). You model should show the major components of the system.

A.8. Deployment

e-Macao-18-1-541

Deployment Model

e-Macao-18-1-542

Course Outline

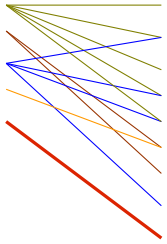
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-543

Deployment Diagrams 1

1) show the configuration of run-time processing nodes and the components that live on them.
2) model the distribution, delivery, and installation of the parts that make up the physical system
3) involve modelling the topology of the hardware on which the system executes

e-Macao-18-1-544

Deployment Diagrams 2

4) essentially focus on a system's nodes, and include:
   a) nodes
   b) dependencies and associations relationships
   c) components
   d) packages

e-Macao-18-1-545

Nodes

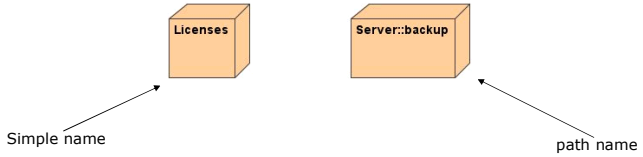
1) model the topology of the hardware on which the system executes
2) represent the hardware on which components are deployed and executed
3) may be stereotyped to allows for specific kinds of processors and devices.

e-Macao-18-1-546

Nodes Names

Every node must have a name that distinguishes it from other nodes.

A name is a textual string which may be written as a simple name or as a path name.



e-Macao-18-1-547

Nodes and Components

Components

1) participate in the execution of a system.

2) represent the physical packaging of otherwise logical elements

Nodes

1) execute components

2) represent the physical deployment of components

The relationship deploys between a node and a component can be shown using a dependency relationship.

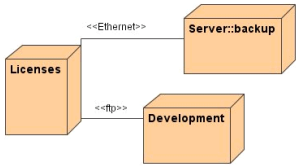
e-Macao-18-1-548

Organizing Nodes

Nodes can be organized:

1) in the same manner as classes and components
2) by specifying dependency, generalization, association, aggregation, and realization relationships among them.

The most common kind of relationship used among nodes is an association representing a physical connection among then.



e-Macao-18-1-549

Processors and Devices

A processor is a node that has processing capability. It can execute a component.

A device is a node that has no processing capability (at least at the level of abstraction showed).

e-Macao-18-1-550

Modelling Nodes

Procedure:

1) identify the computational elements of the system's deployment view and model each as a node
2) add the corresponding stereotype to the nodes
3) consider attributes and operations that might apply to each node.

e-Macao-18-1-551

Distribution of Components

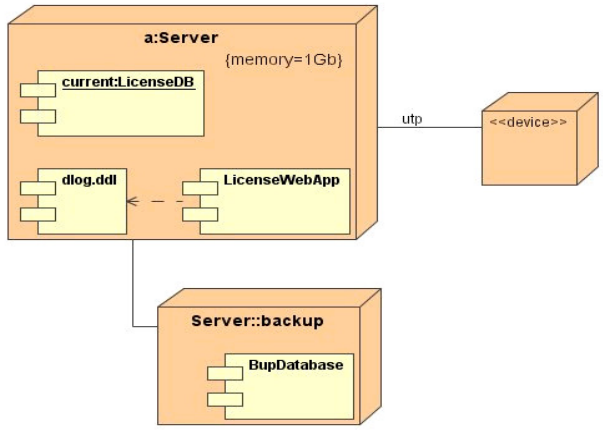
To model the topology of a system it is necessary to specify the physical distribution of its components across the processors and devices of the system.

Procedure:

1) allocate each component in a given node
2) consider duplicate locations for components, if it is necessary
3) render the allocation in one of these ways:
   a) don't make visible the allocation
   b) use dependency relationship between the node and the component it's deploy
   c) list the components deployed on a node in an additional compartment

e-Macao-18-1-552

Example



e-Macao-18-1-553

Summary

Deployment diagrams model the topology of the hardware on which the system executes and the software installed on each of the hardware components.

Deployment diagrams include nodes, packages, components and their relationships.

Deployment diagrams may be used to model different kind of systems such as embedded, client-server or distributed systems.

e-Macao-18-1-554

Task 55:

Consider task 54, show how these components will be deployed at the agency. In your model, annotate the nodes according to described them.

A.9. Unified Process

e-Macao-18-1-555

Unified Process

e-Macao-18-1-556

Course Outline

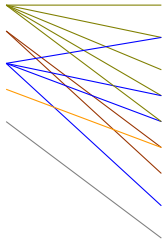
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-557

Overview

1) Software Development Process

2) Unified Process Overview

3) Unified Process Structure
   1) Building Blocks
   2) Phases
   3) Workflows

e-Macao-18-1-558

Why a process ?

A process defines:
1) who is doing what
2) when to do it
3) how to reach a certain goal
4) the inputs and outputs for each activity

e-Macao-18-1-559

Development Process

Is a framework which guides the tasks, people and define output products of the development process.

It is a framework because:
1) provides the inputs and outputs of each activity
2) does not restrict how each activity must be performed
3) must be tailored for every project

There is no universal process.

e-Macao-18-1-560

Unified Process - Overview

Key elements:

1) iterative and incremental

2) use case-driven

3) architecture-centric

e-Macao-18-1-561

Iterative and Incremental 1

The design process is based on iterations that address different aspects of the design process.

The iterations evolve into the final system (incremental aspect).

The process does not try to complete the whole design task in one go.

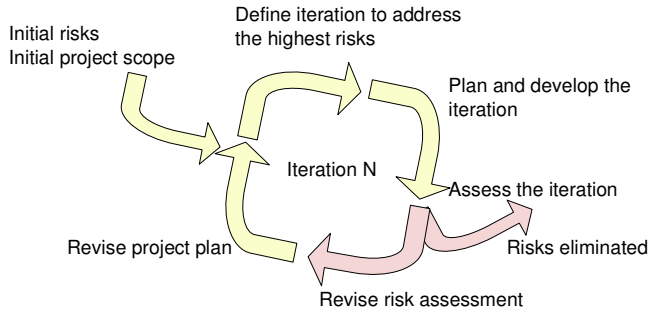
How to do it?

1) plan a little
2) specify, design and implement a little
3) integrate, test and run
4) obtain feedback before next iteration

e-Macao-18-1-562

Iterative and Incremental 2

Technical risks are assessed and prioritized early and are revised during each iteration.



e-Macao-18-1-563

Use Case-Driven

Use cases are used for:

1) identify users and their requirements
2) aid in the creation and validation of the architecture
3) help produce definitions of test cases and procedures
4) direct the planning of iterations
5) drive the creation of user documentation
6) direct the deployment of the system
7) synchronize the content of different models
8) drive traceability throughout models

e-Macao-18-1-564

Architecture-Centric

Problem:

- with the iterative and incremental approach different development activities are done concurrently

Solution:

- the system's architecture ensures that all parts fit together

"An architecture is the skeleton on which the muscles (functionality) and skin (user-interface) of the system will be hung".

e-Macao-18-1-565

Unified Process - Structure

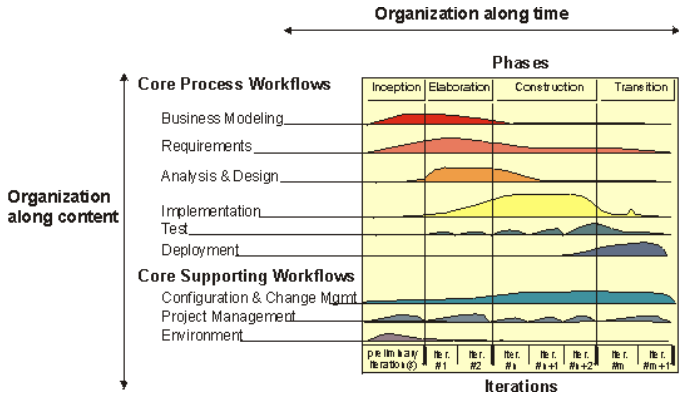
Two dimensions:

1) time → division on the life cycle into phases and iterations

2) process components → production of a specific set of artifacts with well-defined activities called workflows

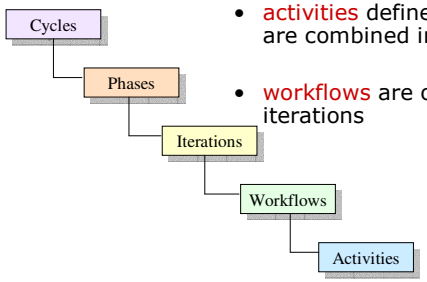
e-Macao-18-1-566

The Development Process



e-Macao-18-1-567

Building Blocks

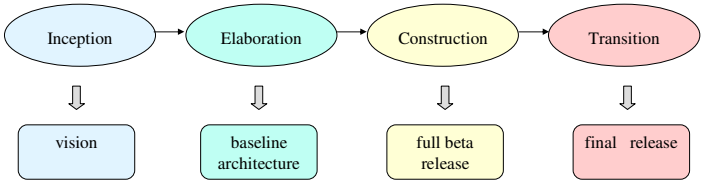


- activities define detailed work and are combined in workflows
- workflows are organized into iterations
- each iteration identifies some aspect of the system and are organized into phases
- phases can be grouped into cycles
- cycles focus on the generation of successive releases

e-Macao-18-1-568

Life Cycle Phases

Phases and major deliverables of the Unified Process



e-Macao-18-1-569

Inception

Focuses on the generation of the business case that involves:

1) identification of core uses cases
2) definition of the actual scope
3) identification of risky difficult parts of the system

The main objectives are:

1) to prove the feasibility of the system to be built
2) to determine the complexity involved in order to provide reasonable estimates

Outputs:

1) the vision of the system
2) very simplified use case model
3) tentative architecture
4) risks identified
5) plan for the elaboration phase

e-Macao-18-1-570

Elaboration

Involves:

1) understanding how requirements are translated into the internals of the system
2) producing the baseline architecture
3) capturing the majority of the use cases
4) exploring further the risks identified earlier and identifying the most significant
5) specifying any non-functional requirements specially those related to reliability and performance

Outputs:

1) the system's architecture
2) detailed use case model
3) set of plans for the construction phase

e-Macao-18-1-571

Construction

Involves:

1) completing the analysis of the system
2) performing the majority of the design and the implementation

Outputs:

1) implemented software product as a full beta release. It may contain some defects
2) associated models

An important aspect for the success of this phase is to monitor the critical aspects of the projects, specially significant risks.

e-Macao-18-1-572

Transition

Involves:

1) deployment of the beta system
2) monitoring user feedback and handling any modifications or updates required

Output:

1) the formal release of the software

e-Macao-18-1-573

UP - Workflows 1

1) requirements: focuses on the activities which allow to identify functional and non-functional requirements

2) analysis: restructures the requirements identified in terms of the software to be built

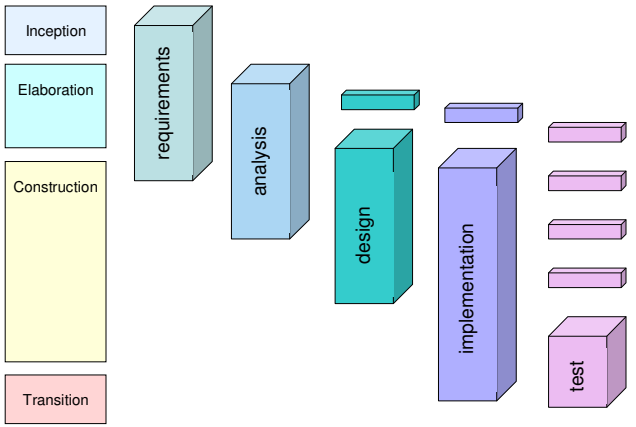
3) design: produces a detailed design

4) implementation: represents the coding of the design in a programming language, and the compilation, packaging, deployment and documentation of the software

5) test: describes the activities to be carried out for testing

e-Macao-18-1-574

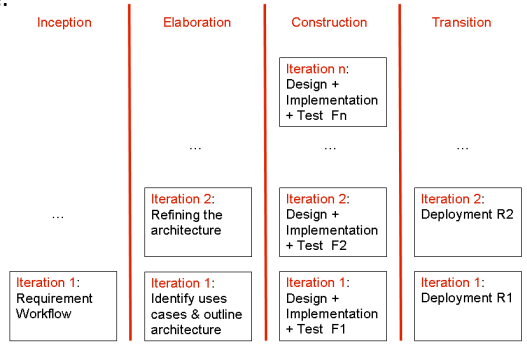
Workflows and Phases



e-Macao-18-1-575

Phases and Iterations

The iterations of workflows occur one or more times during a phase.



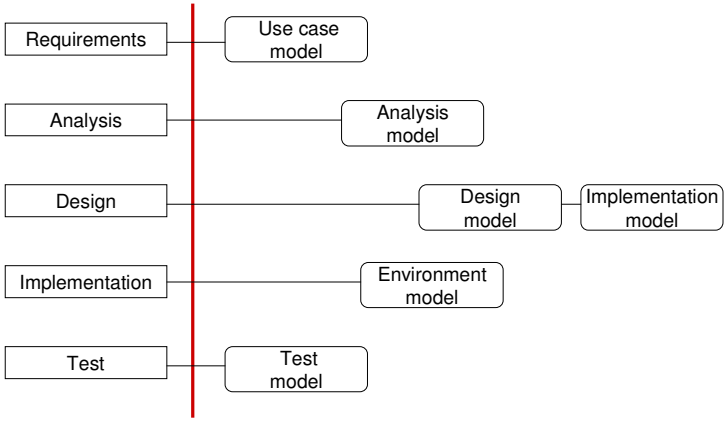
e-Macao-18-1-576

Workflows and Activities

Workflows	Activities
Requirements	Find actors and use cases, prioritize use cases, detail use cases, prototype user interface, structure the use case model
Analysis	Architectural analysis, analyze use cases, explore classes, find packages
Design	Architectural design, trace use cases, refine and design classes, design packages
Implementation	Architectural implementation, implement classes and interfaces, implement subsystems, perform unit testing, integrate systems
Test	Plan and design tests, implement tests, perform integration and system tests, evaluate tests

e-Macao-18-1-577

Workflows and Models



e-Macao-18-1-578

Summary 1

A software development process is a framework that provides guidance to carry out the different activities needed to produce a software product.

Every software development process must be parameterized for each individual project.

There is no universal process.

e-Macao-18-1-579

Summary 2

The main features of the Unified Process are:

1) use case-driven
2) architecture-centric
3) iterative and incremental

With respect to time dimension, the lifecycle is divided into phases and iteration.

There are four main phases: inception, elaboration, construction and transition.

A specific set of artifacts are produced with well-defined activities called workflows.

There are five main workflows: requirements, analysis, design, implementation and test.

A.10. UML Tools

UML Tools

e-Macao-18-1-581

Course Outline

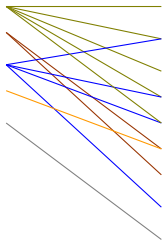
1) Object Orientation

2) UML Basics

3) UML Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

4) Unified Process

5) UML Tools



UML Diagrams:
1. use case
2. class
3. object
4. sequence
5. state
6. component
7. collaboration
8. activity
9. deployment

e-Macao-18-1-582

Overview

1) Why UML CASE tools?

2) Benefits

3) Different tools

4) Evaluating UML CASE tools

5) Main Features of:
   a) Enterprise Architect
   b) MagicDraw
   c) Poseidon
   d) Rational Rose

e-Macao-18-1-583

Why UML CASE Tools

1) tools offer benefits to everyone involved in a project:
   a) analysts can capture requirements with use case model
   b) designers can produce models that capture interactions between objects
   c) developers can quickly turn the model into a working application

2) UML case tool, plus a methodology, plus empowered resources enable the development of the right software solution, faster and cheaper

e-Macao-18-1-584

Different UML CASE Tools

Tools vary with respect to:

1) UML modelling capabilities
2) project life-cycle support
3) forward and reverse engineering
4) data modelling
5) performance
6) price
7) supportability
8) easy of use
9) …

e-Macao-18-1-585

Evaluating CASE Tools

Different Criteria for evaluating CASE tools:

1) repository support
2) round-trip engineering
3) HTML documentation
4) UML support
5) data modelling integration
6) versioning
7) model navigation
8) printing support
9) diagrams views
10) exporting diagrams
11) platform

e-Macao-18-1-586

Different UML Tools

- Enterprise Architect
  - organization: Sparx Systems
  - web-site: http://www.sparxsystems.com.au/
- MagicDraw
  - organization: No Magic Inc.
  - web-site: http://www.nomagic.com/
- Poseidon
  - organization: Gentleware
  - web-site: http://www.gentleware.com/
- Rational Rose
  - organization: IBM
  - web-site: http://www-306.ibm.com/software/rational/

e-Macao-18-1-587

Enterprise Architect

Criteria	Features
Platform	Windows
UML Compliance	Support for all 13 UML 2.0 diagrams
Usability	Replication capable – Comprehensive and flexible documentation
Development Environment	Multi-user enabled – Allows to replicate and share projects
Outputs & Code Generation	C++, Java, C#, VB, VB.Net, Delphi, PHP – HTML and RTF document generation - Forward and reverse database engineering
Data Repository	Models are stored in a data repository - Checks data integrity in the data repository – Provides a project browser
Other features	Allows scripting to extend functionality – Project estimation tools – User definable patterns

e-Macao-18-1-588

Magic Draw

Criteria	Features
Platform	Any where Java 1.4 is supported
UML Compliance	Support for UML 1.4 notation and semantics
Usability	Replication capable – Customizable views of UML elements – Customizable elements properties
Development Environment	Multi-user enabled – Lock parts of the model to edit – Commit changes – Model versioning and rollback
Outputs & Code Generation	Code generation and reverse engineering to C++, Java, C# - RTF and PDF document generation
Data Repository	Provides a project browser
Other features	Friendly and customizable GUI – Hyperlinks can be added to any model element

e-Macao-18-1-589

Poseidon

Criteria	Features
Platform	Platform independent – Implemented in Java
UML Compliance	Supports all 9 diagrams of UML 1.4
Usability	Replication capable – Internationalization an localization for several languages
Development Environment	Collaborative environment based on client-server architecture – Locking of model parts – Secure transmission of files
Outputs & Code Generation	VB.Net, C#, C++, CORBA IDL, Delphi, Perl, PHP, SQL DDL – Round trip engineering for Java – Diagram export as gif, ps, eps and svg.
Data Repository	Uses MDR (Meta Data Repository) developed by Sun and based on the JMI (Java Metadata Interface) standard
Other features	Allows to import Rational Rose files

e-Macao-18-1-590

Rational Rose

Criteria	Features
Platform	Windows
UML Compliance	Not fully supported UML 1.4
Usability	The add-in feature allows to customize the environment – User configurable support for UML, OMT and Booch 93.
Development Environment	Parallel multi-user development through repository and private support
Outputs & Code Generation	C++, Visual C++, VB6, Java – Documentation generation – Round trip engineering
Data Repository	Maintains consistency between the diagram and the specification, you may change any of them and automatically updates the information.
Other features	Can be integrated with other Rational products such as RequisitePro, Test Manager

e-Macao-18-1-591

Summary

There exist several CASE Tools supporting Object Oriented modelling with UML.

Different criteria should be considered when evaluating a software tool.

Four tools were presented: Enterprise Architect, Magic Draw, Poseidon and Rational Rose.

A.11. Summary

Course Summary

e-Macao-18-1-593

Object Orientation

Object Orientation is about viewing and modelling the world/system as a set of interacting and interrelated *objects.*

Four principles:

1) abstraction
2) encapsulation
3) modularity
4) hierarchy

e-Macao-18-1-594

Object Oriented Concepts

- Object
- Class
- Attribute
- Operation
- Interface
- Implementation
- Association
- Aggregation
- Composition
- Generalization
- Super-class / Sub-class
- Abstract class / Concrete-class
- Discriminator
- Polymorphism
- Realization

e-Macao-18-1-595

UML Basics

Unified Modeling Language – standard published by OMG

Building Blocks:

1) elements
2) relationships
3) diagrams

e-Macao-18-1-596

UML Elements 1

Structural:

1) class
2) interface
3) collaboration
4) use case
5) active class
6) component
7) node

e-Macao-18-1-597

UML Elements 2

Behavioural:

1) interactions
2) state-machines

Grouping:

1) package

Annotation:

1) notes

e-Macao-18-1-598

UML - Diagrams

Static

1) Class
2) Object
3) Component
4) Deployment

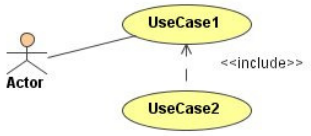
Dynamic

1) Use Case
2) Sequence
3) Collaboration
4) Statechart
5) Activity

e-Macao-18-1-599

Use Case Diagram

Shows the functions of the system (external view).
Components:
1) use cases
2) actors
3) relationships:
   a) use cases:
      – generalization
      – dependency: includ
   b) actors:
      – generalization
   c) use cases – actors:
      – association



e-Macao-18-1-600

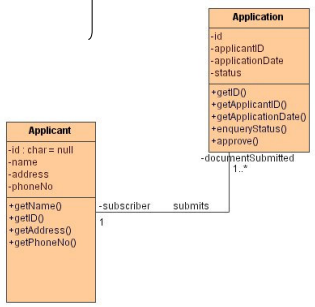
Class Diagram

Shows the classes and their relationships.
Main components:
1) classes
   • attributes – types, default value
   • operations – arguments,
2) relationships
   • association
   • aggregation
   • composition
   • generalization
   • dependency

visibility

multiplicity

roles



e-Macao-18-1-601

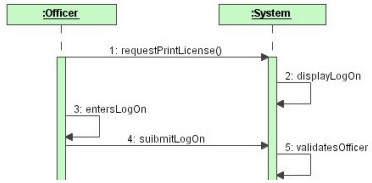
Object Diagram

Shows the instances for a given class diagram in a point of time.
Components:
1) objects
2) links

e-Macao-18-1-602

Sequence Diagram

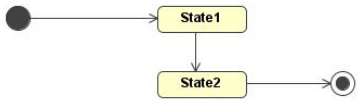
Shows interactions between objects – (ordered in time).
Main components:
1) object lifelines:
   - object
   - lifeline
2) interactions - messages



e-Macao-18-1-603

Statechart Diagram

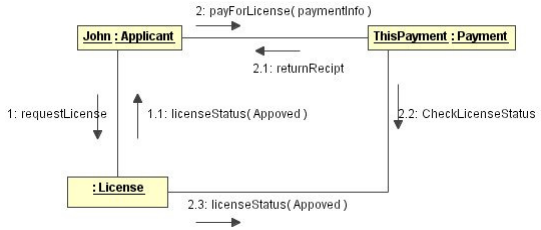
Shows the state of an object(s) in response to events.
Main components:
1) states:
   - pseudo-states:
     – initial – final
     – static/dynamic branch points
   - simple states
   - composite states
     – mutually exclusive sub-states
     – concurrent sub-states
2) events



e-Macao-18-1-604

Collaboration Diagrams

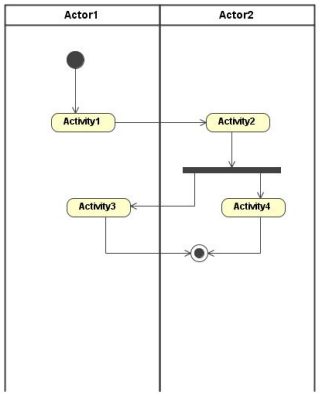
Shows interaction between objects – (ordered in time).
Main components:
a) classes – associations
   objects – links
b) messages



e-Macao-18-1-605

Activity Diagram

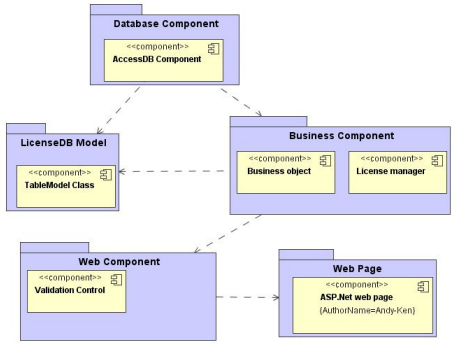
Focuses on the internal execution of activities.
Main components:
1) states
2) transitions
3) decisions
4) forks – joins
5) swimlanes



e-Macao-18-1-606

Component Diagram

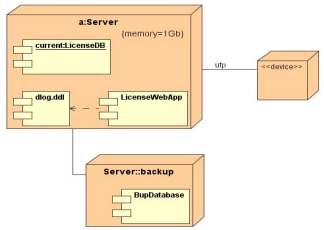
Shows structural replaceable parts of the system.
Main components:
1) components
2) interfaces
3) packages



e-Macao-18-1-607

Deployment Diagram

Shows the nodes and components of the system.
Main components:
1) nodes:
   - processor
   - devices
2) components
3) packages



e-Macao-18-1-608

Development Stages

1) Requirements
2) Analysis

requirements modelling

3) Design → architecture and detailed design
4) Implementation
5) Deployment

e-Macao-18-1-609

Development 1

- Requirements Modelling:
  - functional and non-functional requirements
  - template and glossary
  - diagrams:
    1) use case diagram
    2) class diagram
    3) sequence diagram
    4) statechart diagram

e-Macao-18-1-610

Development 2

- Architecture:
  - concepts:
    - subsystem
    - cohesion – coupling
    - layering
  - diagrams:
    - collaboration diagram
    - component diagram
  - architecture styles:
    - repository
    - MVC
    - client-server

e-Macao-18-1-611

Development 3

- Design:
  - activities
  - diagrams:
    - class diagrams
    - activity diagrams
    - sequence diagrams
    - statecharts diagrams
  - patterns

e-Macao-18-1-612

Development 4

- Implementation:
  - diagram:
    - component diagram

e-Macao-18-1-613

Development 5

- Deployment:
  - diagram:
    - deployment diagram

e-Macao-18-1-614

Unified Process

1) iterative and incremental

2) use case-driven

3) architecture-centric

e-Macao-18-1-615

UP Workflows

1) Requirements

2) Analysis

3) Design

4) Implementation

5) Test

e-Macao-18-1-616

UML Tools

- Magic Draw
- Rational Rose
- Poseidon
- Enterprise Architect

The End

e-Macao-18-1-618

Acknowledgements

We would like to thank Dr. Tomasz Janowski and all the members of the eMacao team for their valuable comments and help in preparing this material.

B. Assessment

B.1. Set 1

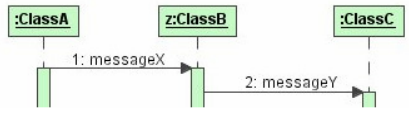
1	Divide a complex system into small, self-contained pieces that can be managed independently. How is it called?	
	a	Abstraction
	b	Modularity
	c	Encapsulation
	d	Hierarchy
	Answer: B	

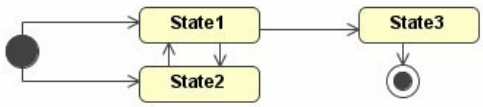
2	In order to model the relationship "a course is composed of 5 to 20 students and one or more instructors", you could use:	
	a	Aggregation
	b	Association
	c	Composition
	d	Realization
	Answer: A	

3	Which of the following statements are true?	
	a	All operations defined in a sub-class are inherited by the super-class
	b	Generalization allows abstracting common features and defining them in a super-class
	c	A super-class is a class that must not have associations
	d	Association is a "part-of" relationship
	Answer: B	

4	What is the relationship between these two use cases?	
	a	Generalization
	b	Extend
	c	Include
	d	Association
	Answer: C	

5	The following diagram shows that there is an interaction between:	
	a	An object of ClassA and an object of ClassB
	b	An object of ClassA and object z of ClassB
	c	Object z of ClassB and an object of ClassB
	d	An object of ClassC and an object of ClassA
	Answer: B	



6	Which of the following statements are true for the following diagram?			
	a	State1 is always the first state of the object after the initial state	c	State3 is always the last state of the object before the final state
	b	State2 is always the last state of the object before the final state	d	During its life, the object is in at least five states
	Answer: C			

7	How many *File*s objects for each *Directory* object?	
	a	0 or 1
	b	2
	c	1 or 2
	d	Many
	Answer: C	

8	Which of the following statement is true for the following diagram	
	a	A is a kind of B
	b	B is a kind of B
	c	A is part of B
	d	B depends on A
	Answer: None	

9	Which of these diagrams shows interactions between objects?	
	a	Activity diagram
	b	Class diagram
	c	Sequence diagram
	d	Component diagram
	Answer: C	

10	A statechart diagram describes:	
	a	Attributes of objects
	b	Nodes of the system
	c	Operations executed on a thread
	d	Events triggered by an object
	Answer: D	

11	An interface is:	
	a	A set of objects used to provide a specific behaviour
	b	A set of classes used on a collaboration
	c	A set of attributes used on an operation
	d	A set of operations used to specify a service of a class or component
	Answer: D	

12	The sequence diagram models:
	a The order in which the class diagram is constructed
	b The way in which objects communicate
	c The relationship between states
	d The components of the system
	Answer: B

13	The activity diagram:
	a Focuses on flows driven by internal processing
	b Models the external events stimulating one object
	c Focuses on the transitions between states of a particular object
	d Models the interaction between objects
	Answer: A

14	The deployment diagram shows:
	a Objects of a system
	b Distribution of components on the nodes in a system
	c Functions of a system
	d Distribution of nodes
	Answer: B

15	Unified Process is a software development methodology which is:
	a Use-case driven
	b Component-driven
	c Related to Extreme Programming
	d None in only one iteration
	Answer: A

B.2. Set 2

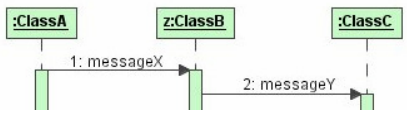
1	Ordering abstractions into a tree-like structure. How is it called?	
	a	Abstraction
	b	Modularity
	c	Encapsulation
	d	Hierarchy
	Answer: D	

2	In order to model the relationship "a hotel has rooms", between Hotel and Room, you could use:	
	a	Aggregation
	b	Association
	c	Composition
	d	Realization
	Answer: C	

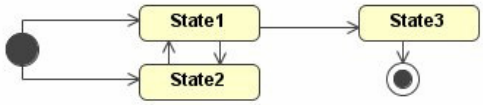
3	Which of the following statements are true?	
	a	All operations defined in a super-class are inherited by the sub-class
	b	Generalization allows abstracting common features and defining them in a sub-class
	c	A super-class is a class that must not have associations
	d	Association is a "kind-of" relationship
	Answer: A	

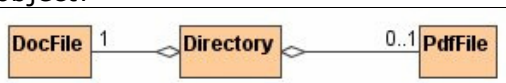
4	What is the relationship between these two use cases?	
	a	Include
	b	Extension
	c	Generalization
	d	Association
	Answer: C	

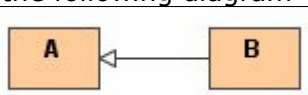
5	The following diagram shows that there is an interaction between:	
	a	An object of ClassC and an object of ClassA
	b	Object z of ClassB and an object of ClassB
	c	An object of ClassA and object z of ClassB
	d	An object of ClassA and an object of ClassC
	Answer: C	



6	Which of the following statements are true for the following diagram?			
	a	State2 is always the first state of the object after the initial state	c	State3 is always the last state of the object before the final state
	b	State1 is always the last state of the object before the final state	d	During its life, the object is in at least five states
	Answer: C			



7	How many *File*s objects for each *Directory* object?	
	a 0 or 2	
	b 0 or 1	
	c 1 or 2	
	d Many	
	Answer: C	

8	Which of the following statement is true for the following diagram	
	a B is a kind of A	
	b A is part of B	
	c A is a kind of B	
	d B depends on A	
	Answer: A	

9	Which of these diagrams shows interactions between objects?
	a Sequence diagram
	b Class diagram
	c Activity diagram
	d Component diagram
	Answer: A

10	A statechart diagram describes:
	a Operations executed on a thread
	b Nodes of the system
	c Attributes and operations of an object
	d Events triggered by an object
	Answer: D

11	An interface is:
	a A set of classes used on a collaboration
	b A set of operations used to specify a service of a class or component
	c A set of attributes used on an operation
	d A set of objects used to provide a specific behaviour
	Answer: B

12	The sequence diagram models:
	a The order in which the class diagram is constructed
	b The relationship between objects
	c The way in which objects communicate
	d The components of the system
	Answer: C

13	The activity diagram:
	a Models the interaction between objects
	b Models the external events stimulating one object
	c Focuses on the transitions between states of a particular object
	d Focuses on flows driven by internal processing
	Answer: D

14	The deployment diagram shows:
	a Objects of a system
	b Functions of a system
	c Distribution of components on the nodes in a system
	d Distribution of nodes
	Answer: C

15	Unified Process is a software development methodology which is:
	a Component-driven
	b Iterative and incremental
	c Related to Extreme Programming
	d Done in only one iteration
	Answer: B

Object-Oriented Analysis and Design with UML

Adegboyega Ojo and Elsa Estevez

UNU-IIST

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

The Course: Objectives

1) present concepts of Object Oriented paradigm, as well as Object Oriented Analysis (OOA) and Design (OOD).

2) introduce the syntax, semantics, and pragmatics of UML and how to integrate it with the Unified Process

3) show how to articulate requirements using use cases

4) present how to develop structural and behavioural diagrams for the different views supported in UML

5) introduce concepts of Patterns and Frameworks and their applications in architecture and design tasks

6) present a comparative analysis of some major UML tools suitable industrial strength development

The Course: Literature

1) OMG Unified Modeling Language Specification, Object Management Group.

2) UML Bible, Tom Pender, John Wiley and Sons, 2003.

3) Object-Oriented Analysis and Design using UML, Simon Bennet, Steve McRobb and Ray Farmer, McGraw-Hill, 2002.

4) Guide to Applying the UML, Sinan Si Alhir, Springer, 2002.

5) Object-Oriented Software Engineering, Bernd Bruegge, Allen H Dutoit, Prentice Hall, 2000.

6) The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, 1999.

7) OO Software Development Using UML, UNU-IIST Tech Report 229.

The Course: Approach

1) provide a general foundation of object orientation

2) discuss the basic building blocks of the UML – model elements, relationships, diagrams

3) present the UML diagrams as they relate to the core workflows of any development process:
    a) Requirements: Use Case, Class, Object, Sequence and Statechart
    b) Architecture: Collaborations and Components
    c) Design: Class, Sequence, Statechart and Activity
    d) Implementation: Components and Deployment

4) diagram features are presented in increasing details as required for each workflow

Object Oriented Concepts

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

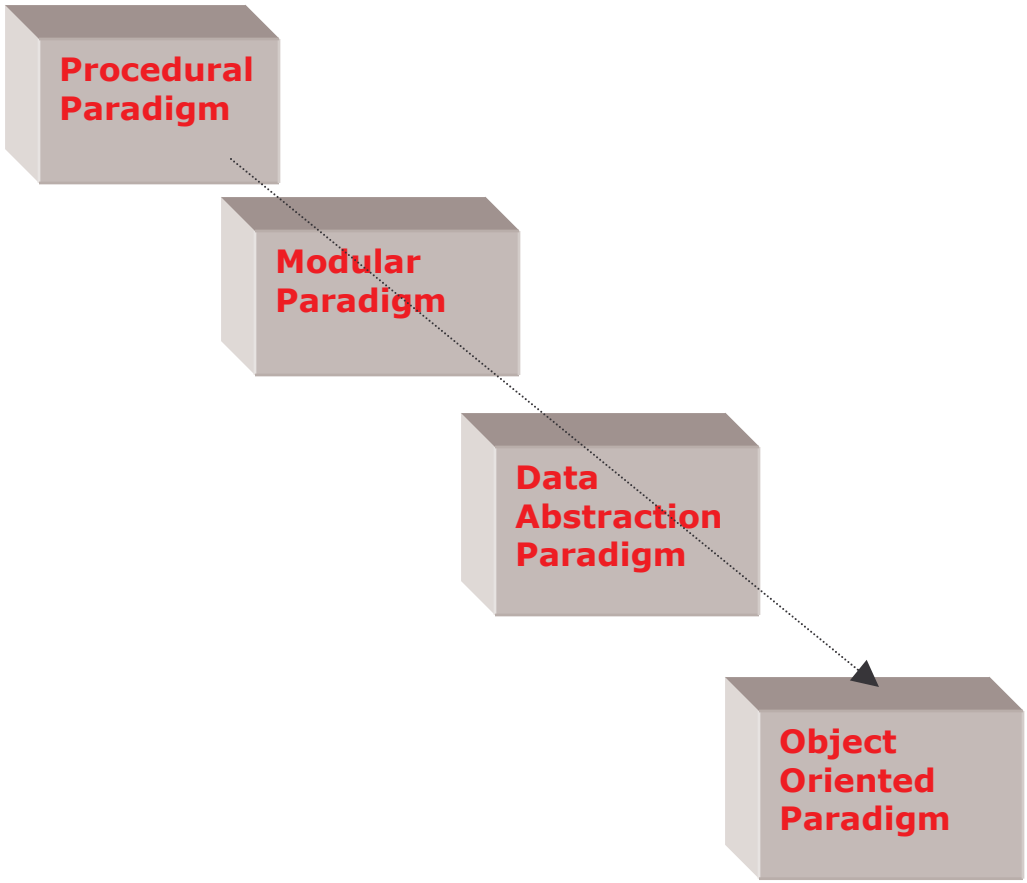
8) Summary

Overview: OO Concepts

1) Background and Principles of Object Orientation
2) Object Oriented Concepts
3) Object Oriented Analysis (OOA)
4) Object Oriented Design (OOD)

Background and Principles

The Road to OO



What is OO?

Definition

Object orientation is about viewing and modelling the world (or any system) as a set of interacting and interrelated objects.

An approach characterized by the following features:

1) views the universe of discourse as consisting of interacting objects
2) describes and builds systems consisting of representation of objects

Principles of OO

1) Abstraction

2) Encapsulation

3) Modularity

4) Hierarchy

Abstraction

A model that includes most important aspects of a given system while ignoring less important details.

Abstraction allows us to manage complexity by concentrating on essential characteristics that makes an entity different from others.

Customer

Salesman

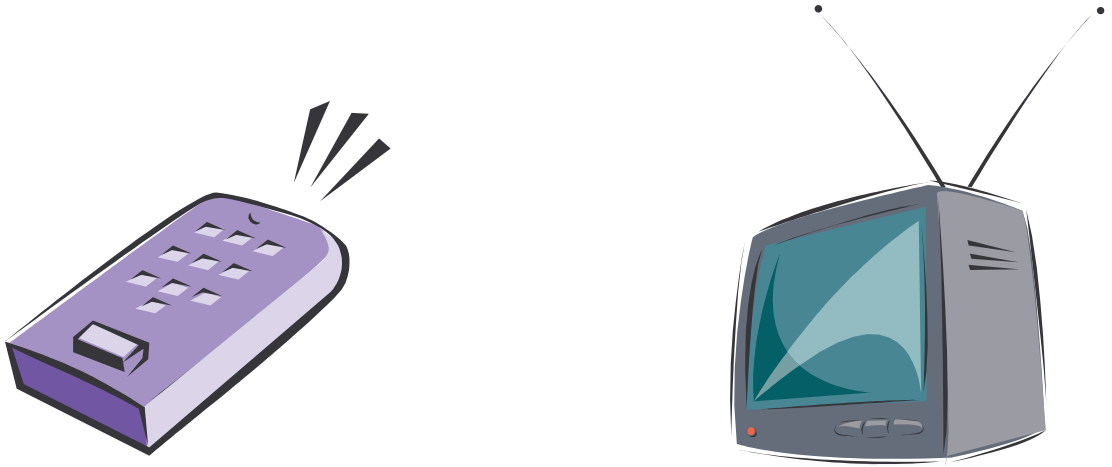
Product

An example of an order processing abstraction

Encapsulation 1

1) Encapsulation separates implementation from users or clients.

2) Clients depend on interface.



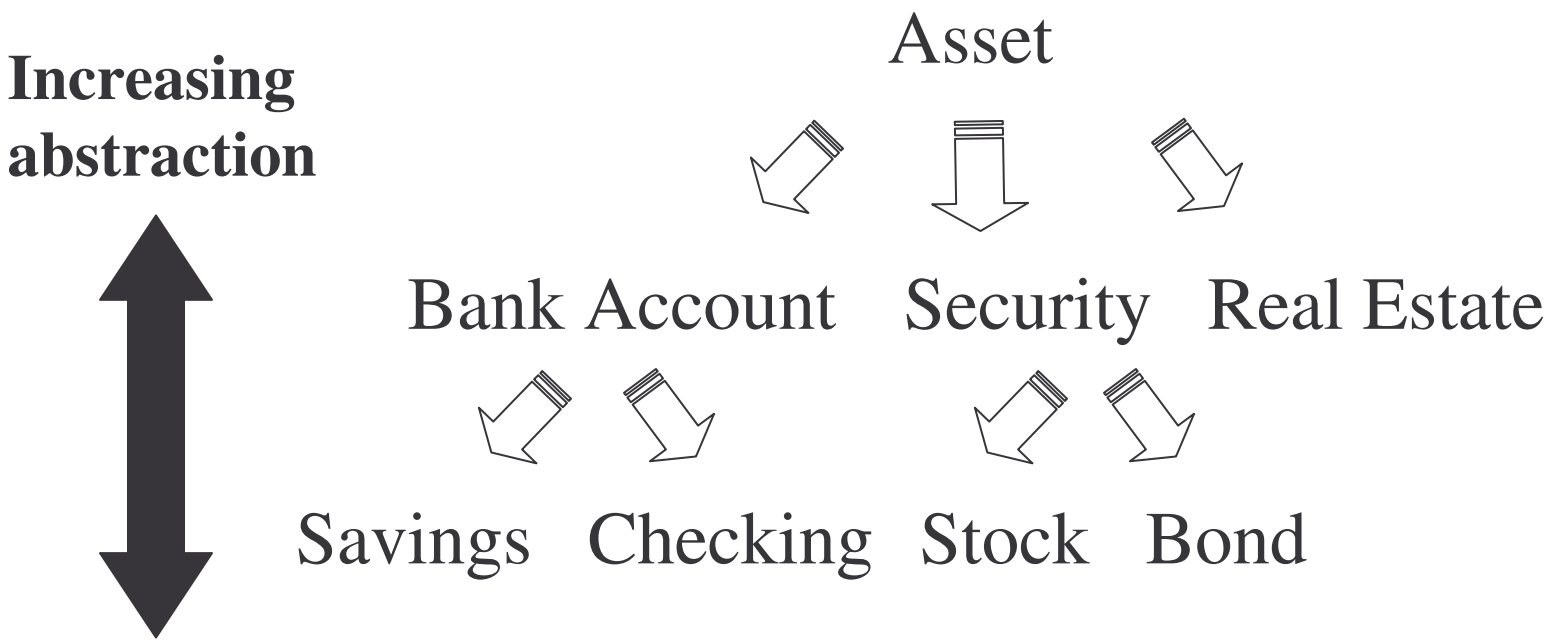
Courtesy Rational Software

Modularity

Modularity deals with the process of breaking up complex systems into small, self contained pieces that can be managed easily.

Hierarchy

Is an ordering of abstractions into a tree like structure.



Concepts

Objects

What is an object?

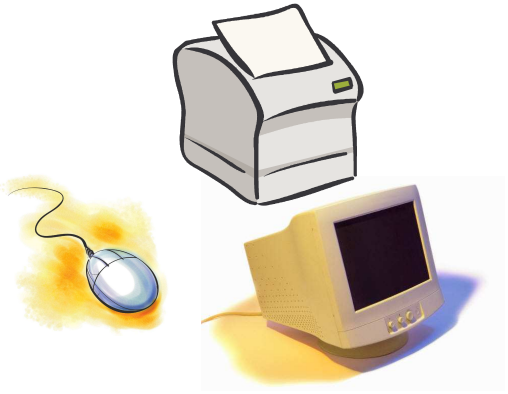
1) any abstraction that models a single thing

2) a representation of a specific entity in the real world

3) may be tangible (physical entity) or intangible

4) examples: specific citizen, agency, job, location, order …

Example: Objects



Object Definition

Two aspects: information and behaviour

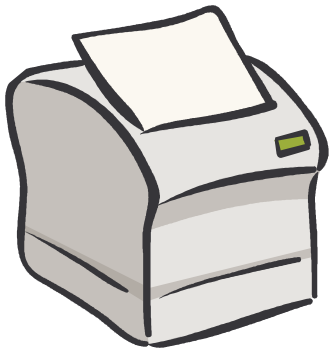
Information:

1) has a unique identity
2) has a description of its structure or the information that is used to create it
3) has a state representing its current condition, e.g. values of some of its features

Behaviour:

1) what can the object do?
2) what can be done to it?

Example: Object Definition



1) information:
   a) serial number
   b) model
   c) speed
   d) memory
   e) status

2) behaviour:
   a) print file
   b) stop printing
   c) remove file from queue

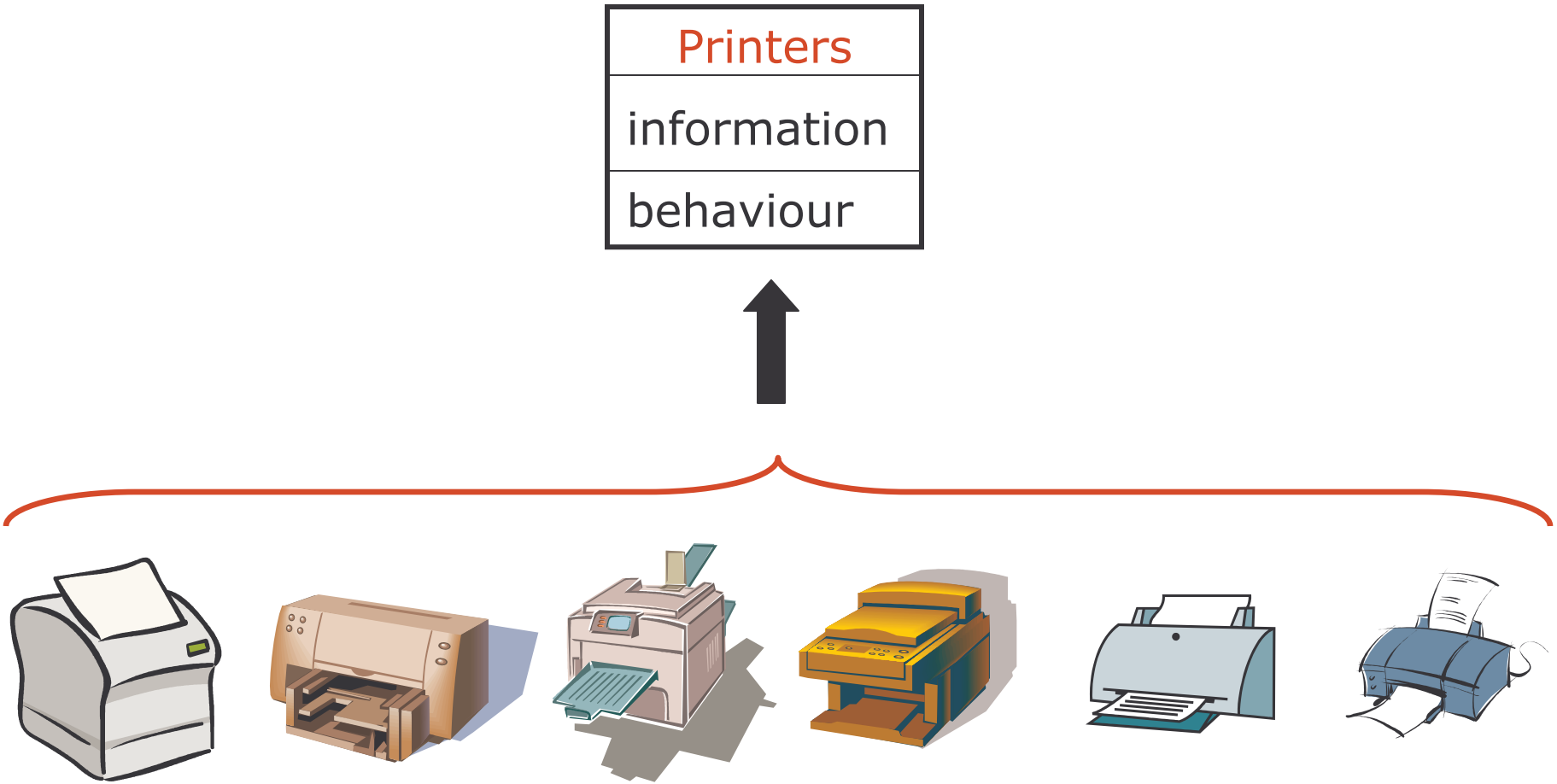
Classes

What is a class?

1) any uniquely identified abstraction of a set of logically related instances that share the same or similar characteristics
2) rules that define objects
3) a definition or template that describes how to build an accurate representation of a specific type of objects
4) examples: agency, citizen, car, …

Objects are instantiated (created) using class definitions as templates.

Example: Classes



Attributes

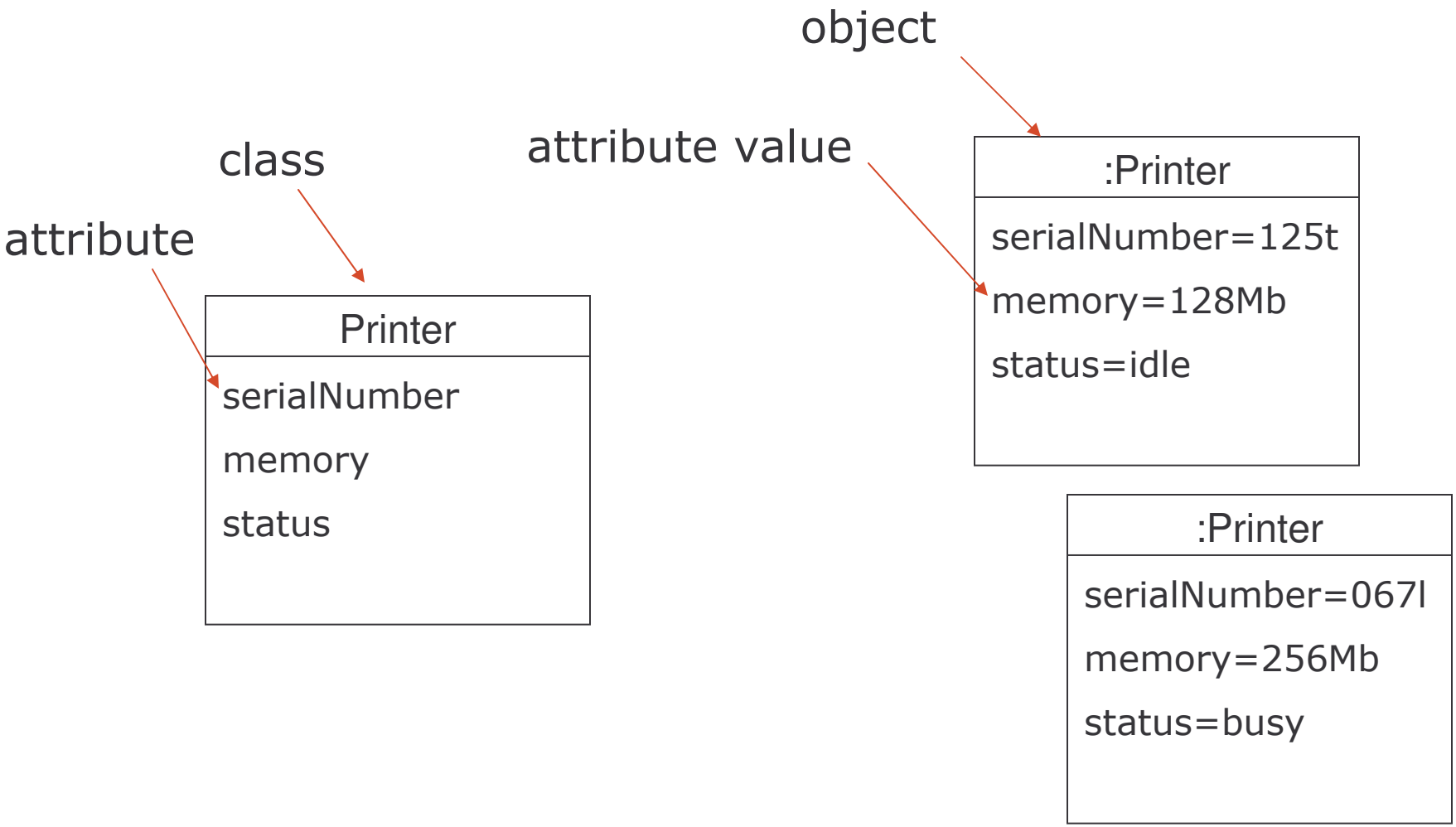
Definition
Attribute is a named property of a class that describes a range of values that instances of the class may hold for that property.

An attribute has a type and defines the type of its instances.

Only the object itself should be able to change the values of its attributes.

The given set of values of the attributes defines the state of the object.

Example: Attributes



Operations

Definition

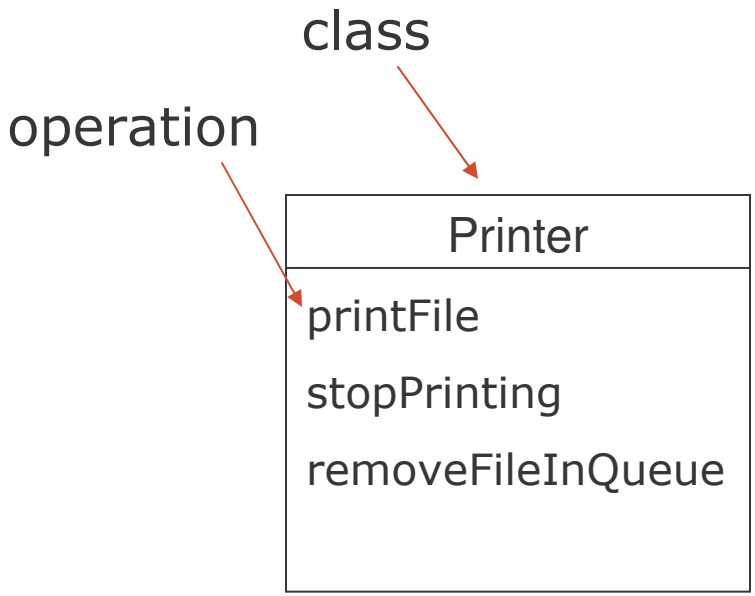
Operation is the implementation of a service that can be requested from any object of the class.

An operation could be:

Question - does not change the values of the object

Command – may change the values of the object

Example: Operations

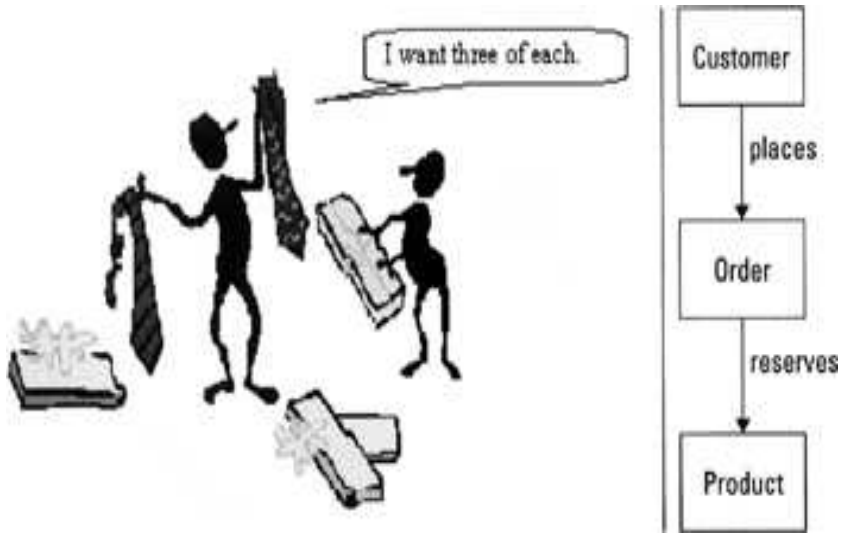


Applying Abstraction

Abstraction in Object Orientation:

1) use of objects and classes in representing reality

2) software manages abstractions based on the changes occurring to real-world objects



Courtesy: XML Bible

Encapsulation 2

1) hallmark of object orientation

2) behaviour is defined once and stored inside objects

3) emphasizes two types of information:

    a) interface information - minimum information for using the object

    b) implementation information - information required to make an object work properly

Interface

1) minimum information required to use an object

2) allows users to access the object's knowledge

3) must be exposed

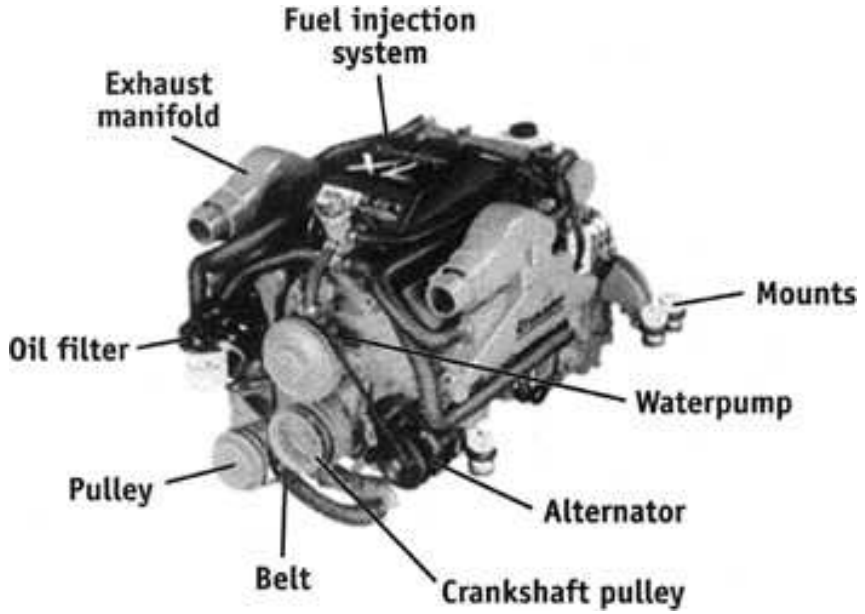
4) provides no direct access to object internals

Implementation

1) information required to make an object work properly

2) a combination of the behaviour and the resources required to satisfy the goal of the behaviour

3) ensures the integrity of the information on which the behaviour depends



Encapsulation Requirements

1) to expose the purpose of an object
2) to expose the interfaces of an object
3) to hide the implementation that provides behaviour through interfaces
4) to hide the data within an object that defines its structure and supports its behaviour
5) to hide the data within an object that tracks its state

Encapsulation Benefits

- separation of an interface from implementation, so that one interface may have multiple implementations

- data held within one object cannot be corrupted by other objects

Associations and Links

Association:

- expresses relationships between classes
- defines links between instances of classes (objects)

Link:

1) expresses relationships between objects

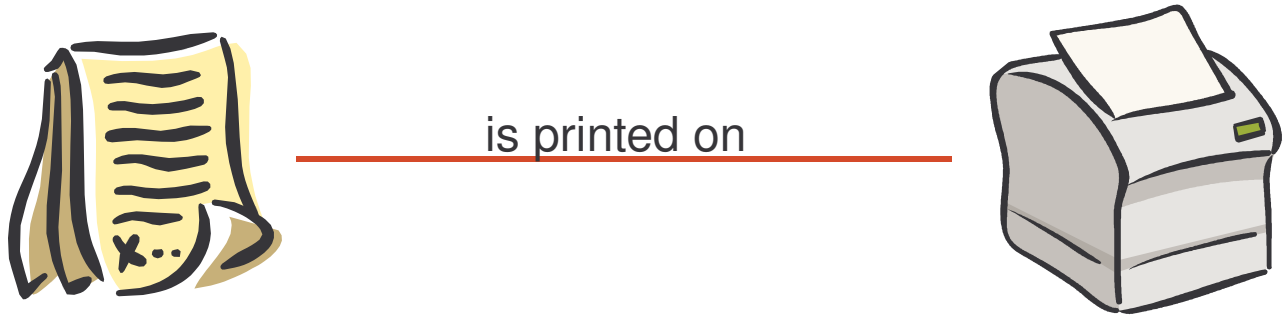
There are different kinds of relationships between classes:

1) association
2) aggregation
3) composition

Example: Associations - Links

a) Association:

b) Link:



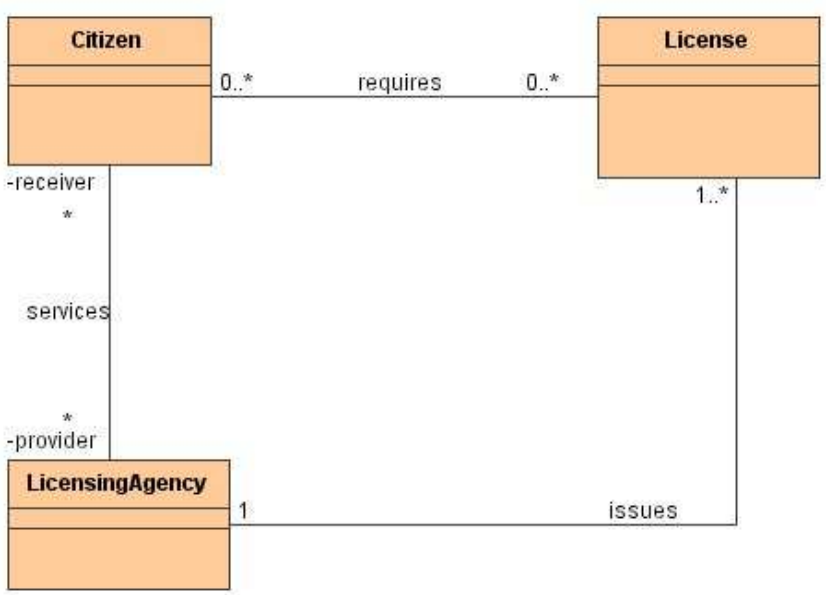
File: F100034.txt

Printer: ljd4

Relationship: Association

Features:

1) the simplest form of relationship between classes
2) a peer-to-peer relationship
3) one object is generally aware of the existence of other object
4) implemented in objects as references



Example: Associations

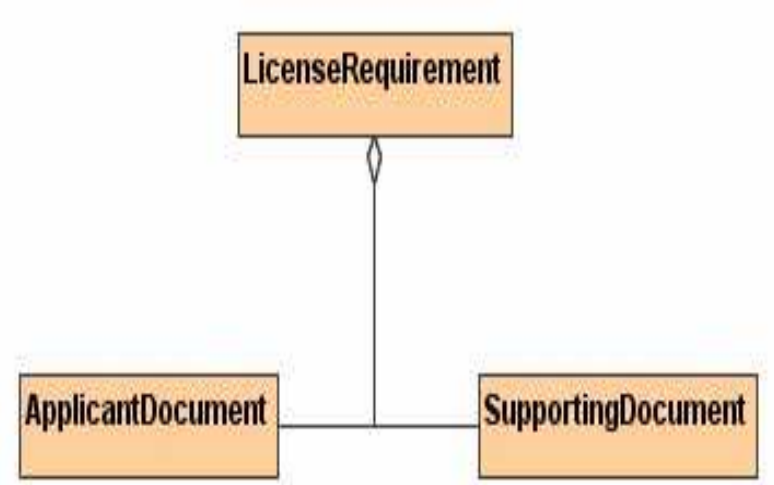
Association between classes A and B:

1) A is a physical/logical part of B
2) A is a kind of B
3) A is contained in B
4) A is a description of B
5) A is a member of B
6) A is an organization subunit of B
7) A uses or manages B
8) A communicates with B
9) A follows B
10) A is owned by B
11) …

Relationship: Aggregation

Features:

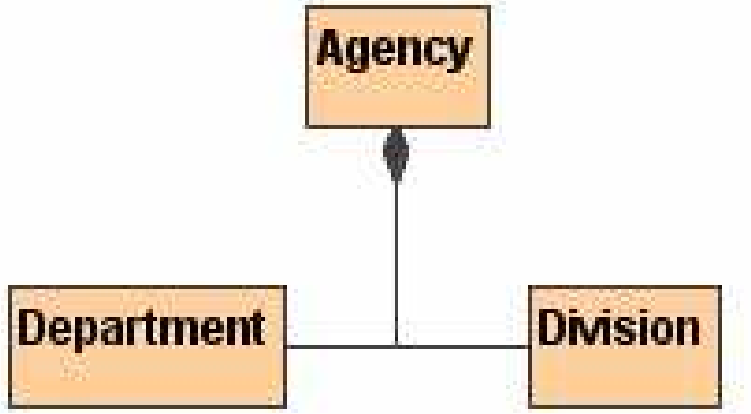
1) a more restrictive form of "part-of" association
2) objects are assembled to create a new, more complex object
3) assembly may be physical or logical
4) defines a single point of control for participating objects (parts)
5) the aggregate object coordinates its parts



Relationship: Composition

Features:

1) a stricter form of aggregation
2) lifespan of parts depend on the on lifespan of the aggregate object
3) parts cannot exist on their own
4) there is a create-delete dependency of the parts on the whole



Inheritance or Generalization

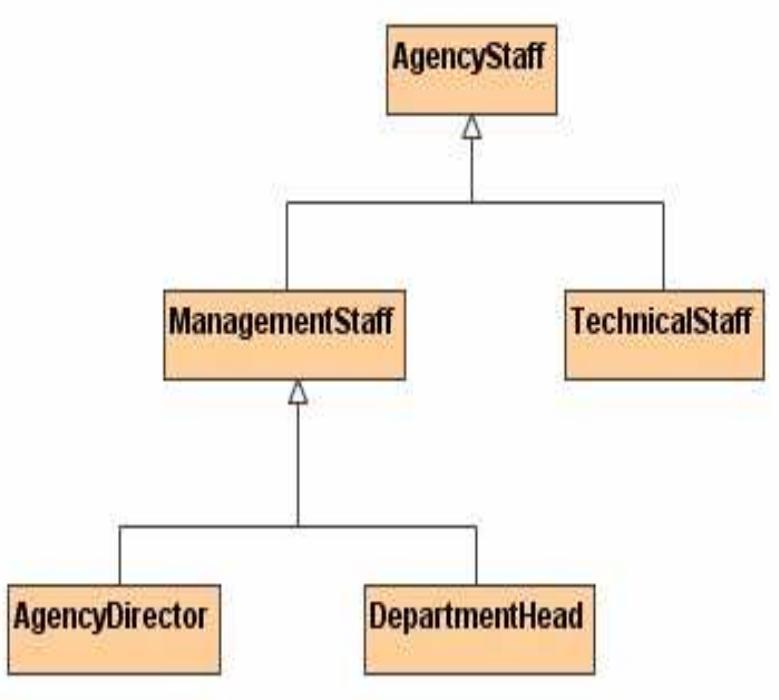
1) a process of organizing the features of different kinds of objects that share the same purpose

2) equivalent to "kind-of" or "type-of" relationship

3) also referred to as inheritance

4) specialization is the opposite of generalization

5) not an association!



Super-Class and Sub-Class

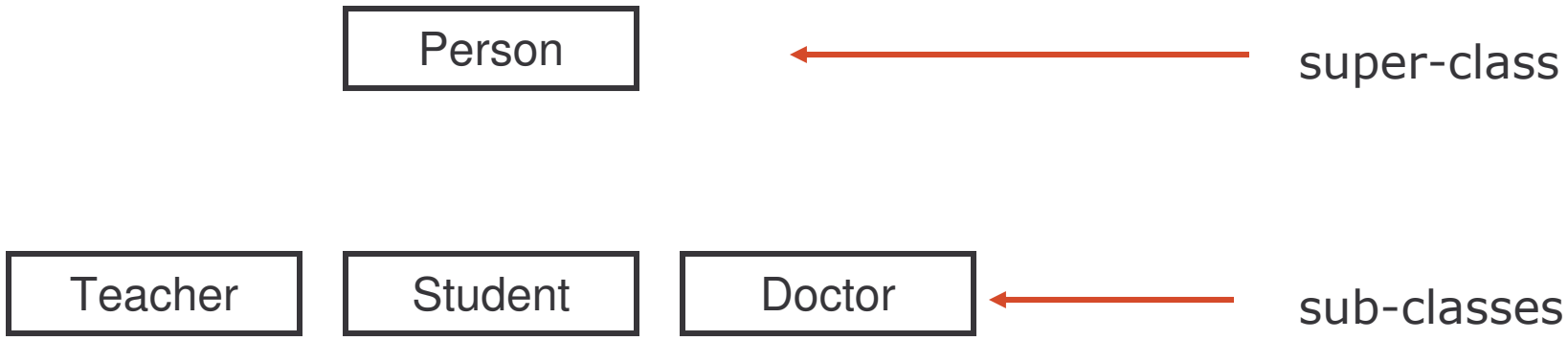
Definition
Super-class is a class that contains features that are common to two or more classes.

A super-class is similar to a superset. e.g. agency-staff.

Definition
Sub-class is a class that contains features of its super-class or super-classes, and perhaps more.

A class may be a sub-class and a super-class at the same time. e.g. management-staff.

Example:Super- and Sub-Class



Abstract and Concrete Class

Abstract class

1) a class that lacks a complete implementation
2) provides operations without implementing methods
3) cannot be used to create objects, i.e. cannot be instantiated
4) a concrete sub-class must provide methods for unimplemented operations

Concrete class

1) has methods for all operations
2) can be instantiated
3) methods may be defined in the class or inherited from a super-class

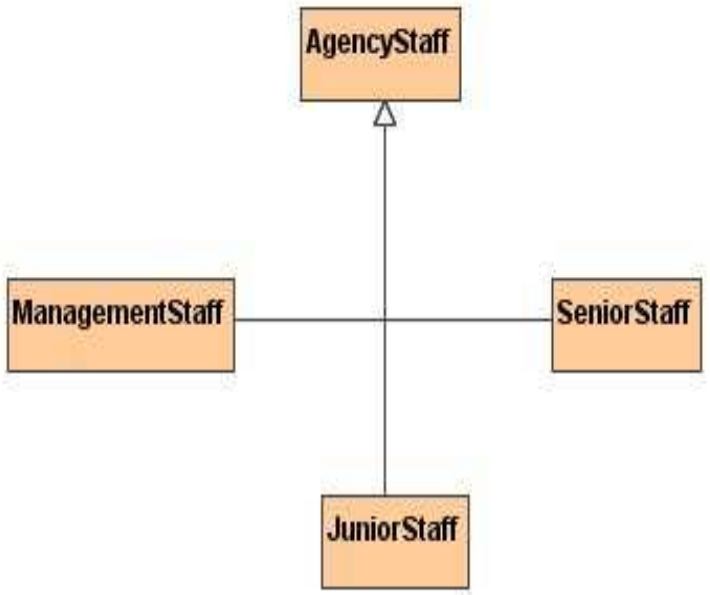
Final / Leaf Class

Leaf class

1) cannot have any sub-classes

2) a leaf node of the generalization hierarchy

Example: management staff, senior staff and junior staff are final or leaf classes.

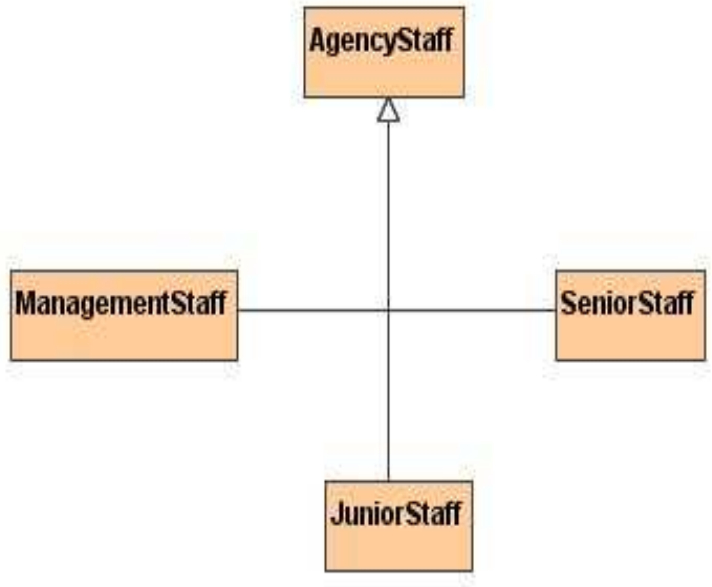


Discriminator

Discriminator

attributes that define sub-classes

Example: “status” of agency staff is a possible discriminator to derive “management”, “senior” and “junior” sub-classes



Discrimination Criteria

1) attribute type

1) attribute values allowed

2) operation or interface

3) method or implementation

4) association

Polymorphism

What is it?

1) ability to dynamically choose the method for an operation at run-time or service time

2) facilitated by encapsulation and generalization:

   a) encapsulation: separation of interface from implementation

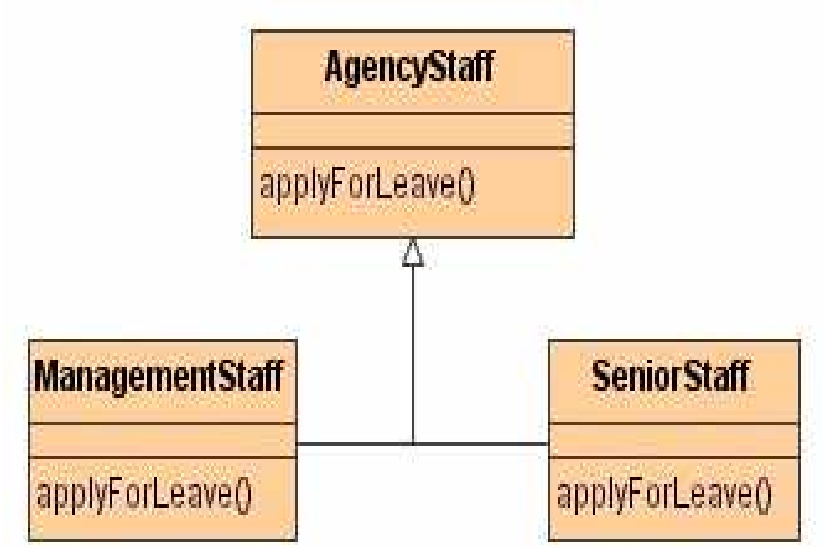
   b) generalization: organizing information such that the shared features reside in one class and unique features in another

3) therefore: operations could be defined and implemented in a super-class, but the re-implemented methods are in unique sub-classes.

Example: Polymorphism

Many ways of doing the same thing!

Example: management-staff and agency-staff can apply for leave, but possibly in different ways



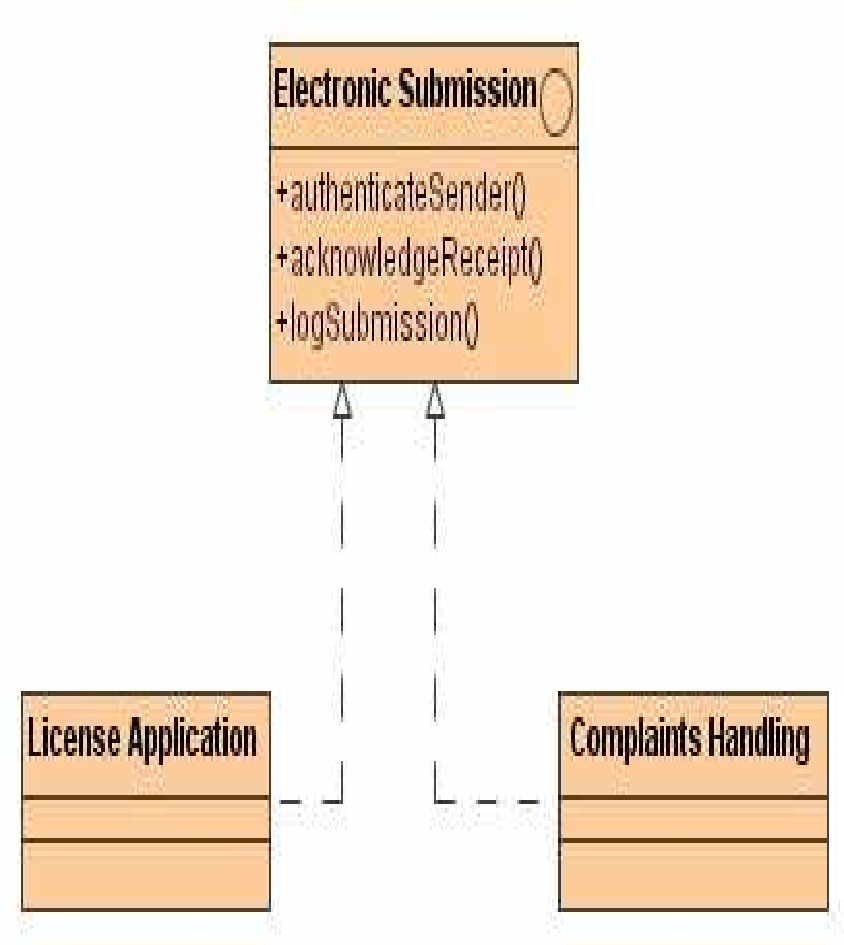
Realization

Realization

1) allows a class to inherit from an interface class without being a sub-class of the interface class

2) only inherits operations

3) cannot inherit methods, attributes, and associations



Object Oriented Analysis

Object Oriented Analysis 1

1) a discovery process

2) clarifies and documents the requirements of a system

3) focuses on understanding the problem domain

4) discovers and documents the key classes in the problem domain

5) concerned with developing an object-oriented model of the application domain

6) identified objects reflect the entities that are associated with the problem to be solved

Object Oriented Analysis 2

Definition [SEI]
Object Oriented Analysis (OOA) is concerned with developing software requirements and specifications expressed as a system's object model (composed of a population of interacting objects), as opposed to the traditional data or functional views of systems.

Benefits

1) maintainability: simplified mapping to the real world
    a) less analysis effort
    b) less complexity in system design
    c) easier verification by the user

2) reusability: reuse of the analysis artifacts that are independent of the analysis method or programming language

3) productivity: direct mapping to features of OOP languages

Object Oriented Design

Object Oriented Design 1

- process of invention and adaptation
- creates abstraction and mechanisms necessary to meet the system's behavioural requirements determined during analysis
- language independent
- provides an object-oriented model of a software system to implement the identified requirements

Object Oriented Design 2

Definition [SEI]
Object Oriented Design (OOD) is concerned with developing an object-oriented model of a software system to implement the requirements identified during OOA.

The benefits are the same as those in OOA.

Process

Stages in OOD:

1) understand and define the context and modes of use of the system
2) design the system architecture
3) identify the principal objects in the system
4) develop design models
5) specify object interfaces

Note on OOD activities:

1) activities are not strictly linear but interleaved
2) back-tracking may be done a number of times due to refinement or availability of more information

Design Principles

Cohesion and coupling

1) module attributes

2) cohesion - how tightly bound are the internal elements of different modules

3) coupling – to what extent one module is connected with others

4) for reusability, modules should have high cohesion and low coupling

Summary

Summary 1

Object is any abstraction that models a single thing in a universe with some properties and behaviour.

A class is any uniquely identified abstraction of a set of logically related objects that share similar characteristics.

Classes may be related by the following types of relationships:

1) association
2) aggregation
3) composition

Summary 2

Object Orientation is characterized by three fundamental principles:

1) encapsulation – combination of data and behaviour, information hiding, separation of an interface from implementation
2) inheritance – generalization and specialization of classes, forms of hierarchy
3) polymorphism – different implementations for the shared operation depending on the particulars of the involving object in the inheritance hierarchy.

Object Oriented Analysis is concerned with creating requirements specifications and analysis models of the application domain.

Object Oriented Design is concerned with implementing the requirements identified during OOA, in the solution domain.

Exercise

1) Consider any application. Describe it very briefly in text.
2) List five main objects for this domain.
3) Identify at least five classes in the application describing the types of objects mentioned in point 2.
4) Provide concrete examples of the association, aggregation and composition relationships in the domain.
5) Show how one of the identified classes can be specialized or generalized.
6) Explain how encapsulation and polymorphism can aid reusability.

UML Basics

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

UML Basics
Modelling Principles

UML Basics

1) Modelling Principles

2) UML Overview:
   a) Goals of UML
   b) Brief history of UML
   c) Language architecture

3) Building Blocks:
   a) Elements: Structural, Behavioural, Grouping and Annotation
   b) Relationships

3) Building Blocks:
   c) Diagrams:
      - Class
      - Object
      - Use case
      - Interaction: Sequence and Collaboration
      - Statechart
      - Activity
      - Component
      - Deployment

4) Views

What Is Modelling?

1) representation or simplification of reality

2) provides a blueprint of a system

3) includes elements with broad effects and omits those not relevant at a given level of abstraction

Why Modelling?

1) to better understand the system we are developing

2) to provide a model of the structure or behaviour of the system

3) to experiment by exploring multiple solutions

4) to furnish abstraction for managing complexity

5) to document the design decisions

6) to visualize the system “as-is” and ”to-be”

7) to provide a template for constructing a system

Why Modelling Graphically?

1) Graphics reveal data

2) 1 bitmap = 1 megaword [anonymous visual modeler]

   - Courtesy Cris Kobryn – Introduction to UML

Modelling Principles

1) the choice of models has a profound influence on how a problem is attacked and how the solution is shaped

2) every model may be expressed at different levels of abstraction (precision)

3) effective models are connected to reality

4) No single model is sufficient. Non trivial systems are best described with a set of independent but related models

UML Basics
UML Overview

UML Basics

1) Modelling Principles

2) UML Overview:
   a) Goals of UML
   b) Brief history of UML
   c) Language architecture

3) Building Blocks:
   a) Elements: Structural, Behavioural, Grouping and Annotation
   b) Relationships

3) Building Blocks:
   c) Diagrams:
      - Class
      - Object
      - Use case
      - Interaction: Sequence and Collaboration
      - Statechart
      - Activity
      - Component
      - Deployment

4) Views

What Is The UML?

1) UML is a language for visualizing, specifying, constructing and documenting artifacts of software intensive systems.

2) Examples of artifacts: requirements, architecture, design, source code, test cases, prototypes, etc.

3) UML is suitable for modelling various kinds of systems: enterprise information systems, distributed web-based applications, real-time embedded system, etc.

UML – Specification Language

1) provides views for development and deployment

2) UML is process independent

3) recommended for use with processes that are:
    a) use-case driven
    b) architecture-centric
    c) iterative
    d) incremental

Goals of UML

1) provide modelers with a ready to use, expressive and visual modelling language to develop and exchange meaningful models

2) provide extensibility and specialization mechanisms to extend core concepts

3) support specifications that are independent of particular programming languages and development processes

4) provide a formal basis for understanding the specification language

5) encourage/growth of the object tools market

6) supports higher level of development with concepts such as components frameworks or patterns

Brief History of UML

1) started as a unification of the Booch method and the Rumbaugh method - *Unified Method v. 0.8* (1995), and in 1996 Jacobson joined them to produce UML 0.9

2) UML Partners worked with the Amigos to propose UML as a standard modelling language to OMG in 1996.

3) in 1997, the UML partners tendered their initial proposal (UML 1.0) to OMG, and 9 months later they submitted the final proposal (UML 1.1)

4) minor revision is UML 1.4 adopted in May 2001, and most recent revision is UML 1.5 published in March 2003.

5) awaiting UML 2.0 official release.

UML Language Architecture 1

UML language architecture was provided by OMG to align UML with other OMG's technologies.

UML architecture follows the meta-modelling architecture of OMG's Meta-Object Facility (MOF).

Four layers:

1) meta-metamodel layer
2) metamodel layer
3) model
4) user objects

UML Language Architecture 2

Layer	Description	Example
Meta-metamodel	The infrastructure for a metamodelling architecture. Defines the language for specifying metamodels.	MetaClass, MetaAttribute, MetaOperation
Metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	Class, Attribute, Operations, Component
Model	An instance of a metamodel. Defines a language to define an information domain.	Agency, AgencyCode, numberUnits, staffStrength
User object (or user data)	An instance of a model. Defines a specific information model.	<CPSP, 5, 100>

UML Basics
Building Blocks

UML Basics

1) Modelling Principles

2) UML Overview:
   a) Goals of UML
   b) Brief history of UML
   c) Language architecture

3) Building Blocks:
   a) Elements: Structural, Behavioural, Grouping and Annotation
   b) Relationships

3) Building Blocks:
   c) Diagrams:
      - Class
      - Object
      - Use case
      - Interaction: Sequence and Collaboration
      - Statechart
      - Activity
      - Component
      - Deployment

4) Views

UML Building Blocks

Three basic building blocks:

1) elements: main "citizens" of the model
2) relationships: relationships that tie elements together
3) diagrams: mechanisms to group interesting collections of elements and relationships

These building blocks will be used to represent large and small complex structures.

Elements

Four basic types of elements:

1) structural
2) behavioural
3) grouping
4) annotation

They will be used to specify well-formed models.

Structural Elements

1) static part of the model to represent conceptual or physical elements

2) “nouns” of the model

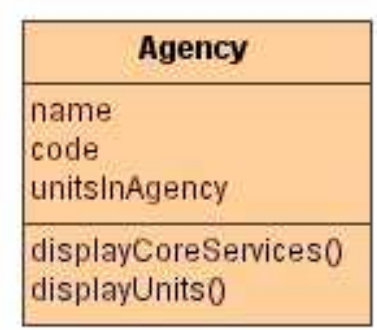
3) seven kinds of structural elements:
    a) class
    b) interface
    c) collaboration
    d) active class
    e) use case
    f) component
    g) node

Class 1

1) description of a set of objects that share the same attributes, operations, relationships and semantics

2) implements one or more interfaces

3) graphically rendered as a rectangle usually including a name, attributes and operations



4) can be also used to represent actors, signals and utilities

Interface

1) collection of operations that specifies a service of a class

1) describes the externally visible behaviour (partial or complete) of a class

2) defines a set of operation signatures but not their implementations

3) rendered as a circle with a name.

Collaboration

1) defines an interaction between elements

2) several elements cooperating to deliver a behaviour rather than individual behaviour

3) includes structural and behavioural dimensions

4) represents implementations of patterns that make up a system

5) represented as a named ellipse drawn with a dashed line

Use Case

1) description of a sequence of actions that a system performs to deliver an observable result to a particular actor

2) used to structure the behavioural elements in a model

3) realized by collaboration

4) graphically rendered as an ellipse drawn with a solid line

Active Class

1) a class whose objects own one or more processes or threads and therefore can initiate an action

2) class whose objects have concurrent behaviour with other objects

3) graphically, an active class is rendered just like a class drawn with a thick line

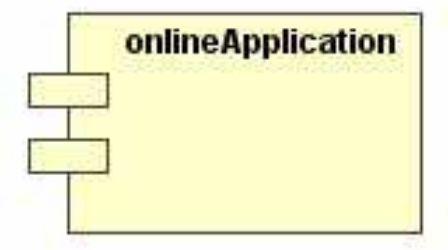
4) it also can be used to represent processes and threads

Component

1) physical replaceable part of a system that conforms to and provides the realization of a set of interfaces

2) represents deployment components such as COM+ or Java Beans components

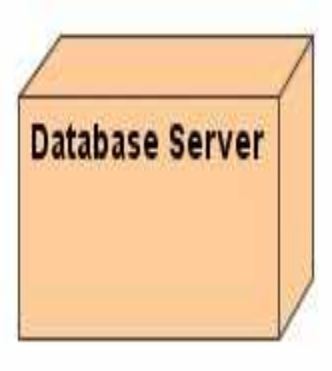
3) represents a physical packaging of logical elements such as classes, interfaces and collaborations



4) it also can be used to represent applications, files, libraries, pages and tables.

Node

1) a physical element that exists at run time
2) represents a computational resource with memory and processing capacity
3) a set of components may reside in a node
4) components may also migrate from one node to another
5) graphically modelled as a cube.



Behavioural Elements

1) represent behaviour over time and space

2) “verbs” of the model

3) two kinds of behavioural elements:

    a) interaction

    b) state machine

Interaction

1) a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose

2) specifies the behaviour of a set of objects

3) involves a number of other elements:
   - messages, action sequences (behaviour invoked by a message) and links (connection between objects)

4) graphically rendered as an arrow

State Machine

1) specifies a sequence of states an object or an interaction goes through during its lifetime and its response to external events

2) may specify the behaviour of an individual class or a collaboration of classes

3) includes a number of elements including states, transition, events and activities

4) presented as a rounded rectangle with a name and sub-states

5) interactions and state machines are associated with structural elements such as classes, collaborations, and objects

Grouping Elements – Packages

1) organizational part of UML
2) the primary mechanism for grouping and decomposition
3) purely conceptual and only available at development time
4) graphically represented as a tabbed folder

Annotation Elements – Notes

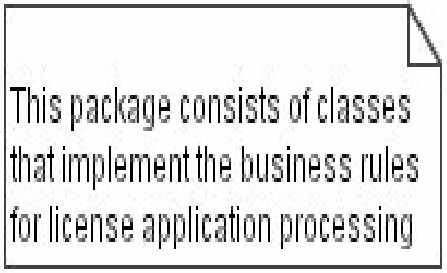
1) comments added to models for better explanation or illumination on specific elements

2) explanatory aspect of UML models

3) notes are used primarily for annotation e.g. for rendering constraints and comments attached to elements or collections of elements

4) presented as a rectangle with a dog-eared corner

5) may include both textual and graphical comments



Relationships

Four basic types of relationships:

1) dependency
2) associations
3) generalization
4) realization

Meanings are consistent with the basic OO relationship types described earlier

Relationship: Dependency

A semantic relationship between two elements in which a change to one element (independent element) may affect the meaning of the other (dependent element)

Given as a directed dashed line possibly with a label

dependent - - - - - - - - - - - - - - → independent

Relationship: Association

1) a structural relationship describing a set of links

2) links are connections between objects

3) aggregation is a special type of association depicting whole-part relationship

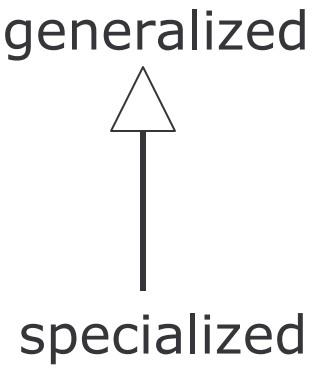
4) association is presented as a solid line, possibly directed, labelled and with adornments (multiplicity and role names)

Relationship: Generalization

1) a relationship in which objects of a specialized element (child) are substitutable for objects of a generalized element (parent)

2) child elements share the structure and behaviour of the parent

3) rendered graphically as a solid line with hollow arrowhead pointing to the parent

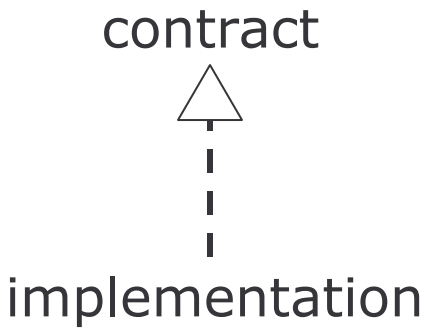


Relationship: Realization

1) a semantic relationship between elements, wherein one element specifies a contract and another guarantees to carry out this contract

2) relevant in two basic scenarios:
    a) interfaces versus realizing classes or components
    b) uses cases versus realizing collaborations

3) graphically depicted as a dashed arrow with hollow head a cross between dependency and generalization



Variations to Relationships

Variations of these four relationship types include:

1) refinement

2) trace

3) include

4) extend

Diagrams

1) a graph presentation of a set of elements and relationships where:
    a) nodes are elements
    b) edges are relationships

2) can visualize a system from various perspective thus a projection of a system

3) UML is characterized by nine major diagrams: a) class, b) object, c) use case, d) sequence, e) collaboration, f) statechart, g) activity, h) component and i) deployment.

Class and Object Diagrams

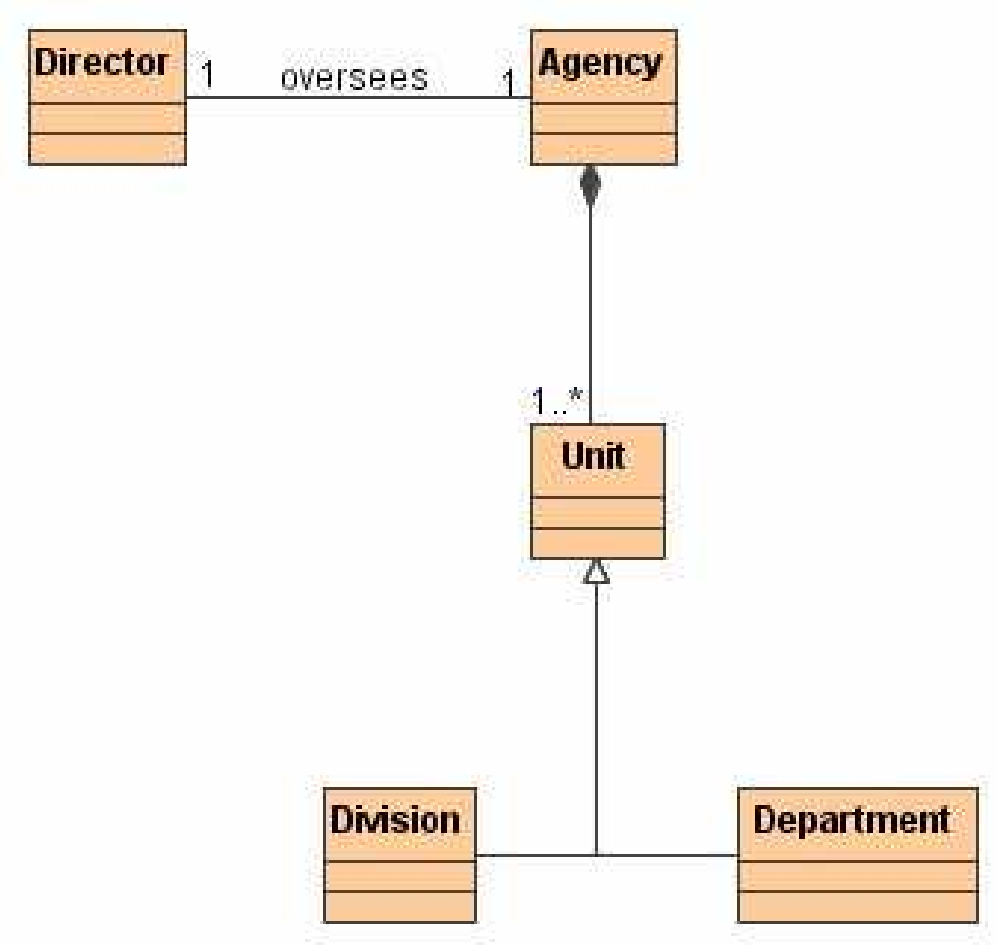
Class Diagrams:

1) show a set of classes, interfaces and collaborations, and their relationships
2) address static design view of a system

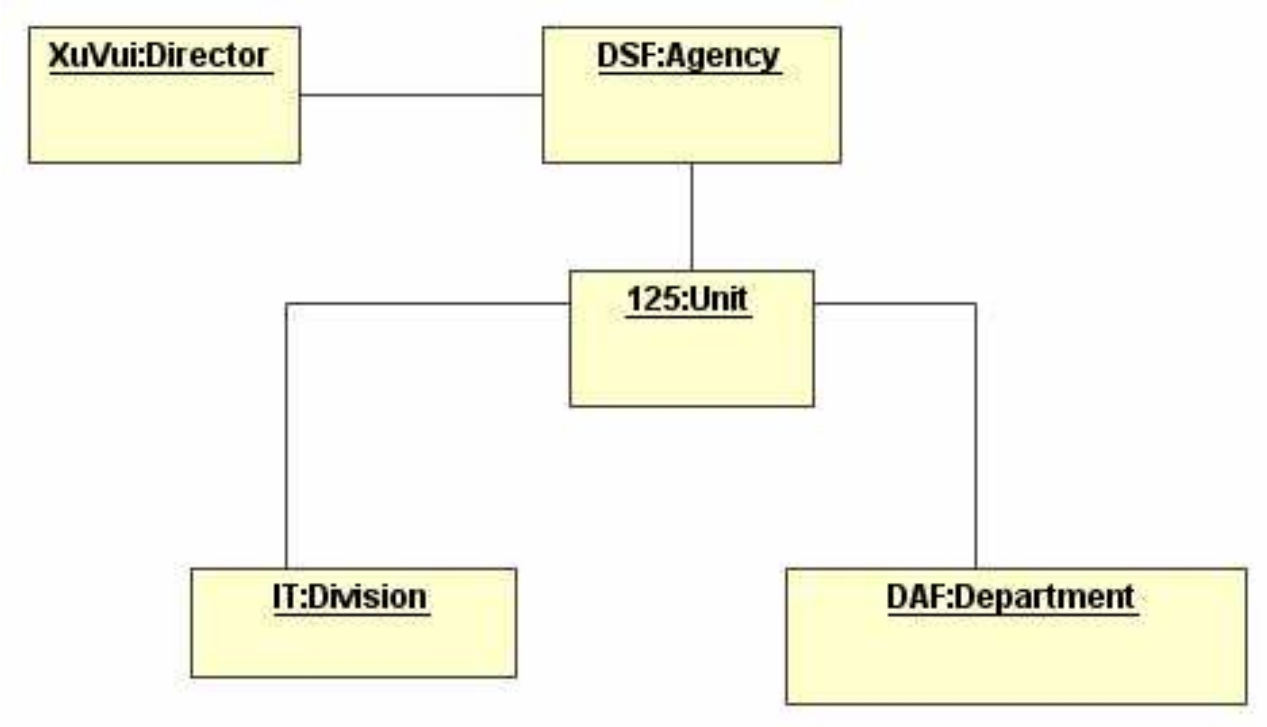
Object Diagrams:

1) show a set of objects and their relationships
2) static snapshots of element instances found in class diagrams

Example: Class Diagram



Example: Object Diagram



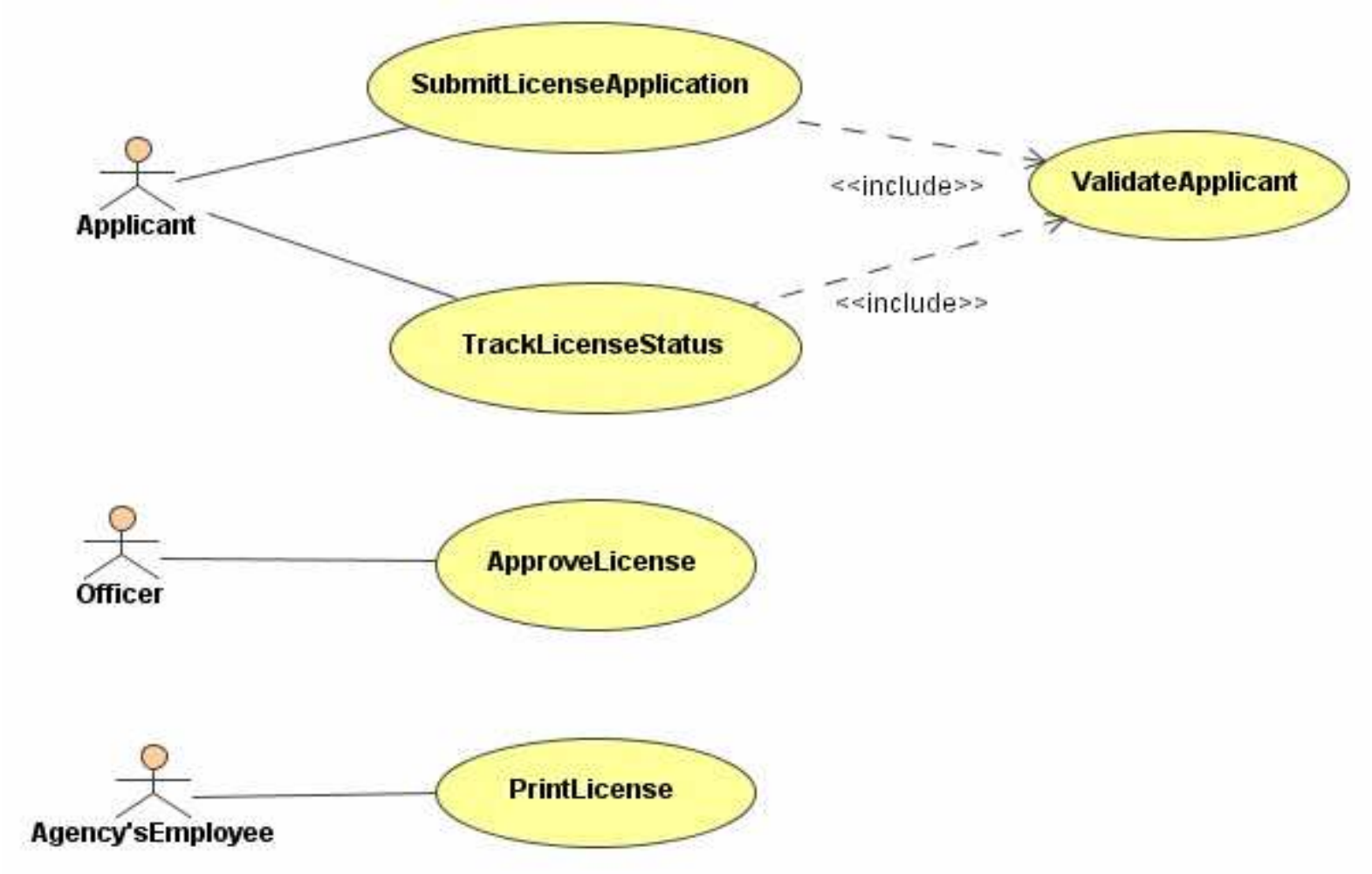
Use Case Diagrams

1) show a set of actors and use cases, and their relationships

2) addresses static use case view of the system

3) important for organizing and modelling the external behaviour of the system

Example: Use Case Diagram



Interaction Diagrams

Sequence Diagrams:

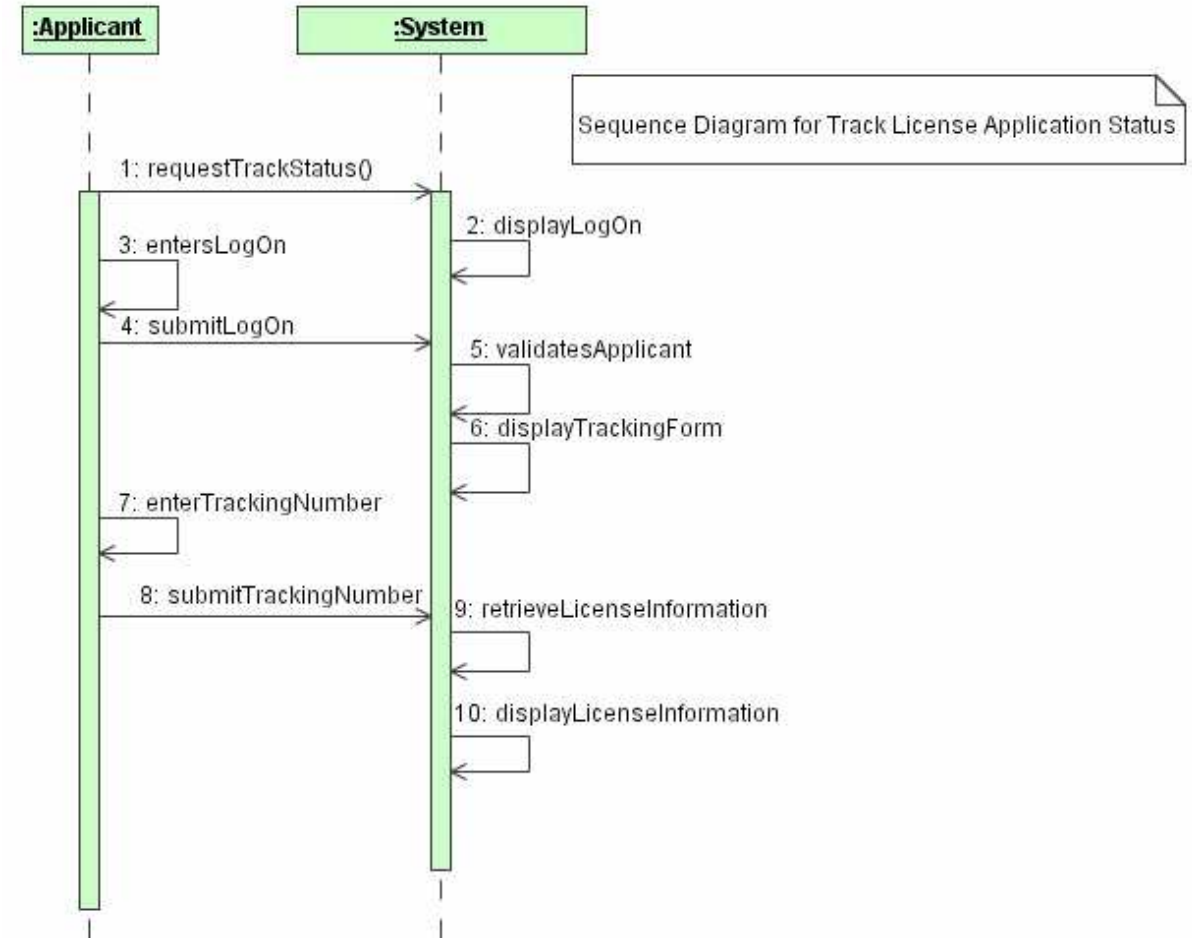
1) show interactions consisting of a set of objects and the messages sent and received by those objects
2) address the dynamic behaviour of a system with special emphasis on the chronological ordering of messages

Collaboration Diagrams:

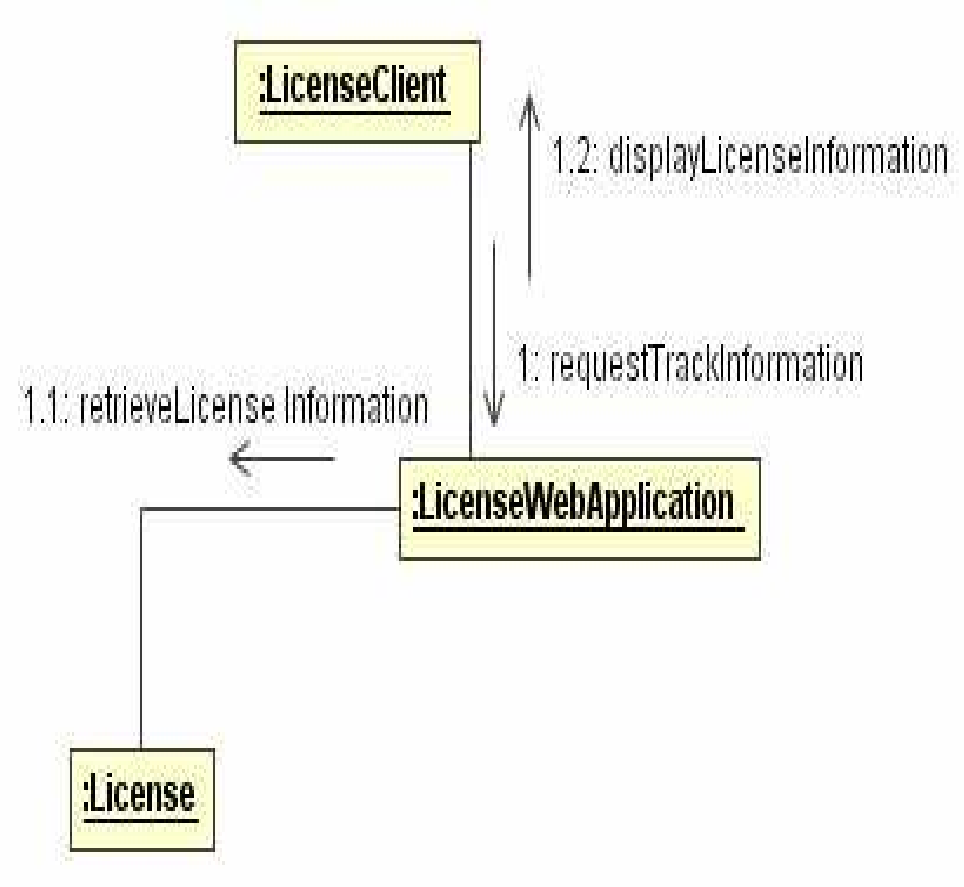
1) show the structural organization of objects that send and receive messages

Sequence and collaboration diagram are isomorphic i.e. one can be transformed into another

Example: Sequence Diagram



Example: Collaboration Diagram



Statechart Diagrams

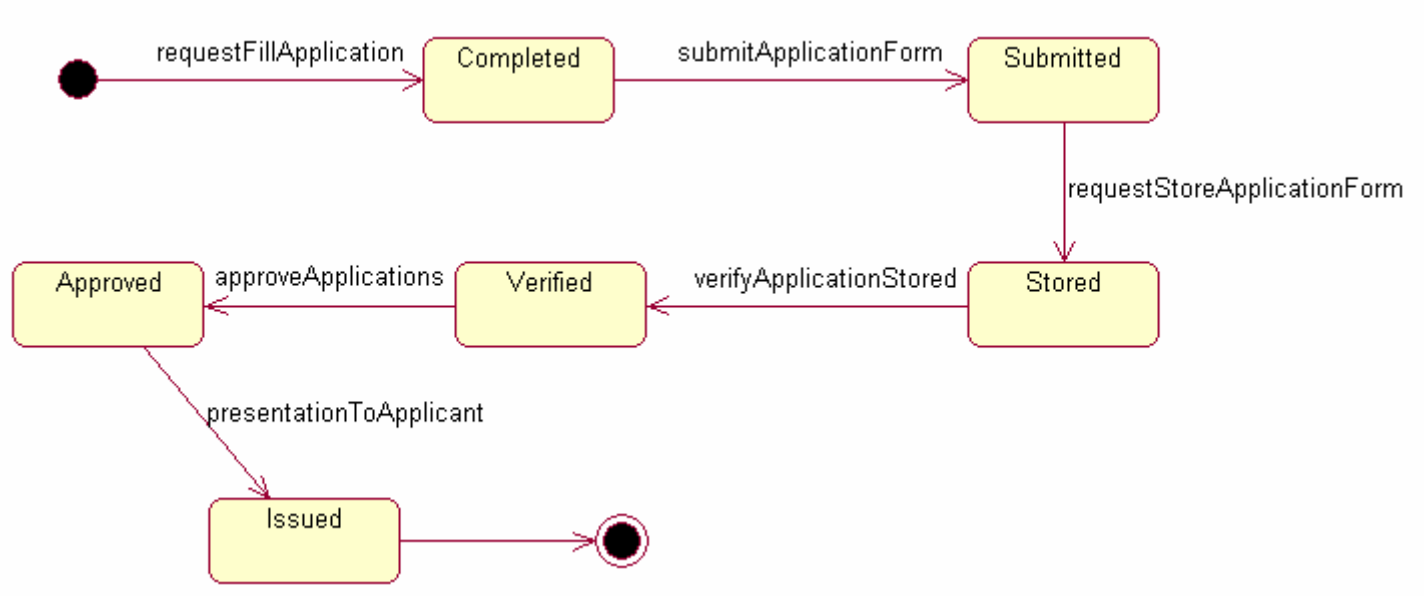
1) show a state machine consisting of states, transitions, events, and activities

2) address the dynamic view of a system

3) important in modelling the behaviour of an interface, class or collaboration

4) emphasise on event-driven ordering

Example: Statechart Diagram



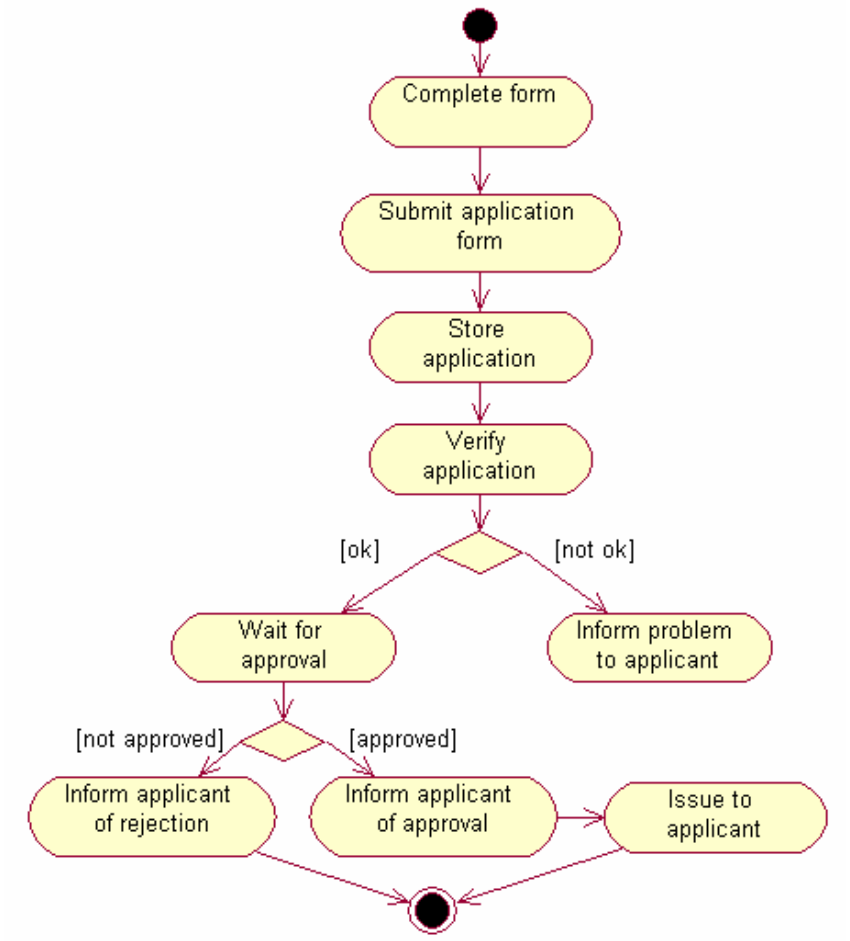
Activity Diagrams

1) a diagram showing control/data flows from one activity to another

2) addresses the dynamic view of a system, useful for modelling its functions

3) emphasises the flow of control among objects

Example: Activity Diagram



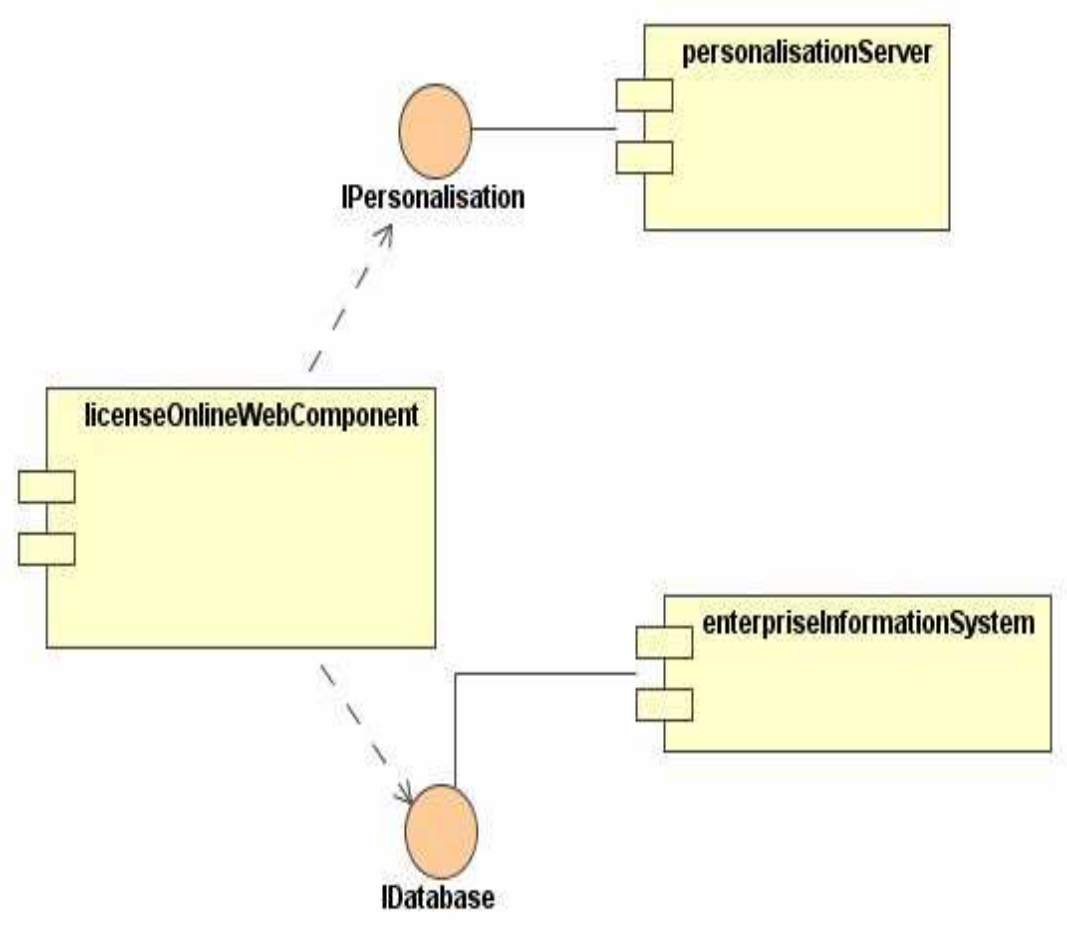
Component Diagrams

1) show the organization and dependencies amongst a set of components

2) address static implementation view of a system

3) a component typically maps to one or more classes, interfaces or collaborations

Example: Component Diagram



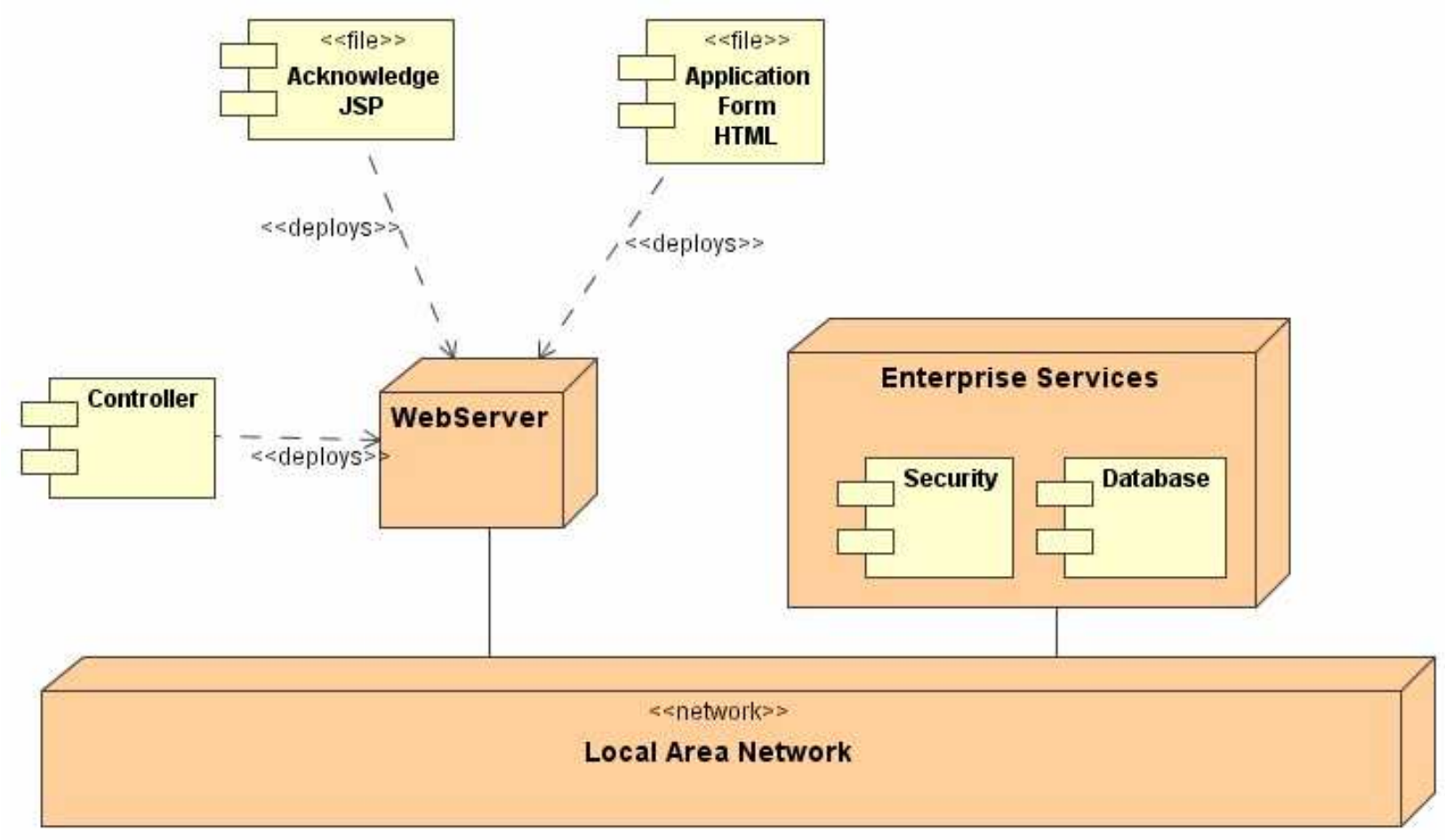
Deployment Diagrams

1) show configuration of run-time processing nodes and the components hosted on them

2) address the static deployment view of an architecture

3) related to component diagrams with nodes hosting one or more components

Example: Deployment Diagram



UML Basics
Views

UML Basics

1) Modelling Principles

2) UML Overview:
   a) Goals of UML
   b) Brief history of UML
   c) Language architecture

3) Building Blocks:
   a) Elements: Structural, Behavioural, Grouping and Annotation
   b) Relationships

3) Building Blocks:
   c) Diagrams:
      - Class
      - Object
      - Use case
      - Interaction: Sequence and Collaboration
      - Statechart
      - Activity
      - Component
      - Deployment

4) Views

Modelling Views 1

1) Use case view: describes the behaviour of the system as seen by its end users, analysts and testers. This view shapes the system architecture.

2) Design view: encompasses the classes, interfaces, interfaces, and collaborations that form the vocabulary of the problem and its solution.

3) Process view: encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view addresses the performance, scalability and throughput of the system.

Modelling Views 2

4) Implementation View: encompasses the components and files that are used to assemble and release the physical system. This view addresses the configuration management of the system's releases.

5) Deployment View: encompasses the nodes that form the system's hardware topology on which the system executes.

Modelling Monolithic Systems

1) Use case view: use case diagrams

2) Design views: class diagrams (structural modelling) and interaction diagrams (behavioural)

3) Process View: none

4) Implementation view: none

5) Deployment view: none

Modelling Distributed Systems

1) Use case view: use case diagrams and activity diagram (behavioural modelling)

2) Design views: class diagrams (structural modelling) interaction diagrams (behavioral modelling), statechart diagram (behavioural)

3) Process View: class diagram (structural modelling) and interaction diagrams (behavioural)

4) Implementation view: component diagrams

5) Deployment view: deployment diagrams

Summary 1

A model provides a blueprint of a system.

UML is a language for visualizing, specifying, constructing and documenting artifacts of software intensive systems.

UML supports five basic views of a system: user, static-structural, dynamic, implementation and environmental modelling views.

UML is process independent but recommended for use with processes that are: use case driven, architecture-centric, iterative and incremental.

Summary 2

There are three building blocks which characterize UML – elements, relationships and diagrams.

Categories of elements in UML include: structural, behavioural, grouping and annotation.

There are four basic types of relationships in UML: dependency, association, generalization and realization.

UML provides 9 diagrams for modelling: class, object, use case, sequence, collaboration, statechart, activity, component and deployment.

There are five different modelling views in UML: use case, design, process, implementation and deployment.

Exercise

1) Why do you think visual modelling in UML is desirable?

2) Briefly describe the purpose of each of the UML diagrams.

3) Which diagrams do you consider essential in your development? Give reasons for your opinion.

Case Study

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

Case Study

This case study will be used to show examples for the different models created during the development phases.

In each model we present a particular aspect of this case study.

e-Delivery of Licensing Services

What is a license?

1) a legal document or instrument that officially permits the holder to undertake some activities

2) required by citizens and businesses

Examples: vehicle, radio, driver, professional, building construction, business, import and export, …

Service Providers

Who provides?

1) government agencies, departments, offices, …
2) issued by one agency or in consultation with other agencies

Service Implementation

1) initiation through application, provided some basic requirements are satisfied

2) submission of relevant documents and information

3) evaluation and consultation with other agencies if necessary

4) decision making - issuance or denial

5) communication to applicant on decision

Motivation

1) effective and efficient coordination:
    a) several units within an agency involved
    b) a number of other agencies may be involved

2) reduction of paper work: several elements move around within and outside the agency to carry out the service

3) citizen centered delivery: citizens may have to face several agencies just to apply for a license

The Goal

Our target with this case study is:

1) to provide a vehicle for direct representation of the domain knowledge being acquired (UML models)

2) to show relevance and importance of UML to effective domain representation and software development

3) to demonstrate a team based approach to requirements gathering and agreement

The Means

To do:

1) analyze the licensing service to determine some basic requirements for its electronic delivery
2) provide Object Oriented (specifically UML) analysis and design models for the application domain

Strategy:

1) concentrate on high level requirements and models to tackle the licensing services in general
2) models will allow room for specialization by specific agencies

Requirements Modelling

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

Requirements Modelling:
Software Requirements

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Requirements Problems

Requirements capture is a critical factor for the success of any development project.

Between 40% - 60% of all defects found in software projects can be traced back to errors made while gathering requirements.

A survey of 8000 projects undertaken by 350 US companies revealed that one-third of the projects were never completed and one-half succeeded only partially i.e. with partial functionalities, major cost over-runs and significant delays.

Failed Projects

Why projects fail:

1) poor requirements
2) lack of user involvement
3) requirement incompleteness
4) changing requirements
5) unrealistic expectations
6) unclear objectives
7) lack of executive support

Requirements Definition

Definition

A requirement is

1) a function that a system must perform
2) a desired characteristic of a system
3) a statement about the proposed system that all stakeholders agree that must be true in order for the customer's problem to be adequately solved.

Requirements Process

Typically includes:

1) elicitation of requirements
2) modelling and analysis of requirements
3) specification of requirements
4) validation of requirements
5) requirements management

The process is not linear!

Functional and Non-Functional

Functional requirements:

1) describe an interaction between the system and its environment
2) describe how a system should behave under certain stimuli

Non-functional requirements:

1) describe the restrictions on the system that limit the choices for its construction as a solution to a given problem

Types of Requirements

1) functional
2) interface
3) data
4) human engineering
5) qualification
6) operational
7) design constraints
8) safety
9) security
10) …

Requirements Specification

Requirements must be expressed formally.

Various standards are available:

1) IEEE P1233/D3 Guide
2) IEEE Std. 1233 Guide
3) IEEE std. 830-1998
4) ISO/IEC 12119-1994
5) IEEE std 1362-1998 (ConOps)

Requirement specification is expected to be:

1) correct
2) consistent
3) feasible
4) verifiable
5) complete
6) traceable

Example: Template

Requirement #: Requirement Type: Event/use case #:

Description:

Rationale:

Source:
Fit Criteria:

Customer Satisfaction: Customer Disatisfaction:
Dependencies: Conflicts:
Supporting Materials:
History:

Volere

Example: Functional

Ref. Id	Description	
F1	Allow online application for license	
	F1.1.	Allow applicant download application form
	F1.2	Applicant will be able to complete license application form online
	F1.3	Applicant will be able to upload completed application form
	F1.4	System will send an acknowledgement of the receipt of application to the applicant
	F1.5	Applicant will be able to upload documents necessary for application
F2	Applicant will be able to track progress of its application	
	F2.1	Applicant will be able to determine the status of its application online
	F2.2	Applicant will be able to make enquiries on the status of its application online
	F2.3	System will inform applicant of changes in status of its application automatically

Example: Non-Functional

Ref. Id	Description	
NF1	Applicant will be able to use the system interface with no intuition	
	NF1.1	System will provide tips on all controls (buttons etc.) available on forms
	NF1.2	Applicant will have access to context sensitive help on interface elements
	NF1.3	Explicit information on type of data and range of values acceptable for data entry fields will be available for all data entry boxes.
NF2	Applicant will be able to complete data entry forms over multiple sessions	
	NF2.1	Applicant will be able to log in and initiate sessions
	NF2.2	Applicant will be able to save a session and resume later
F3	The average burden time for the applicant in completing license application online form will be 15 minutes	

Requirements Modelling:
Use Case Modelling

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Use Cases

1) describe or capture functional requirements

2) represent the desired behaviour of the system

3) identify users (“actors”) of the system and the associated processes

4) are the basic building blocks of use case diagrams and use case models

5) tie requirements phase to other development phases

Definition

A use case:

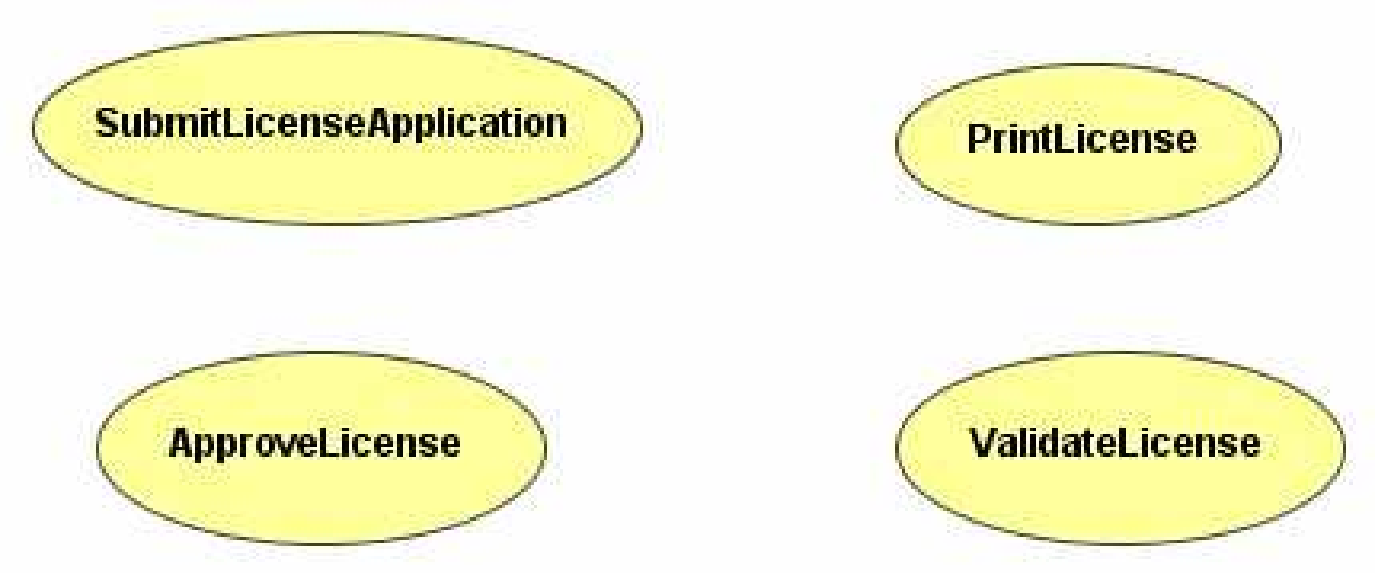
1) is a collection of task-related activities describing a discrete chunk of a system
2) is a description of a set of actions sequences that a system performs to obtain an observable result to an actor
3) describes the system from an external usage viewpoint

Key attributes: description, action sequence, includes variants, produces observable results

Use cases do not describe:

1) user interfaces
2) performance goals
3) non-functional requirements

Example: Use Cases



Use Case Relationships 1

Use cases are organized by relationships:

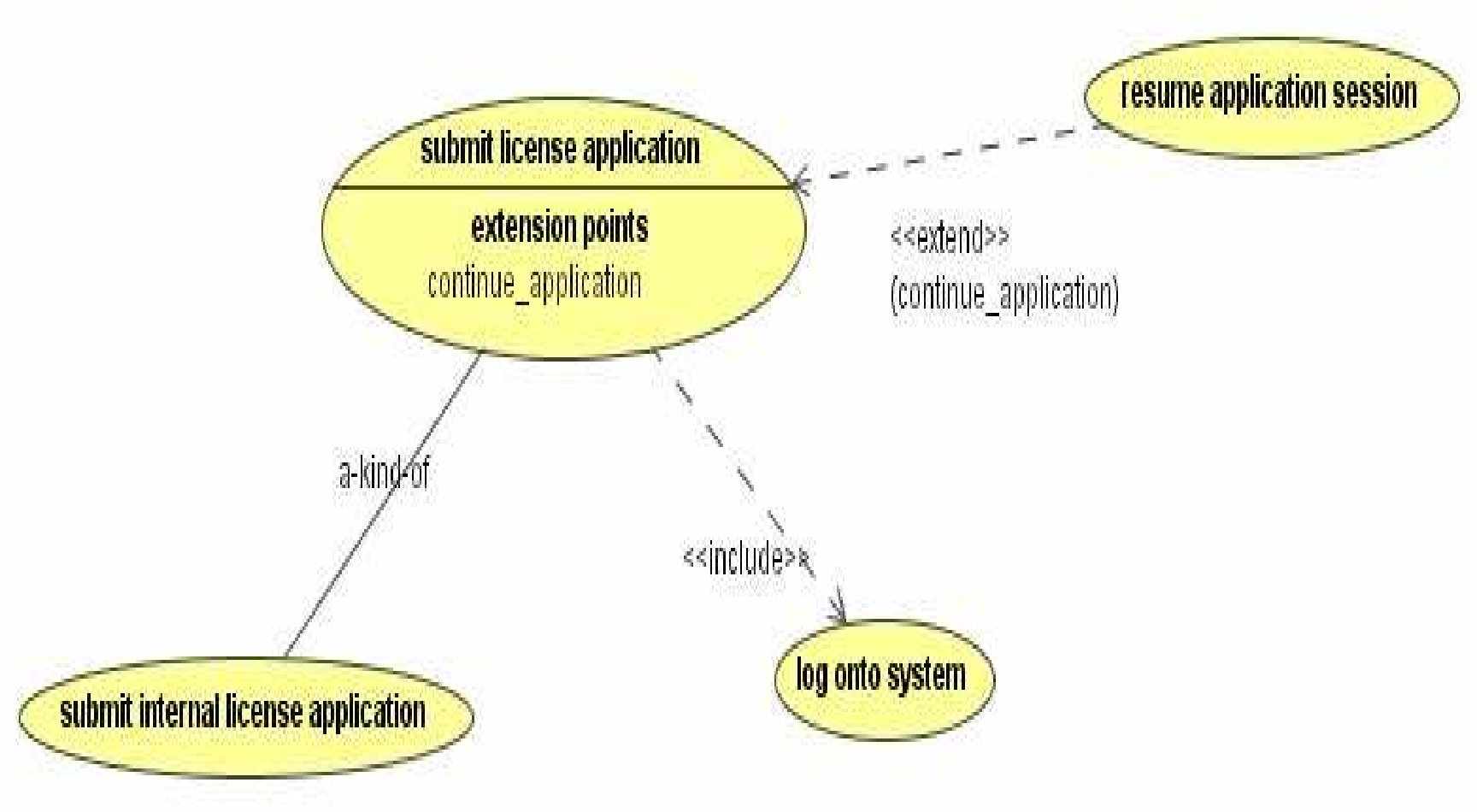
1) generalization
   a) the same meaning as before
   b) more specialized use cases are related to more general use cases

2) include
   a) the base use case explicitly incorporates the behaviour of another use case at a location specified in the base
   b) the included relationship never stands alone, but is only instantiated as part of some large base of the use cases that include it
   c) rendered as the "include" stereotype

Use Case Relationships 2

3) extend
   a) the base use case implicitly incorporates the behaviour of another use case at a location specified by the extending use case (extension point)
   b) base use case may stand alone and usually executes without regards to extension points
   c) depending on system behaviour, the extension use case will be executed or not
   d) rendered as the "extend" stereotype

Example: Use Case Modelling



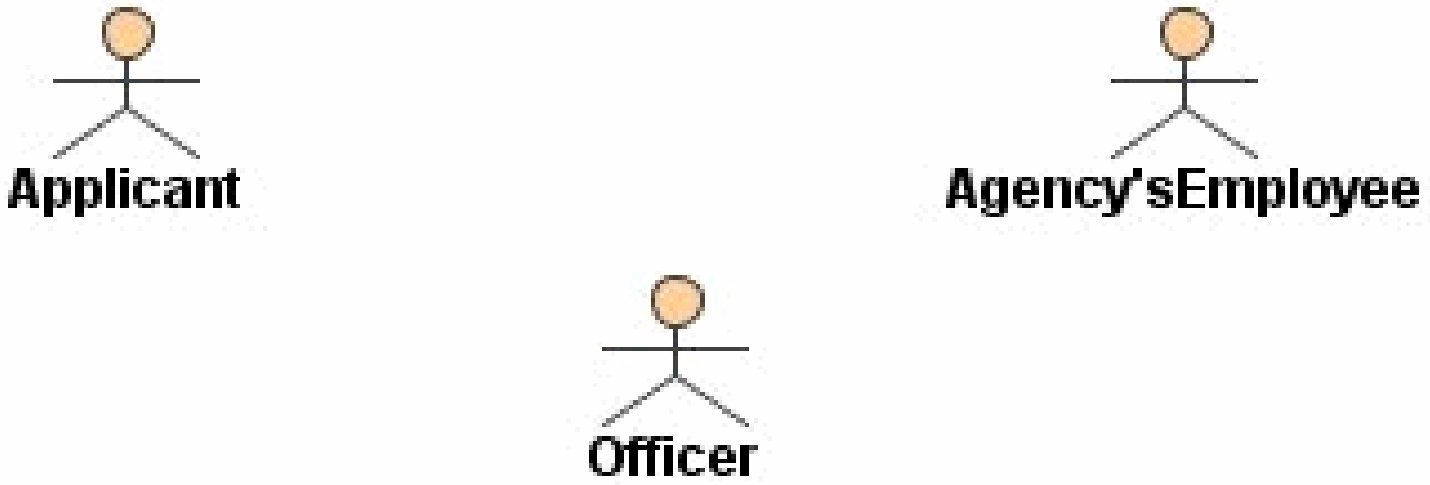
Actors

Definition
Actor is anyone or anything that interacts with the system causing it to respond to business events.

Actor:

1) is something or somebody that stimulates the system to react or respond to its request
2) is something we do not have control over
3) represents a coherent set of roles that the entities external to the system can play
4) represents any type of a system's user

Example: Actors



Notes about Actors

1) actors stimulate the system with input events or receive something from the system

2) actors communicate with the system by sending messages to it and receiving messages from the system as they perform the use cases

3) actors model anything that needs to interact with the system to exchange information – human users, computer systems, etc.

4) a physical user may act as one or several actors as it interacts with the system, several individual users may act as different instances of one and the same actor

Types of Actors

Initiator versus participant:

- When there is more than one actor in a use case, the one that generates the stimulus is called the initiator and the others are participants

Primary versus secondary:

- The actor that directly interacts with the system is called the primary actor, others are called secondary actors.

Glossary 1

1) a set of terms that are defined and understood to form the basis for communication

2) is a dictionary for modelling

3) its purpose is to clarify the meaning of terms or to have the shared understanding of the terms amongst team members

4) is created during requirements definition, use case identification, conceptual modelling

5) is maintained throughout development

Glossary 2

It is usually the central place for:

1) definitions of key concepts
2) clarification of ambiguous terms and concepts
3) explanations of jargons
4) description of business events
5) description of software actions

There is no specific format for glossaries:

1) reference identification for terms
2) definitions
3) categories
4) cross references

Example: Glossary

Ref. Id	Name	Description	Category	Cross reference
U001	Submit License Application	Describes a functional requirement which specifies that the system must allow user submit application for license.	Use Case	CB001
A001	Applicant	An entity (citizen, business, visitor, agency etc.) who is applying for a license. The applicant may be an internal or an external applicant.	Actor	A002, A003
A002	External Applicant	An entity (citizen, business, visitor, agency etc.) who is applying for a license. This entity excludes the employees of the licensing.	Actor	A001
A003	Internal Applicant	A staff member of the agency who requires the licensing service.	Actor	A001
C1001	Agency	An administrative unit of government that provides a set of differentiated services to citizens businesses and other government agencies.	Concept	
CB001	Capture Online Submission	A structural and behavioural entity which implements the “submit application license” use case.	Collaboration	U001

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Use Case Diagrams

1) is a diagram that shows a set of use cases and actors, and their relationships
2) is central to modelling the behaviour of the system
3) is used to visualize the behaviour of a system, so that users can comprehend how to use that system, and developers can understand how to implement it
4) puts everything together
5) commonly contains:
    a) use cases
    b) actors
    c) dependency, generalization and association relationships

Identifying Use Cases

Use cases describe:

1) the functions that the user will want from the system to accomplish

2) the operations that create, read, update, delete information

3) how actors are notified of the changes to the internal state of the system and how they notify the system about external events

Identifying Actors

To determine who are the actors, we try to answer the following questions:

1) who uses the system?

2) who gets information from the system?

3) who provides information to the system?

4) who installs, starts up or maintains the system?

Naming Use Cases

Use concrete verb-noun phrases:

1) a weak verb may indicate uncertainty, a strong verb may clearly identify the action taken
   a) strong verbs: create, merge, calculate, migrate, activate
   b) weak verbs: make, report, use, copy, organize, record…

2) a weak noun may refer to several objects, a strong noun clearly identifies only one object
   a) strong nouns: property, payment, transcript …
   b) weak nouns: data, paper, report, system

Naming Actors

1) group individuals according to how they use the system; identify the roles they adopt when using the system

2) each role is a potential actor

3) name each role and define its distinguishing characteristics

4) not equate job titles with roles; roles cut across job titles

5) use common names for an existing system; avoid inventing new names

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Use Case Definition Template

Fields	Description
Use Case Name	Name of the use case
Actors	Role names of people or external entities initiating and participating in the use case
Purpose	The intention of the use case
Overview	A brief description of the usage of the process
Precondition	A condition that must hold before a use case can begin
Variation	Different ways to accomplish use case actions
Exceptions	What might go wrong during the execution of the use case
Policies	Specific rules that must be enforced by the use case
Post-conditions	Condition that must prevail after executing the use case
Priority	How important is the use case?
Frequency	How often is the use case performed?
Cross reference	Relate use cases and functional requirements

Case Study: Use Cases

Use Case	Overview
Submit License Application	A citizen, business or employee of a licensing agency visits the website of the licensing agency providing the online licensing service. The applicant logs-on into the site. The applicant then completes the on-line application form. This may be done over a number of sessions. Finally the applicant submits the data entered through the form into the agency license database. The applicant receives an acknowledgement of the receipt of application with an application number for tracking. The applicant may choose to download the application form and complete it off-line. The applicant later uploads the completed form.
Track license application status	The applicant who has already successfully submitted a license application logs onto the agency site. On successful entry the applicant enters the license application assigned to it. The status of its application is then displayed.
Review license application documents	The processing officer retrieves all relevant records associated with a particular application. The officer notifies all entities within the agency whose inputs are needed to act on the case. The officer also sends the necessary documents to all supporting agencies.

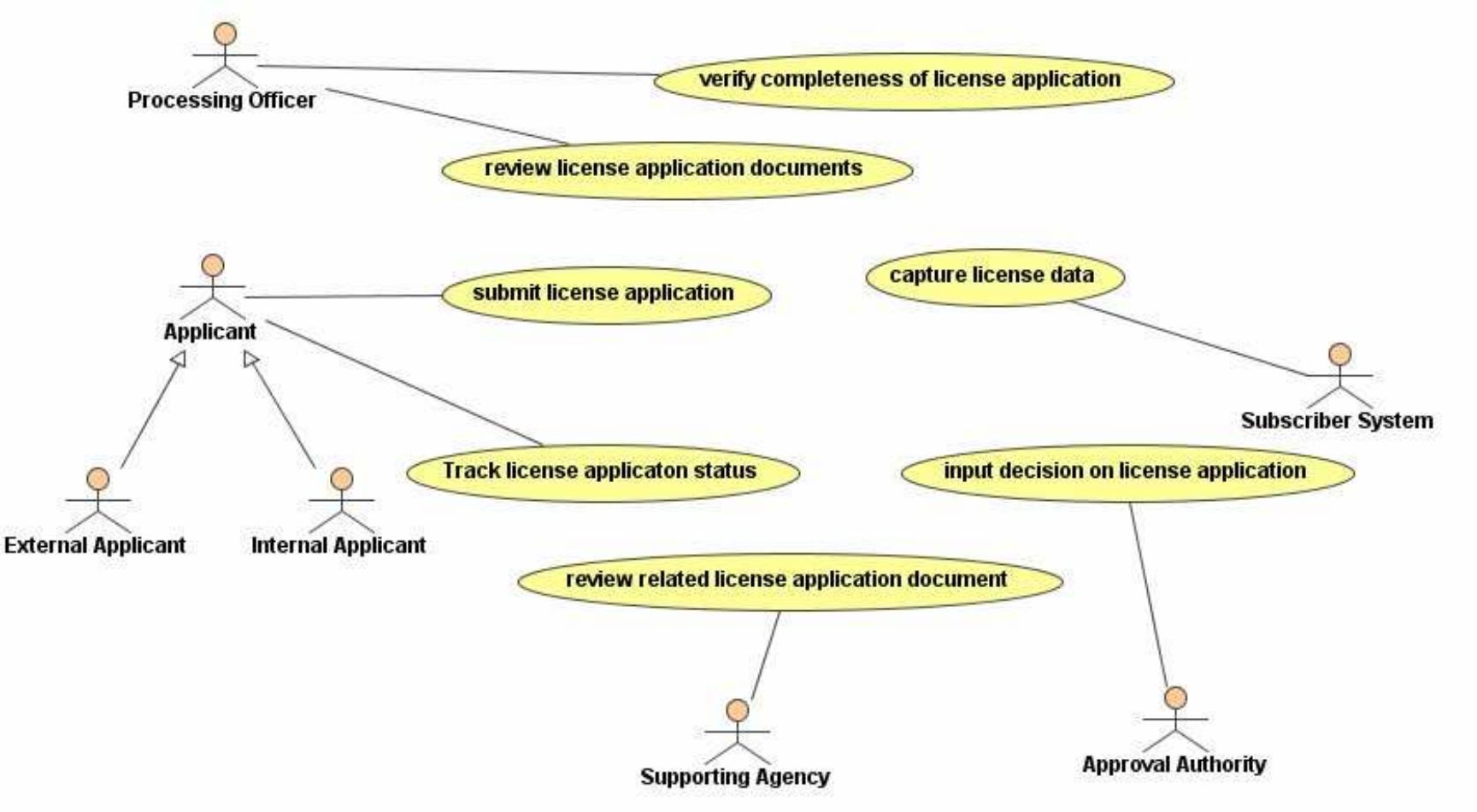
Case Study – Possible Actors

Actor	Description
External Applicant	An entity (citizen, business, visitor, agency etc.) who is applying for a license. This entity excludes the employees of the licensing agency.
Internal Applicant	A staff member of the agency who requires the licensing service
Processing Personnel	Any agency staff responsible for processing the application
Approving Authority	The entity (e.g. the agency director, sub-director or board) within the agency responsible for the final approval of the license application.
Supporting Agencies	Other agencies that provide technical inputs or opinions to the licensing agency on the license application
Licensing Agency	The agency that receives the license application and issues the license
Website visitor	Any entity (citizens, business, system, etc.) visiting the licensing website
Subscriber System	External systems that request licensing information from the license database in the licensing agency
System Administrator	IT personnel in the licensing agency responsible for administering the licensing system

Case Study: Description

Fields	Description
Use Case Name	Submit License Application
Actors	Applicant (External Applicant or Internal Applicant)
Purpose	Capture the online license application
Overview	A citizen, business or employee of a licensing agency visits the website of the licensing agency providing the online licensing service. The applicant logs-onto the site. The applicant then completes the online application form. This maybe done over a number of sessions. Finally the applicant submits the data entered through the form into the agency license database. The applicant receives an acknowledgement of the receipt of application with an application number for tracking.
Precondition	The Applicant fulfils the basic requirements for applying for a license
Variation	External applicant may submit its application at the agency location or other designated venues.
Exceptions	Applicant submits incomplete information
Policies	Acknowledgement of the receipt of application must be sent to the applicant within the time limit specified in the performance pledge relating to this service
Post-conditions	An acknowledgement must be sent to the applicant upon successful submission of its application and the application must be available in the license database.
Priority	5 (the highest). In scale 1 to 5.
Frequency	50 per/day
Cross reference	F1 (F1.1 – F1.5), NF2 (NF2.1 and NF2.2), NF3

Case Study: Use Case Diagram



Requirements Modelling:
Conceptual Modelling

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Concept Definition

What is a concept ?

- an idea, thing or object

A concept can be:

1) represented symbolically
2) defined or described
3) exemplified

What is an instance?

- each concrete application of the concept

Example: Concepts

1) agency

2) license

3) officer

4) applicant

5) internal applicant

6) external applicant

Concept Identification

Concepts can be:

1) physical or tangible objects
2) places
3) documents, specifications, design or descriptions
4) roles of people
5) container of other things
6) organizations
7) processes
8) catalogs
9) …

Concepts may be identified from requirements definitions and use cases.

Conceptual Model and Classes

Conceptual model:

1) captures the concepts in a domain in an abstract way
2) important part of OO requirements analysis

Classes:

1) equivalent to concepts in UML
2) an abstraction of a set of objects
3) objects are concrete entities existing in space and time (persistence)

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Class Diagrams

1) the most widely used diagram of UML

2) models the static design view of a system

3) also useful in modelling business objects

4) used for specifying the structure (attributes and operations), interfaces and relationships between classes that form the foundation of system architecture

5) primary diagram for generating codes from UML models

Class

1) a description of a set of objects that share the same attributes, operations, relationships and semantics

2) a software unit that implements one or more interfaces

3) graphically rendered as a rectangle usually including a name, attributes and operations

Example: Classes

Class Notation

Basic notation: a solid-outline rectangle with three compartments separated by horizontal lines.

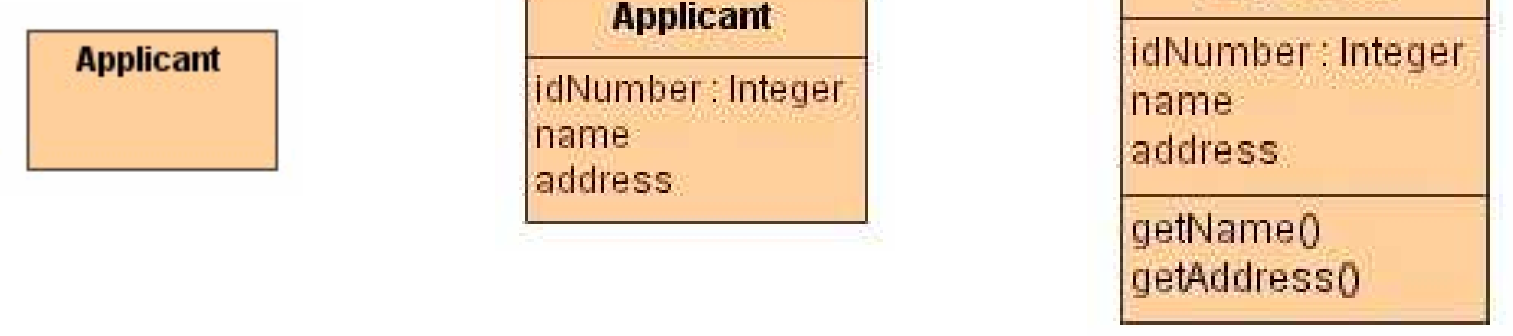
Three compartments:

1) the top compartment holds the class name and other general properties of the class
2) the middle compartment holds a list of attributes
3) the bottom compartment holds a list of operations

Alternative styles for presentation:

1) suppress the attributes compartment
2) suppress the operation compartment

Example: Class Notation



Stereotypes for Classes 1

The list of attributes or operations in a class may be organized using stereotypes.

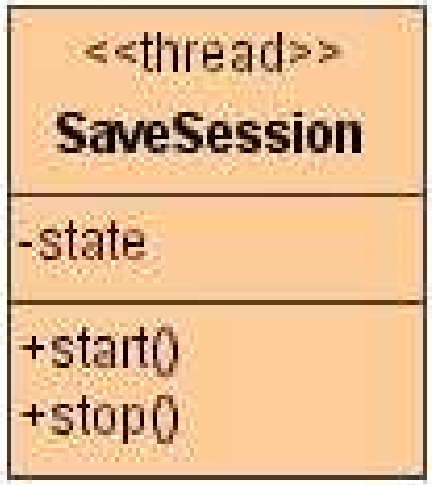
Stereotype:

1) a new class of a metamodel element introduced at the modelling time
2) represents a sub-class of an existing metamodel element with the same form (attributes and operations) but with different intent
3) notation – the name of a metamodel element within the matched guillemets e.g. <<changeintent>>

Stereotypes for Classes 2

UML provides a number of stereotypes of classes and data types such as:

1) thread
2) event
3) entity
4) process
5) utility
6) metaclass
7) powerclass
8) enumeration
9) …



Example: Stereotypes

Stereotype	Description
Thread	An active class which specifies a lightweight flow that can execute concurrently with other threads within the same processes.
Process	An active class which specifies a heavyweight flow that can execute concurrently with other processes.
Control	A class which owns almost no information about itself. It represents a behaviour rather than resources and directs the behaviour of other objects almost having no behaviour of its own.
Entity	A class which represents a resource in the real world. It describes its features and their current condition (their state) and preserves its own integrity regardless of where and when it is used.
Utility	A class whose attributes and operations are all class scoped. That is a class which no instance.
Metaclass	A classifier whose objects are all classes.
Powerclass	A classifier whose objects are the children of a given parent.
Enumeration	A user defined data type that defines a set of values that do not change.

Modelling Scope 1

Different scopes can be specified for class features (attributes and operations):

1) a feature appears in each instance of the class (or a classifier generally)

2) there is just a single instance of the feature for all instances of a class (classifier)

Modelling Scope 2

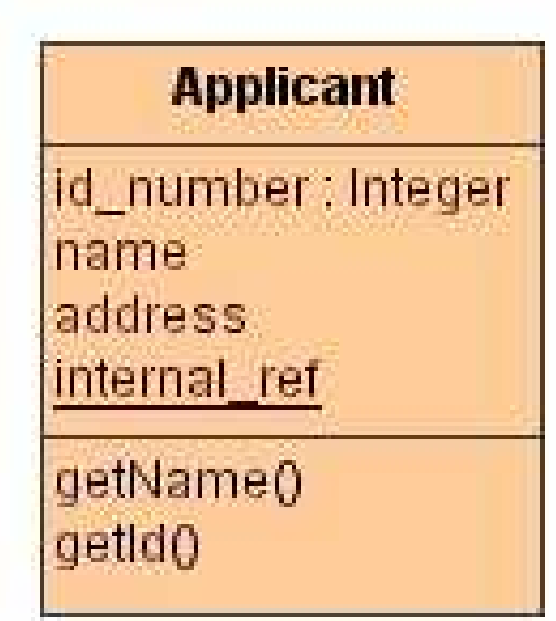
Instance scope:

- each instance holds its own value

Class scope:

- a single value for all instances of the class

Underlining the feature's name indicates the classifier scope.



Types of Classes

Abstract:

- cannot have direct instances
- the name is written in italics

Root:

- cannot be a sub-class

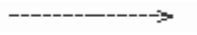
Leaf:

- cannot be a super-class

Class Relationships 1

Construct	Description	Syntax
Association	A relationship between two or more classifiers that involves connection among instances.	
Aggregation	A special form of association that specifies a whole-part relationship between the aggregate(whole) and the component part.	

Class Relationships 2

Construct	Description	Syntax
Generalization	A taxonomic relationship between a more general and a more specific element.	
Dependency	A relationship between two modelling elements, in which a change to one modelling element (the independent element) will affect the other modelling element (the dependent element).	

Multiplicities for Classes

1) shows how many objects of one class can be associated with one object of another class

2) example: a citizen can apply for one or more licenses, and a license is required by one citizen

Multiplicities for Attributes

1) can specify how may instances of an attribute can be associated with one instance of the class

2) example: an applicant will have 2 addresses

Multiplicities Syntax

Value	Description
0..0	Zero
0..1	Zero or one
0..*	Zero or more
1..1	One
1..*	One or more
*	Unlimited number
<literal>	Exact number (Example: 4)
<literal>..*	Exact number or more (Example: 4..* indicating 4 or more)
<literal>..<literal>	Specified range (Example: 4..13)
<literal>..<literal>, <literal>	Specified range or exact number (Example: 4..13,31 indicating 4 through 13 and 31)
<literal>..<literal>,<literal>..<literal>	Multiple specified ranges (Example: 4..13, 31-41)

Roles

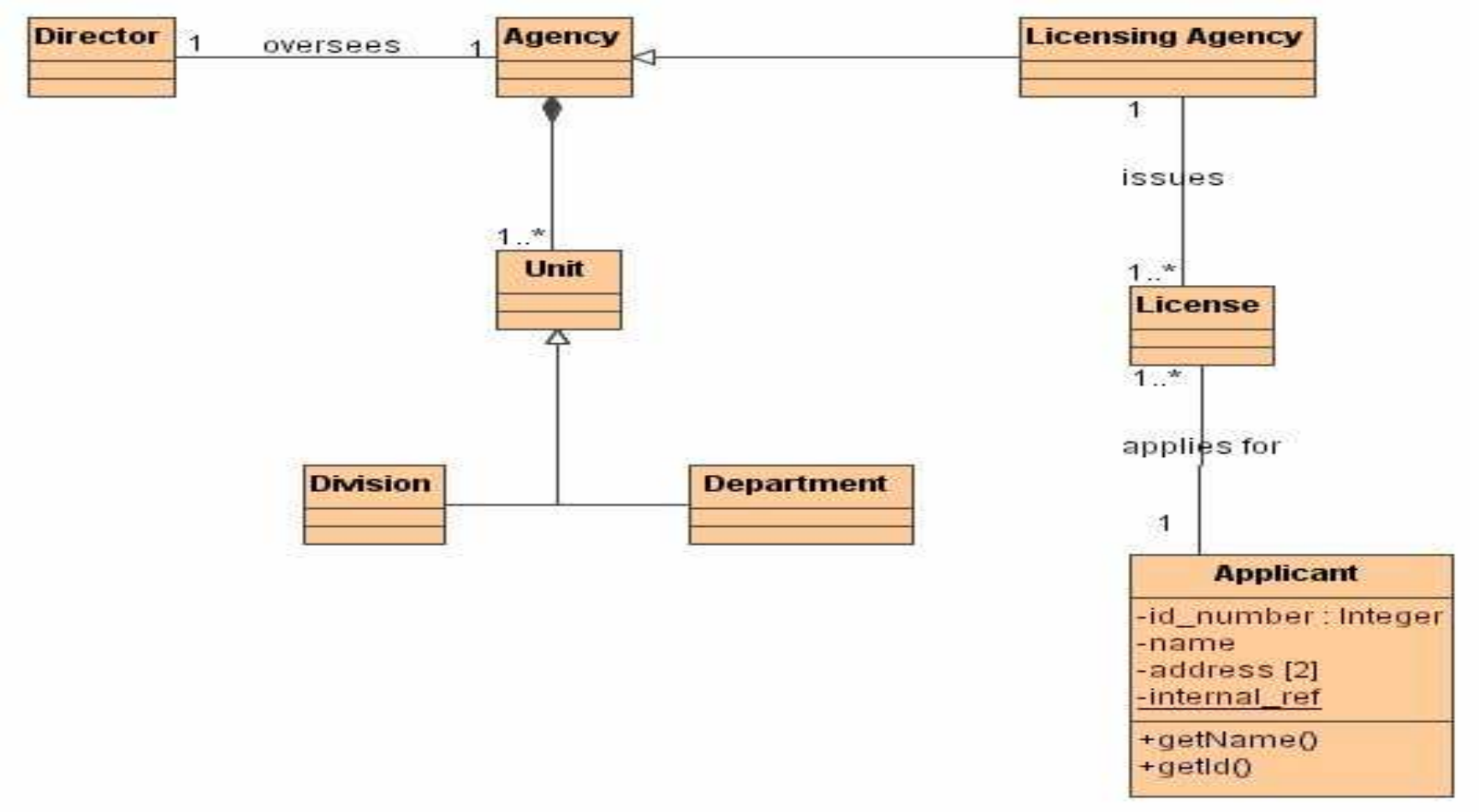
A role names a behaviour of an entity participating in a particular relationship.

Notation:

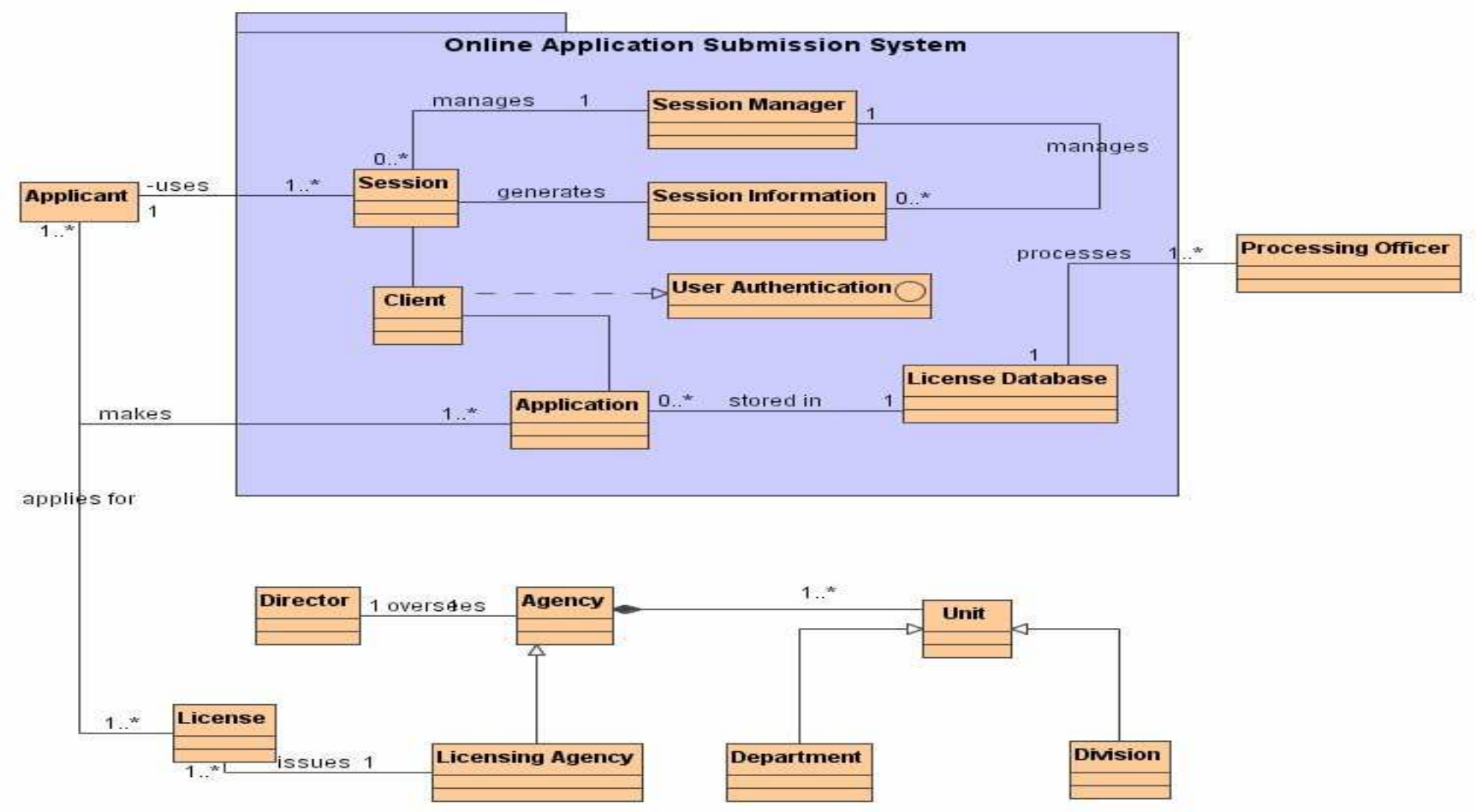
- add the name of the role at the end of the association line next to the class icon

Example: a citizen may play different roles depending on the context, e.g. can owns a car, or be an employee, or …

Example: Class Diagrams 1



Example: Class Diagrams 2



Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Object Diagram 1

1) models the instances of classes contained in class diagrams

2) shows a set of objects and their relationships at a point in time

3) modelling object structures involves taking a snapshot of a system at a given moment in time

4) is an instance of a class diagram or the static part of an interaction diagram

Object Diagram 2

5) is used to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships

6) contains:

    a) objects

    b) links

Creating an Object Diagram

1) identify the function or behaviour you want to model that results from the interaction of a set of classes, interfaces and other artifacts

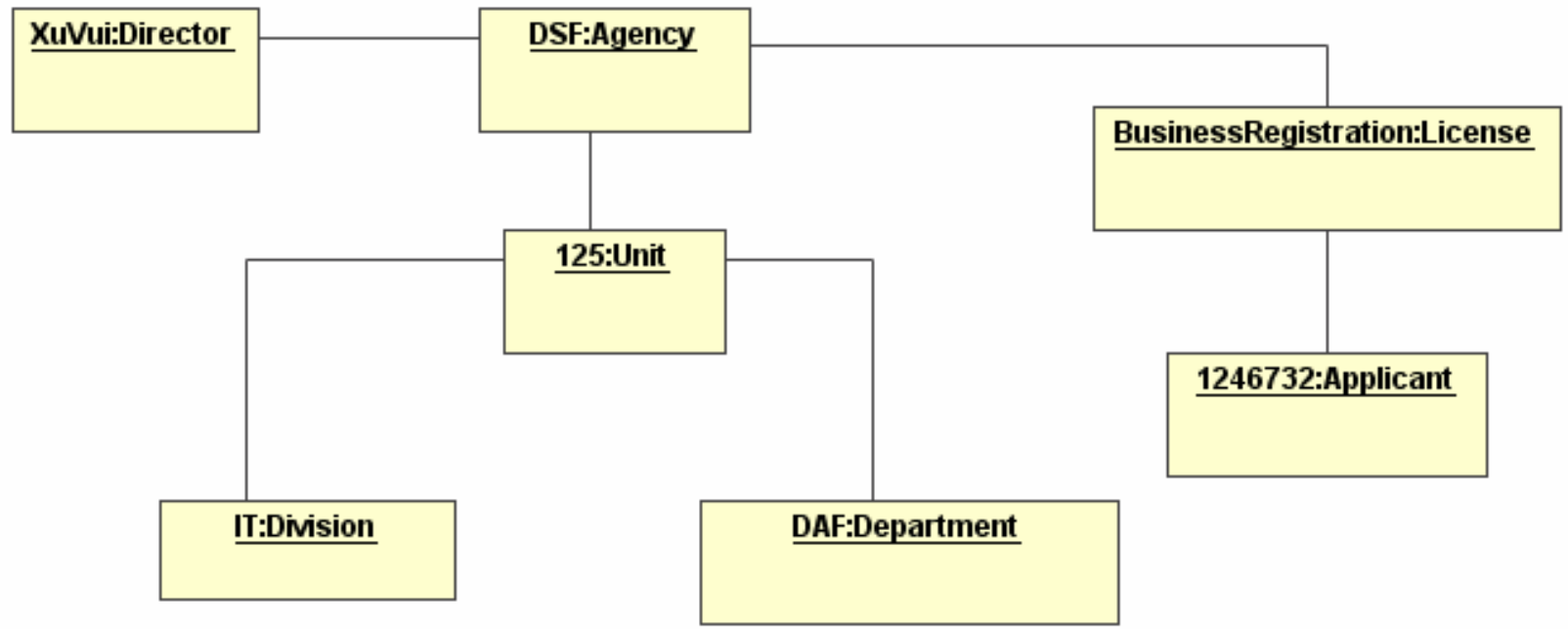
2) for each function or behaviour, identify the artifacts that participate in the collaboration as well as their relationships

3) consider one scenario that invokes the function or behaviour. Freeze the scenario and render each participating object

4) expose the state and attribute values of each object, as necessary to understand the scenario

5) expose the links among these objects

Example: Object Diagram



Requirements Modelling: Behavioural Modelling

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Behavioural Diagrams 1

1) represent how objects behave when you put them to work using the structure already defined in structural diagrams

2) model how the objects communicate in order to accomplish system tasks

3) describe how the system:
    a) responds to actions from the users
    b) maintains internal integrity
    c) moves data
    d) creates and manipulates objects, ...

Behavioural Diagrams 2

4) describe discrete pieces of the system, such as individual scenarios or operations

Note:

no need to specify behavioural diagrams for all system behaviours as simple behaviours may not need a visual explanation of the communication required to accomplish them

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Scenarios

Definition
Scenario is a textual description of how a system behaves under a specific set of circumstances.

The behaviour described in the use cases is the basis for building scenarios.

Scenarios also provide a basis for developing test cases and acceptance-level test plans.

Example: Scenarios 1

1) consider the use case “Track License Application”

2) there are at least two possible scenarios:

   a) the applicant enters the license application number; the system retrieves the information related to it; and the system displays this information

   b) the applicant enters the license application number but this number does not exist in the agency’s database, in which case the system displays an error message

Example: Scenarios 2

Scenario: the applicant enters the license application number; the system retrieves the information related to it; and the system displays this information

Steps:

1) Applicant requests to track status of license application
2) System displays the log on form
3) Applicant enters log on information
4) Applicant submits log on information
5) System validates applicant
6) System displays form to enter the tracking number
7) Applicant enters the tracking number
8) Applicant submits license number
9) System retrieves license information
10) System displays license information

Sequence Diagrams

1) show how objects interact by sending messages among them in order to provide a specific behaviour

2) address the dynamic behaviour of a system with special emphasis on the chronological ordering of messages

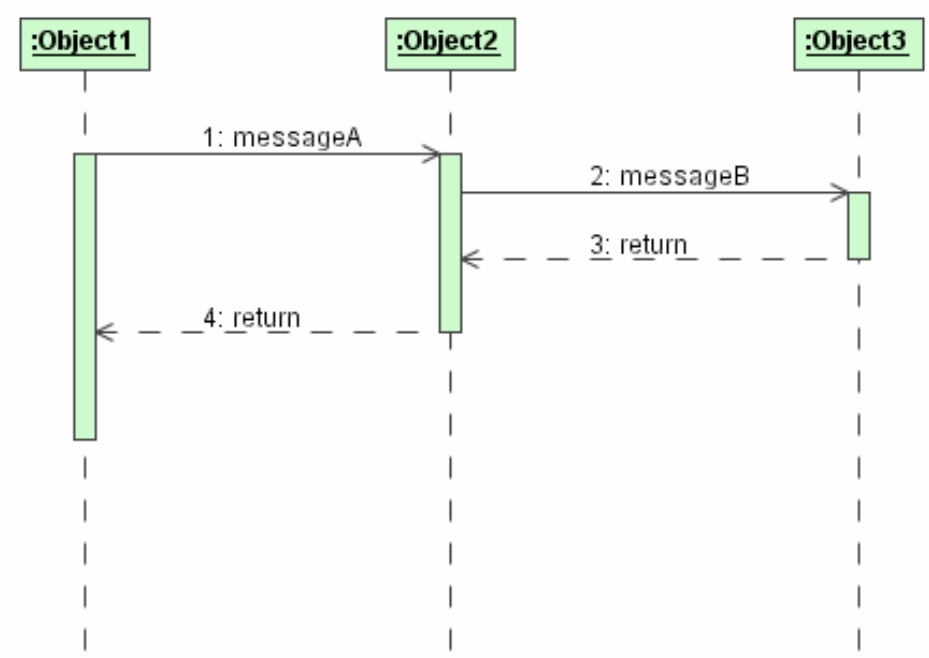
3) show a set of messages arranged in time sequence

4) are used to show the behaviour sequence of a use case

Sequence Diagram - Elements

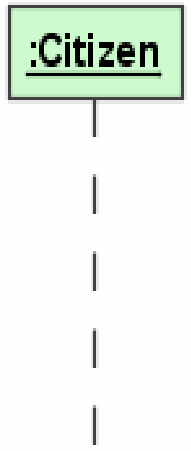
Two elements are used to build sequence diagrams:

1) object lifelines
2) messages or stimuli



Object Lifeline

The notation combines the object icon and a timeline.



The timeline is a line that runs from the beginning of a scenario at the top of the diagram to the end of the scenario at the bottom of the diagram.

If an object is created and destructed during the messages sequence, then the lifeline represents the whole lifecycle of the object.

Message

Definition
Message is a description of some type of communication between objects.

It is a unit of communication between objects.

The sender object may invoke an operation, raise a signal or cause the creation or destruction of the target object.

Notation: is modelled as an arrow where the tail of the arrow identifies the sender and the head points to the receiver.

Stimulus

Definition

Stimulus is an item of communication between two objects, and has the following characteristics:

1) it is associated with both a sending and a receiving object
2) it travels across a link
3) it may invoke an operation, raise a signal (asynchronous message), create or destroy an object
4) it may include parameters/arguments in the form of either primitive values or object references
5) it is associated with the procedure that causes it to be sent

Example: Stimulus

1) Example 1 – invoking an operation: The citizen object submits the application form by sending a message to the system. The system receives the message and executes the associated procedure.

2) Example 2 – create an object: when the System receives the message from the applicant submitting the application form, it creates an object to support the new session.

Messages and Stimuli

A message is the formal specification of a stimulus.

A stimulus is an instance of a message.

The specification includes:

1) the roles that the sender object and the receiver object play in the interaction
2) the procedure that dispatches the stimulus

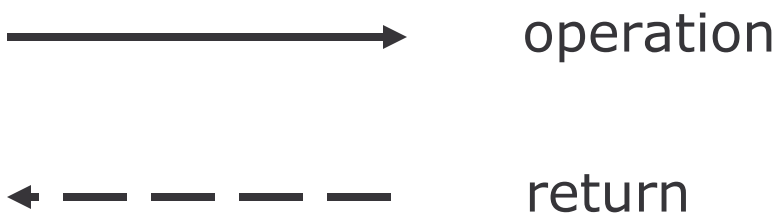
For each message we need to define:

1) the name of the invoked operation and its parameters
2) the information returned from the message.

Operations and Returns

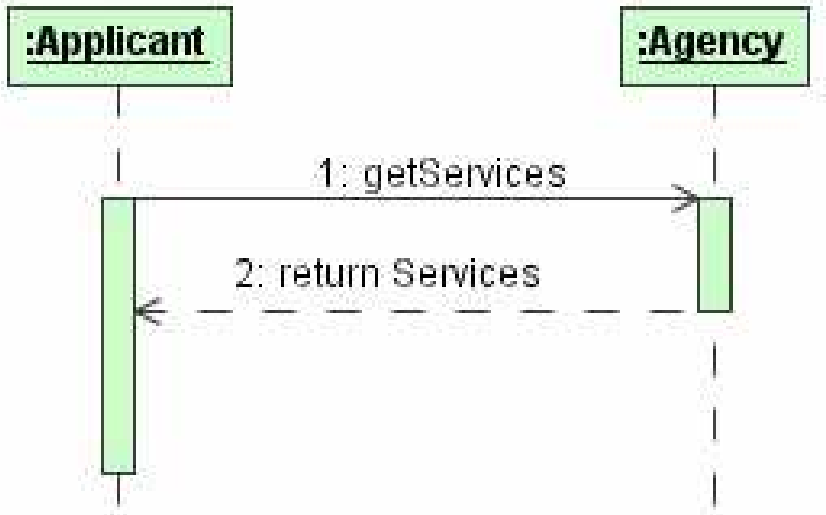
The operation specifies the procedure that the message invokes on the receiving object.

The return contains the information passed back from the receiver to the sender. An empty return is valid.



Example: Operations - Returns

Example: The *Applicant* object sends a message to the *Agency* object to get the information of which are the different services its provides. The *Agency* object returns this information.



Signals

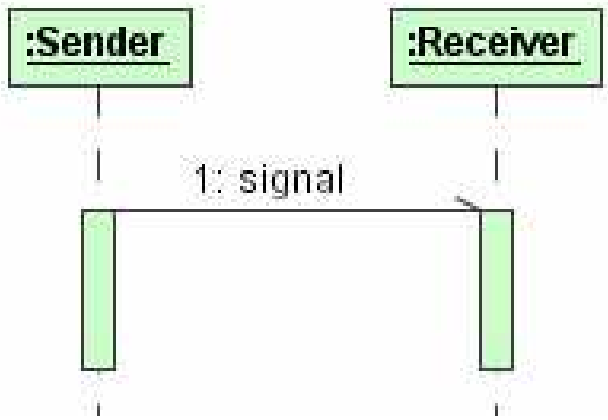
1) an object may raise a signal through a message

2) a signal is a special type of a class associated with an event that can trigger a procedure within the receiving object

3) a signal does not require a return from the receiving object

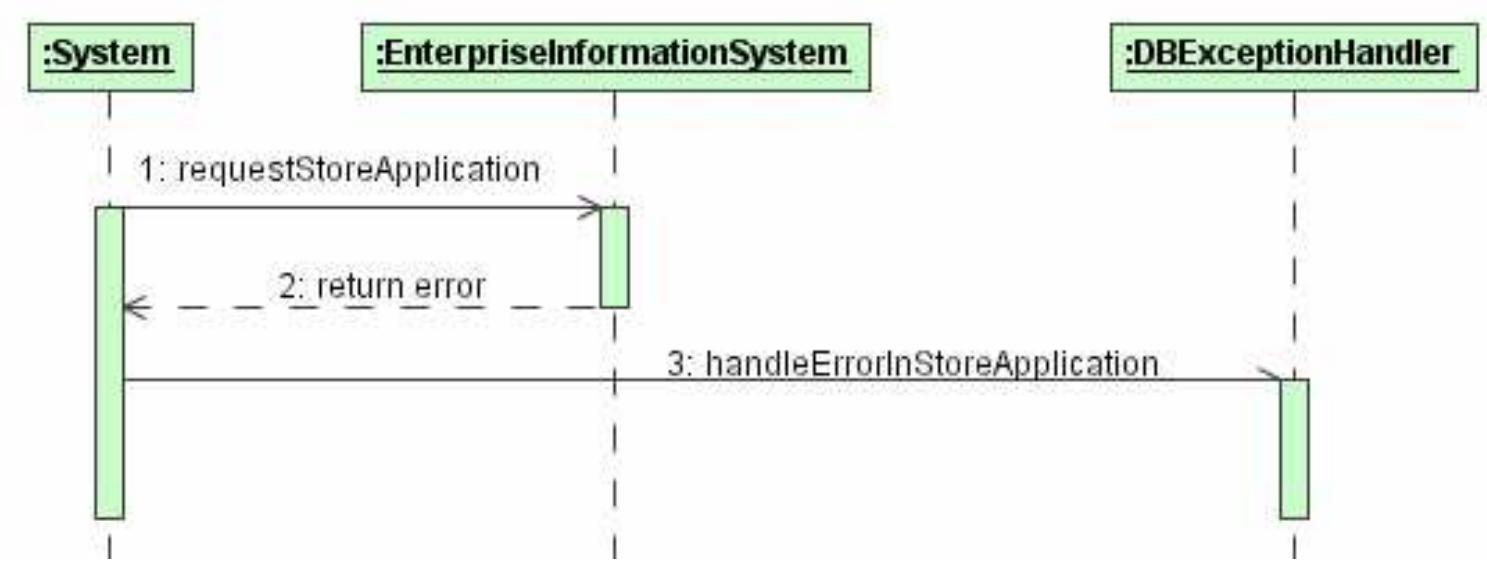
4) an exception is a special type of a signal

5) throwing an exception means sending out a message containing an object that describes the error condition



Example: Signal

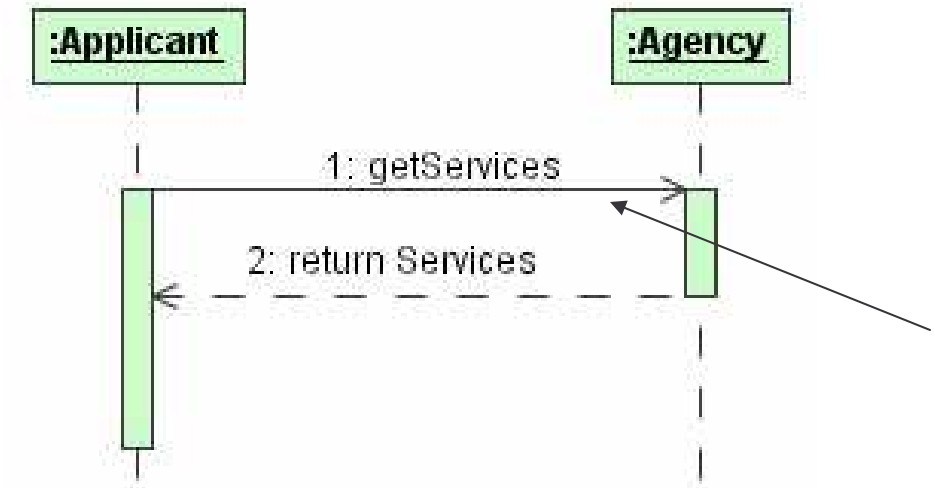
Each time the System object receives a message indicating that some abnormal situation occurred with the database, it raises a signal to the object DBExceptionHandler for the complete treatment of the error.



Identification of Messages

A message number or a message name is used to properly identify messages.

Example:



In this example, we identify the message getServices with 1.

Example: Message Syntax

Example:

2.1, 2.5, 6 / 8: getCitizenAddress * [foreach ApplicationForm] *return* text *:=* getCitizenPersonalAddress(Citizen.Id:Integer)

In this example, we are specifying message number 8 called getCitizenAddress.

This message will be executed more than once (*), one time for each ApplicationForm.

Each message will call the operation getCitizenPersonalAddress of the receiving object, sending as parameter the CitizenId that is of type Integer, and will return a value of type text.

For the execution of this message, it is required that messages 2.1, 2.5 and 6 have already been executed.

Message Syntax 1

predecessors '/' sequence-term iteration [condition] return ':=' operation

where:

1) *predecessors* refers to a list of comma-separated sequence numbers of messages that must come before the current message
   - predecessors on the diagram are assumed so they don't need to be included

2) *sequence-term* may be either a number or a name that identifies the message

Message Syntax 2

3) *iteration* refers to the need to execute one or more messages in a sequence more than once

3) one message: add an iteration symbol (*) and a condition to control the number of iterations
4) many messages: enclose the set of messages in a box

4) *condition* is used to specify the control of the iteration and is expressed as a text enclosed within square brackets

5) *return* may include a list of values sent back to the sender by the receiver

6) *operation* defines the name of the operation and optionally its parameters and a return value

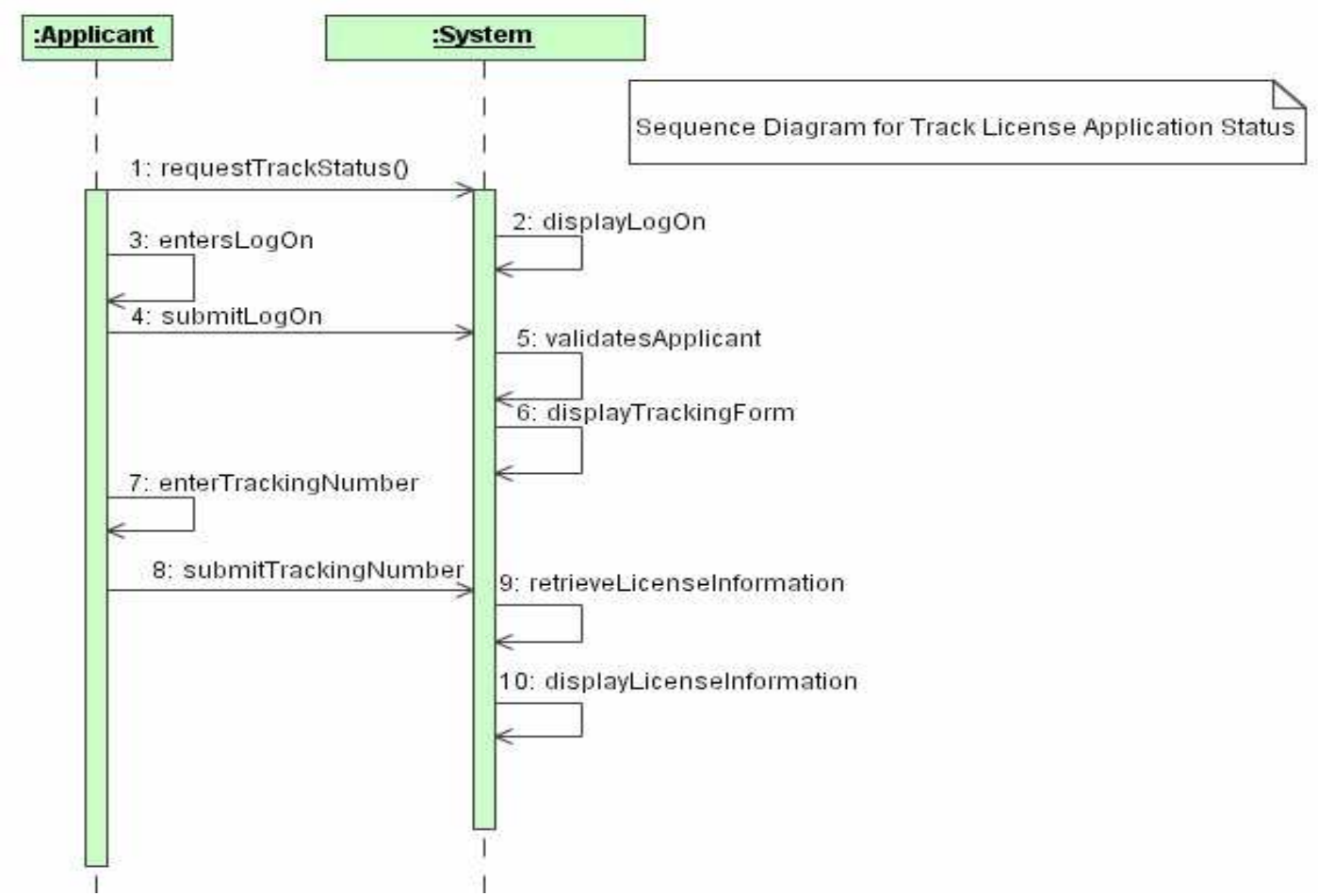
Case Study Model Example 1

Recall the example explained for scenarios: an applicant tracks the status of a license already applied through the system and the system displays the license information.

Procedure:

1) Applicant requests to track status of license application
2) System displays the log on form
3) Applicant enters log on information
4) Applicant submits log on information
5) System validates applicant
6) System displays form to enter the tracking number
7) Applicant enters the application the tracking number
8) Applicant submits tracking number
9) System retrieves license information
10) System displays license information

Case Study Model Example 2



Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Statechart Diagrams

Statechart diagrams define a notation for describing state machines.

State machines capture the changes in an object throughout its lifecycle as they occur in response to external events.

The statechart diagram identifies both the external events and internal events that can change the object's state.

The scope of a statechart is the entire life of one object.

The foundation of statecharts is the relationship between states and events.

States

Definition
State is the current condition of an object reflected by the values of its attributes and its links to other objects.

Notation:

Initial State

The initial state identifies or points to the state in which an object is created or constructed.

The initial state is called a pseudo-state because it does not really have the features of an actual state, but it helps clarify the purpose of another state of the diagram.

Notation:

Final State

The final state is the state in which once reached, an object can never do a transition to another state.

Also, the final state may mean that the object has actually been destroyed and can no longer be accessed.

Notation:

Events

Definition
Event is an occurrence of a stimulus that can trigger a state transition.

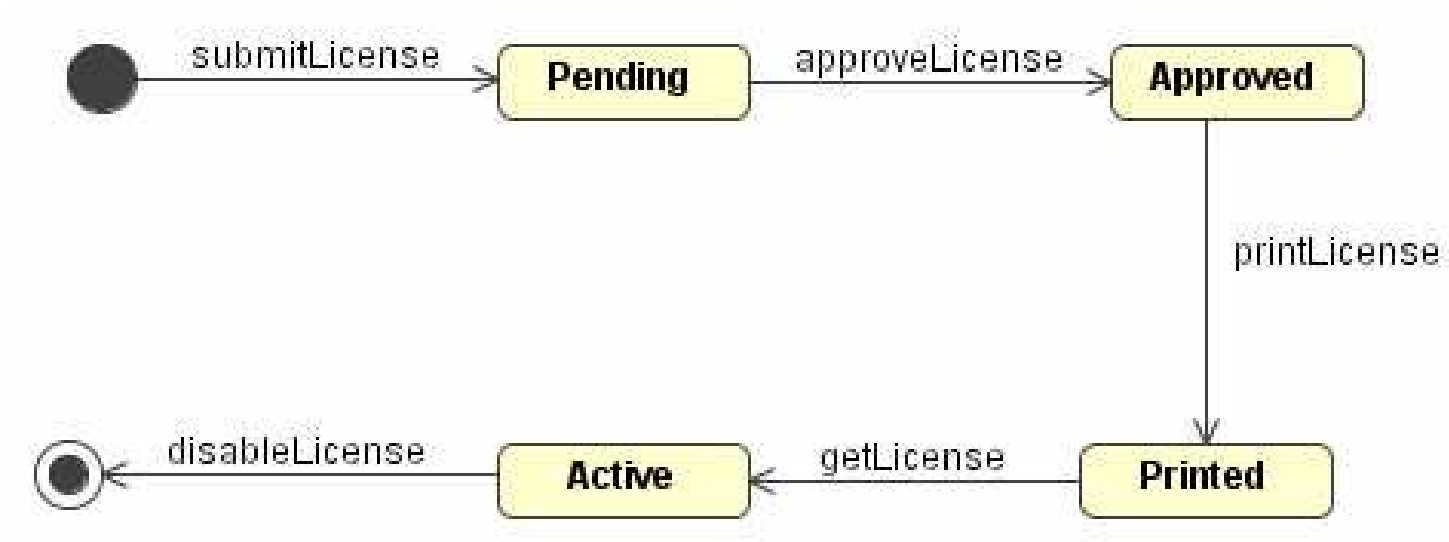
An event may be:

1) the receipt of a signal, e.g. the reception of an exception or a notice to cancel an action
2) the receipt of a call, that is the invocation of an operation, e.g. for changing the expiration date of a license or for printing approved licenses.

An event on a statechart diagram corresponds to a message on a sequence diagram.

Notation: 

Example: Statechart Diagram



Special Events

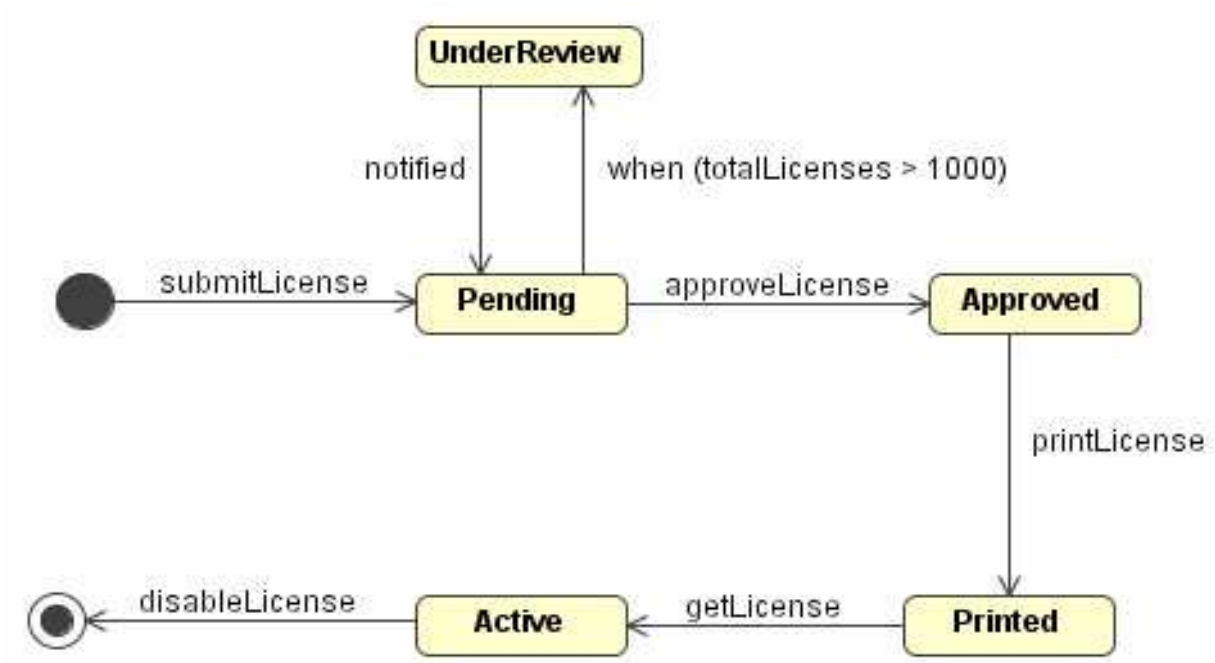
An event may also be the recognition of some condition in the environment or within the object itself, such as:

1) change event: A predefined condition becoming true.
   Example: when the amount of submitted licenses by day is over a predefined quantity, an action such as sending a notice is required.

2) elapsed-time events: The passage of a designated period of time. Example: for printed licenses not reclaimed by anybody after some period, some action is required to destroy those licenses.

Example: Change Event

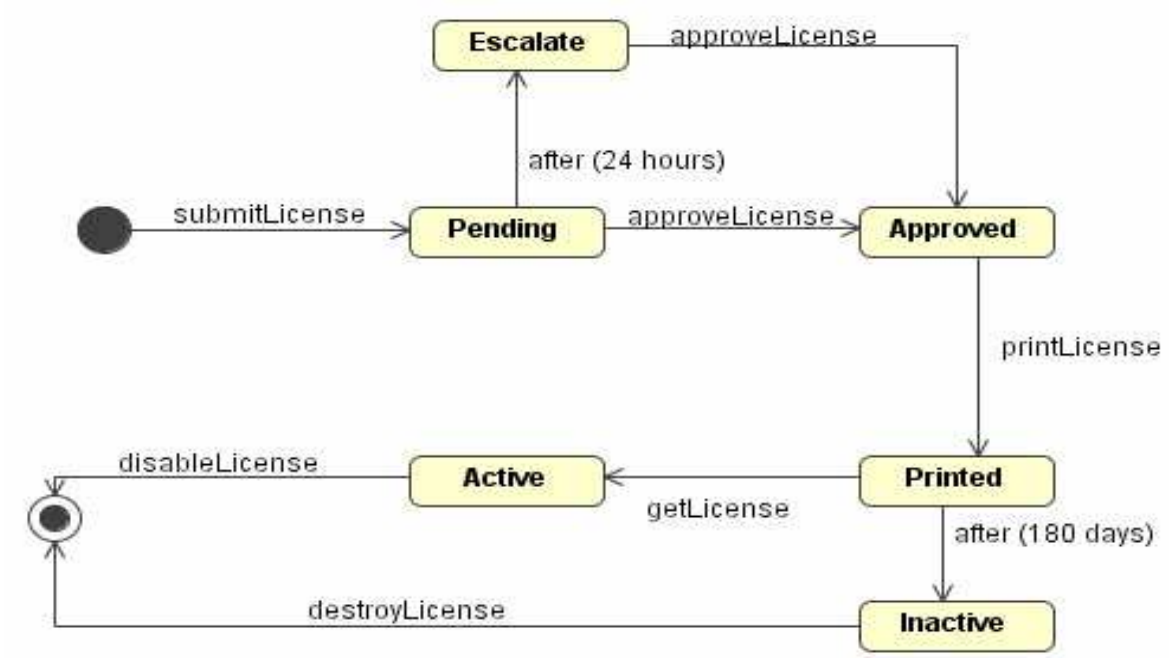
A change event is an event where the transition is based on a change in the object or a change in the environment.



The event-name is the keyword when followed by a boolean expression enclosed in parenthesis.

Example: Elapsed-Time Event

An elapsed-time event is an event where the transition is triggered because of the passage of time.



The event-name is the keyword "after" followed by a numerical expression enclosed in parenthesis that represents the amount of time.

Event syntax 1

Example:

approveLicense(License.Id) [requirements=ok] / setExistLicense(true)

where:

1) the event name is approveLicense

2) the event parameter is License.Id

3) the guard condition specifies that *requirements* (attribute of the object) should be *ok*. This condition determines whether the receiving object should respond to the event or not

4) the action executed in the receiving object is a call to the method setExistLicense sending true as parameter

Event syntax 2

event-name ‘(‘ [comma-separated-parameters-list] ‘) ‘
[‘[‘guard-condition’]’] / [action-expression]

where:

1) event-name identifies the event

2) parameters-list defines the data values that are passed with the event for use by the receiving object in its response to the event

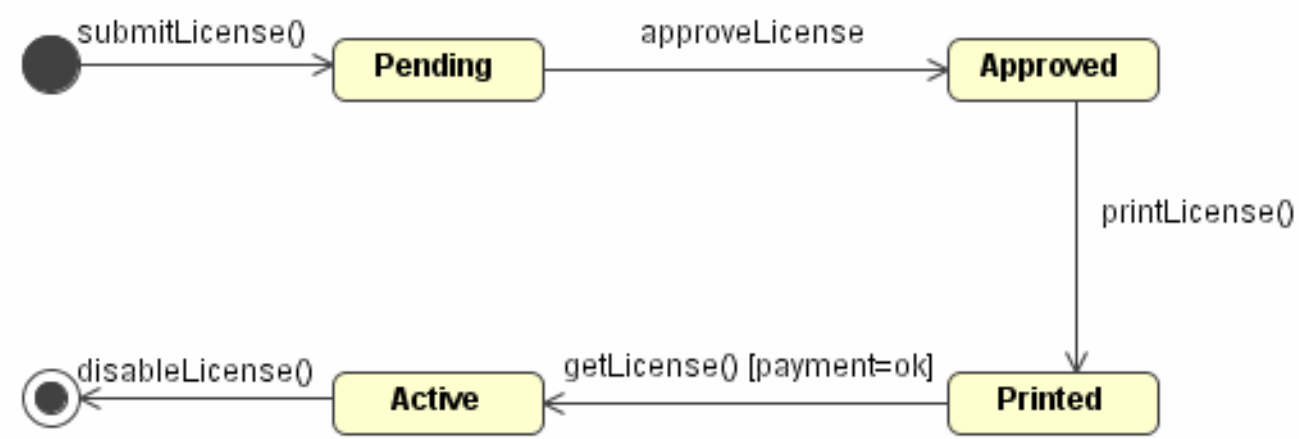
3) guard-condition determines whether the receiving object should respond to the event

4) action-expression defines how the receiving object must respond to the event

Guard Condition

Typically, an event is received and responded to unconditionally.

However, the receipt of an event may be conditional; the test needed is called the guard condition.



Event Actions

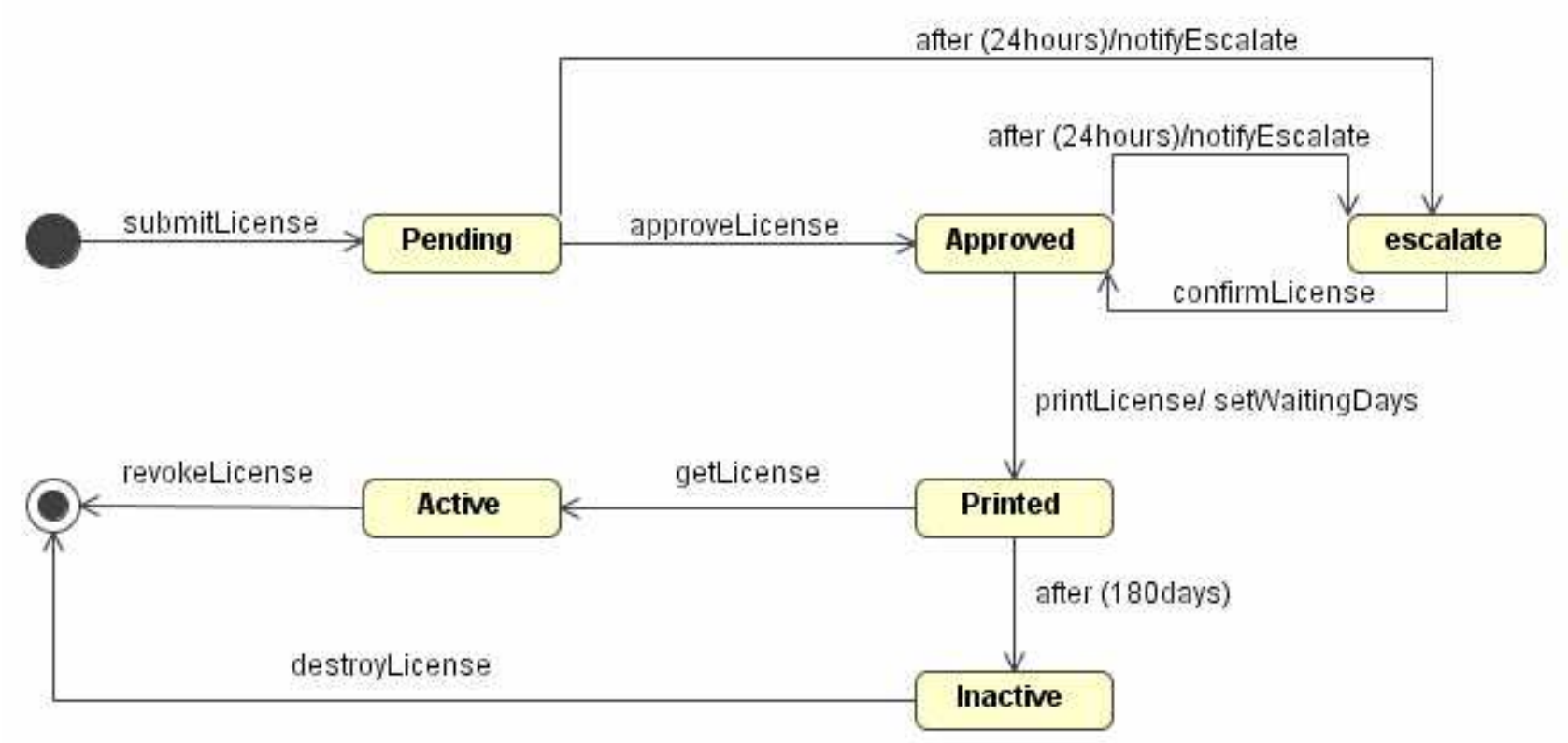
The response to an event has to explain how to change the attribute values that define the object's state.

The behaviour associated with an event is called an action expression.

An action expression is part of a transition event, it is a part of the change from one state to another.

An action expression is an atomic model of execution, and is referred to as run-to-completion semantics.

Example: Event Actions

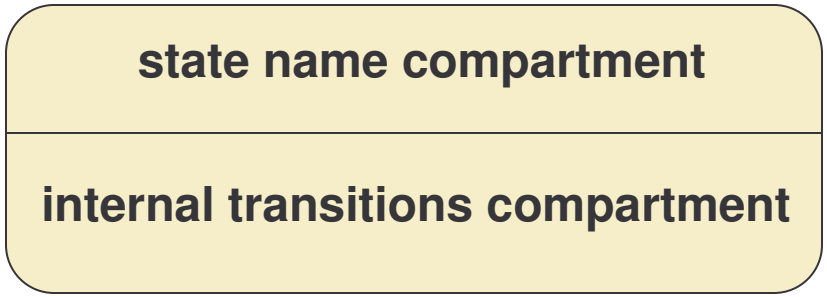


Internal Compartment

The state icon can be expanded to model what the object can do while it is in a given state.

The notation splits the state icon into two compartments, the name compartment and the internal transitions compartment.

The internal transitions compartments contains information about actions, activities and internal transitions specific to that state.



Entry Actions

More than one event can change the object to the same state.

When the same action takes place in all events that goes into the state, it is possible to write the action only once as an entry action.

Notation:

1) use the keyword entry followed by a slash and the action or actions that need to be performed every time you enter the state
2) entry actions are placed in the internal transitions compartment.

Exit Actions

The same simplification can be used for actions associated with events that trigger a transition out of a state.

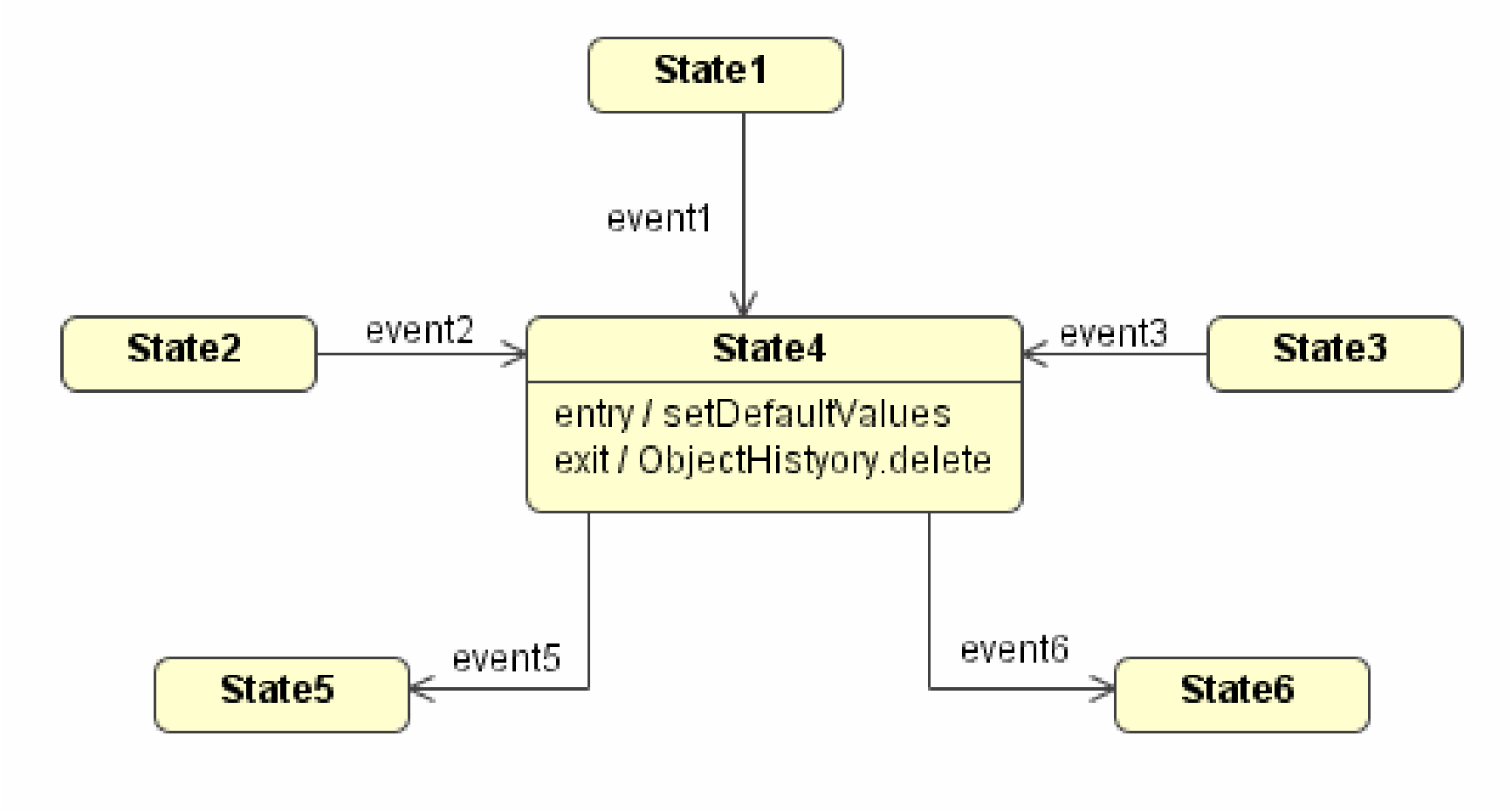
They are called exit actions.

Notation:

1) use the keyword exit followed by a slash and the action or actions that are performed every time you exit the state
2) exit actions are placed in the internal transitions compartment.

It may only be used when the action takes places every time you exit the state.

Example: Entry / Exit Actions



Activities 1

Activities are processes performed within a state.

Activities may be interrupted because they do not affect the state of the object.

Notation:

1) use the keyword do followed by a slash and one or more activities
2) activities are placed in the internal transitions compartment.

Activities 2

An activity should be performed from the time the object enters the state until either the object leaves the state or they finish.

If an event produces a transition out of the activity state, the object must shut down properly and exit the state.

Internal Transitions

An event that can be handled completely without a change in state is called an internal transition.

It also can specify guard conditions and actions.

Notation:

1) uses the keyword internal transition followed by a slash and one event action
2) they are placed in the internal transitions compartment

Order of Events

1) if an activity is in process in the current state, interrupt it and finish it in a proper way

2) execute the exit action(s)

3) execute the actions associated with the event that triggered the transition

4) execute the entry action(s) of the new state

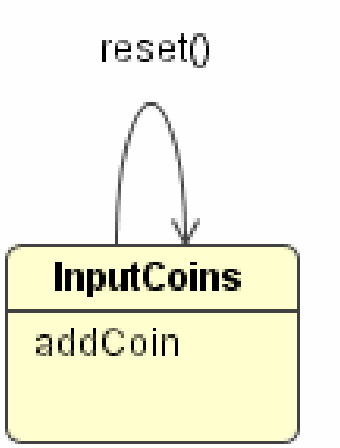
5) begin executing the activity or activities of the new state.

Self Transition

A self-transition is an event that is sufficiently significant to interrupt what the object is doing.

It forces the object to exit the current state and return to the same state.

The result is to stop any activity within the object, exit the current state and re-enter the state.



Important Features

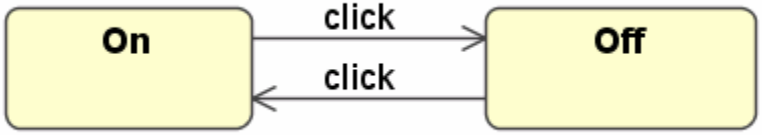
Within the statechart diagram:

1) an object need not know who sent the message
2) an object is only responsible for how it responds to the event

Focusing on the condition of the object and how it responds to the events, which object sends the message becomes irrelevant and the model is simplified

The state of the object when it receives an event can affect the object's response

event + state = response



Defining Send Events

Sometimes the object modelled by the statechart needs to send a message to another object, in this case the outgoing event must define which is the receiving object. Also, this event must be caught by the receiving object.

A message send to another object is called a send event.

Notation: provide the object name before the action expression with a period separating both.

Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Relating Diagrams 1

1) the sequence diagram models the interactions between objects

2) the statechart diagram models the effect that these interactions have on the internal structure of each object

3) the messages modelled in the sequence diagrams are the external events that place demands on objects

Relating Diagrams 2

4) the objects internal responses to those events that cause changes to the objects' states are represented in the statechart diagram

5) not all objects need to be modelled with a statechart diagram

6) the objects that appear in many interactions and are target of many events are good candidates to be modelled with a statechart diagram

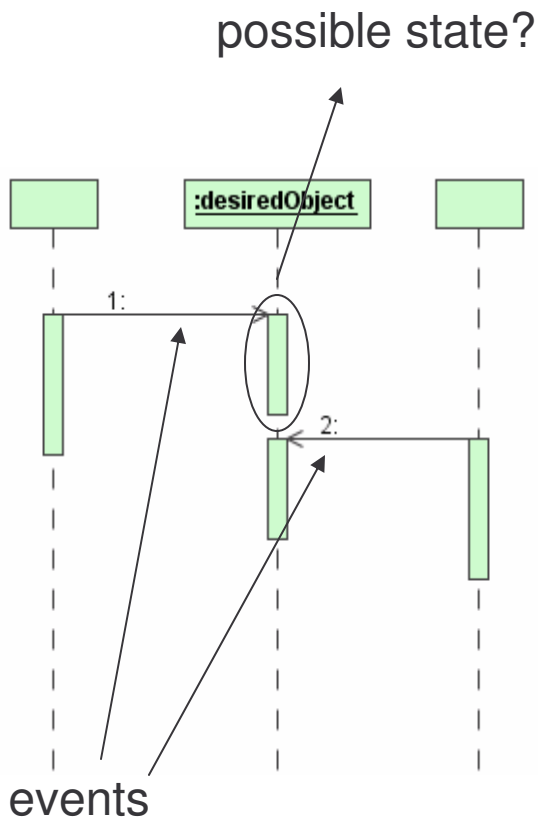
Deriving Statechart Diagrams

1) identify the events directed at the lifeline of the desired object

2) identify candidate states by isolating the portions of the lifeline between the incoming events

3) name the candidate states using adjectives that describe the condition of the object during the period of time represented by the gap

4) add the new state and events to the statechart diagram



Requirements Modelling

1) Software Requirements
   a) Problems
   b) Process
   c) Types

2) Use Case Modelling:
   a) Concepts
   b) Use Case Diagrams
   c) Templates

3) Conceptual Modelling:
   a) Concepts
   b) Class Diagram
   c) Object Diagram

4) Behavioural Modelling:
   a) Behavioural Diagrams
   b) Sequence Diagrams
   c) Statechart Diagrams
   d) Relation between them

5) Summary

Summary 1

A requirement is a function that a system must perform or a desirable characteristic of a system.

There are different kind of requirements such as functional and non-functional.

Most project failures can be traced back to errors made in requirements gathering and specification.

Summary 2

Use cases are descriptions of a set of action sequences that a system performs to obtain an observable result to the environment.

Use cases may be related using the generalization, include and extend relationships.

Actors are entities that interact with the system causing to respond to business events.

Use case diagrams shows a set of use cases, actors and their relationships.

Summary 3

Conceptual modelling helps in understanding application domains.

Classes are equivalent to Concepts in UML.

Conceptual Class diagrams describe (showing some important attributes of the classes, but no method) and relate the concepts of a domain.

Object diagrams model the instances of classes contained in class diagrams.

Summary 4

Behavioural modelling specifies how objects work together to provide a specific behaviour using their structure.

Sequence diagrams show how objects interact during time in order to deliver a discrete piece of the system functionality.

Statechart diagrams capture the changes in an object or a set of related objects as they occur in response to events.

Exercise: Requirements 1

Section A: Class Exercise

Consider the Online Licensing Service case study presented earlier.

1) Use Case Modelling:

a) Using the use case diagram presented in slide 177, provide a detailed use case specification for any two of the use cases shown in the diagram.

Exercise: Requirements 2

Section A: Class Exercise

2) Conceptual Modelling using Class Diagrams:

a) List five different concepts (not included in the diagram presented in slide 202 and 203) related to this domain.

b) Describe the attributes of the classes that represent these five concepts.

c) Provide a simple conceptual model using a class diagram which relates these concepts with others already presented in slide 202.

d) Present an object diagram based on the class diagram provided in the previous question specifying the objects states.

Exercise: Requirements 3

Section A: Class Exercise

3) Behavioural Modelling using Sequence and Statechart Diagrams:

a) Describe two scenarios for the use case detailed in point one of this exercise.

b) Present the sequence diagrams for these scenarios.

c) Select an object that has a relevant behaviour and provide the statechart diagram to model the different states of its lifecycle. If possible, try to include an activity, a change-event and an elapsed-time event.

Exercise: Requirements 4

Section B: Project

4) List the core functional requirements of the system you wish to develop. For each requirement listed, provide the requirement identifier, description, and cross references.

5) List five non functional requirements of the system using the format specified in question 4.

6) Identify the use cases for your system.

7) Identify the actors related to each of the use cases identified in question 6.

Exercise: Requirements 5

Section B: Project

8) Provide a detailed specification for the uses cases listed in question 6 using the template provided in slide 173.

9) Provide a use case model to show the relationships between actors and user cases for your system.

10) Provide a glossary that describes the core concepts of your application domain (at least 10 concepts). You may suggest any glossary format of your choice.

11) Use a class diagram to relate these concepts identified in question 10. In addition, update the glossary to define the meanings of the relationships used in this diagram.

Exercise: Requirements 6

Section B: Project

12) Identify the scenarios with relevant behaviour.

13) Provide sequence diagrams for different scenarios of the use cases mentioned in the previous question.

14) Identify objects which may present different states.

15) Provide statecharts for the objects mentioned previously.

Architecture Modelling

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

Software Architecture Concepts

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Software Architecture 1

Definition

An architecture is:

1. a set of significant decisions about the organization of a software system
2. the selection of structural elements and their interfaces by which the system is composed together with a behaviour as specified in the collaborations amongst elements
3. the composition of these structural and behavioural element into progressively larger subsystem
4. the architectural style that guide this organization – the elements and their interfaces, their collaboration and composition

Software Architecture 2

Architecture involves:

1) structural organization of a system from its components,
2) ensemble of elements that collaborate to constitute the system
3) how elements interact to provide the system's overall behaviour or required functionality

Architectural concerns:

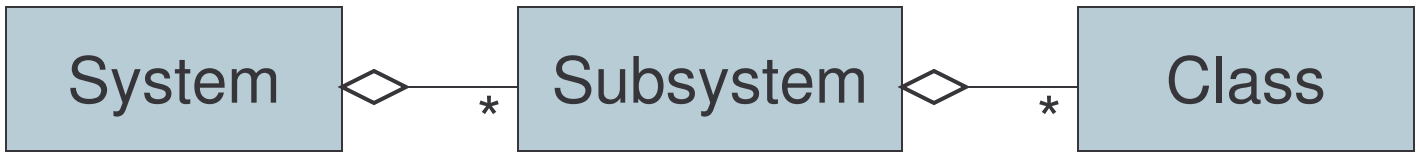
1) structural or static
2) behavioural

Subsystems and Classes

A solution domain may be decomposed into smaller parts called subsystems.

Subsystems are composed of solution domain classes (design classes).

Subsystems may be recursively decomposed into simpler subsystems.



Subsystems and Services 1

A subsystem is characterized by the services it provides to other subsystems.

A service is a set of related operations that share a common purpose.

Example of a service:

notification service involves the following operations – send notices, lookup notifications channels, subscribe and un-subscribe to a channel

The set of operations of a subsystem are available to other subsystem through the subsystem's interface.

Subsystems and Services 2

Subsystem interface or API includes:

1) name of the operations
2) parameters
3) high-level behaviour
4) return values

The notion of services allow to focus more on the interfaces rather than the implementation to minimize the impact of change.

Coupling

Definition

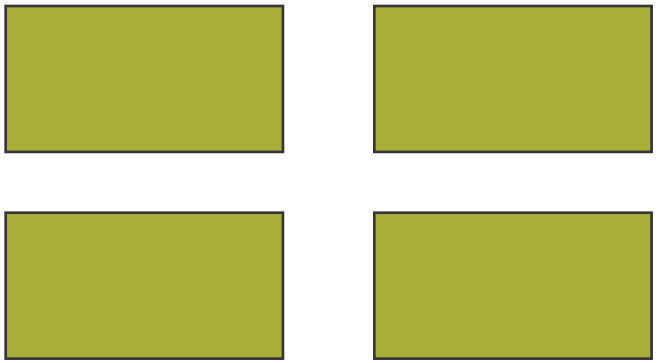
Coupling is the strength of dependencies between two sub-systems.

Subsystem independence allows for easy understanding, modification and maintenance.

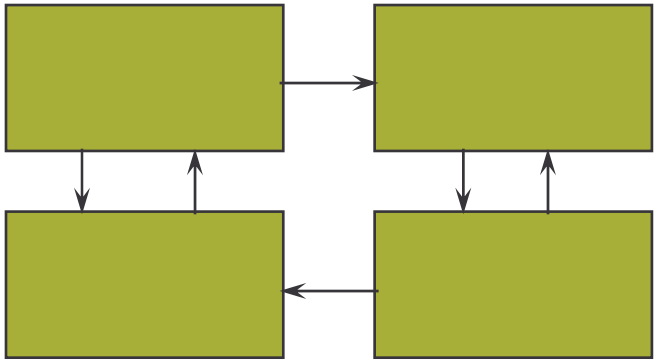
Loose coupling minimizes the impact of one subsystem on others.

A desirable property of subsystem decomposition is that the resulting subsystems be loosely coupled.

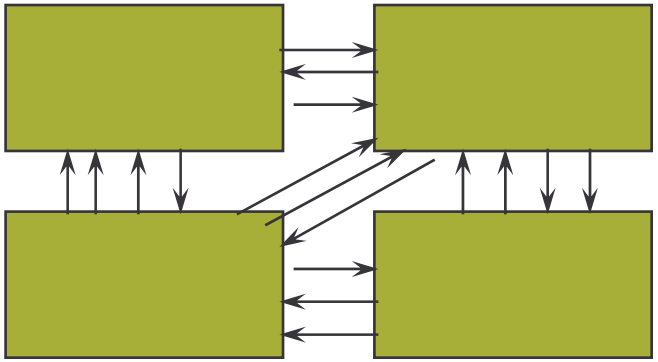
Coupling: Example



Uncoupled -
no dependencies



Loosely coupled -
some dependencies



Highly coupled -
many dependencies

Cohesion or Coherence

Definition

Coherence or cohesion is the strength of dependencies within a subsystem.

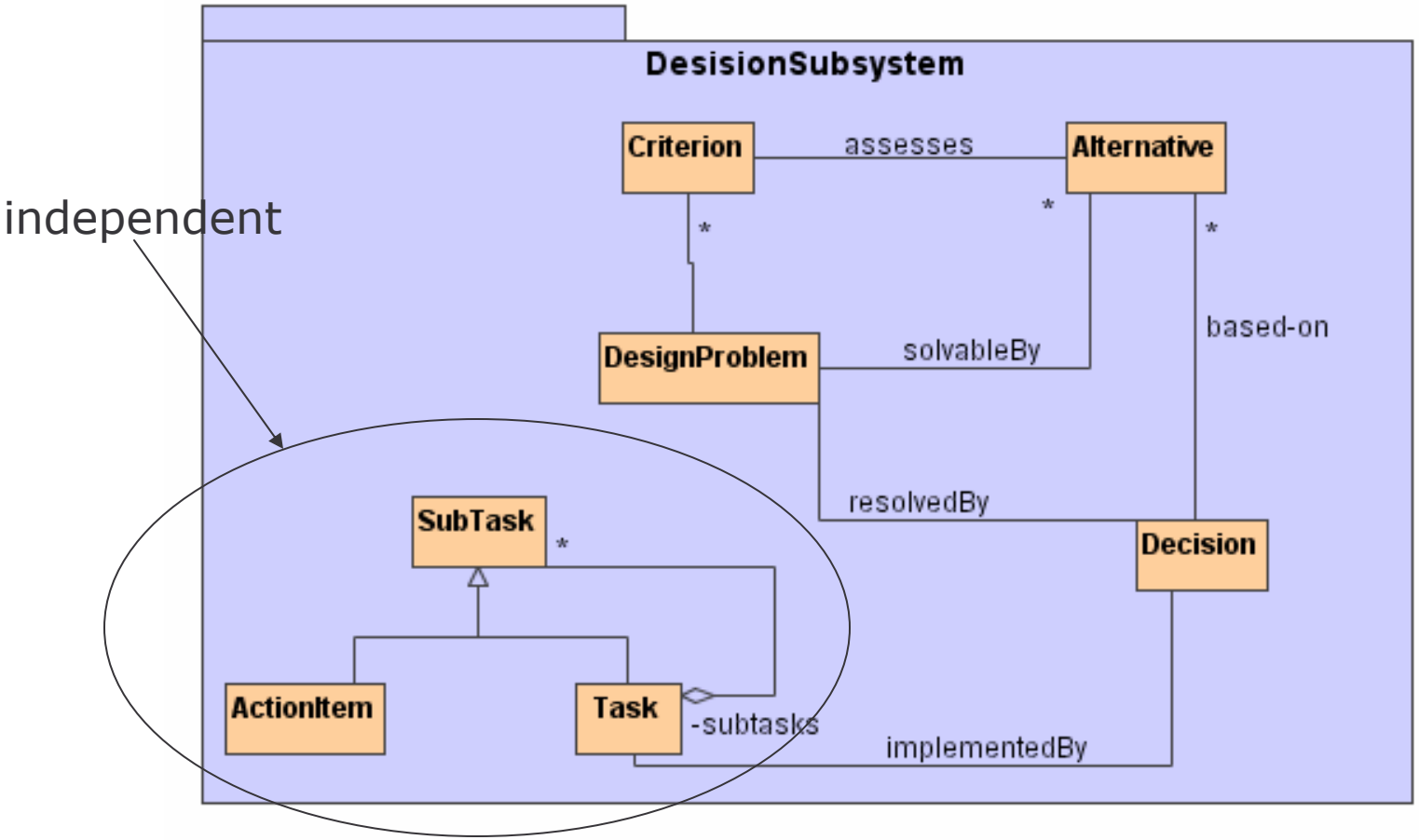
1) the internal "glue" with which a subsystem is constructed

2) a component is cohesive if all its elements are directed toward a task and the elements are essential for performing the same task

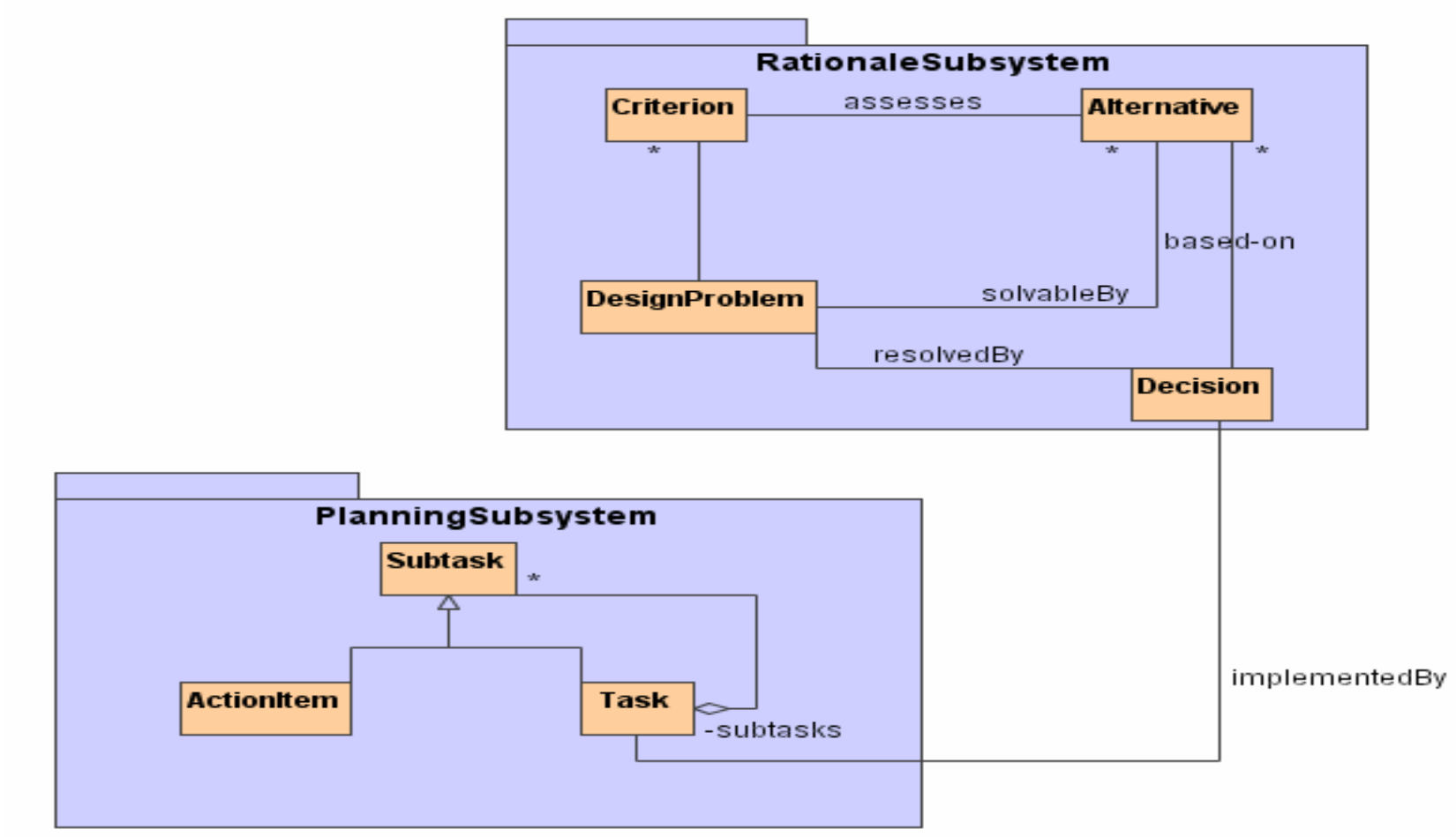
3) if a subsystem contains unrelated objects, coherence is low

4) high cohesion is desirable

Low Cohesion



Example: High Cohesion



Layering

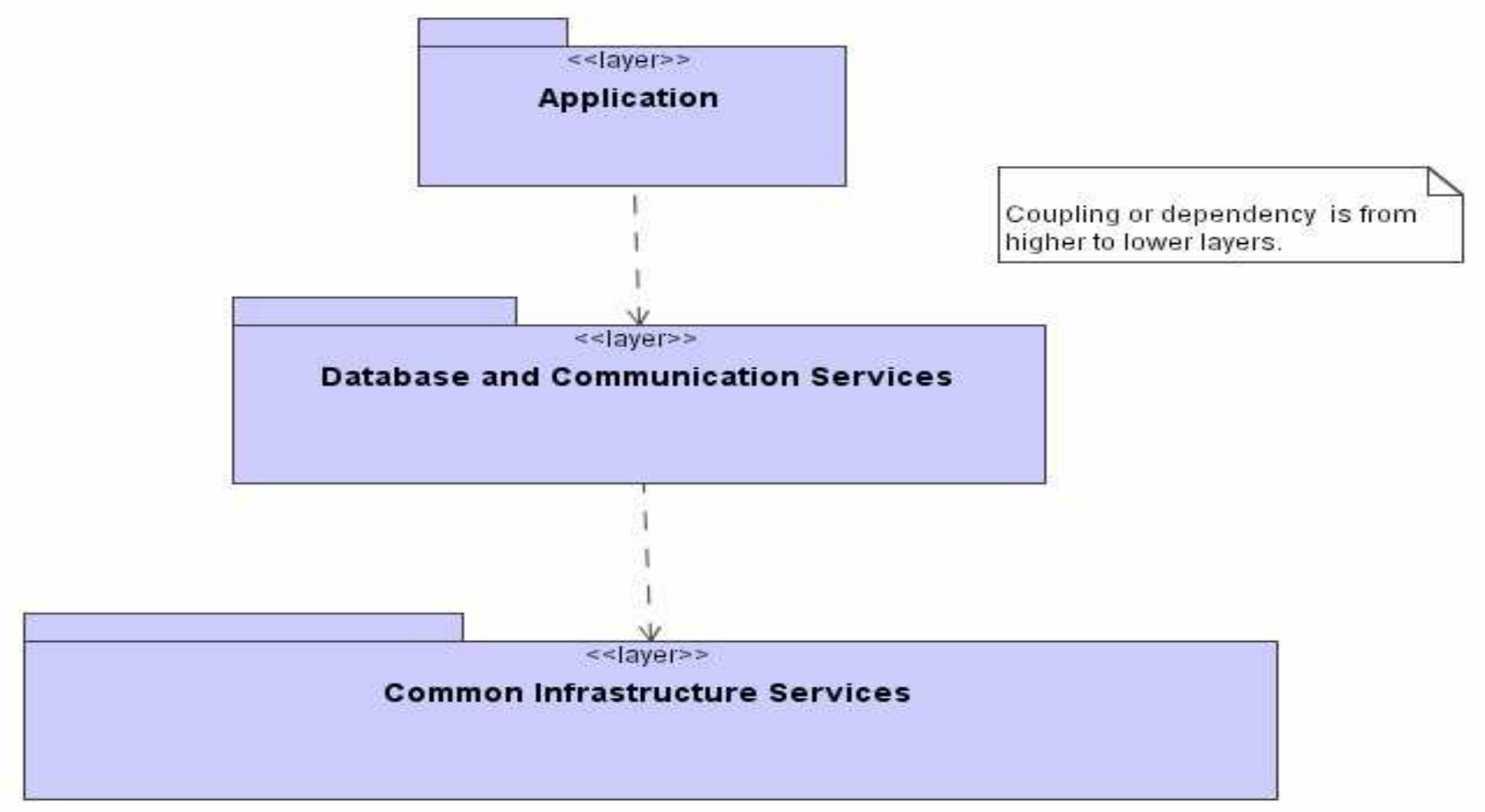
1) A strategy for dividing system into subsystem.

2) Layering divides a system into a hierarchy of subsystems.

3) There are two common approaches to layering:
   a) responsibility driven: layers have well-defined responsibilities, such that they fulfill specific roles in the overall scheme of things

   b) reuse driven: layers are designed to allow for maximum reuse of system elements, by making higher level layers use services of lower level layers

Example: Layered Architecture



Packages

Architecture Modelling

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

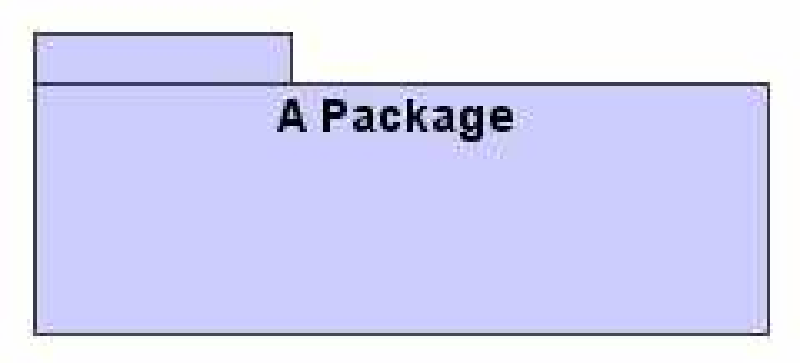
9) Summary

Packages

A general purpose mechanism for organizing modelling elements (e.g. classes) into groups.

It groups elements that are semantically close and that tend to change together.

Well structured packages are loosely coupled and very cohesive, with tightly controlled access to the package's content.



Owned Elements

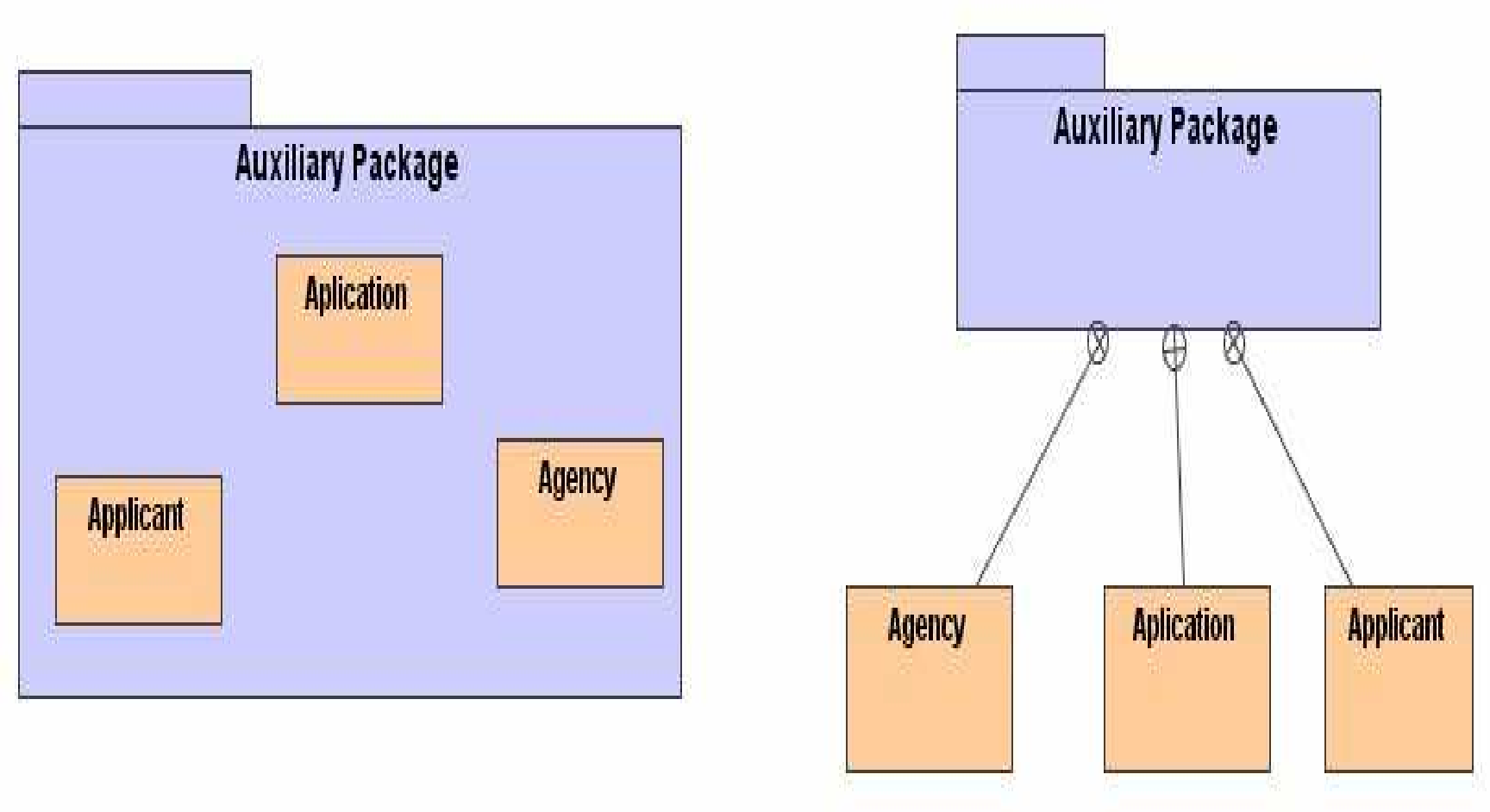
A package may own other elements for instance: classes, interfaces, components, nodes, collaborations, use cases, diagrams and other packages.

“Owning” is a composite relationship.

Elements are declared within the package and are destroyed when the package is destroyed.

A package forms a namespace, thus elements of the same kind must be named uniquely. For example you cannot have two classes in the same package with the same name.

Example: Owned Elements



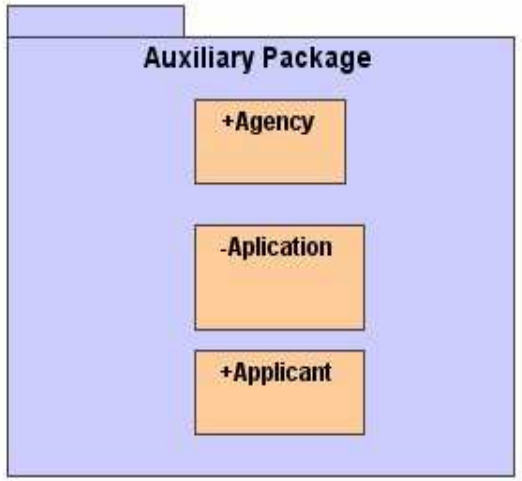
Visibility

Visibility of owned elements can be controlled in a similar way as visibility for attributes and operations of classes.

Elements owned by the package is visible to contents of any package that imports the owning or enclosing package.

Protected elements can also be seen by children and private elements cannot be seen outside their owning package.

For example an element of any package that imports the “Auxiliary package” will have access to the “Agency” element but not “Application”.



Importing 1

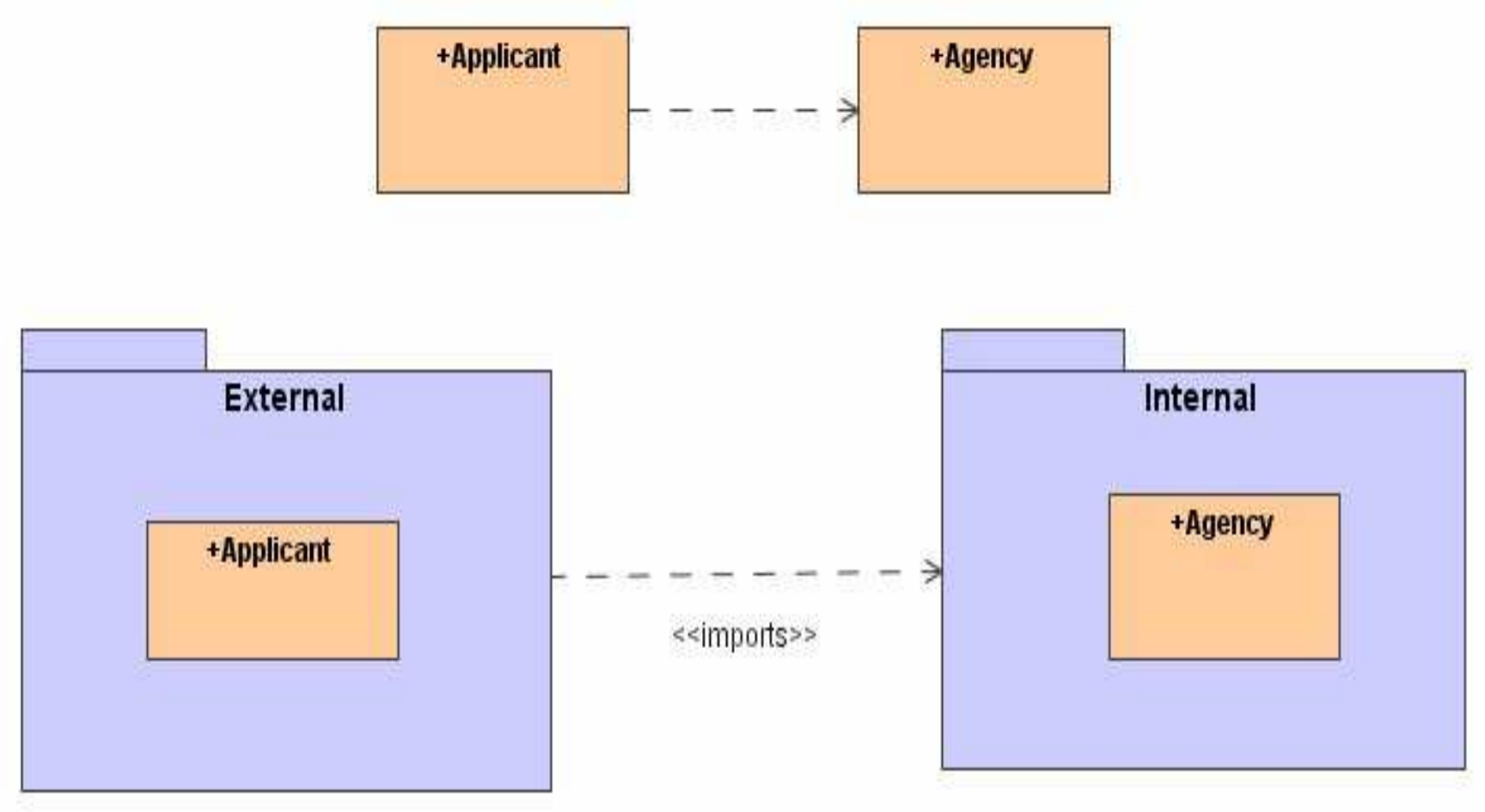
Suppose *A* and *B* are two peer classes (A requires B), which are organized into separate packages.

Also suppose that *A* and *B* are both declared as public in their respective packages.

Class A can only access class B when A's package imports B's Package.

Importing grants a one way permission for elements in one package to access elements in the imported package.

Importing 2


+Applicant
+Agency
External
+Applicant
<<imports>>
Internal
+Agency

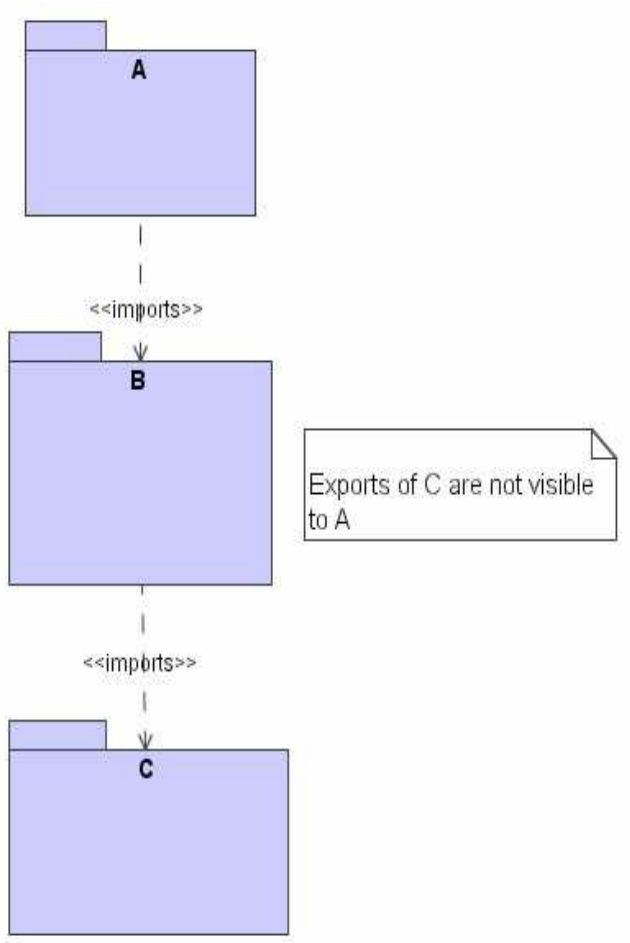
Exporting

Exports are the contents of a package that are visible only to other packages that explicitly imports it.

For example *Agency* is an export of package *Internal*

Import and access dependencies are not transitive.

If an element is visible within a package, then it is visible to all packages nested within the package.



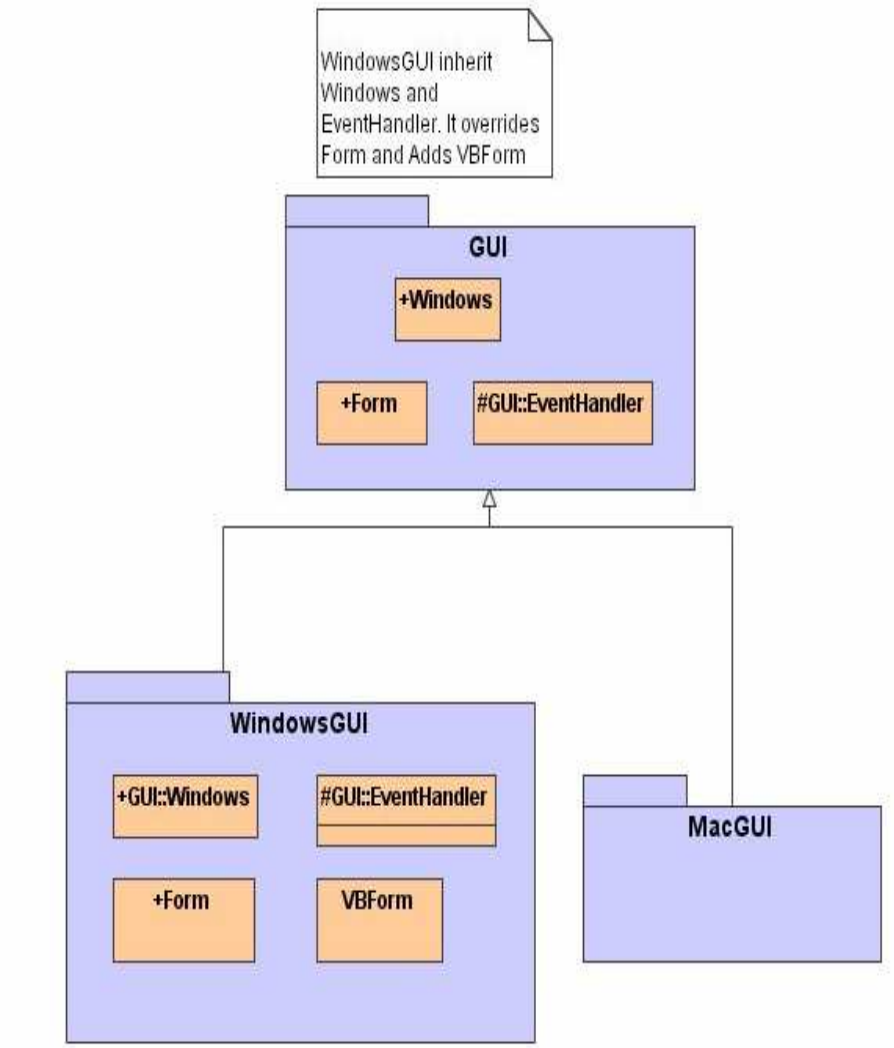
Generalization

Generalization is the second kind of relationship that can exist between packages.

Generalization is used to specify a family of packages and is similar to generalization between classes.

Packages involved in generalization relationships follow the same substitutability rule as classes.

WindowsGUI may be used wherever *GUI* is used.

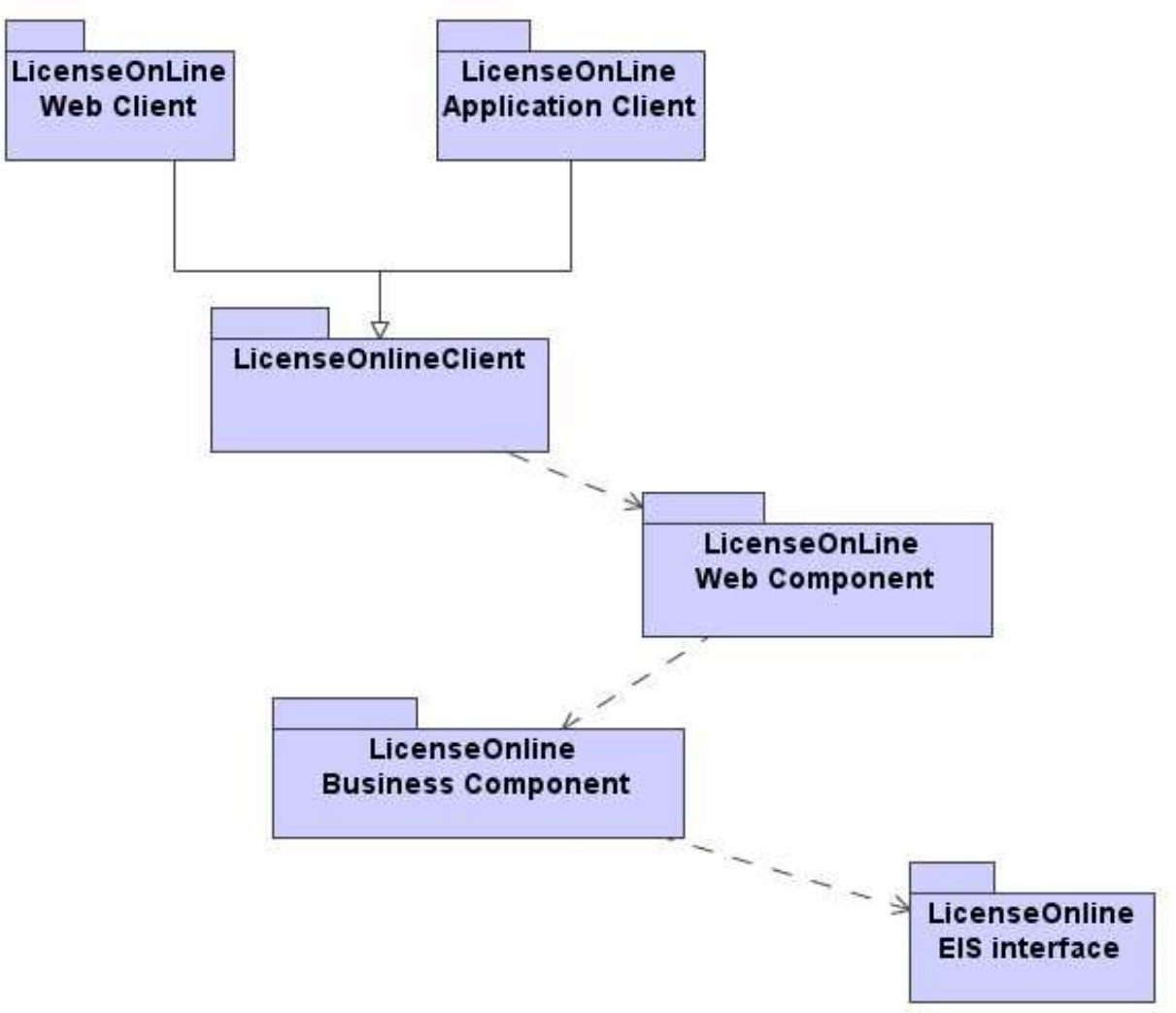


Package Stereotypes

UML provides five stereotypes that apply to packages:

1) Façade: a package that is only a view on some other packages
2) Framework: a package that consists mainly of patterns
3) Stub: specifies a package that serves as a proxy for the public contents of another package
4) Subsystem: a package representing an independent part of the entire system being modelled
5) System : specifies a package representing the entire system being modelled

Case Study: Packages



Collaboration Diagrams

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Collaboration

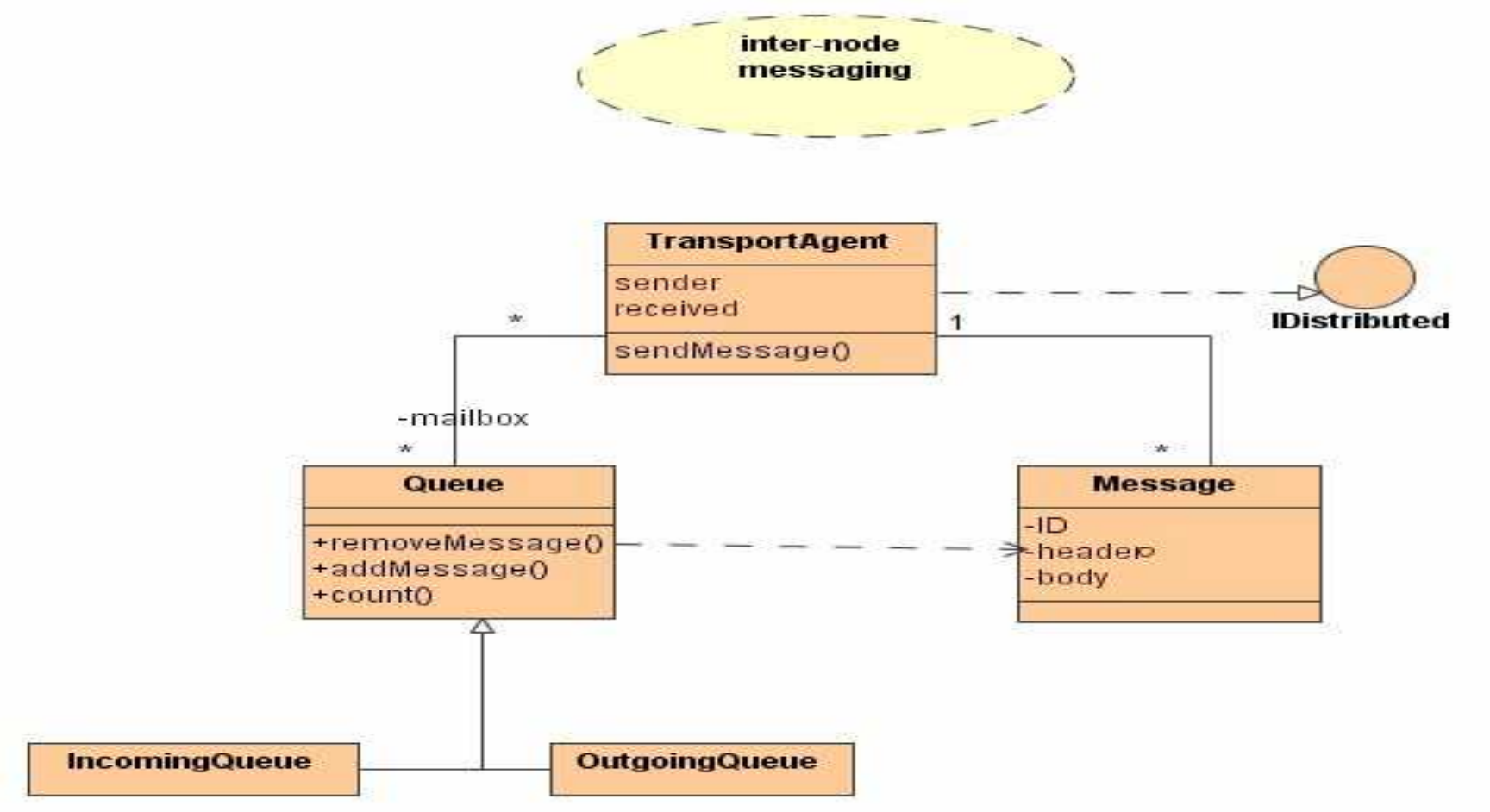
Definition

A collaboration is a society of classes, interfaces, and other elements that work together to deliver or provide some cooperative behaviour that is bigger than the sum of all its parts.

Collaboration in the context of software architecture allows you to name a conceptual chunk that encompasses both static and dynamic aspects.

It specifies the realization of a use case and operations by a set of classifiers and associations playing specific roles used in a specific way.

Example: Collaboration



Inter-node Messaging Collaboration and its details

Collaboration Names

Every collaboration has a name that distinguishes it from other collaborations.

Collaboration names are nouns or short noun phrases, typically first letter of the noun is capitalized.

Example: *"Inter-node Messaging"* or *"Application Submission"*

Collaboration – Structural

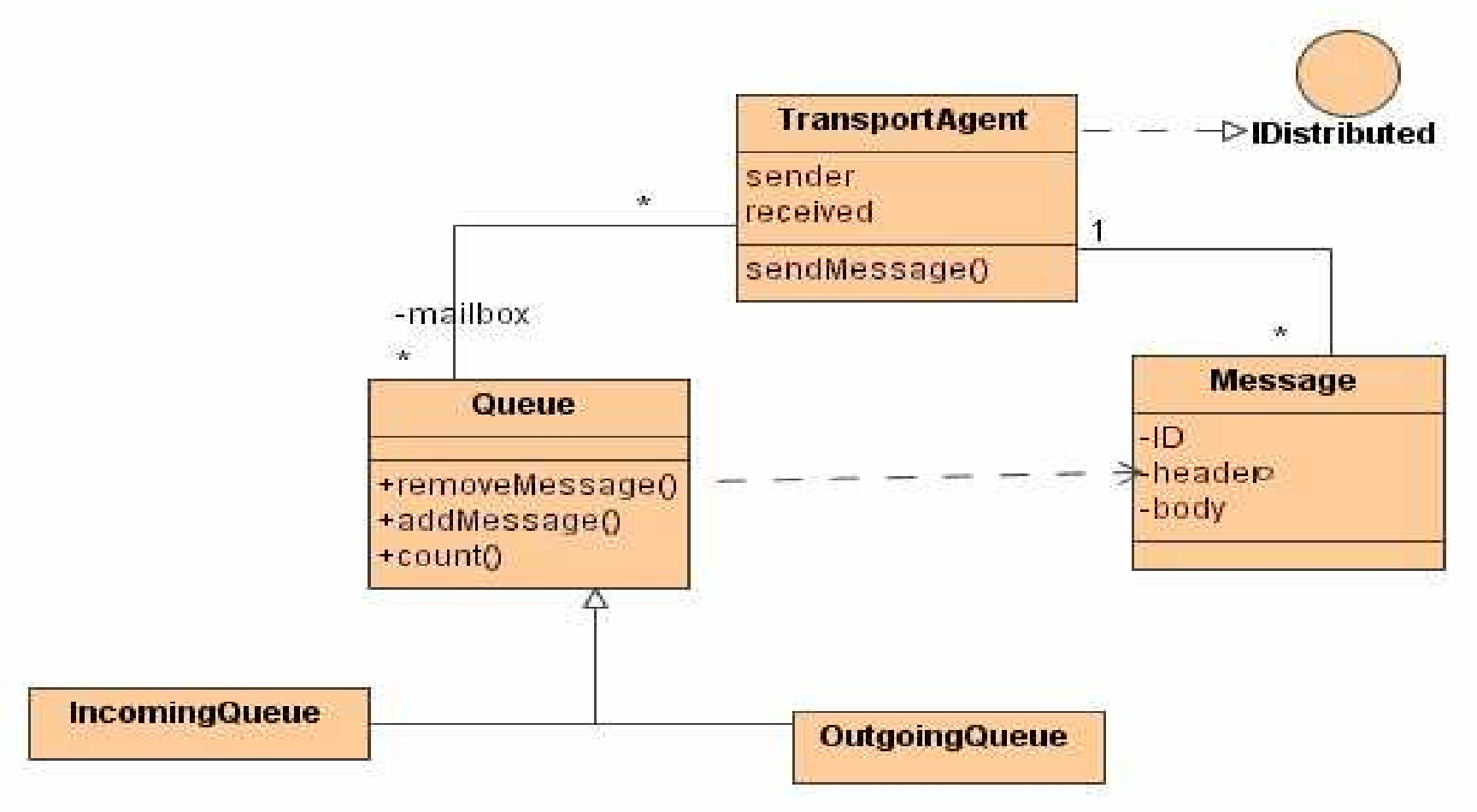
Structural aspects of collaborations specifies the classes, interfaces, and other elements that work together to carry out a named collaboration.

It includes a combination of classifiers, such as classes, interfaces, components and nodes.

Classifiers (Classes, components, nodes etc.) may be organized using all the usual UML relationships including association, generalization, and dependencies.

Unlike packages or subsystems, a collaboration does not own any of its structural elements , it only references the classes, interfaces, components etc. declared elsewhere.

Example: Collaboration 1



Structural View of Collaboration

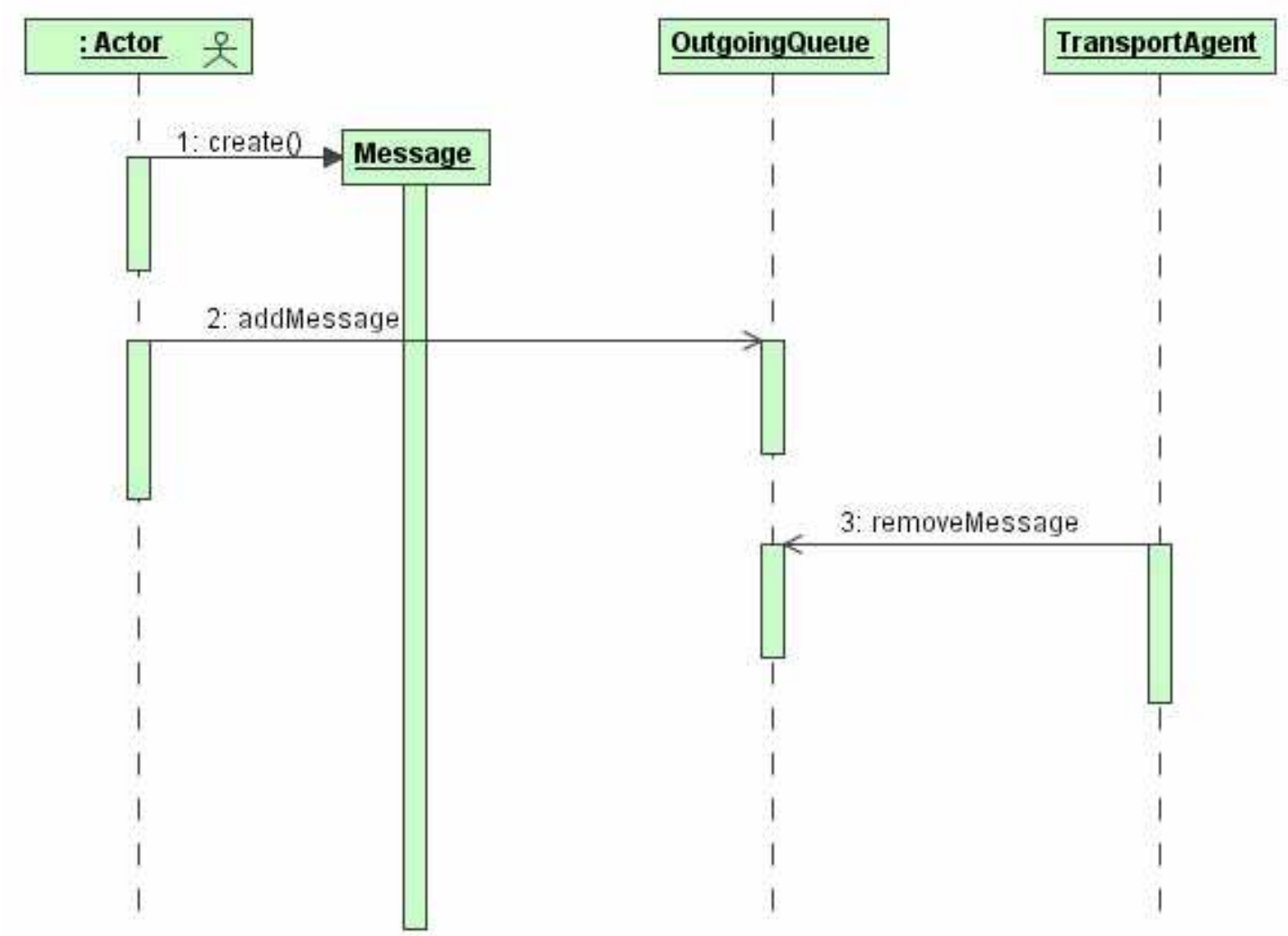
Collaboration – Behavioural

The behavioural aspect of a collaboration is rendered using interaction diagram.

An interaction diagram specifies an interaction that represents a behaviour composed of a set of messages that are exchanged among a set of objects to accomplish a specific purpose.

An interaction context is provided by its enclosing collaboration which establishes the classes, interfaces, components, nodes and other structural elements whose instances participate in that interaction.

Example: Collaboration 2



Organizing Collaborations

Collaborations are essential in system architectures.

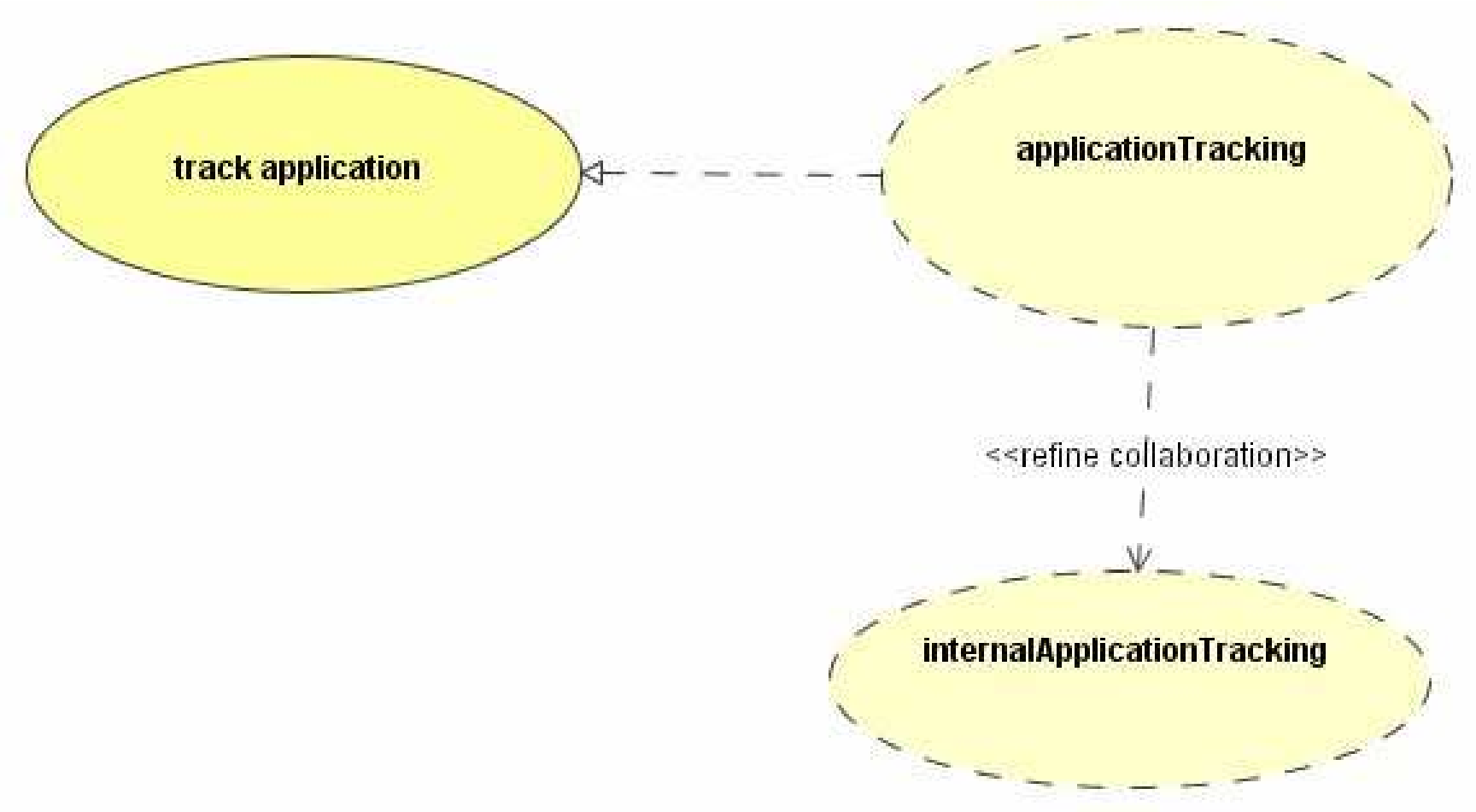
A well structured OO system is composed of modestly sized regular set of collaborations.

There are two sets of relationships in collaborations:

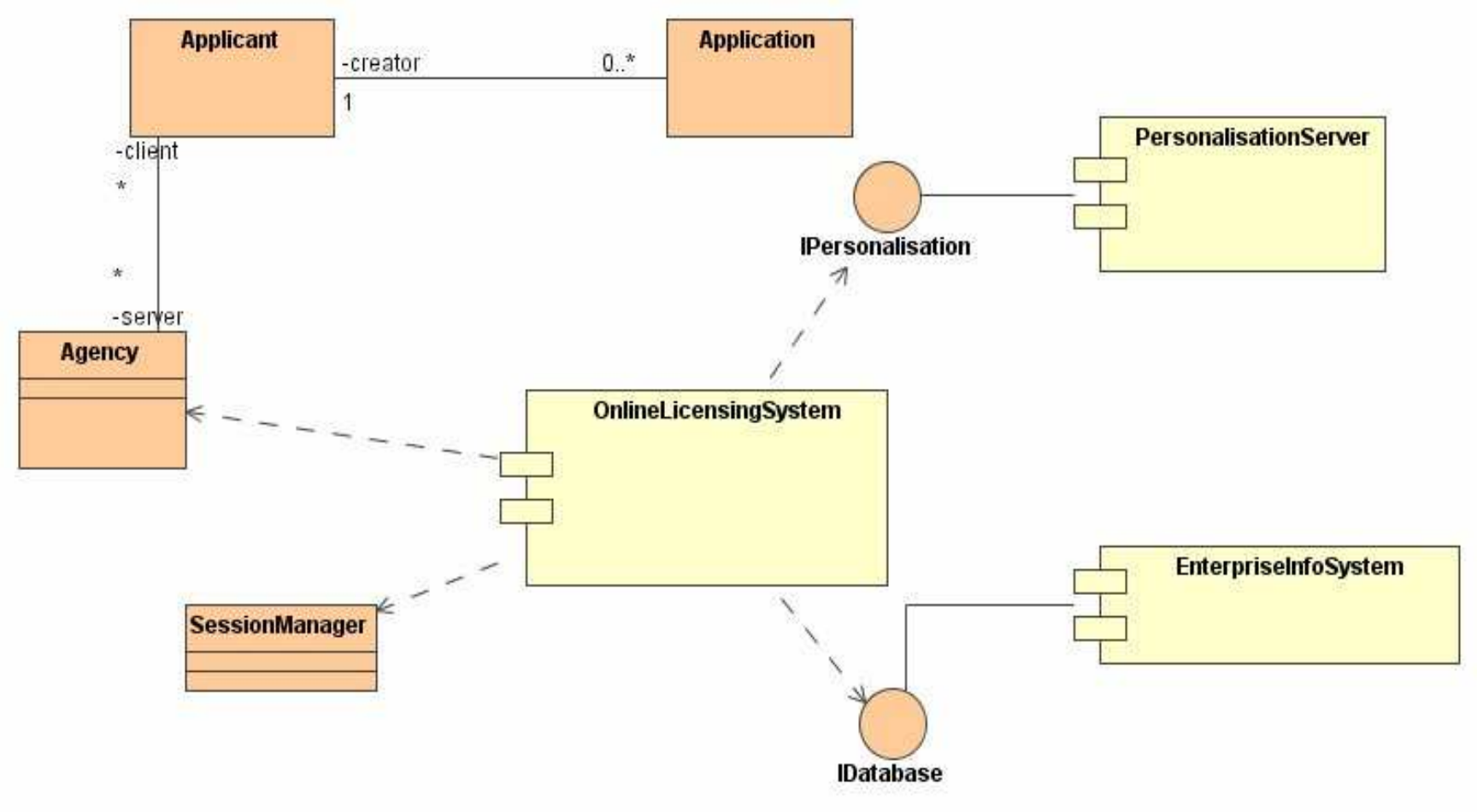
a) relationship between collaboration and what it realizes (use cases or operations)
b) relationship between collaborations

These relationships correspond to realization and refinement respectively.

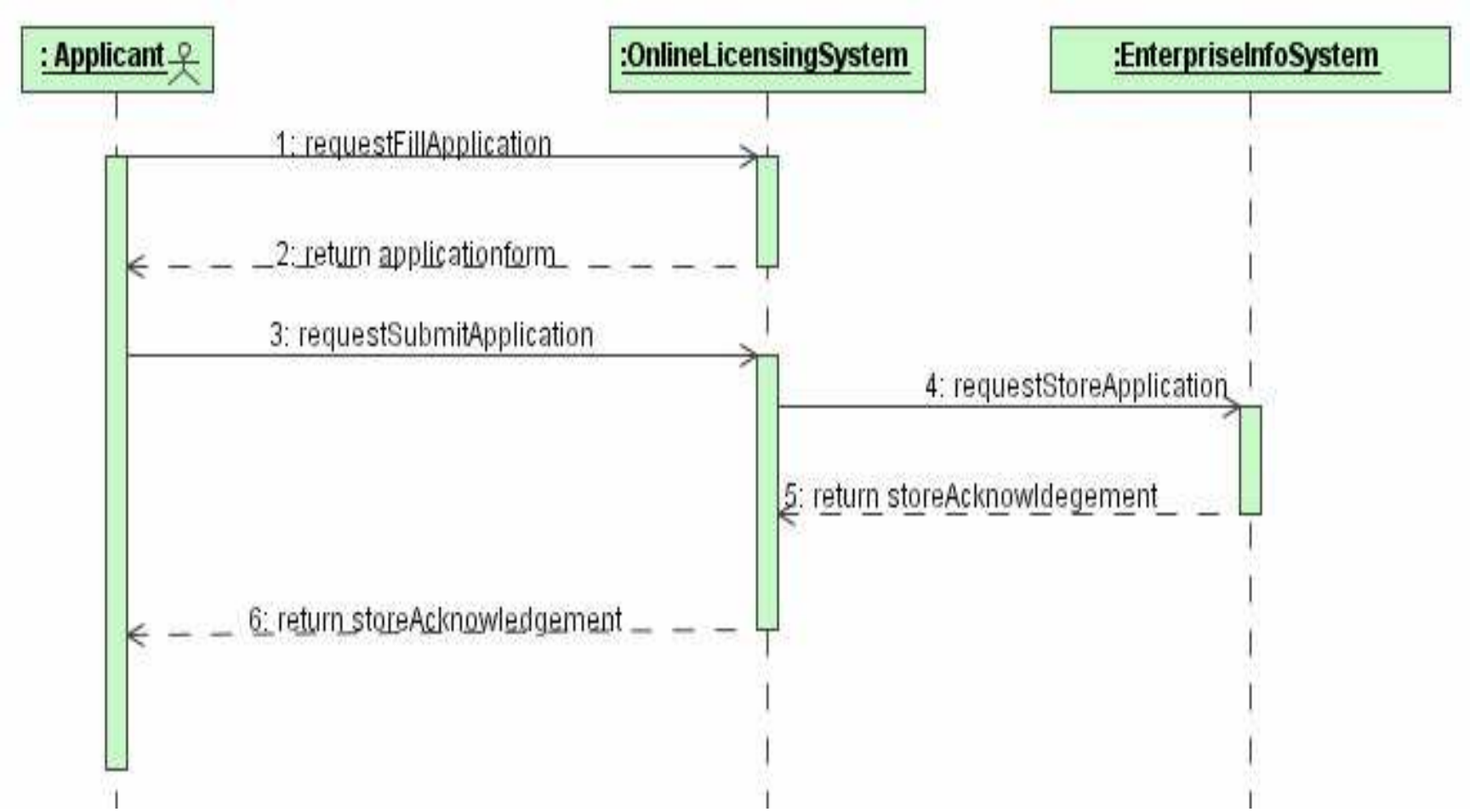
Organizing Collaborations



Case Study: Structural



Case Study: Behavioural



Behavioural View of a Collaboration which implements a "submit application online" use case

Collaboration Diagram

Definition

A collaboration diagram shows the interactions organized around the structure of a model, using either classifiers (e.g. classes) and associations or instances (e.g. objects) and links.

It presents a collaboration, containing a set of roles to be played by instances as well as their required relationships given in a particular context.

It also presents an interaction which defines a set of messages (stimuli) specifying the interaction between instances playing a certain role within a collaboration to achieve a desired result.

Notation 1

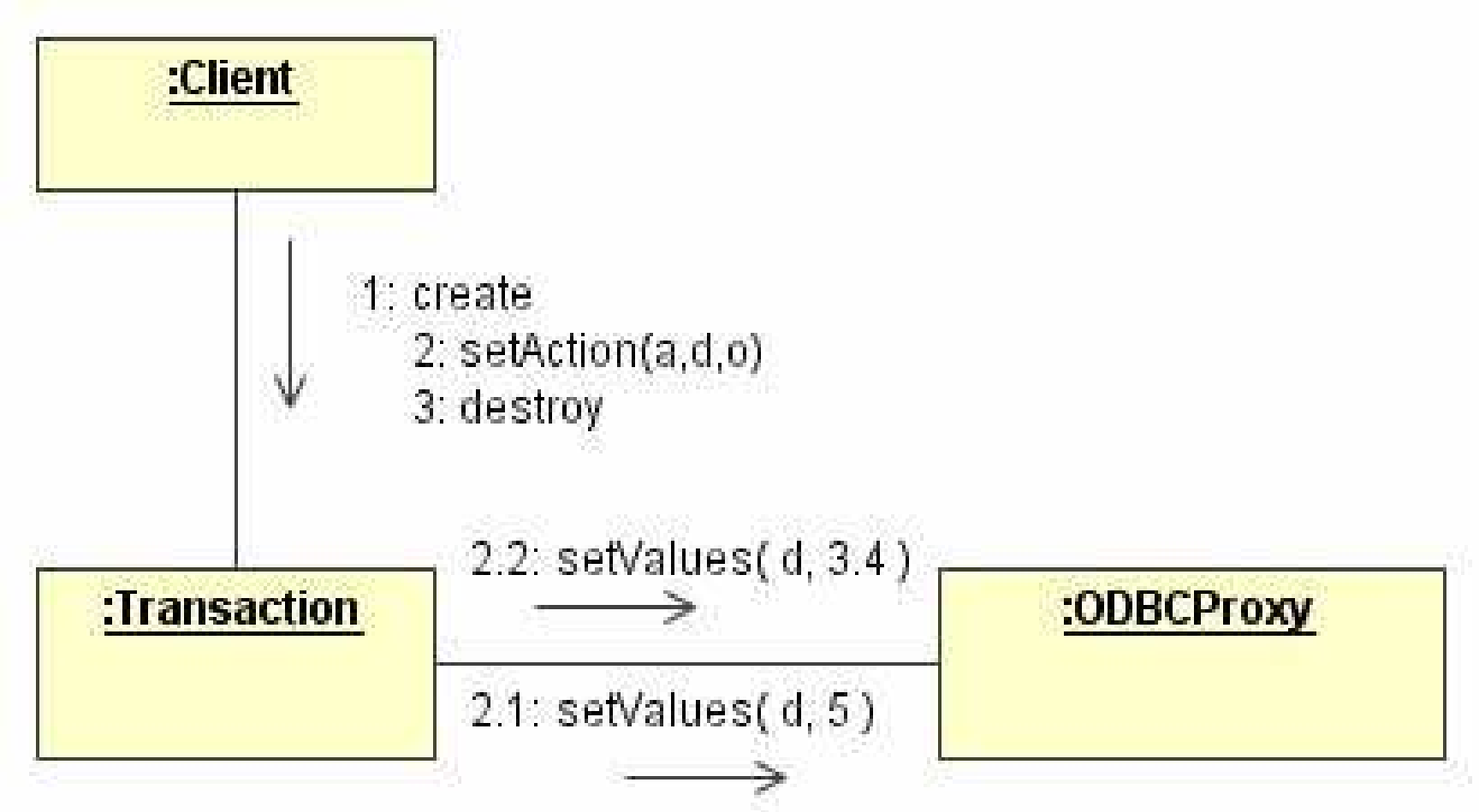
1) A collaboration diagram shows a graph of either instances linked to each other or classifiers and associations.

2) Navigability is shown using arrow heads on the lines representing links.

3) An arrow next to a line indicates a stimuli or message flowing in the given direction.

4) The order of interaction is given with a number starting from *1.*

Notation 2


:Client
1: create
2: setAction(a,d,o)
3: destroy
:Transaction
2.2: setValues(d, 3.4)
2.1: setValues(d, 5)
:ODBCProxy

Notation 3

A collaboration diagram without any interaction shows the "context" in which an interaction can occur.

A collection of standard constraints may be used to show whether an instance or a link is created or destroyed during the execution:

- {new} – instances and links created during execution
- {destroyed} – instances and links destroyed during execution
- {transient} - instances and links created and destroyed during the execution

Collaboration Diagrams Types

Collaboration diagrams may be presented at two levels:

1) instance level – instances and stimuli
2) specification levels – roles

Instance level (or context):

- defined in terms of a static structure of instances and their relationships and possibly the stimuli

Specification level:

- shows a collaboration

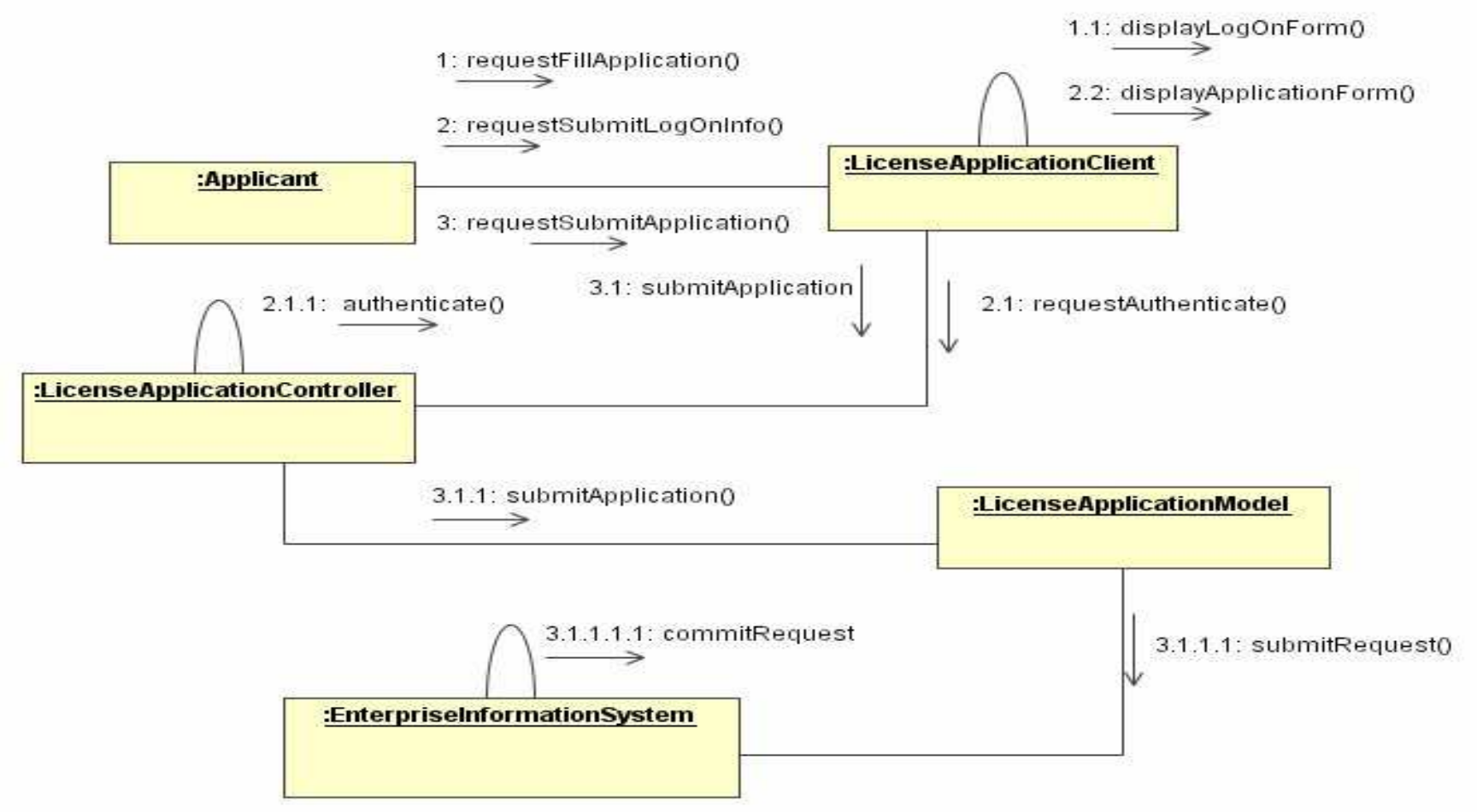
Instance Diagram

It shows a CollaborationInstanceSet – a collection of object boxes and lines mapping to instances and links, respectively.

It may include arrows attached to lines that corresponds to stimuli communicated over links.

It also shows the instances relevant to the realization of an Operation.

Example: Instance Level



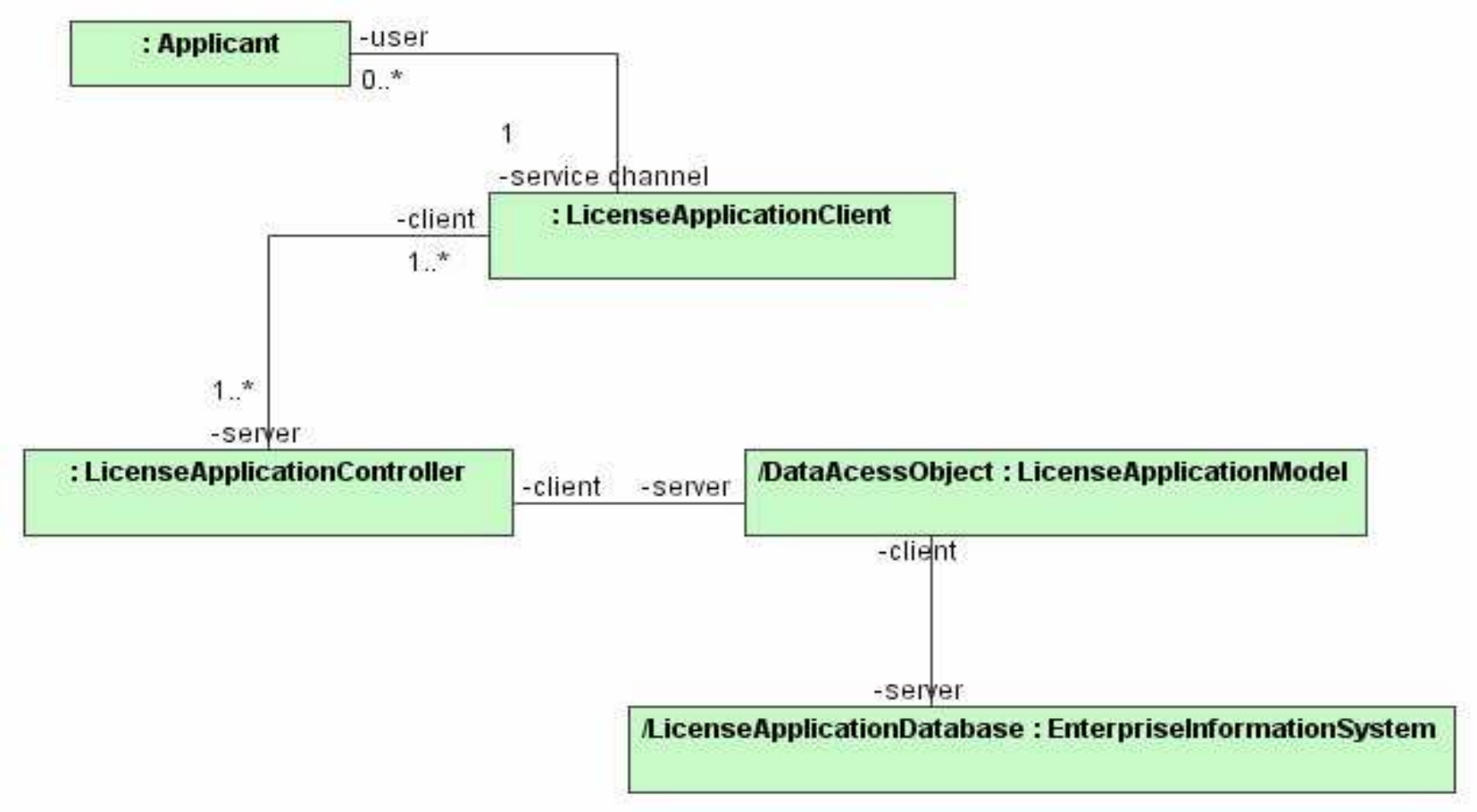
Specification Diagram

It shows a collaboration.

It provides the roles defined within a collaboration.

The arrows attached to lines map into messages.

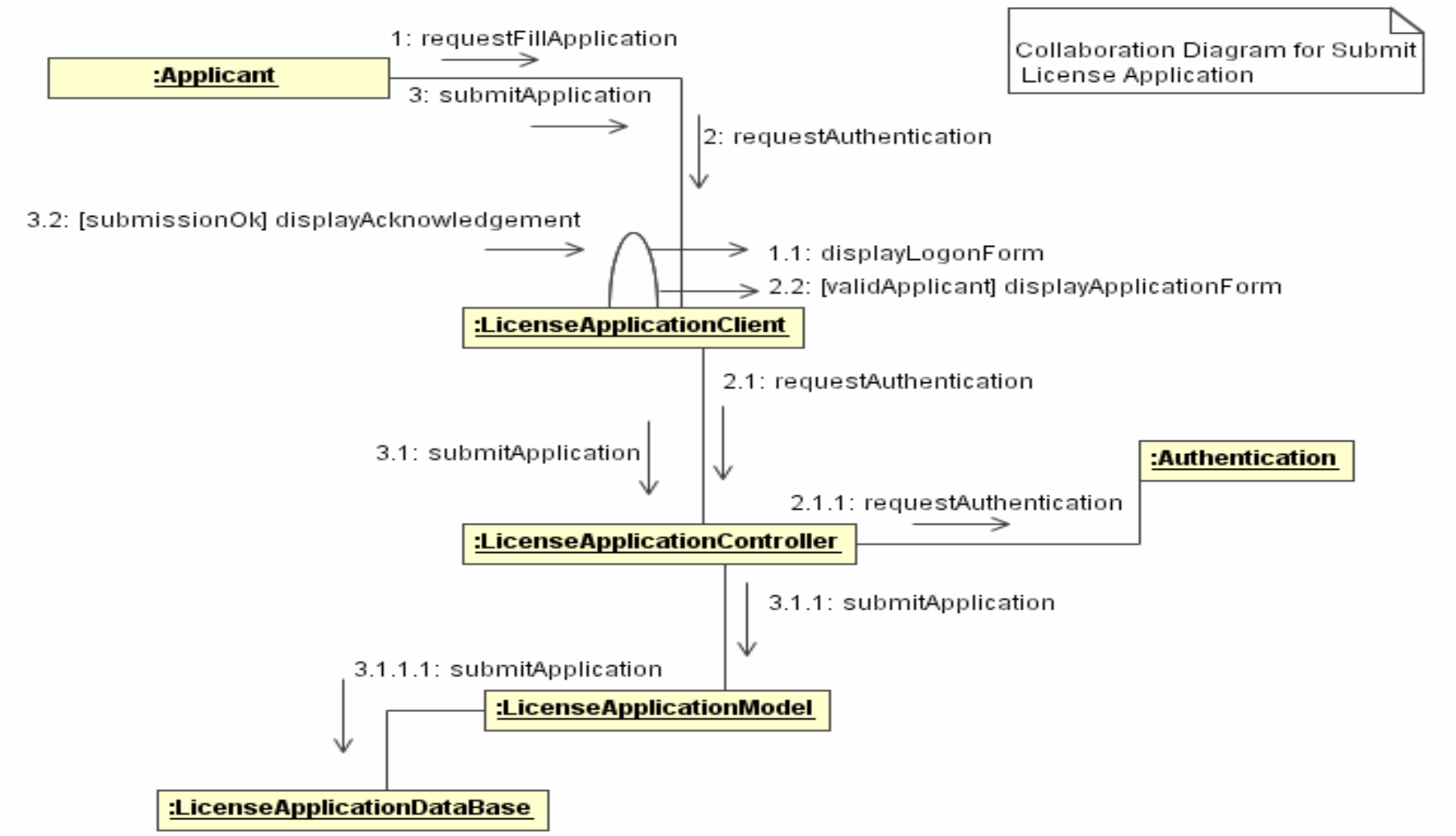
Example: Specification Level



Collaboration Roles Syntax

Syntax	Explanation
:C	Un-named instance originating from the Classifier C
/R	Un-named instance playing the role R
/R:C	Un-named Instance originating from C, playing role R
O/R	Instance named O playing role R
O:C	Instance named O originating from the Classifier C
O/R:C	An instance named O originating from the Classifier C, playing role R
O	An instance named O

Example: Case Study



Component Diagrams

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Component

Definition

A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.

It encapsulates the implementation of classifiers residing in it.

A single element may reside in multiple components.

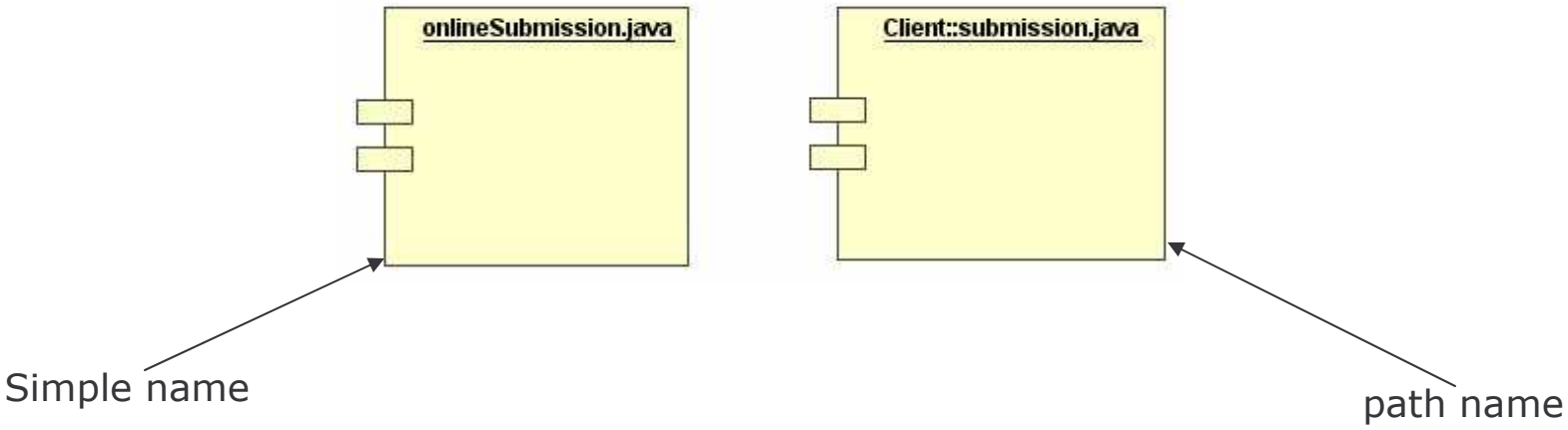
A component does not have its own features, but a mere container for its elements.

Components are replaceable or substitutable.

Component Names

Every component must have a name that distinguishes it from other components.

Name is a textual string which may be written as a simple name or path name.



Component and Classes 1

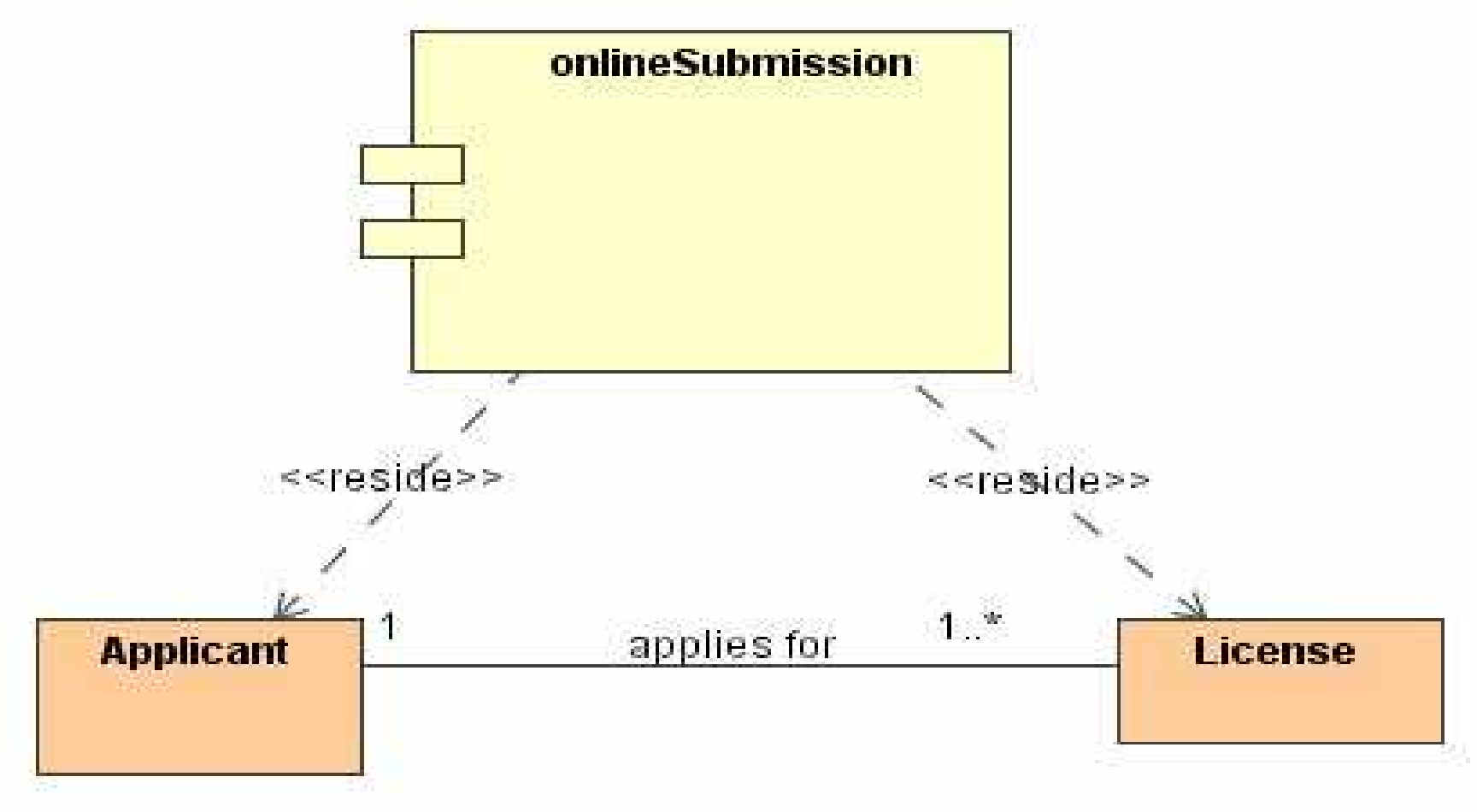
similarities

1) both may realize a set of interfaces
2) both may participate in dependencies, generalizations and associations
3) both may be nested
4) both may have instances
5) both may participate in an interaction

differences

1) classes represent logical abstraction while components represent physical things
2) components represent the physical packaging of logical components and are at a different level of abstraction
3) classes may have attributes and operations whereas components only have operations reachable only through their interfaces

Component and Classes 2



Components and Interfaces

Definition

An interface is a collection of operations that are used to specify a service of a class or components.

Using component technologies (COM+, CORBA, EJB), physical implementation may be decomposed by specifying interfaces that represent the major seams of the system.

Components that realize these “seams” are created or provided in implementation.

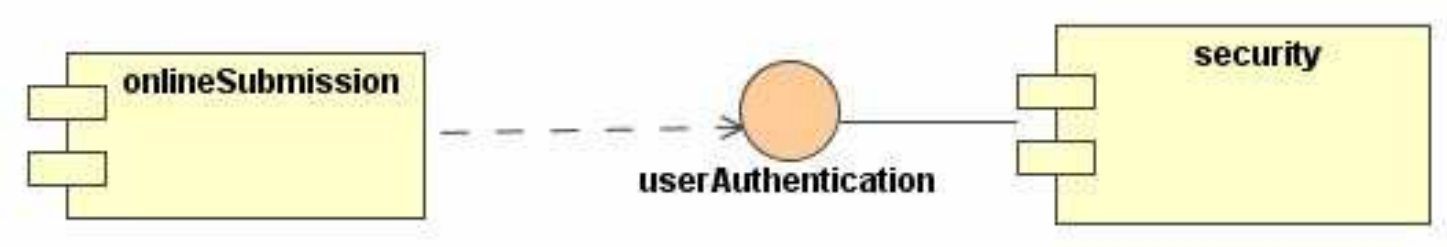
Interfaces allow for the deployment of systems whose services are somewhat location independent and replaceable.

Component Notation 1

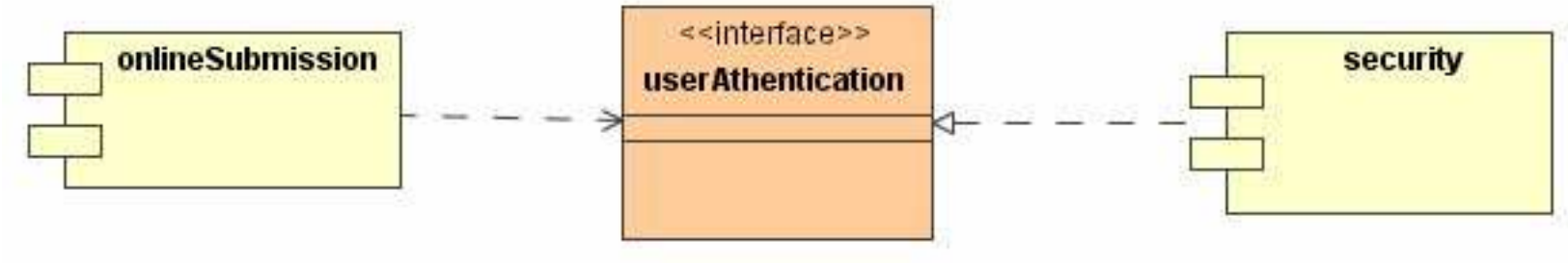
Relationship between a component and its interface may be shown in two ways:

1) style 1
    a) interface in elided iconic form
    b) component that realize the interface is connected to the interface using an elided realization relationship
2) style 2
    a) interface is presented in an enlarged form, possibly revealing its operations
    b) realizing component is connected to it using a full realization relationship

Component Notation 2



Style 1: elided iconic or short hand



Style 2: long hand

Export Interfaces

Definition

An interface that a component realizes is called an export interface, meaning an interface that the component provides as a service to other components.

Components may export more than one interfaces.

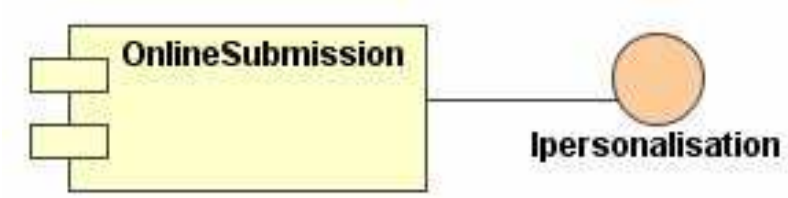
Import Interface

Definition

An interface that a component uses is called the import interface, meaning the interface that the component conforms to and builds on.

Components may import more than one interface.

They may also import and export interfaces at the same time.



Component Replaceability

The key intent of any component based OS facility is to permit the assembly of system from binary replaceable parts.

A system can be:

a) created out of components

b) evolved by adding new components and replacing old ones without rebuilding the system

Interfaces allow easy reconfiguration of component based systems.

Types of Components

There are three basic types of components.

1) Deployment:

component necessary and sufficient to form an executable system such as DLLs and EXEs

2) Work product component:

residue of development process consisting of things like source code files and data files from which deployment components are created

3) Execution component:

created as a consequence of an executing system

Component Stereotypes 1

1) executable: specifies that a component may be executable on a node

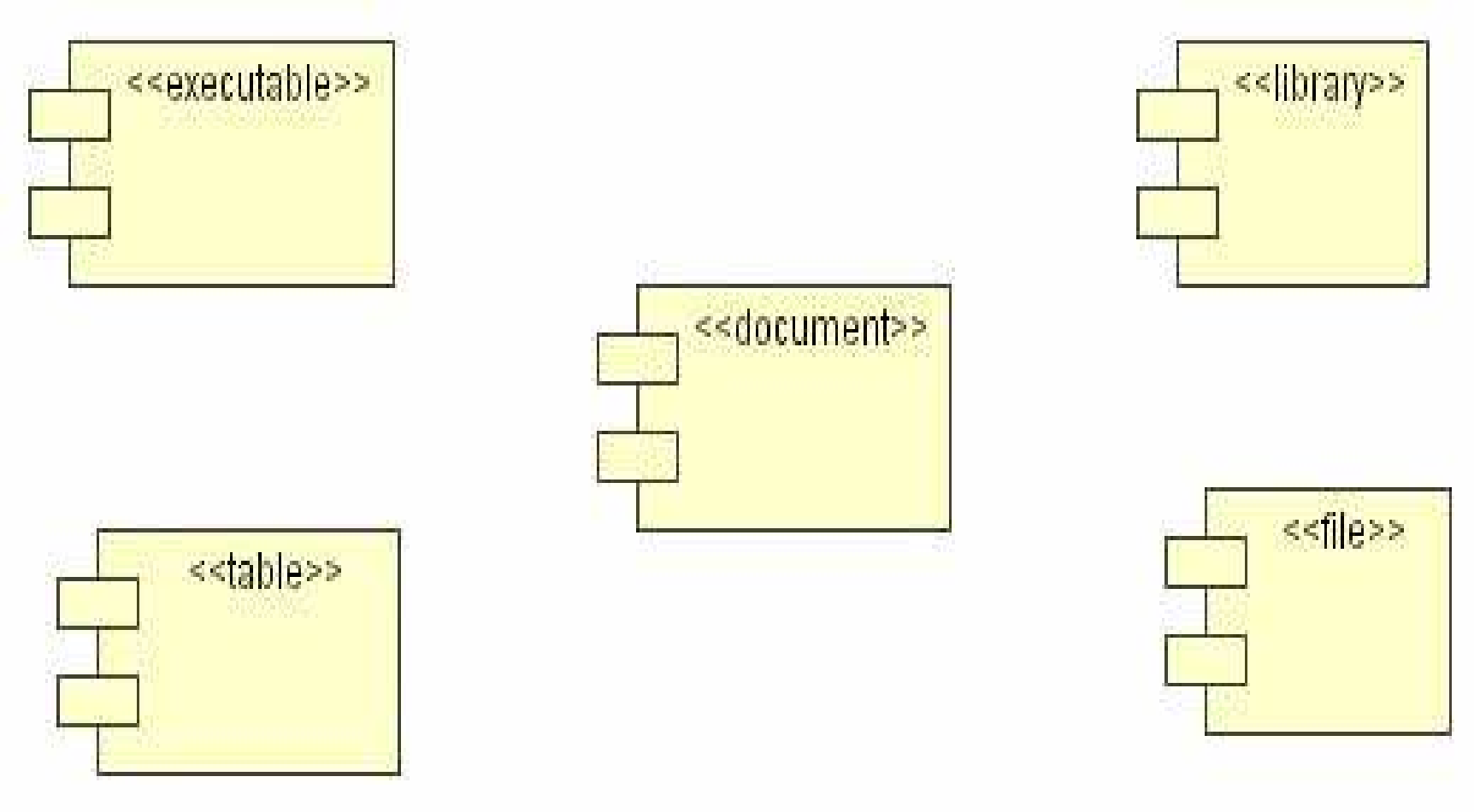
2) library: specifies a static or dynamic object library

3) table: specifies a component that represent a database table

4) file: specifies a component that represents a document containing source code or data

5) document: specifies a document that represent a document

Component Stereotypes 2



Component Diagram

Definition

A component diagram shows the dependencies among software components including the classifiers that specify them and the artifacts that implements them.

It models the physical implementation of the software specified by the logical requirements of the class diagram.

It indicates the relationship between the classes that specify the requirements for the components and the artifacts that implements them.

Component Diagram Notation 1

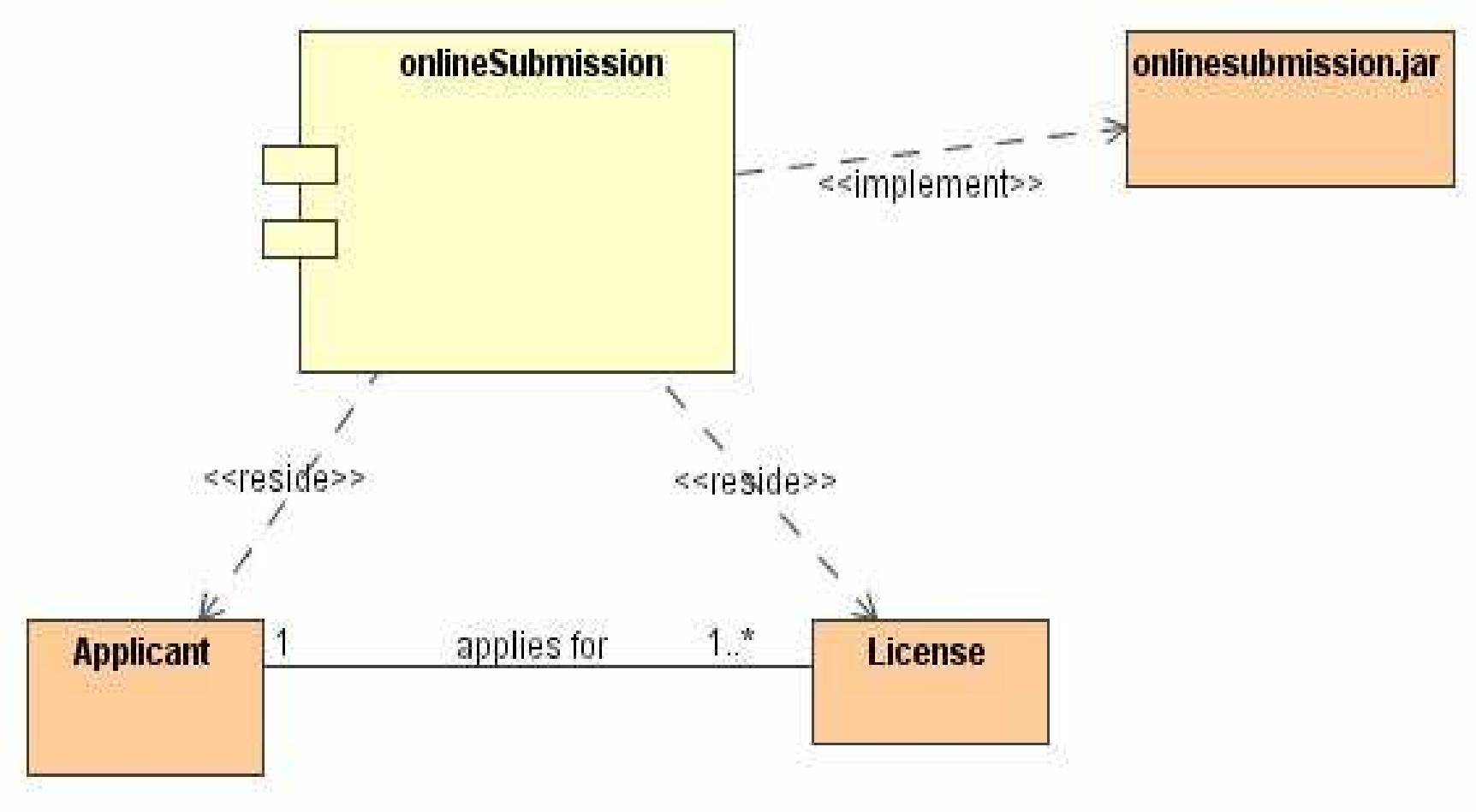
A graph of components, classifiers and artifacts connected by dependency relationships.

Components may also be connected using physical containment representing composition.

Classifiers that specify the components can be connected to them by physical containment or by a <<reside>> relationship.

Artifacts that specify components can be connected to them by an <<implement>> relationship.

Component Diagram Notation 2



Component Diagram Content

1) specifying classes

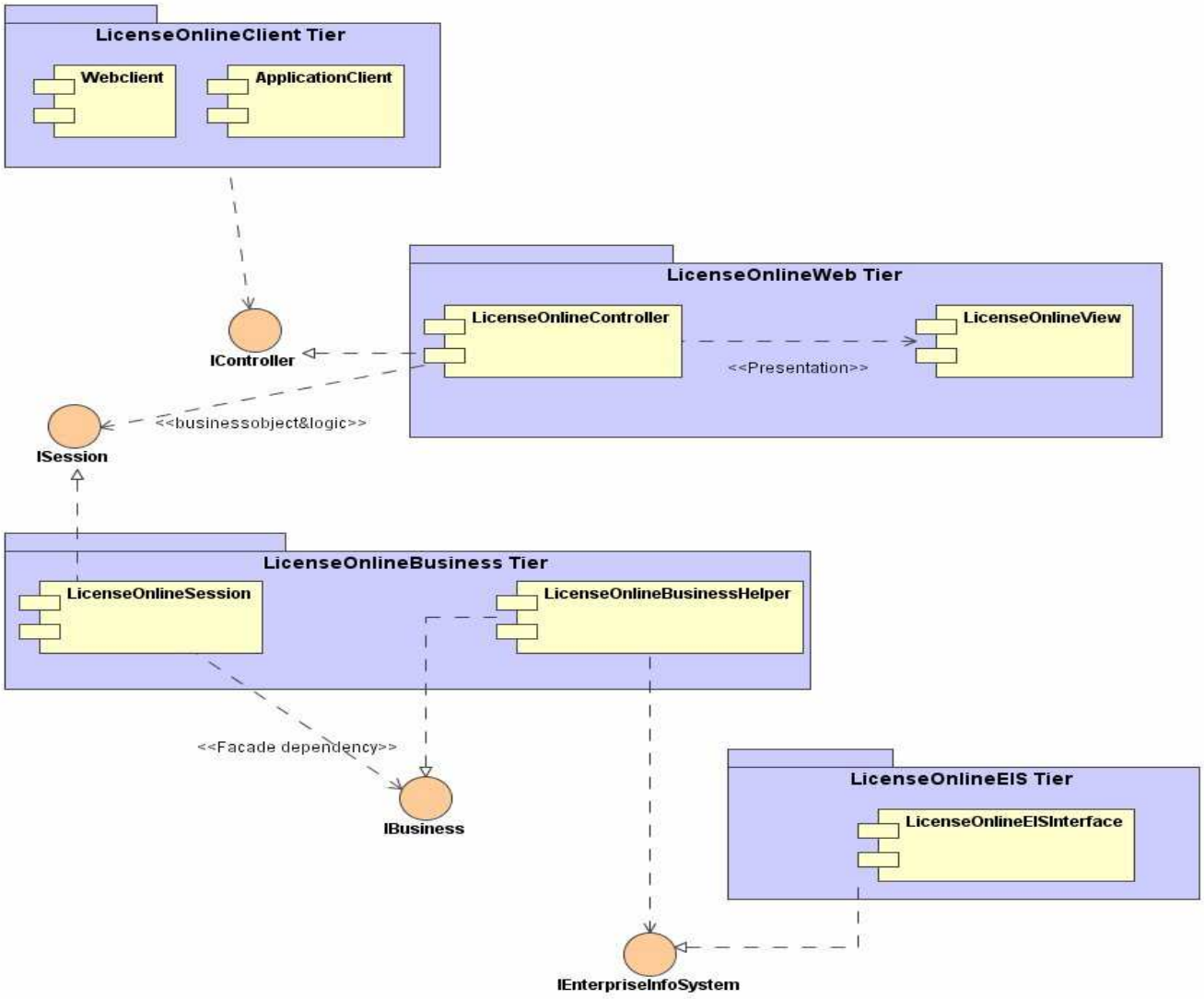
2) implementing artifacts

3) components

4) interfaces

5) dependencies, generalization, association and realization relationships

Example: Component Diagram



Architectural Patterns

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Templates

Definition

A template is a parameterized element with one or more unbound parameters defining a family of elements.

Templates are used to generalize the structure and behaviour of a society of elements.

A template is not directly usable, because it has unbound parameter and must be bound to actual values before it is used through a binding relationship.

Template Notation

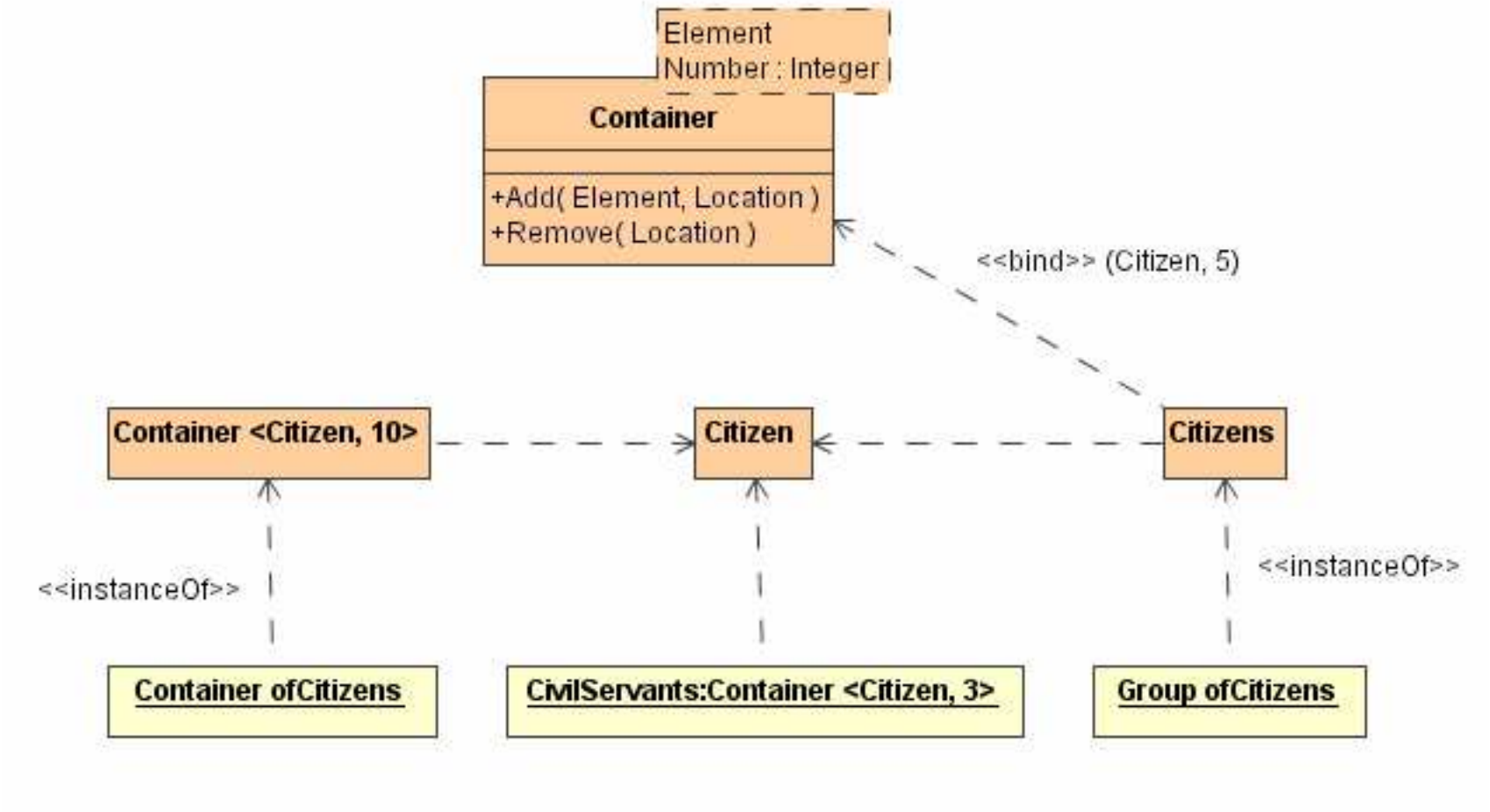
1) a template class is depicted with a small dashed rectangle superimposed on the upper right hand corner of the rectangle for the class

2) the formal parameters are comma separated while the actual parameters are specified through the <<bind>> dependency relations

3) formal parameters may also be listed one per line including the parameter's name, implementation type and an optional value

4) a missing type for a formal parameter indicates a class type

Example: Template



Patterns – What are they?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way the you can use this solution a million times over, without doing it the same way twice."

– Christopher Alexander (Patterns in buildings and towns)

Patterns Definition

Definition

A generalized solution to a problem in a given context where each pattern has a description of the problem, solution context in which it applies, and heuristics including use advantages, disadvantages and trade-offs.

Patterns identify, document, and classify best practices in OOD.

They generalize the use and application of a society of elements.

Solution is described using static structure, identifying participating elements and their relationships, and a dynamic behaviour, identifying how elements collaborate and interact.

Notation and Description

Notation:

a pattern is depicted as a template or parameterized collaboration, where formal parameters are roles used in the collaboration

The description of a pattern uses four elements:

a) pattern name

b) problem

c) solution

d) consequences

Pattern Elements

1) name: a handle which describes the design problem, its solution, and consequences in a word or two

2) problem: describes when to apply the pattern and explains the problem and its context

3) solution: elements that must make up the design, their relationships, responsibilities and collaborations.

4) consequences: associated trade-offs in using the pattern.

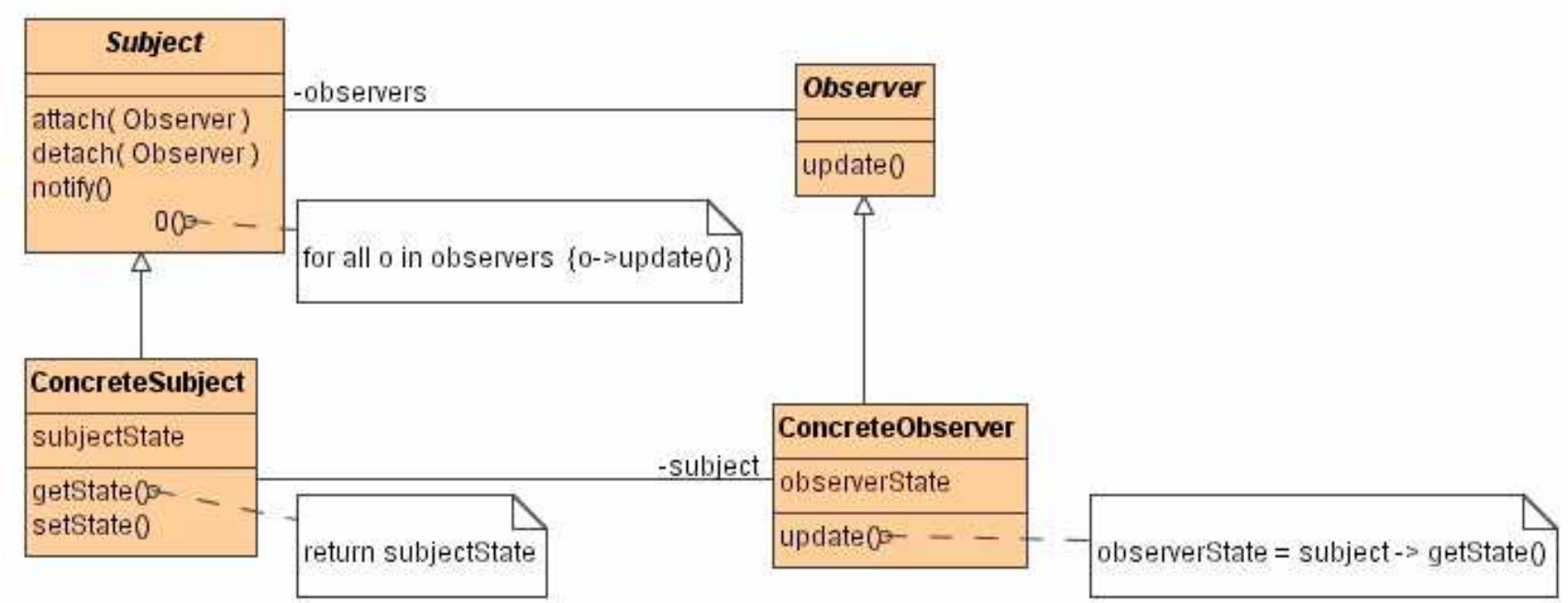
Example: Pattern 1

1) Name: Observer

2) Problem:
   a) When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
   b) When a change to one object requires changing others, and you don't know how many objects need to be changed
   c) When an object should be able to notify other object without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

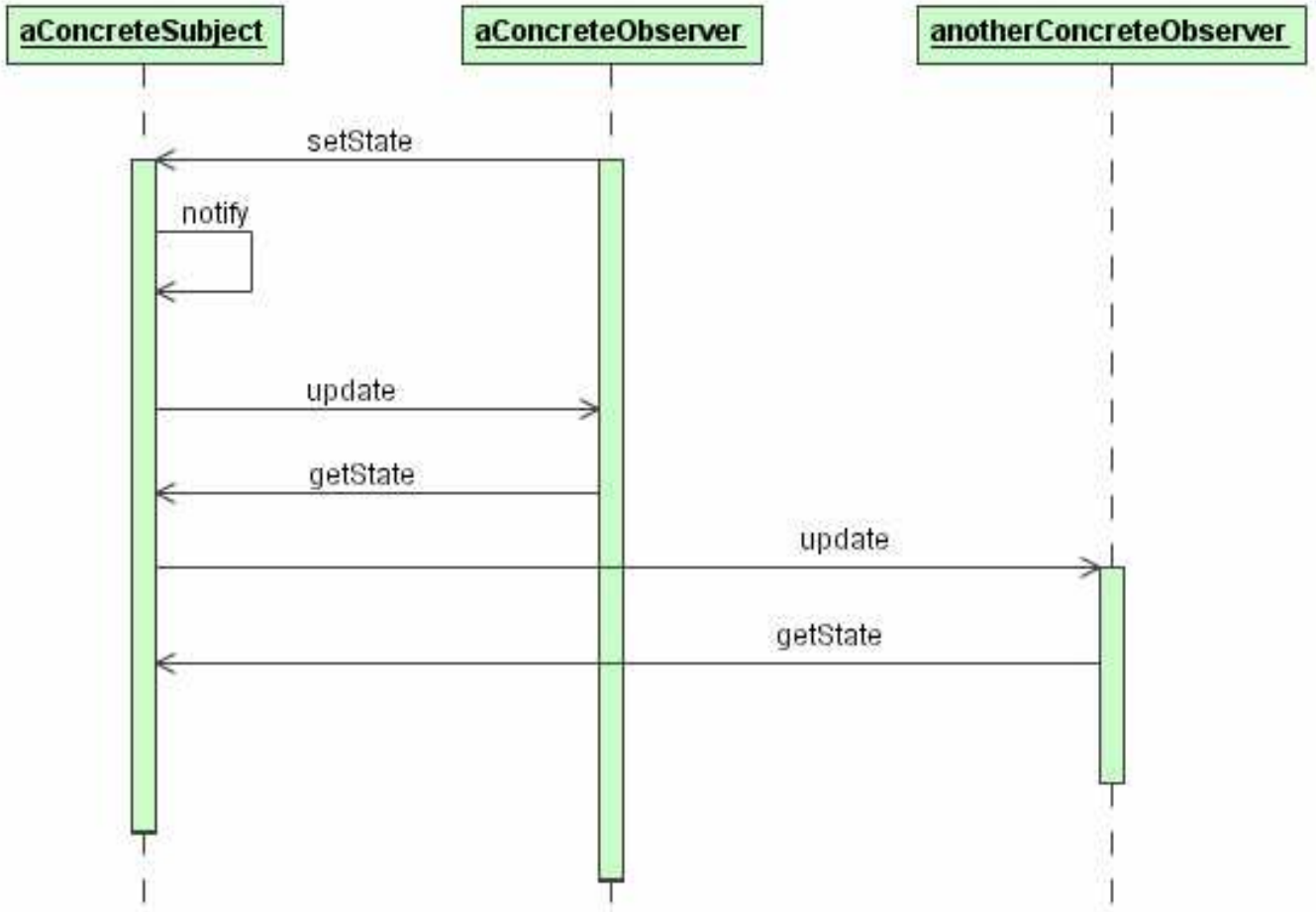
Example: Pattern 2

3) Solution – Structural:



Example: Pattern 3

3) Solution - Behavioural:



Example: Pattern 4

4) Consequences:

a) abstract coupling between Subject and Observer

b) support for broadcast communication

c) unexpected updates

Types of Patterns

Creational Patterns:

apply to instantiation of objects and are concerned with decoupling the type of objects from the process of constructing that object.

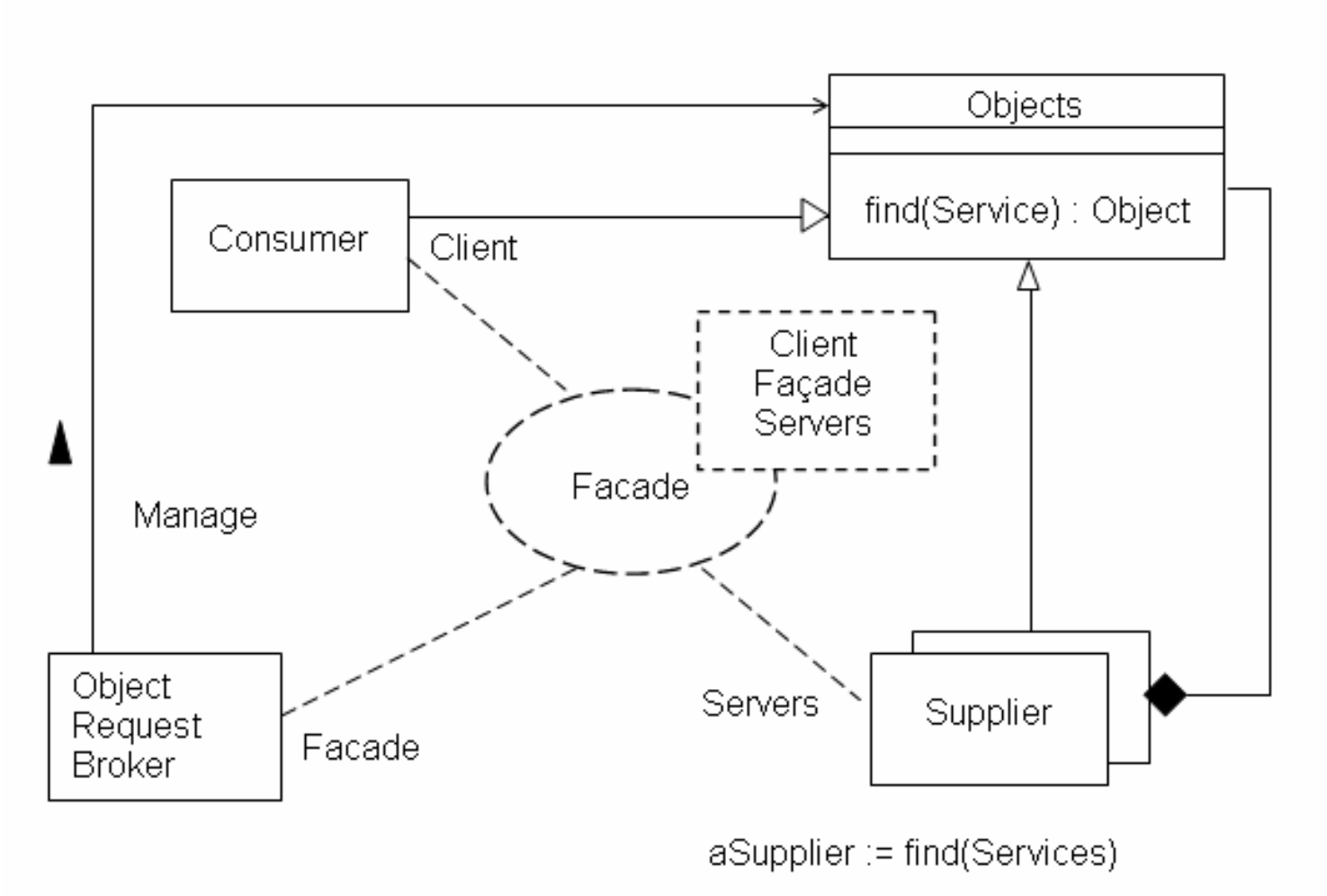
Structural and Architectural patterns:

apply during the organization of a system and concerned with larger structures composed from smaller structures.

Behavioural Patterns:

assigning responsibilities among a collection of objects.

Example: Façade Pattern



Framework 1

Definition

A Framework is a reusable software architecture that provides the generic structure and behaviour for a family of software applications, along with a context that specifies their collaboration and use.

A framework is also a collection of patterns defined as template collaborations with supporting elements.

It is a skeletal solution in which specific element must be plugged in order to establish an actual solution.

Frameworks are depicted as packages stereotyped with the “framework” keyword.

Framework 2

1) frameworks are made up of a set of related classes that can be specialized or instantiated to implement an application

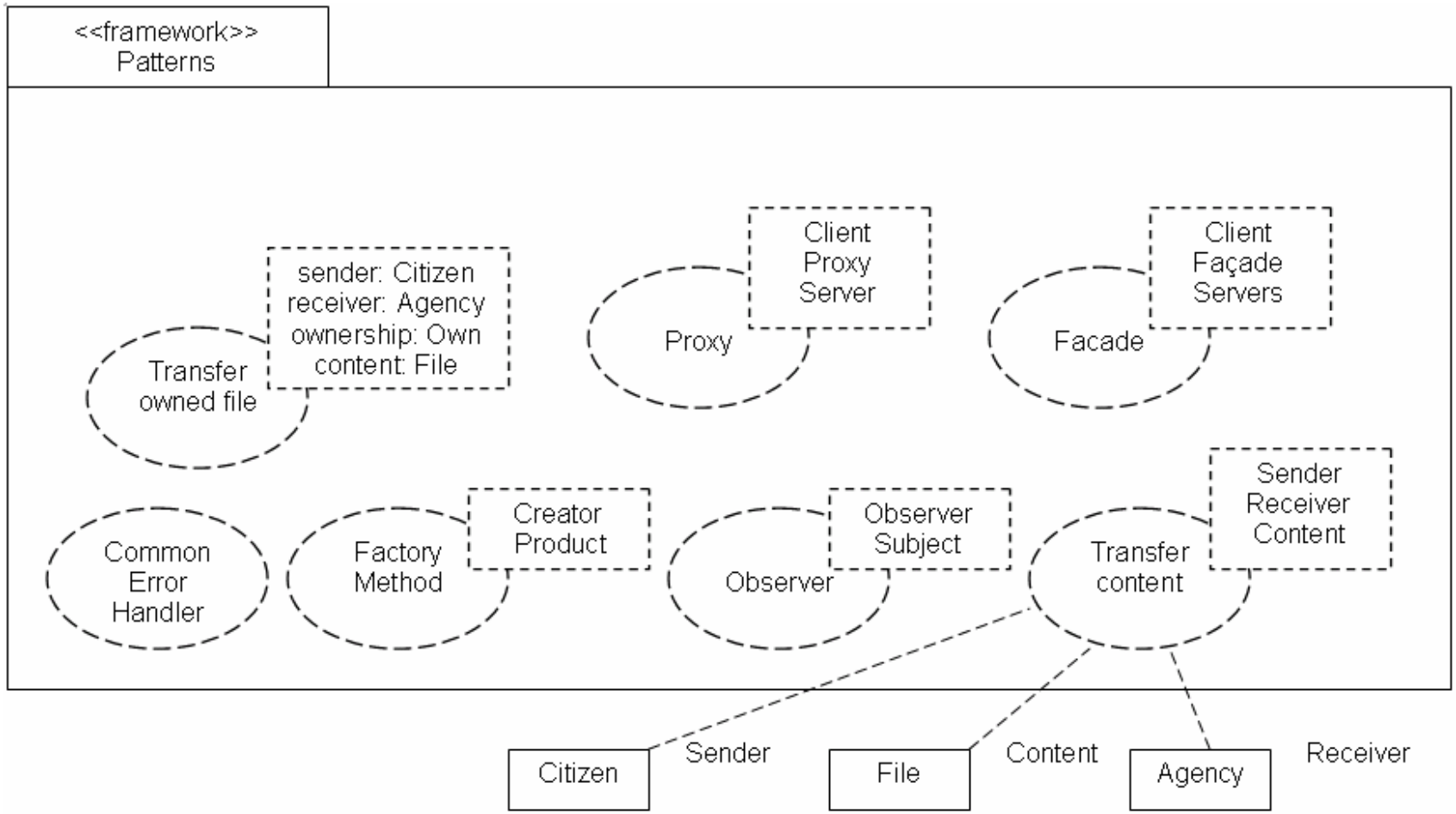
2) they lack the necessary application specific functionality and therefore not immediately useful

3) they may be seen as a prefabricated structure or template of a working application in which “plug-points” or “hot spots” are not implemented or are given overidable implementations

4) patterns can be used to document frameworks

5) they are physical realization of one or more patterns

Example: Framework



Architecture Patterns/Styles

1) Repository
2) Model/View/Controller
3) Client-Server
4) Peer-to-Peer
5) Pipe-and-Filter

Repository Architecture 1

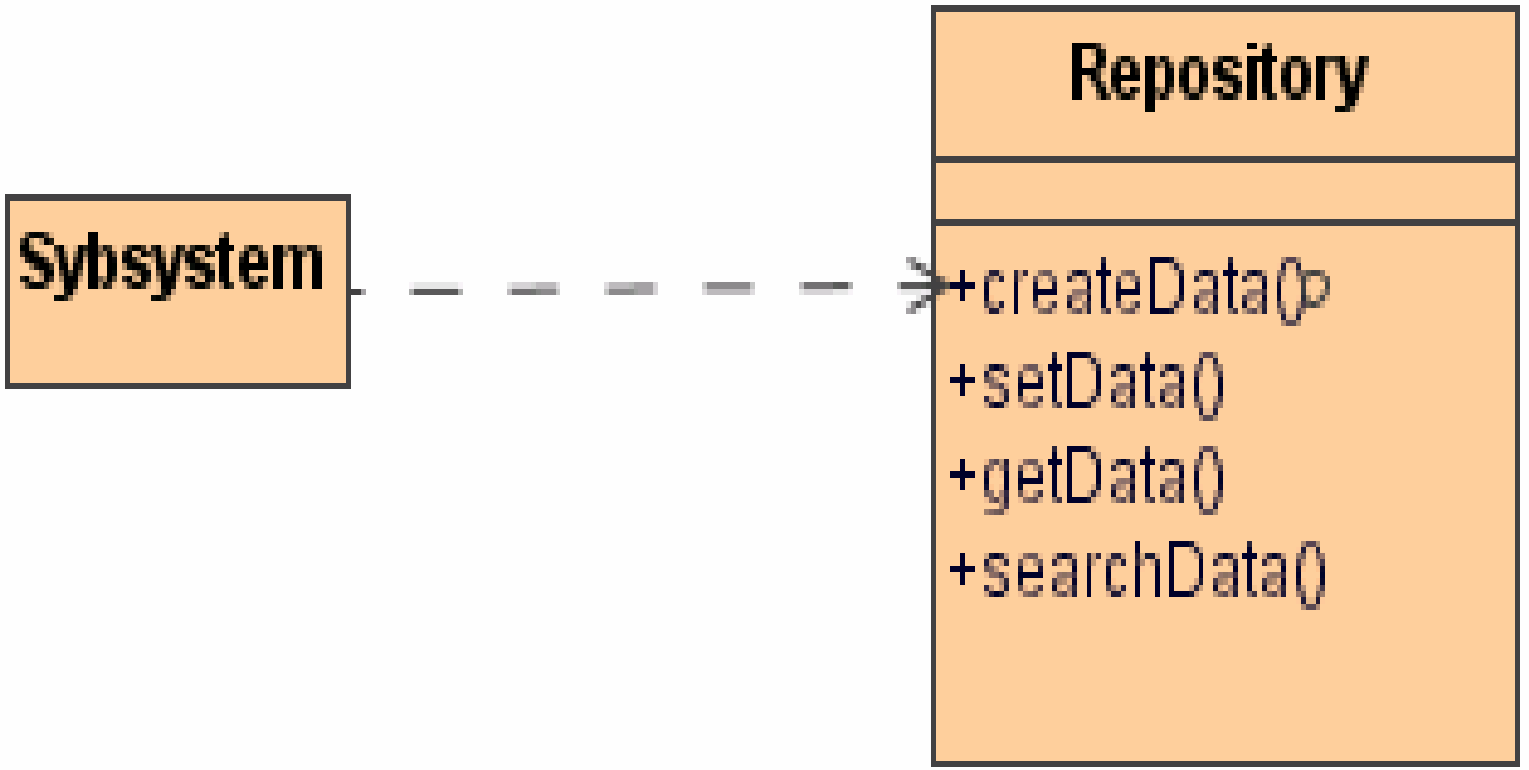
1) subsystems access and modify data from a single data structure called the repository

2) subsystems are relatively independent and interact through the central data structure

3) control flow can be dictated either by the central repository or by the subsystem

4) it is usually employed in database applications, compilers and software development environment

Repository Architecture 2



UML Class Diagram Describing Repository Architecture

MVC Architecture 1

Subsystems are classified into three different types:

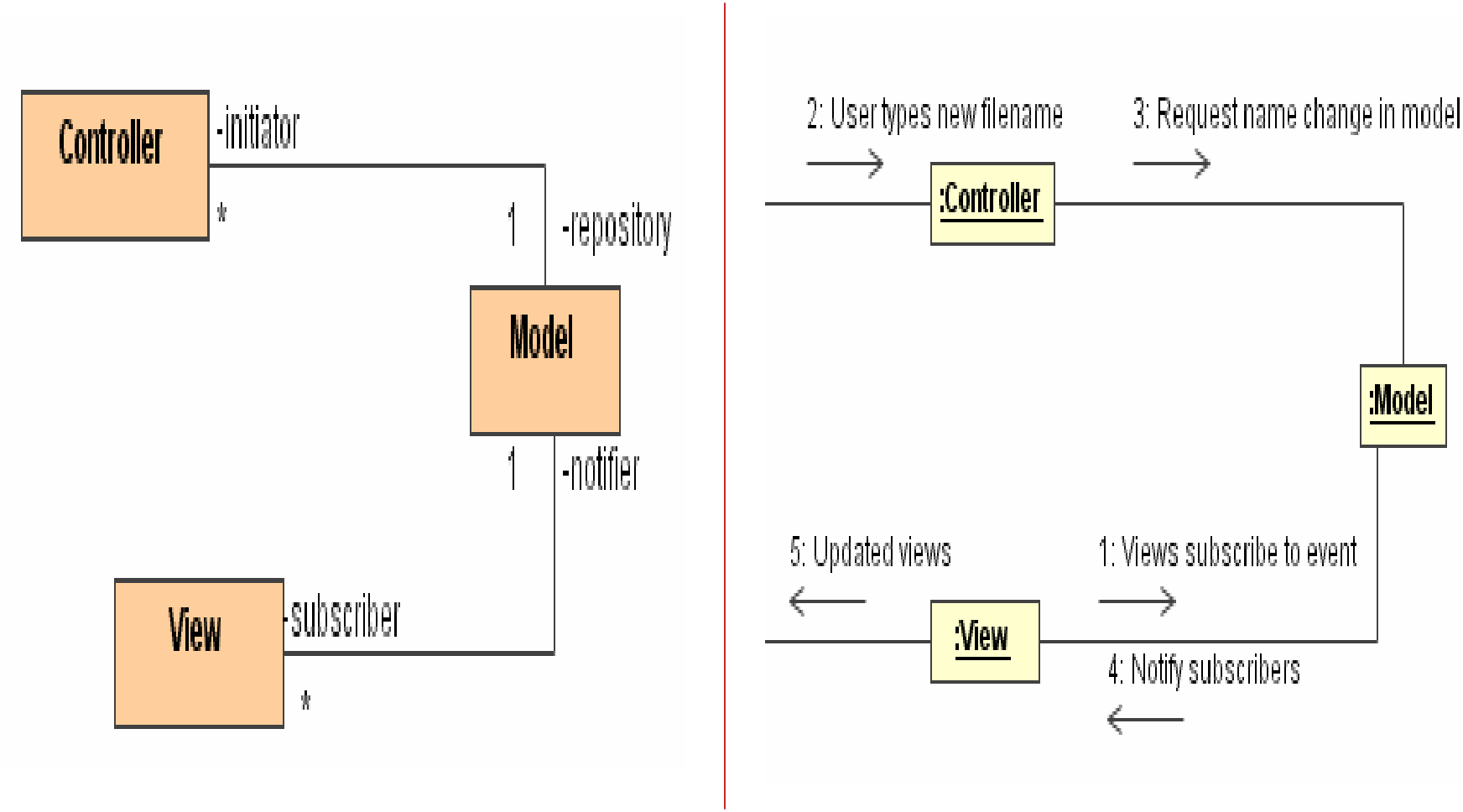
a) model subsystems – maintains domain knowledge
b) view subsystems – displays information to users
c) controller subsystem – controls sequence of interaction

The Model subsystems are written independently of view or controller subsystems.

Changes in model's state are propagated to the view subsystem through the subscribe/notify protocol.

The MVC architecture is a special case of repository architecture; model is the central repository and the controller subsystem dictates control flow.

MVC Architecture 2



Structural and Behavioural Description of MVC Architecture

Client – Server Architecture 1

1) there are two kinds of subsystems – the Server and Client

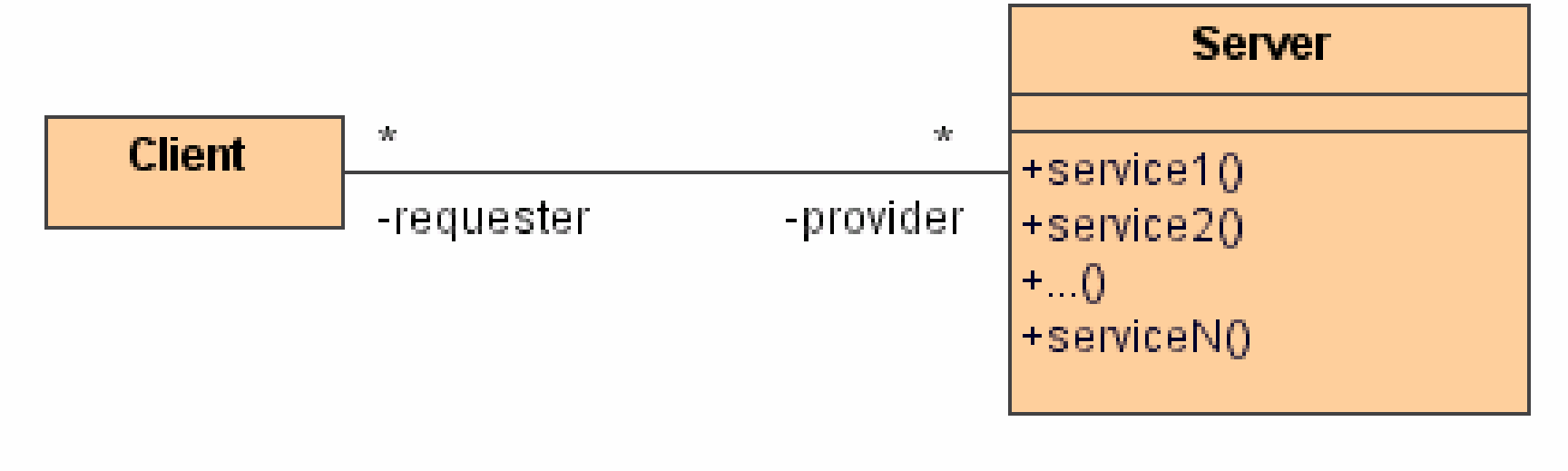
2) the Server subsystem provides services to instances of the other subsystems called clients; which interacts with the users

3) service requests are usually through remote procedure call or some other distributed programming techniques

4) control flow in clients and servers is independent except for synchronization to manage requests and receive results

5) there may be multiple servers subsystems like in the case of the world-wide web

Client – Server Architecture 2



Client – Server Architecture - Structural Description

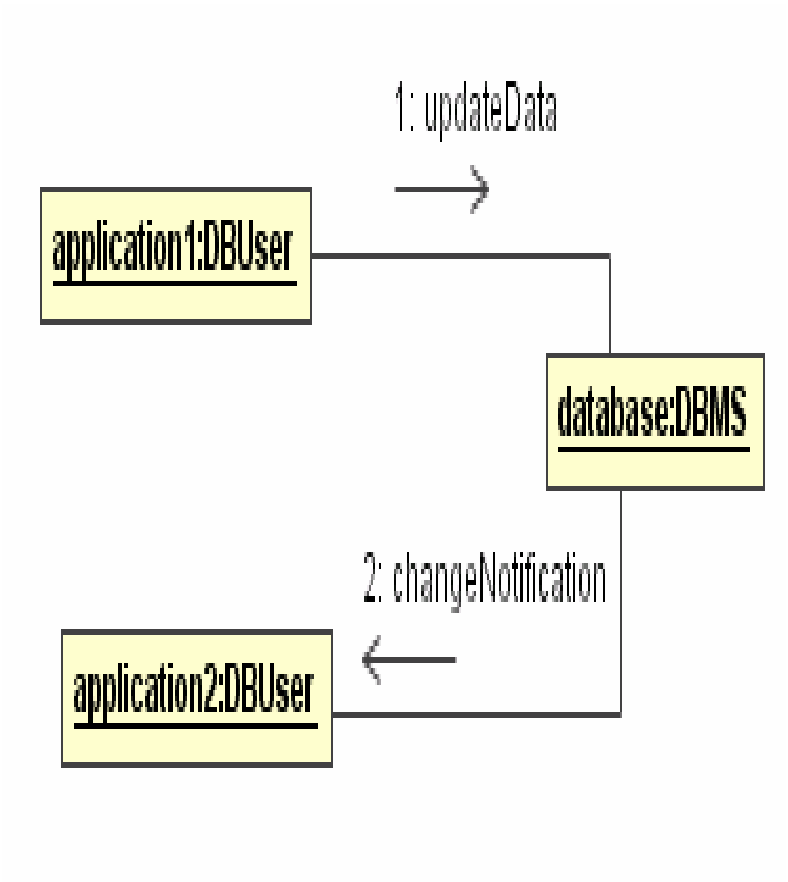
Peer-to-Peer Architecture 1

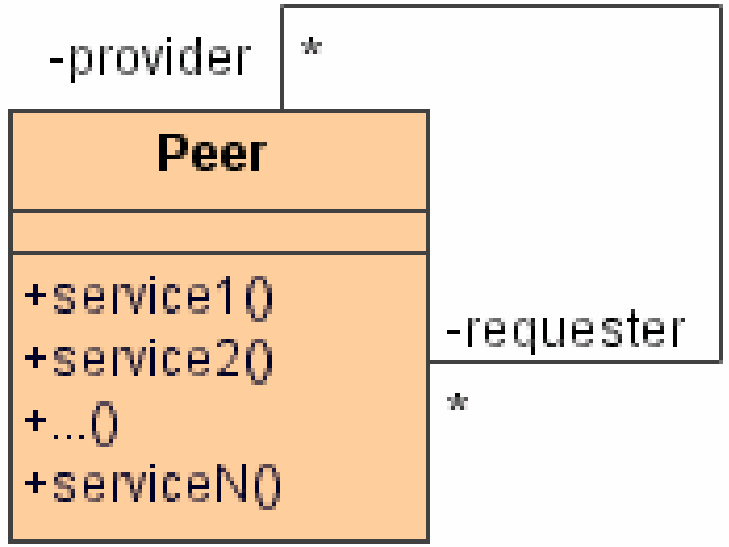
1) a generalization of the client-server architecture in which subsystems can take up the roles of clients and servers dynamically

2) control flow within subsystems is independent from the others except for synchronization on requests

3) Peer-to-Peer architecture is more difficult to design than client server systems as there are possibilities of deadlocks

Peer-to-Peer Architecture 2





Pipe-and-Filter Architecture 1

1) subsystems process data received from a set of input and send results to other subsystems via set of outputs

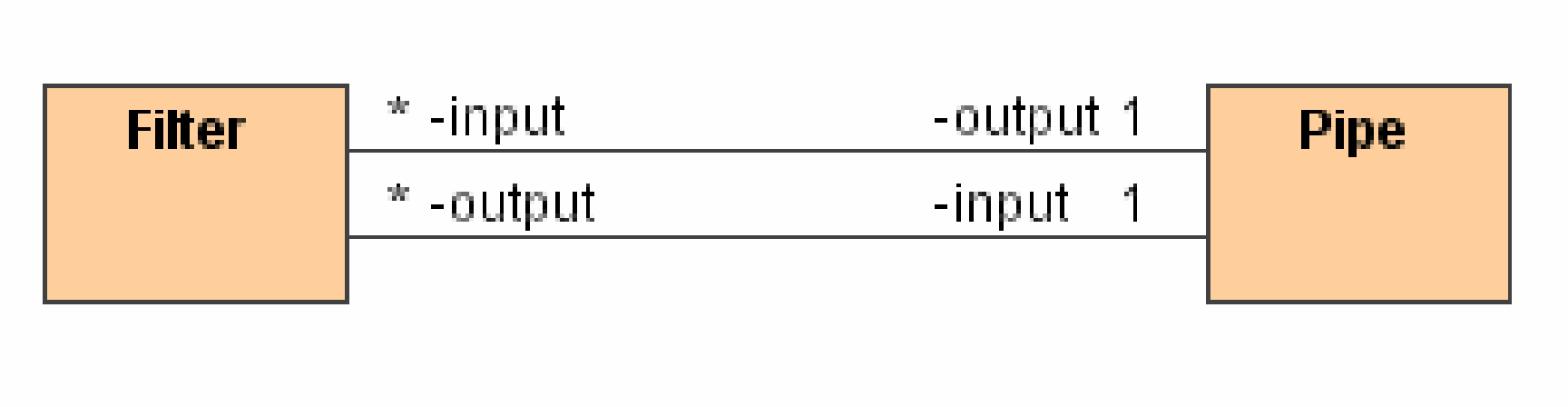
2) subsystems are called filters and the association between filters are called pipes

3) filters only know the content and format of the data received on the input pipes and not the filters that produce them

4) each filter is executed concurrently and synchronization is done via the pipes

5) pipes and filters can be reconfigured as required

Pipe-and-Filter Architecture 2



System Operations Contract

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Contracts

Definition

Contracts are constraints on a class that enable caller and callee to share the same assumption about class. The contract specifies constraints the caller must meet before using the class as well as the constraints that are ensured by the callee when used.

Contracts include there types of constraints:

a) invariant
b) precondition
c) postcondition

Contracts are defined for important system operations.

Contract: Invariant

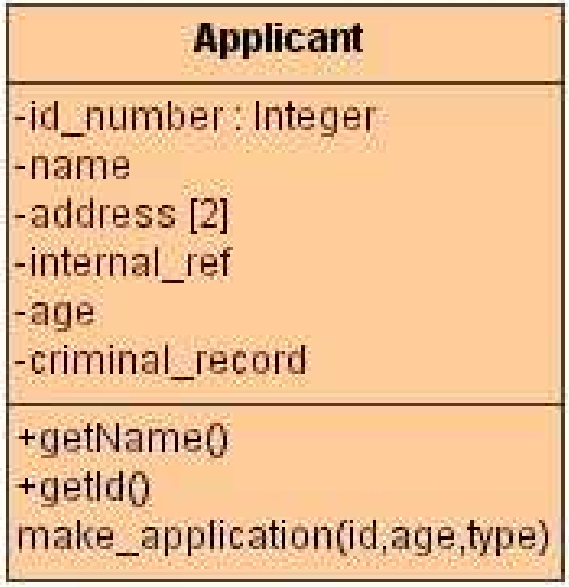
Definition

An Invariant is a predicate that is always true for all instances of a class. Invariant constraints are associated with classes or interfaces. They specify consistency conditions among class attributes.

Contract C01: *make_application*

Cross References: use case - *submit application*

Invariant: *criminal record must be nil*



Contract: Pre-Condition

Definition

A Pre-Condition is a predicate that must be true before an operation is invoked. Pre-conditions are associated with specific operations and is a constraint on the caller.

Contract C01: *submit_application*

Cross References: use case - *submit application*

Precondition: *(1) application has not been submitted earlier; (2) entry in application form is valid*

Contract: Post-Condition

Definition

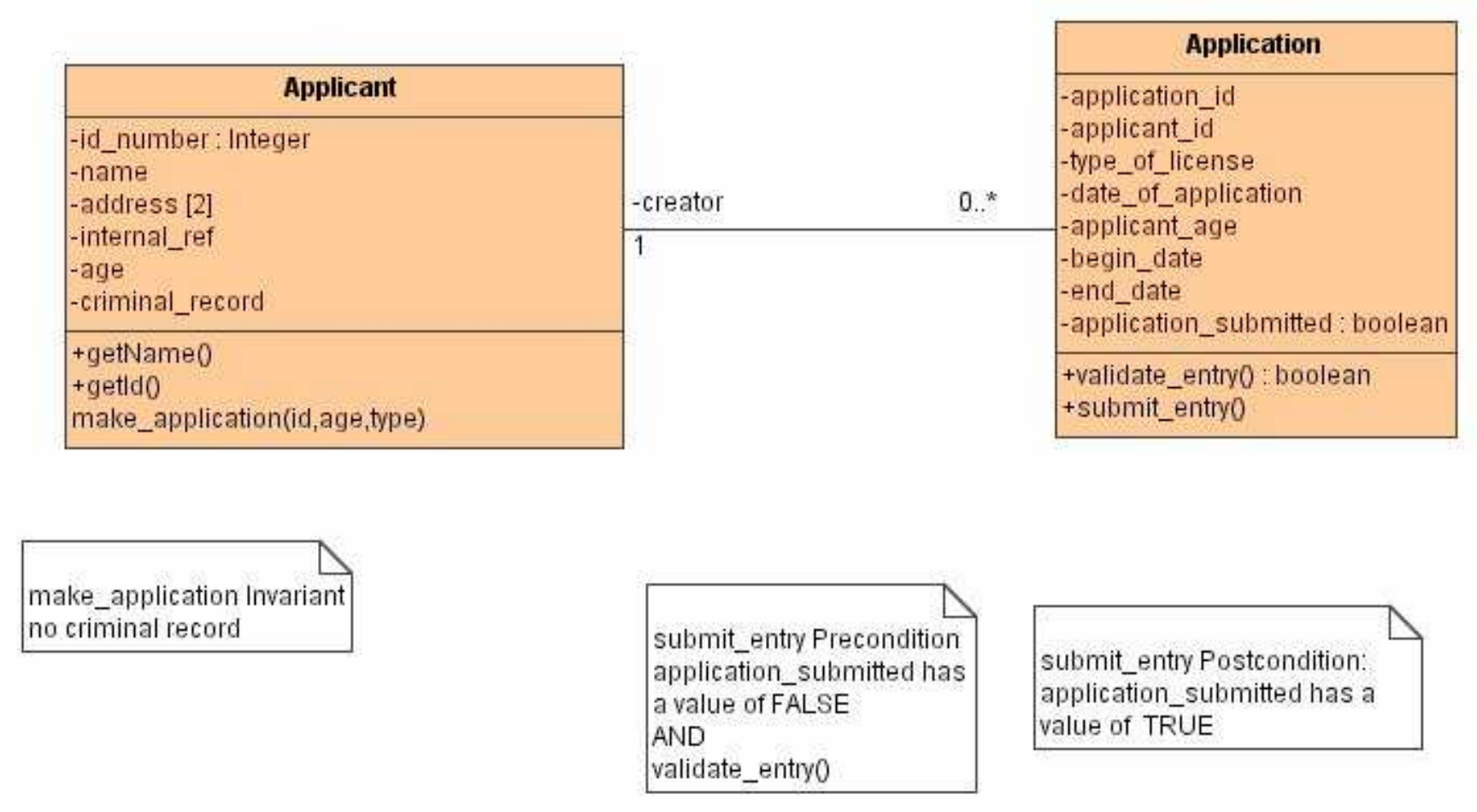
A Post-Condition is a predicate that must be true after an operation is invoked. Post-conditions are associated with specific operations and are used to specify the constraints that the object must ensure after execution of the operation.

Contract C01: *submit_application*

Cross References: use case - *submit application*

Post-condition: *(1) a record of the application is created.*

Example: Contract



GRASP Patterns

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

What is GRASP

General Responsibility Assignment Software Pattern

Responsibility:

1) a contract or an obligation of an object
2) responsibilities are related to the obligations of objects in terms of their behaviour

Two types:

1) doing responsibilities – actions that an object can perform
2) knowing responsibilities – knowledge that an object maintains

GRASP – Responsibilities

1) doing
    a) doing something itself
    b) initiating an action or operation in other objects
    c) controlling and coordinating activities of other objects

2) knowing
    a) knowing about private encapsulated data
    b) knowing about related objects
    c) knowing about things it can derive or calculate

GRASP Patterns

1) Information Expert

2) Creator

3) High Cohesion

4) Low Coupling

5) Controller

Expert 1

Pattern Name : Expert

Solution : assign a responsibility to the information expert – the class that has information necessary to fulfill the responsibility.

Problem : what is the most basic principles by which responsibilities are assigned in OOD?

Example: consider the “Applicant” and “Application” classes in the contract example. Which class should be responsible for submitting an application?

Since the Application class possesses the required information necessary for submitting an application (basic information, begin application and end application dates etc.), we delegate this responsibility to it.

Expert 2

1) Expert pattern is the most applied GRASP pattern in OOD, supporting the common intuition of objects do things related to the information they have

2) fulfillment of responsibility may require information spread across different classes of objects, indicating the possibilities of many partial experts who will collaborate in the task

3) Expert patterns help maintain encapsulation since objects use their own information to fulfill responsibilities

Creator 1

Pattern Name : Creator

Solution : assigns class B the responsibility to create an instance of class A (B is a creator of A objects) if one of the following is true:

- B aggregates A objects
- B contains A objects
- B records instances of A objects
- B closely uses A objects
- B has initializing data that will be passed to A when it is created (thus B is an expert w.r.t. creating A objects)

Problem: what should be responsible for creating a new instance of some class?

Creator 2

Example: still consider the contract example involving the Applicant and Application Class. Obviously the Applicant class closely uses the Application objects. In fact, the *Applicant* possess vital initializing data for the *Application* objects and thus an expert at creating *Application* objects.

So we can assign the Applicant class the responsibility of creating an instance of the Application Class.

Thus the inclusion of the *make_application()* method in the Applicant class

Low Coupling 1

Coupling is a a measure of how strongly one class is connected to, has knowledge of, or relies upon other classes. A class with low or weak coupling is not dependent on too many class.

Pattern Name: Low Coupling

Solution: assign a responsibility so that coupling remains low.

Problem: how to support low dependency and increased reuse?

Low Coupling 2

Example: consider the MVC architecture described earlier, where the controller component will dispatch requests to the model component.

If the model component was implemented as individual classes which will be called by the controller component, then there will be a lot of dependencies involved the architecture of the system.

On the other hand, we can conceive of a “broker or façade” class which takes the request from the controller and identifies appropriate classes to service the request.

Low Coupling 3

1) common forms of coupling from class X to class Y:
    a) class X has an attribute that refers to a class Y itself
    b) class X has a method which references an instance of class Y, or Class Y itself, by an means (parameters or local variable of type Class Y, or the object returned from a message being an instance of class Y
    c) class X is a direct or indirect subclass of class Y

2) coupling may not be too important if reuse is not a goal

3) no absolute measure of when coupling is too high

4) very little or no coupling between classes is rare and not interesting

High Cohesion 1

Cohesion is a measure of how strongly related and focused the responsibilities of a class are.

A class with high cohesion has highly related functional responsibilities and does minimal amount of work.

Pattern Name: High Cohesion

Solution: assign a responsibility so that cohesion remains high

Problem: how to keep complexity manageable?

High Cohesion 2

Example: again consider the contract example. Since applicants make applications one may ask if the “Applicant” and the *Application* classes could be merged.

This would require the *Applicant* class to bear the responsibilities of *making*, *validating* and *submitting* an *application*.

This strategy strongly ties the concept of an *Application* with an *Applicant.*

Controller 1

A controller is a non-user interface object responsible for receiving or handling a system event. A controller defines the method for the systems operations.

Pattern Name: Controller

Solution: assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

a) the overall system, device, or subsystem (façade controller)

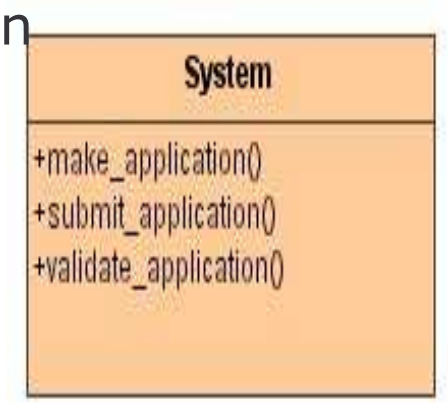
b) a use case scenario within which the system events occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator or <UseCaseName>Session

Controller 2

Problem: Who should be responsible for handling an input system event?

An input system event is an event generated by an external actor. They are associated with system operations which respond to system events.

Example: consider the system operations shown in the System class. We may ask who should be the controller for system events such as *make_application* and *submit_application*?



By the controller pattern, we have some choices:

(1)the overall *LicenseOnline* system

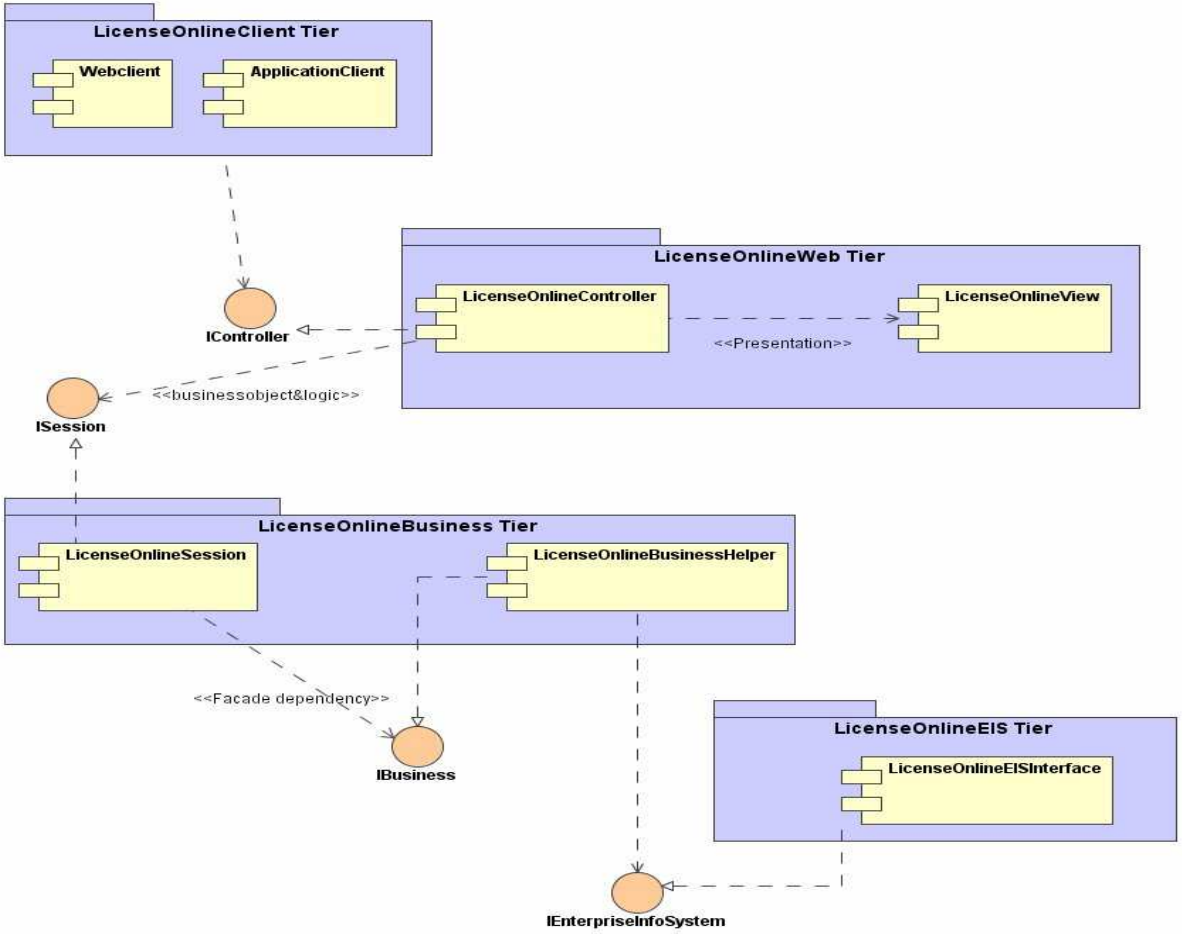
(2) a receiver or handler of all system events of a use case scenario (*SubmitApplicationHandler*)

Architecture Model for Case Study

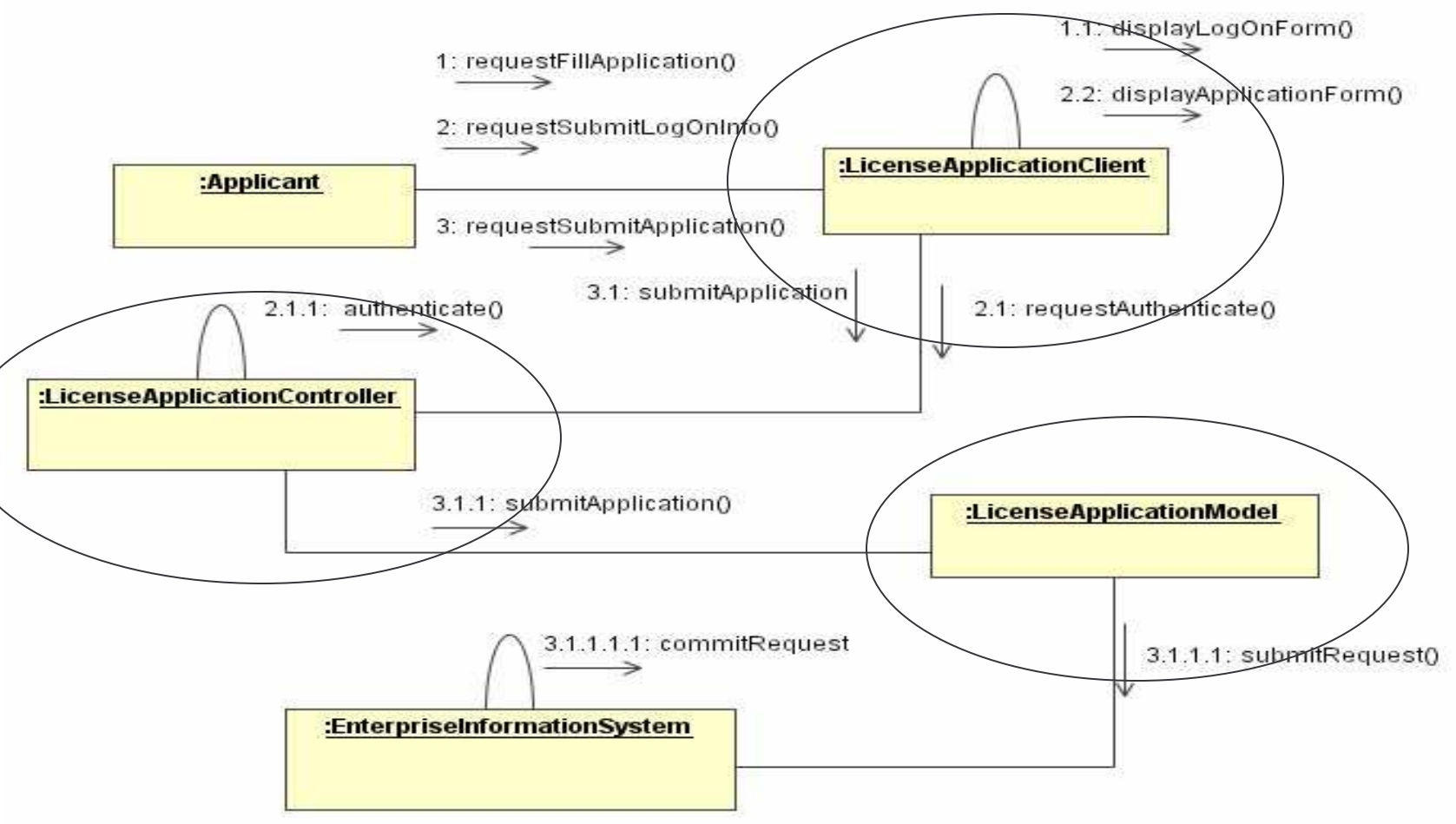
Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Architecture: Structural Model



Architecture: Behavioural Model



Summary

Architecture Modelling

1) Software Architecture Concepts
2) Packages
3) Collaboration Diagrams
4) Component Diagrams
5) Architectural Patterns
6) System Operations Contract
7) GRASP Patterns
8) Architecture Model for Case Study
9) Summary

Summary 1

Architecture is concerned with the structural organization of system components as well as how they interact to provide the system's overall behaviour or functionality.

Packages are general purpose mechanisms for organizing modelling elements into groups.

Well structured packages must be loosely coupled and very cohesive.

Five stereotypes may be applied to packages: façade, framework, stub, subsystem and system.

Summary 2

A collaboration is a society of classes which provides some cooperative behaviour that is more than the sum of all its parts.

Collaborations have both structural and behavioural aspects

Collaborations may realize a use case or an operation.

A collaboration diagram shows interactions organized around the structure of a model, using either classes and associations or instances and links.

Summary 3

A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.

An interface is a collection of operation that are used to specify a service of class or components.

There are three categories of components: deployment, work product and execution.

Components may be stereotyped as executable, library, table, file or document.

Summary 4

A component diagram consists of specifying classes, implementing artifacts, components, interfaces and relationships between these model elements.

Basic architectural patterns or styles include: repository; MVC; client-server; peer-to-peer; and pipe-and-filter.

The General Responsibility Assignment Software Pattern provides five basic patterns: Information Expert, Creator, High Cohesion, Low Coupling and Controller.

Exercise – Project 1

1) Describe a high level architecture of your system using packages.

2) List the various classifiers and associations that are essential in presenting a collaboration diagram for your system.

3) Present the above elements in a specification level collaboration diagram for your system.

4) Identity the major components of your system and relate these components to their specifying classes and defining interfaces.

5) Using the same specification level collaboration diagram drawn in 3, show a conforming instance level collaboration diagram that supports a particular use case of your system.

Exercise – Project 2

6) Consider the architectural patterns discussed earlier, justify your choice of a suitable architecture pattern.

7) Discuss the tradeoffs associated with your system's architectural style.

8) Highlight some principles that underpin responsibility assignment in your system's architecture.

Design Modelling

Overview

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:
   a) Requirements
   b) Architecture
   c) Design
   d) Implementation
   e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

Design Concepts

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

What is System Design?

"*There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies*"

C.A.R. Hoare

Definition

System design is the transformation of analysis models of the problem space into design models (based on the solution space). It involves selecting strategies for building the system e.g. software/hardware platform on which the system will run and the persistent data strategy.

System Design - Overview

1) Analysis produces:
    a) a set of non-functional requirements and constraints
    b) a use case model describing functional requirements
    c) an object model, describing entities
    d) a sequence diagram for each use case showing the sequence of interaction among objects

2) Design results in:
    a) a list of design goals – qualities of the system
    b) software architecture describing the subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, major policy decision such as control flow, access control, and data storage

System Design – Activities

1) definition of design goals
2) decomposition of system into sub-system
3) selection of off-the-shelf and legacy components
4) mapping of sub-systems to hardware
5) selection of persistent data management infrastructure
6) selection of access control policy
7) selection of a global control flow mechanism
8) handling of boundary conditions

Some Design Activities 1

1) Define goals:
    a) derived from non-functional requirements

2) Hardware and software mapping:
    a) what computers (nodes) will be used?
    b) which node does what?
    c) how do nodes communicate?
    d) what existing software will be used and how?

3) Data management:
    a) which data needs to be persistent?
    b) where should persistent data be stored?
    c) how will the data be stored?

Some Design Activities 2

4) Access control:
    a) who can access which data?
    b) can access control be changed within the system?
    c) how is access control specified and realized?

5) Control flow:
    a) how does the system sequence operations?
    b) is the system event-driven?
    c) does it need to handle more than one user interaction at a time?

What is Object Design?

1) system design describes the system in terms of its architecture, such as its subsystem decomposition, its global control flow and its persistency management

2) object design includes:

    a) service specification – precise description of class interface
    b) component selection – identification of off-the-shelf components and solution objects
    c) object model restructuring – transform the object design model to improve understandability and extensibility
    d) object model optimization – transform object model to address performance (response time, memory etc.)

Object Design Activities 1

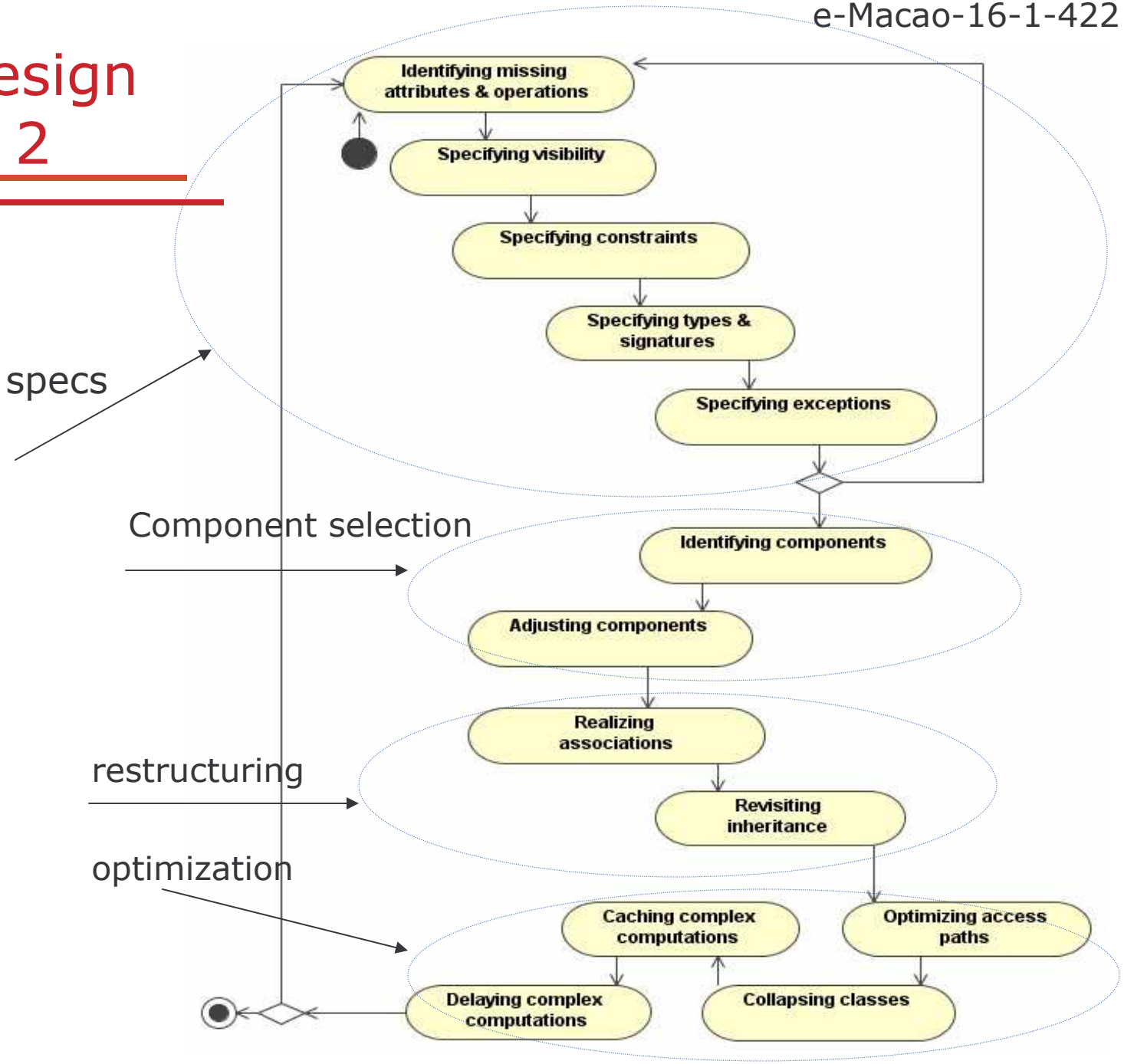
1) specification
   a) missing attributes and operations
   b) type signatures and visibility
   c) constraints
   d) exceptions

2) component selection
   a) adjusting class libraries
   b) adjusting application frameworks

3) restructuring
   a) realizing associations
   b) increasing reuse
   c) removing implementation dependencies

4) optimization
   a) access paths
   b) collapsing objects
   c) caching results of expensive components
   d) delaying expensive components

Object Design Activities 2



Design Pattern

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

Design Patterns 1

Definition

Design patterns are partial solutions to common problems. They name, abstract, and identify the key aspects of common design structure that make them useful for creating reusable object-oriented design.

Design patterns are composed of small number of classes that through delegation and inheritance, provide robust and modifiable solutions.

Some common problems:

a) separating interfaces from a number of alternate implementations
b) wrapping around a set of legacy classes
c) protecting a caller from changes associated with specific platforms

Design Patterns 2

1) design patterns capture expert knowledge and design tradeoffs and support the sharing of architectural knowledge among developers

2) design patterns as a shared vocabulary can clearly document the software architecture of a system

3) allow design engineers relate to one another at a higher level of abstraction

Design Patterns: Description 1

Attribute	Description
Pattern Name and Classification	Conveys the essence of the pattern succinctly.
Intent	A short statement that answers the following questions: what does the design do? Its rationale and intent.
Also Known As	Other well known names for the patterns, if any
Motivation	A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
Applicability	What are the situations in which the design pattern can be applied?
Structure	Class diagram and sequence diagram for the classes involved in the pattern.
Participants	The classes and/or the object participating in the design pattern and their responsibilities.
Collaborations	How participant objects collaborate to carry out their responsibilities.

Design Patterns: Description 2

Attribute	Description
Consequences	How does the pattern support its objectives? What are the tradeoffs?
Implementation	What pitfalls, hints, or techniques you should be aware of when implementing the pattern.
Sample Code	Code fragment to show implementation.
Known uses	Example of patterns found in real systems.
Related Patterns	Which design patterns are closely related to this one?

Design Patterns Catalog

1.	Abstract Factory (C)	13.	Interpreter (B)*
2.	Adapter (S), Adapter (S)*	14.	Iterator (B)
3.	Bridge (S)	15.	Mediator (B)
4.	Builder (C)	16.	Memento (B)
5.	Chains of Responsibility (B)	17.	Observer
6.	Command (B)	18.	Prototype (C)
7.	Composite (S)	19.	Proxy (S)
8.	Decorator (S)	20.	Singleton (C)
9.	Façade (S)	21.	State (B)
10.	Factory Method (C)*	22.	Strategy (B)
11.	Flyweight (S)	23.	Template Method (B)*
12.	Visitor (B)		

*C – creational pattern, S – structural pattern, B – Behavioural pattern, * - Class Scope*

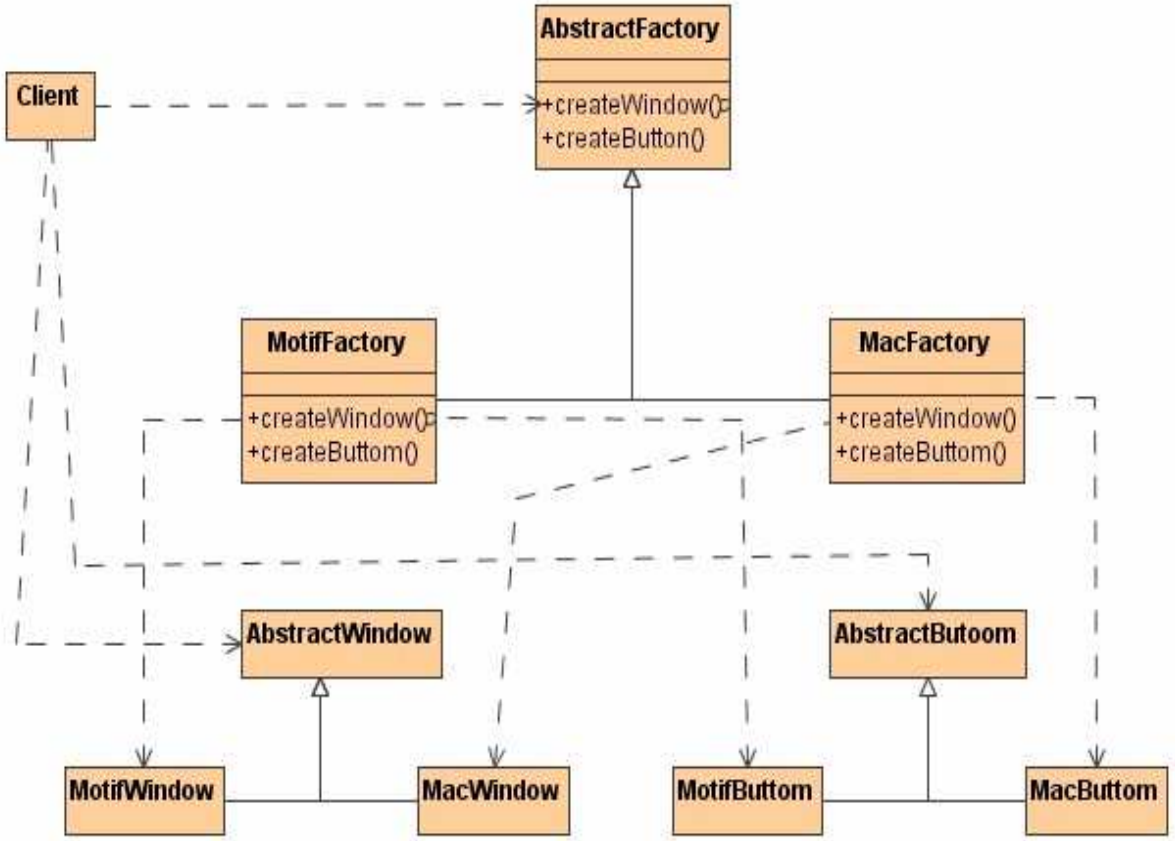
Applying Abstract Factory

Problem:

encapsulating platforms

Solution:

abstract factory provides an interface for creating families of related or dependent objects without specifying their concrete classes



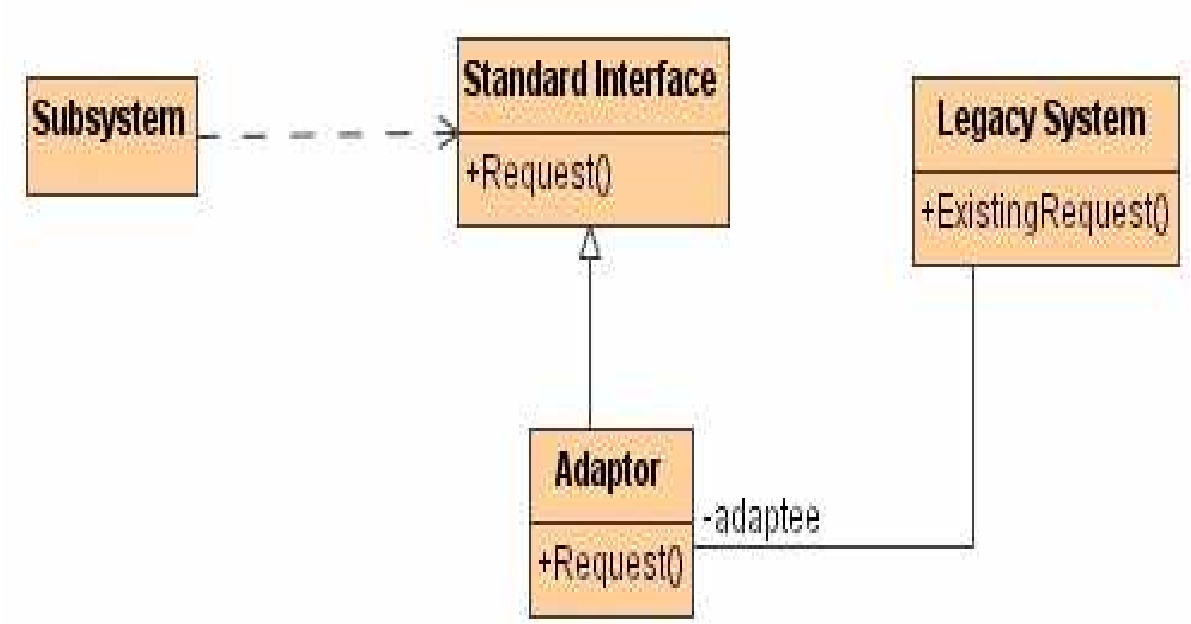
Applying Adapter

Problem:

wrapping around legacy code

Solution:

adapter encapsulates a piece of legacy code not designed to work with the system



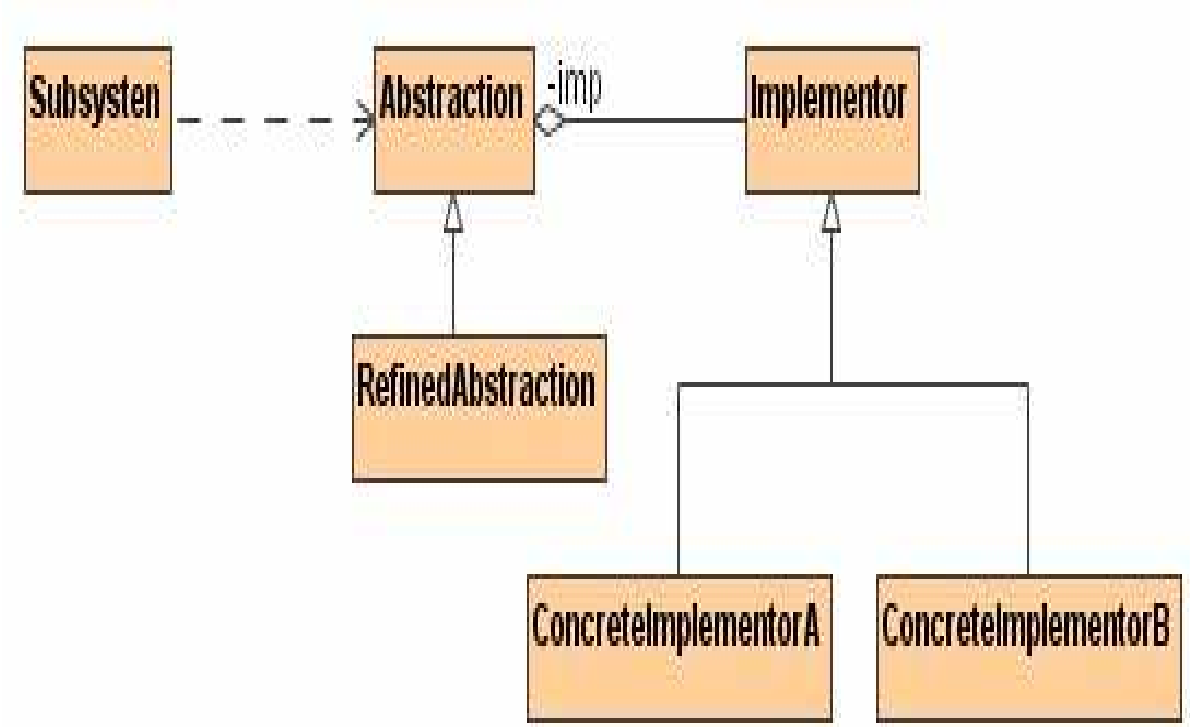
Applying Bridge

Problem:

allowing for alternate implementation

Solution:

bridge decouples the interface from its implementation



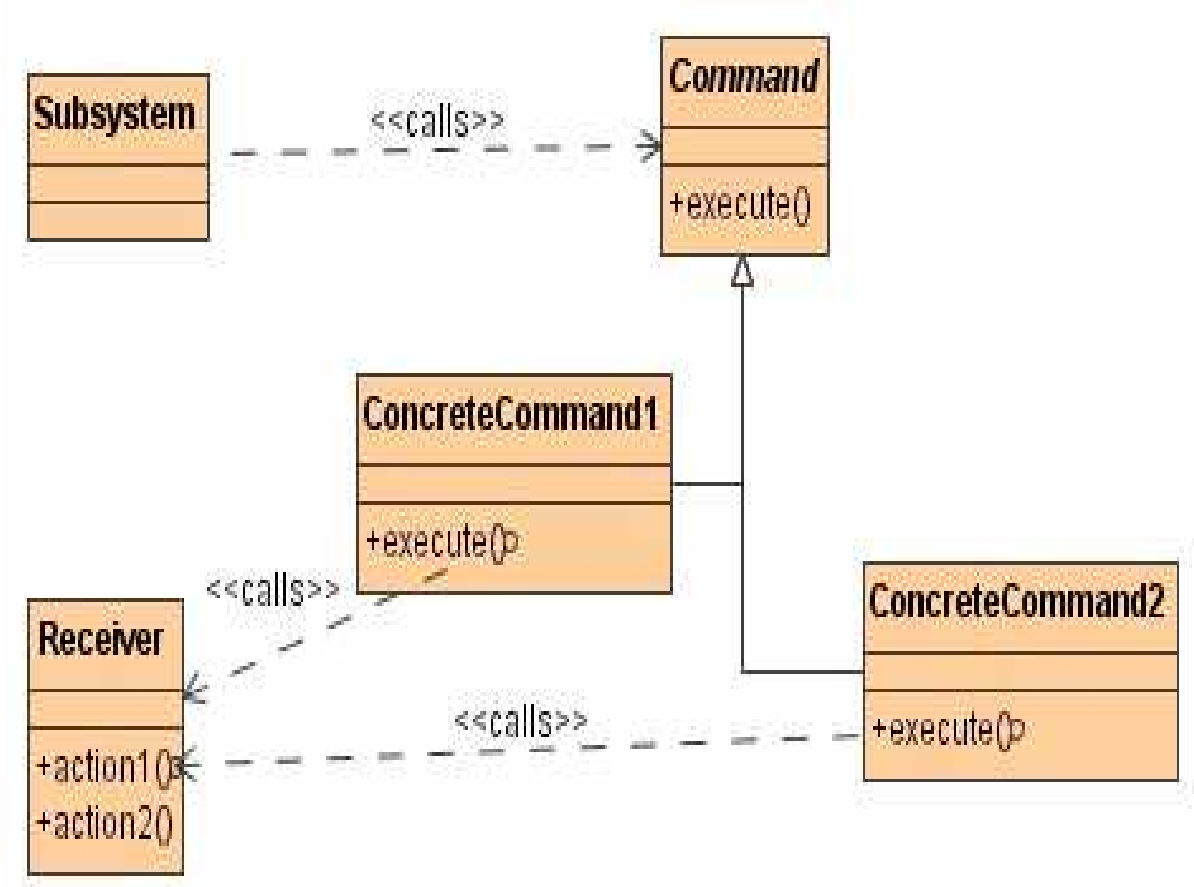
Applying Command

Problem:

encapsulating control

Solution:

command encapsulates a control such that user requests can be treated uniformly



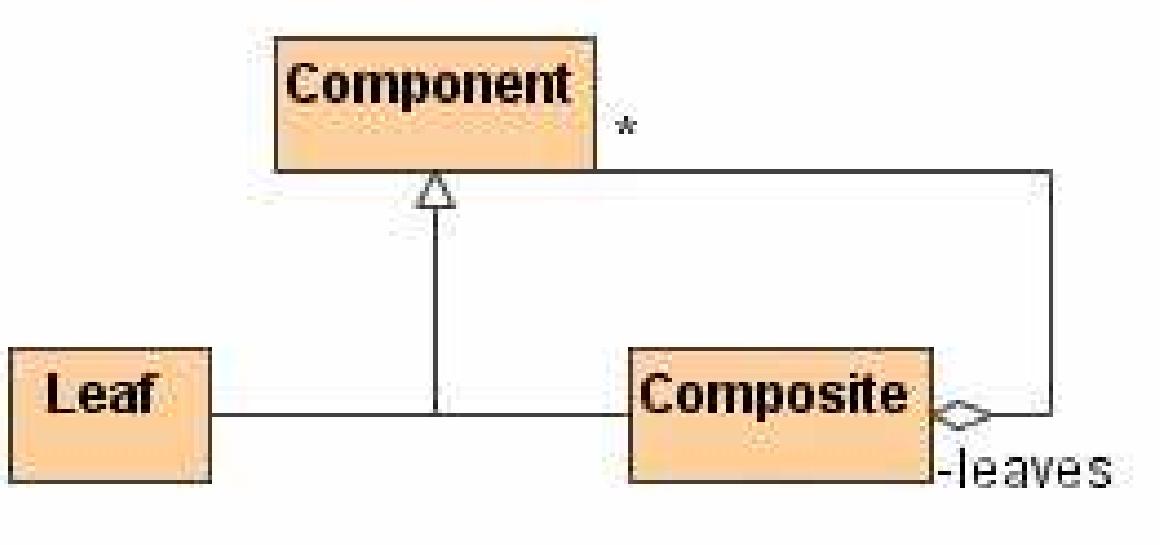
Applying Composite

Problem:

representing recursive hierarchies

Solution:

composite represent recursive hierarchies



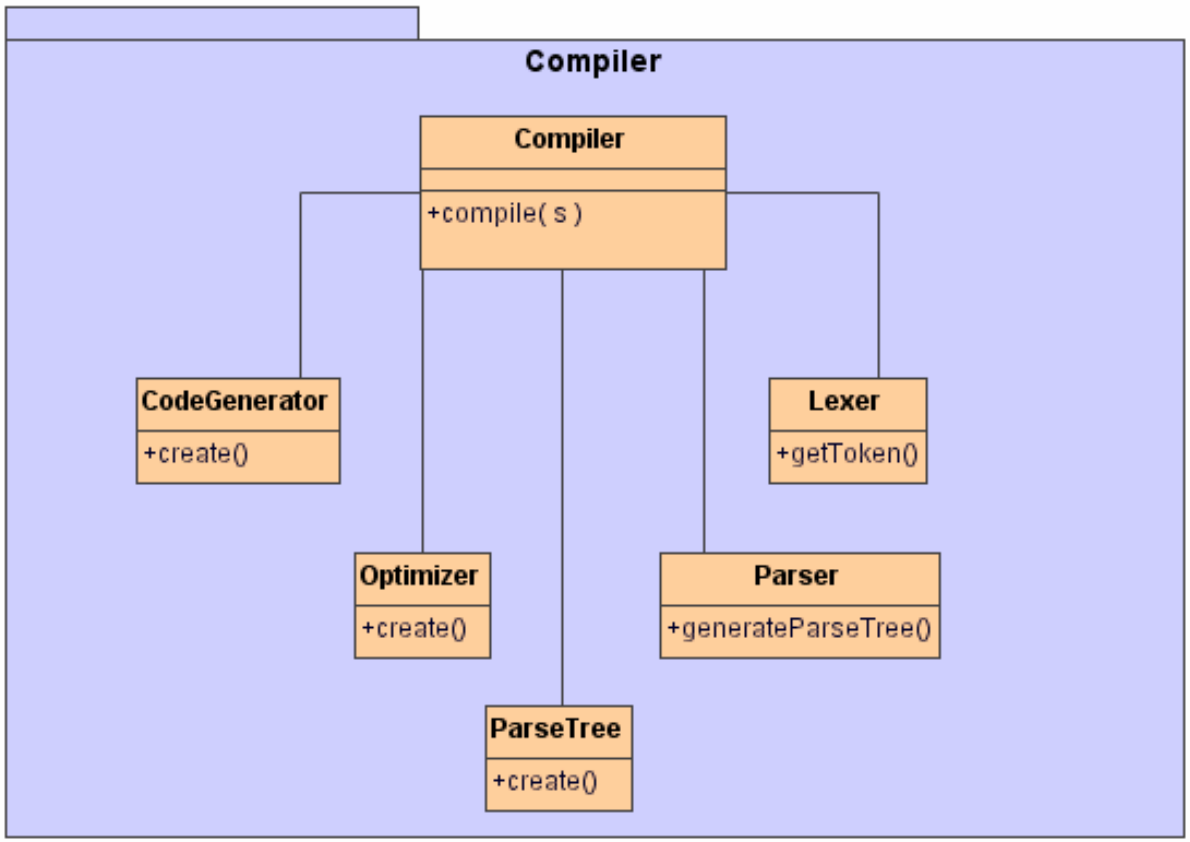
Applying Façade

Problem:

encapsulating subsystems

Solution:

façade reduces dependencies among classes by encapsulating subsystems from with simple unified interfaces



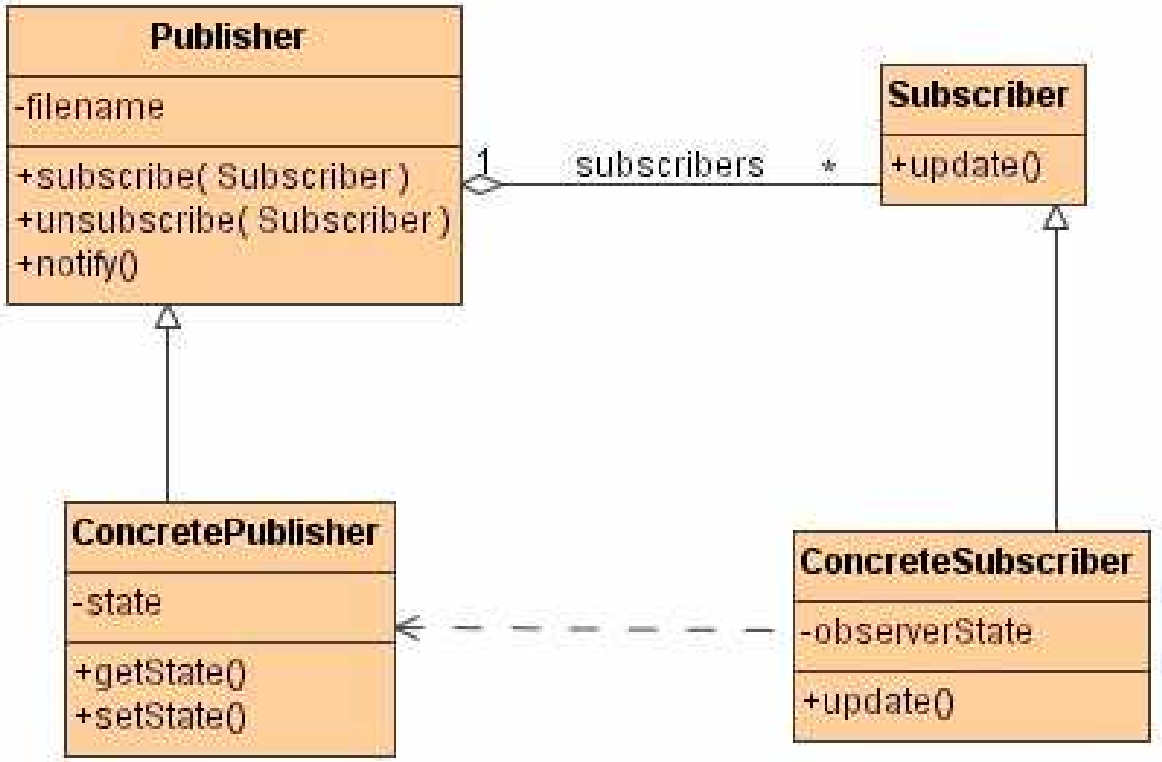
Applying Observer

Problem:

decoupling entities from view

Solution:

observer allows us to maintain consistency across the state of one publisher and many subscribers



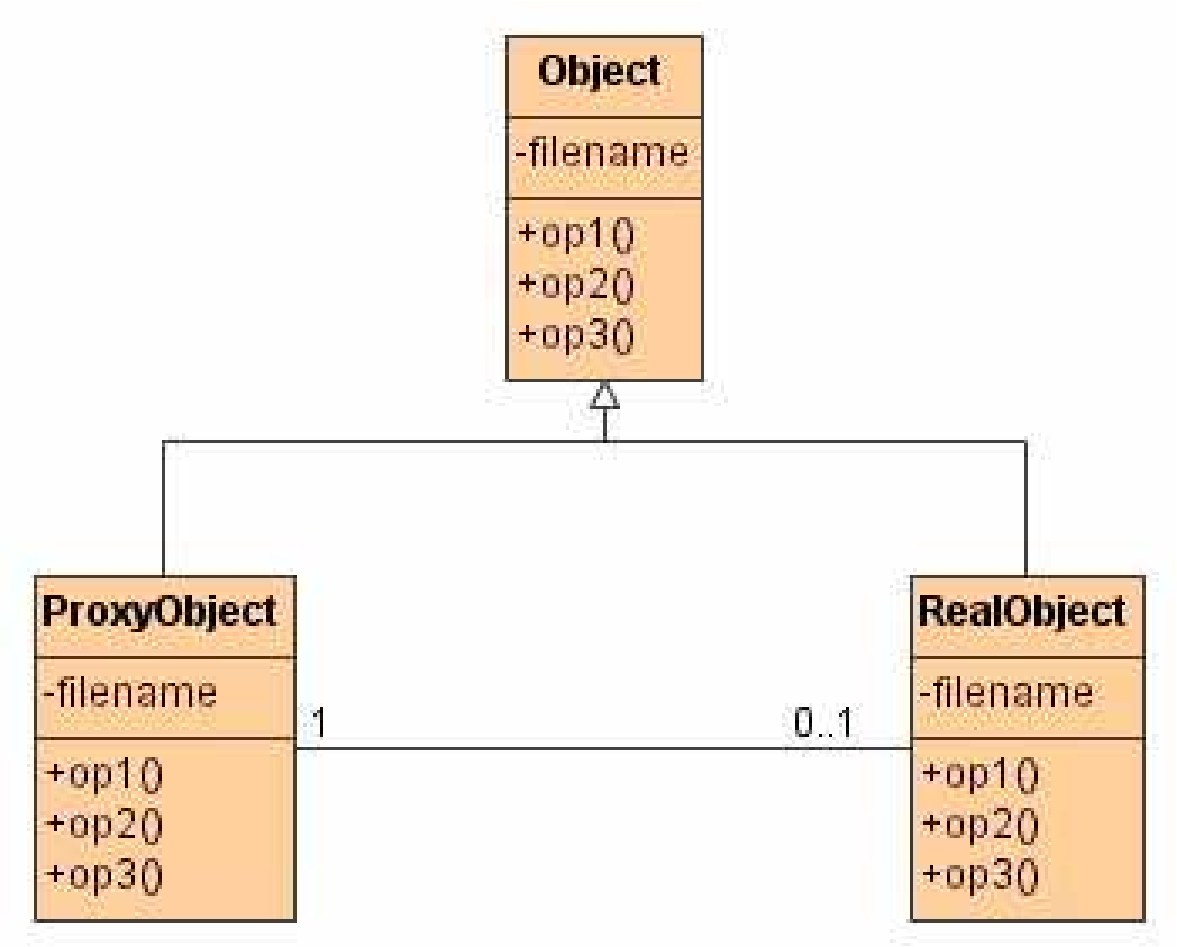
Applying Proxy

Problem:

encapsulating expensive objects

Solution:

proxy improves the performance or the security of a system by delaying expensive computations, … until when needed



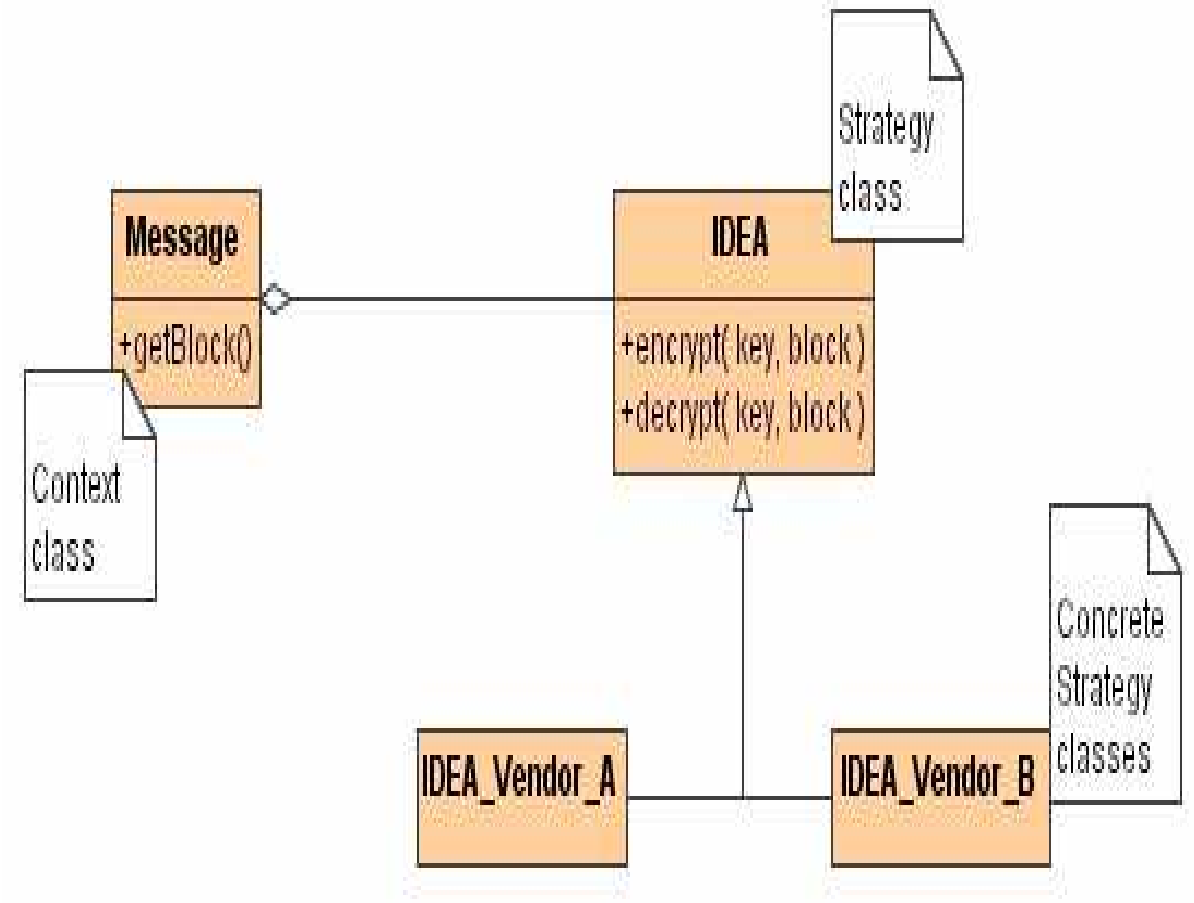
Applying Strategy

Problem:

encapsulating algorithms

Solution:

strategy decouples an algorithm from its implementations



Design Class Diagrams

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

Design Class Diagram

A Design Class Diagram (DCD) provides the specification for software classes and interfaces in an application.

Typical information contained in a DCD includes:

1) classes, associations and attributes
2) interfaces, with their operations and constants
3) methods
4) attributes type information
5) navigability
6) dependencies

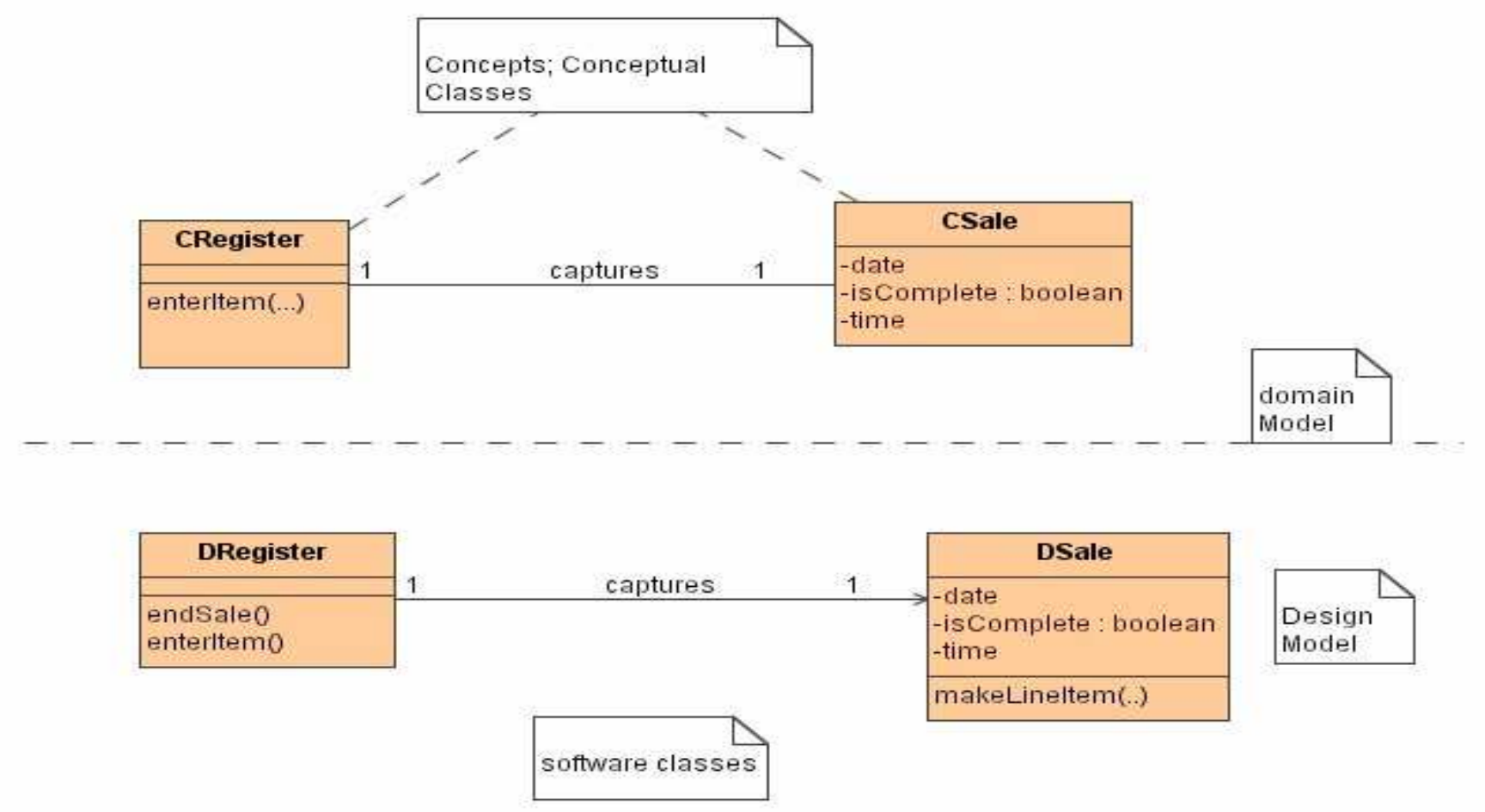
Domain Vs. Design Classes

In a domain model, a class does not represent a software definition. Rather it is an abstraction of a real world-world concept about which we are interested in making statement.

Design Classes express the definitions of classes as software components. In these diagram, a class represent a software class.

In contrasts to conceptual classes in the domain model, design classes in the DCD show definitions for software classes rather than real-world concepts.

Example: CCD and DCD



Creating Design Class Diagrams 1

1) identify all classes participating in object interaction by analyzing the collaborations

2) present them in a class diagram

3) copy attributes from the associated concepts in the conceptual model

4) add methods names by analyzing the interaction diagrams

5) add type information to the attributes and methods

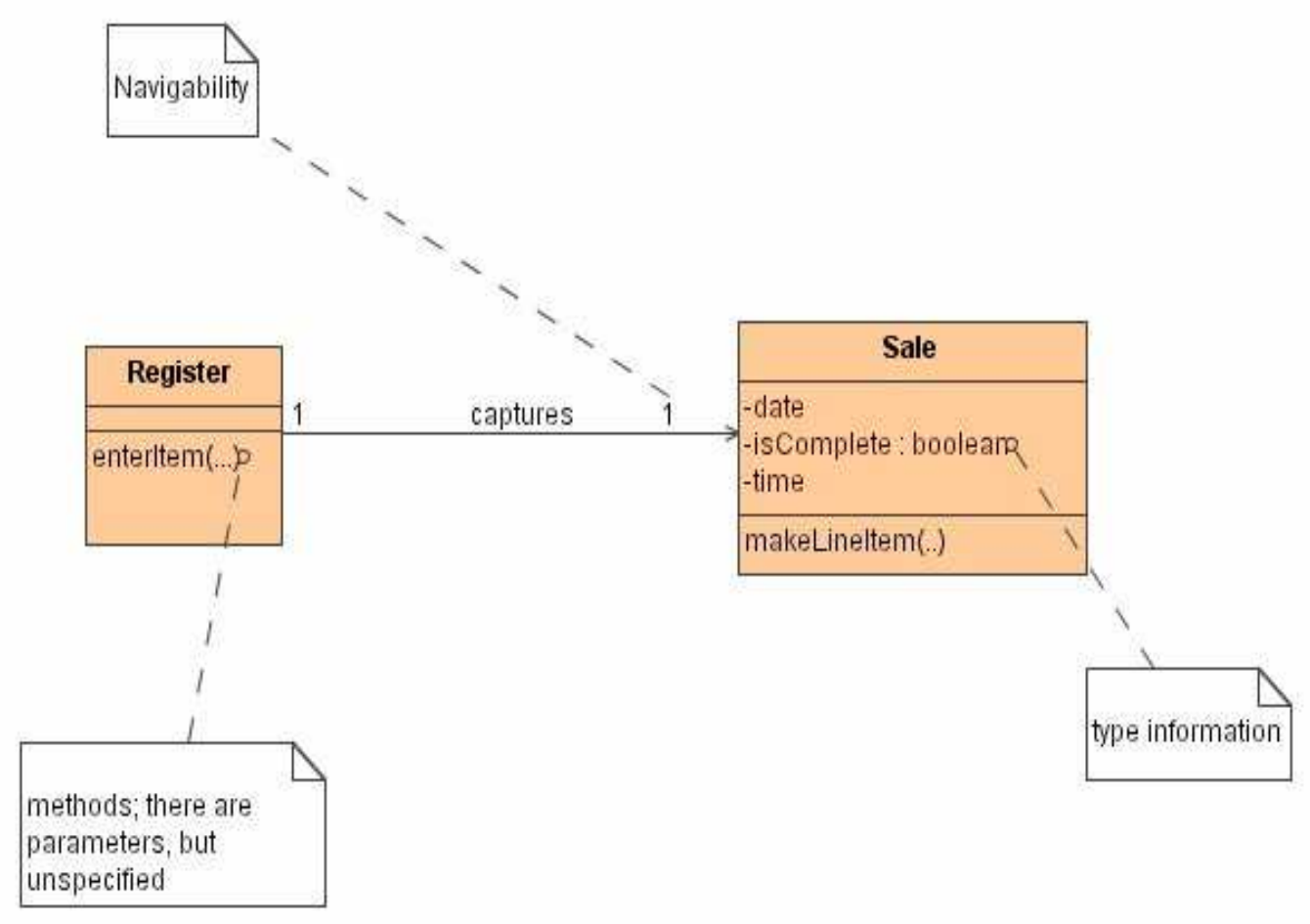
Creating Design Class Diagrams 2

6) add the association necessary to support the required attribute visibility

7) add navigability arrows necessary to the associations to indicate the direction of the attribute visibility

8) add dependency relationship lines to indicate non-attribute visibility

Example: DCD


Navigability
Register
enterItem(...)
1
captures
1
Sale
-date
-isComplete : boolean
-time
makeLineItem(..)
type information
methods; there are parameters, but unspecified

Adding Classes

The first step in creating a design class diagram is to identify those classes that participate in the software solution.

These classes can be found by scanning all the interaction diagrams and listing the participating classes.

Adding Attributes

After defining the classes, attributes must be added.

Attributes come from software requirements artifacts, for example conceptual classes provide most of the attributes applicable to design classes.

Adding Methods

The methods of each class can be identified by analyzing the interaction diagrams.

For example if the message *displayLogOnform* is sent to an instance of the *LicenseApplicationClient* class, then the class *LicenseApplicationClient* must define a method *displayLogOnform.*

Adding Visibility

Visibility:

1) scope of access allowed to a member of a class
2) applies to attributes and operations

UML visibility maps to OO visibilities:

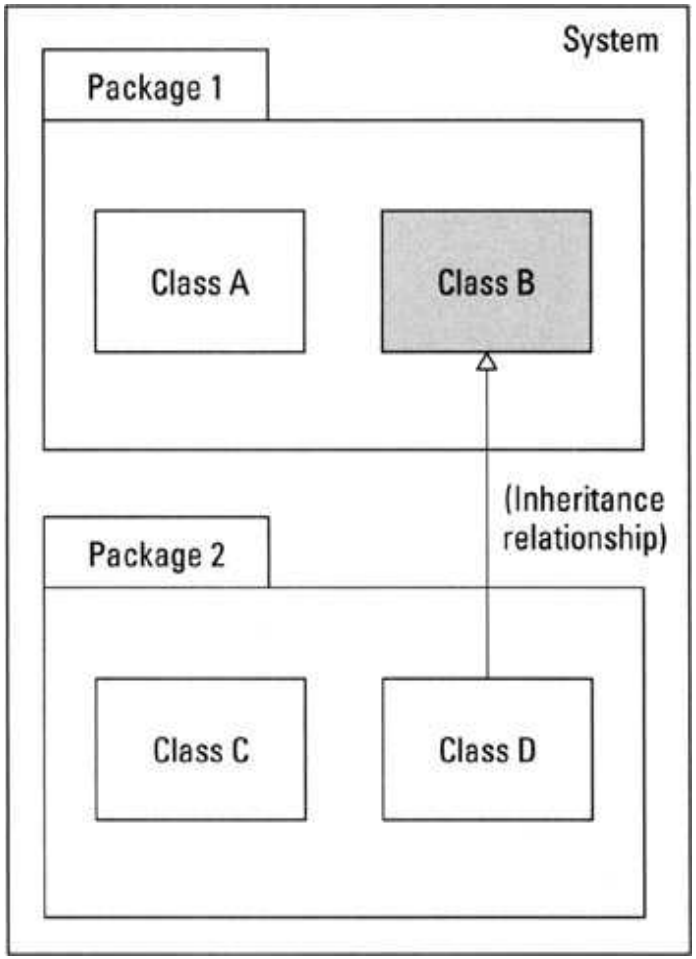
1) Private scope: within a class (-)
2) package scope: within a package (~)
3) public scope: within a system (+)
4) protected scope: within an inheritance tree (#)

Private Visibility

1) private element is only visible inside the namespace that owns it

2) notation is "-"

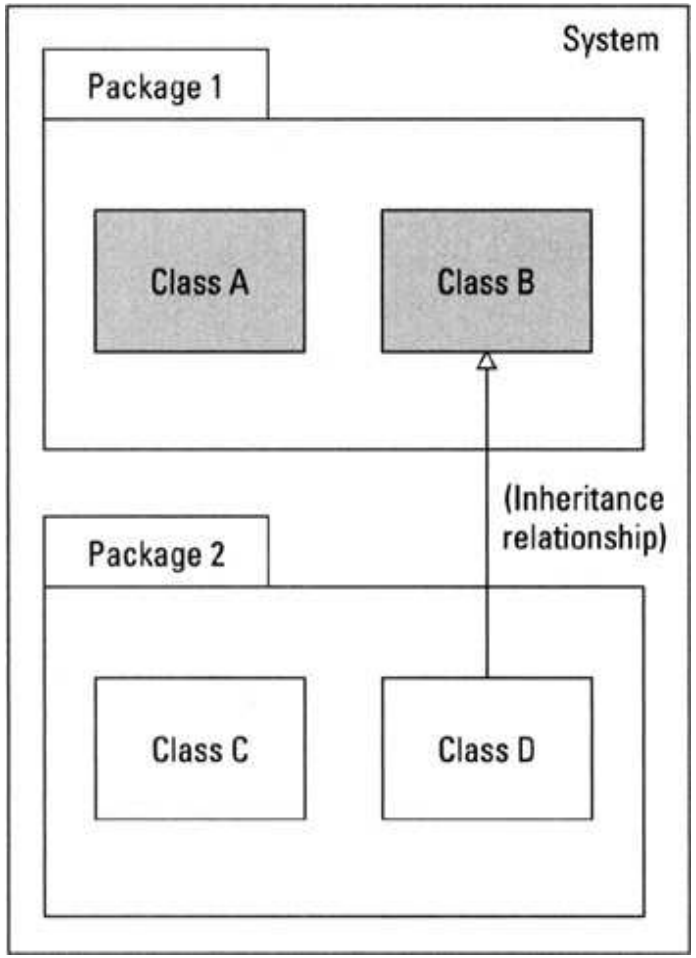
3) useful in encapsulation



Package Visibility

1) package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace

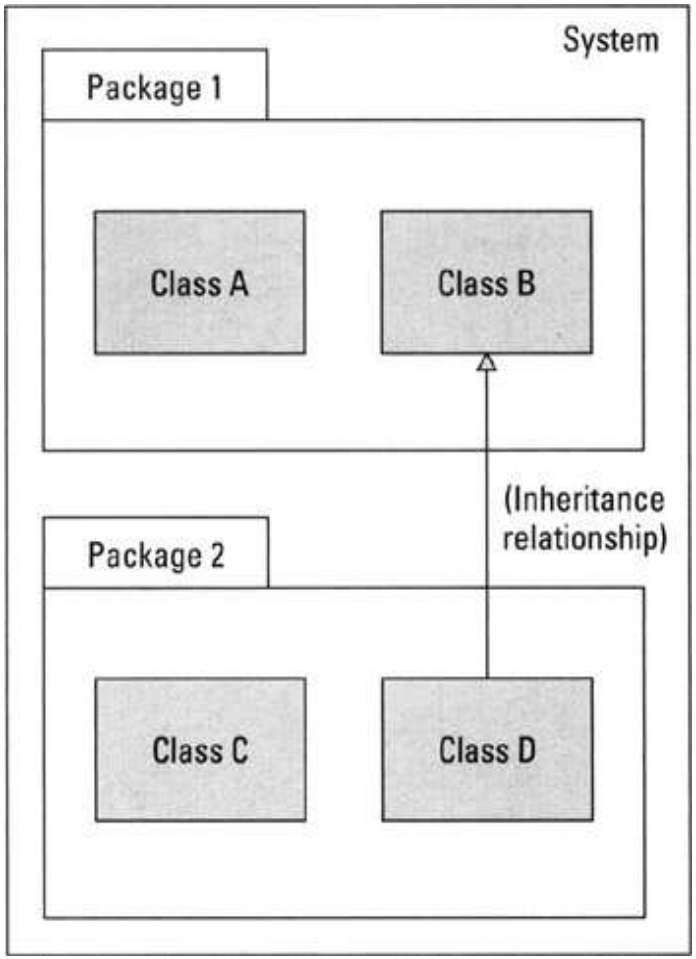
2) notation is "~"



Public Visibility

1) public element is visible to all elements that can access the contents of the namespace that owns it

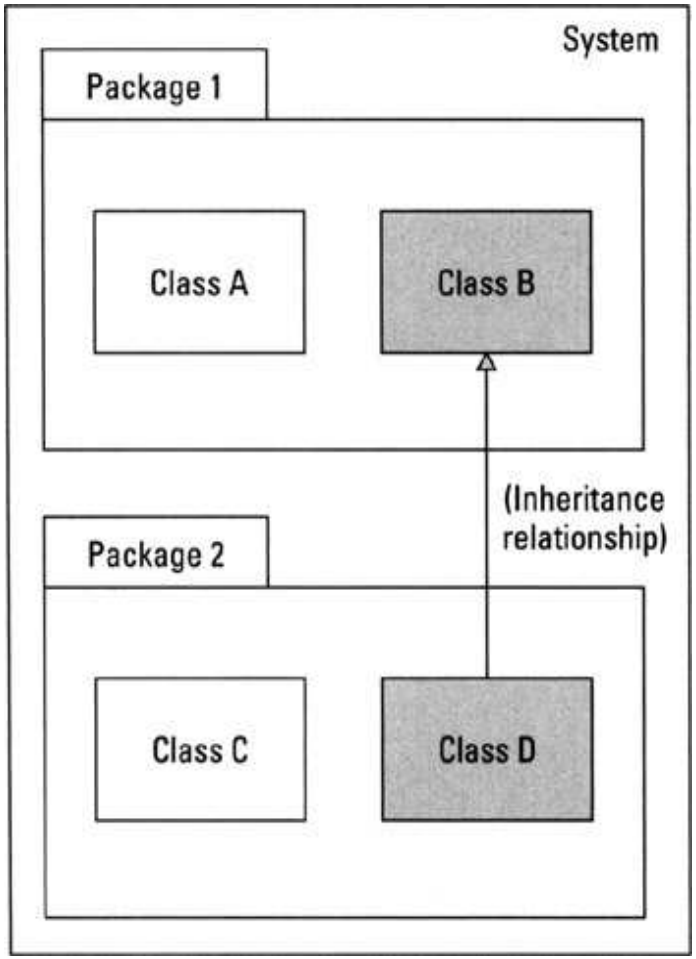
2) notation is “+”



Protected Visibility

1) a protected element is visible to elements that are in the generalization relationship to the namespace that owns it

2) notation is "#"



Adding Associations

The end of an association is called a role.

In DCDs the role may be decorated with a navigability arrow. Navigability is a property of a role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target.

Navigability indicates attributes visibility.

During implementation in an object-oriented programming language it is usually translated as the source class having an attribute that refers to an instance of the target class.

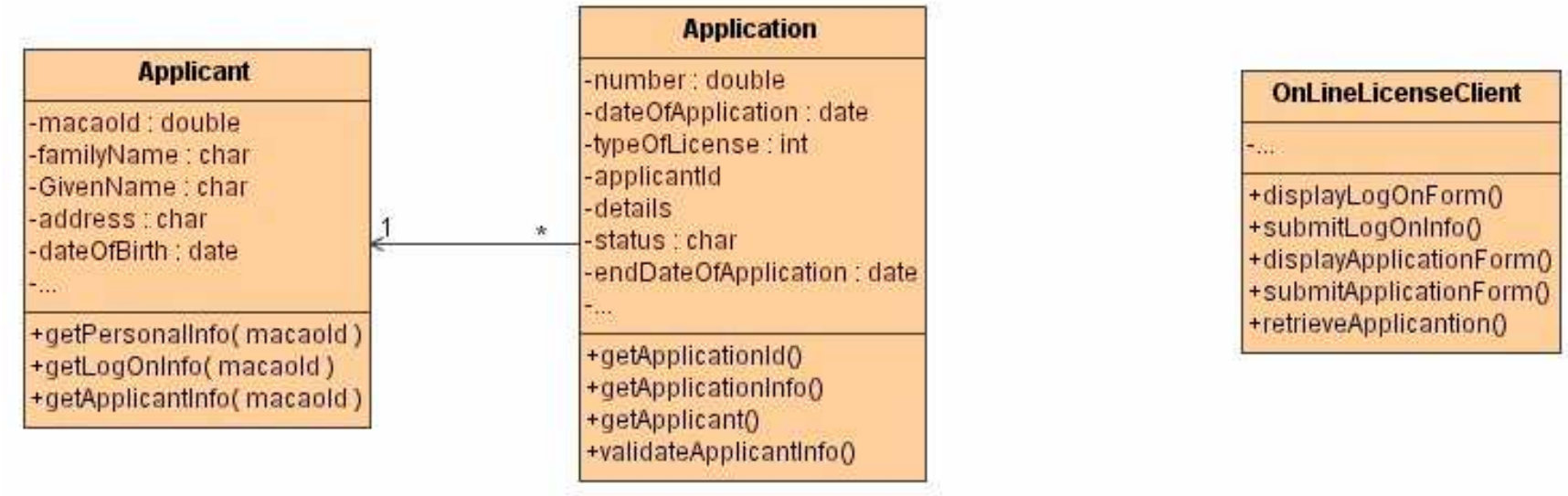
Most if not all associations in a DCD should be adorned with the necessary navigability arrows.

Adding Dependency

In class diagrams, the dependency relationship is useful to depict non-attribute visibility between classes.

For example dependency relationships are useful when there are parameters, global or locally declared visibility.

Example: Case Study



Design Sequence Diagrams

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

Design Sequence Diagrams

They present the interactions between objects needed to provide a specific behaviour.

They capture the sequence of events between participating objects, taking into account temporal ordering and optionally which objects are active at any time.

Design considerations:

a) different types of message

b) timed-messages

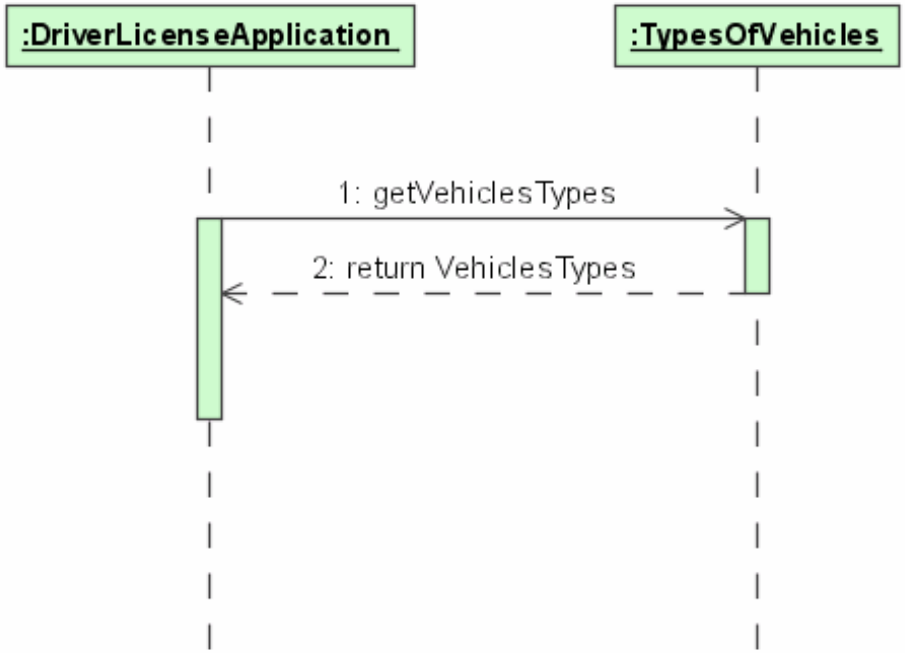
c) activation and deactivation

d) recursion

e) object creation and destruction

Synchronous Messages

Synchronous message: assumes that a return is needed, so the sender waits for the return before proceeding with any other activity.



Asynchronous Messages

1) asynchronous message: there is no need for an answer

2) signals are asynchronous messages

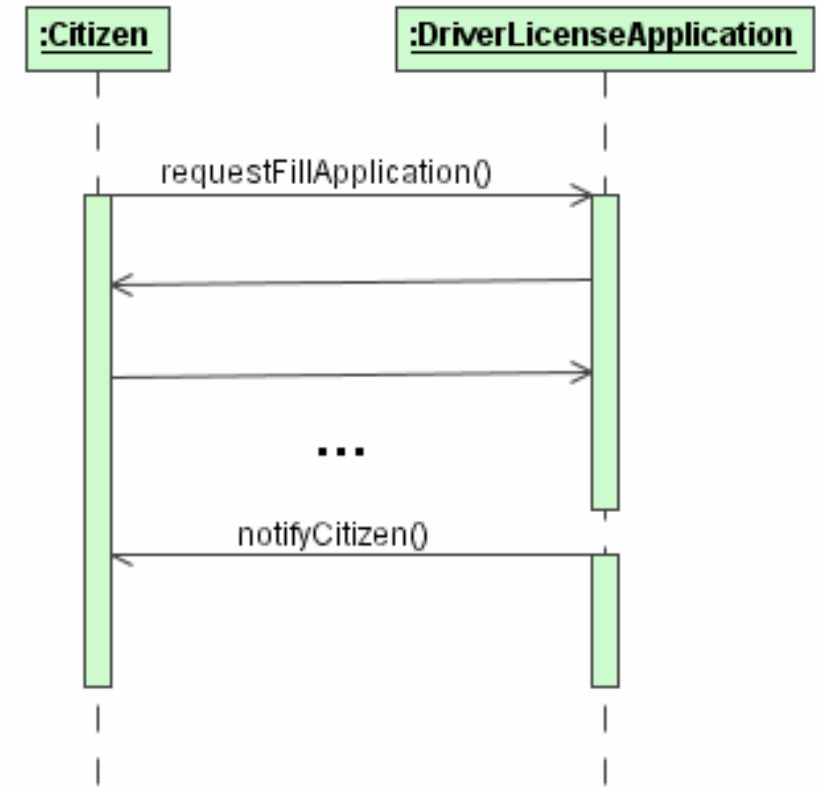
3) the sender is responsible only to get the message to the receiver and it has no responsibility or obligation to wait

4) the receiver may decide to do nothing or to process the received message

5) an asynchronous message is modelled using a solid line and a half arrowhead to distinguish it from the full arrowhead of the synchronous message

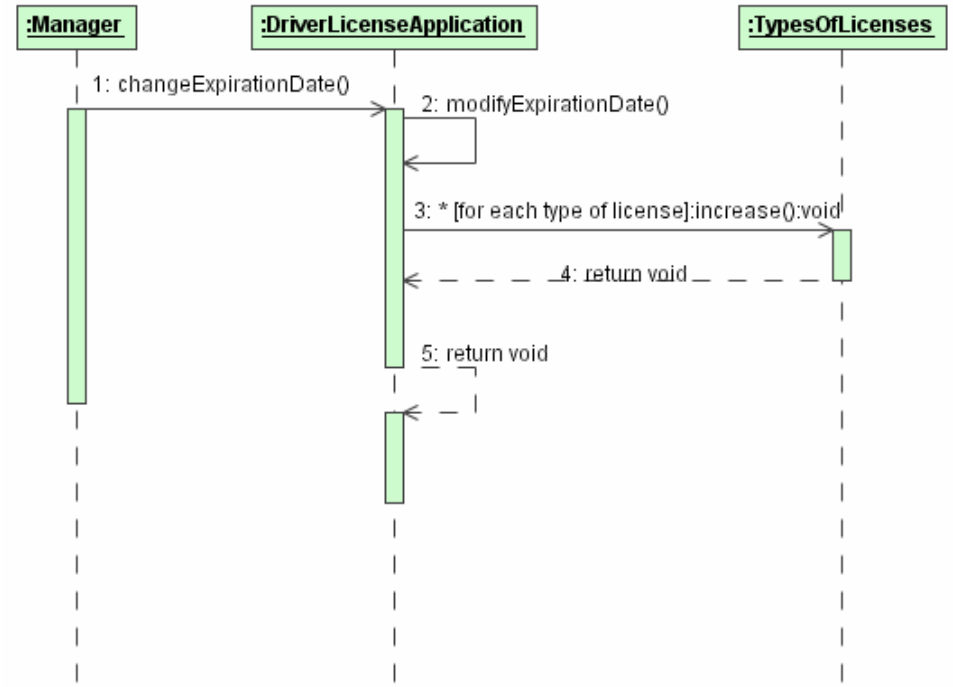
6) notation 

Example: Messages



Self-Reference Message

A self-reference message is a message where the sender and receiver are one and the same object.



1) in a self-reference message the object refers to itself when it makes the call

2) message 2 is only the invocation of some procedure that should be executed

Timed Messages

A message may have any number of user-defined time attributes, such as sentTime or receivedTime.

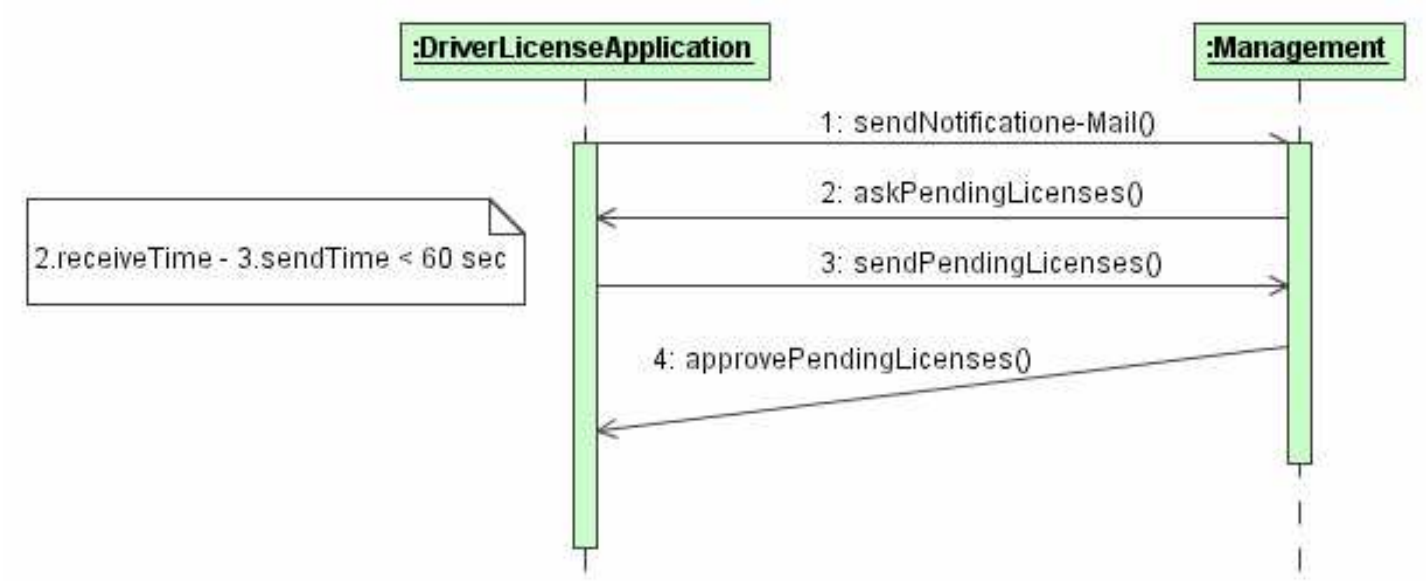
To access these values you need to identify the message that owns the value by using the message number or the message name.

This identification is useful to specify time constraints that may be described at the left margin.

Instantaneous messages are modelled with horizontal arrows.

For messages requiring a significant amount of time, it is possible to slant the arrow from the tail down to the head.

Example: Timed Messages



For messages 1, 2 and 3 the time required for their execution is considered equal to zero.

Message 4 requires more time (time > 0) for its execution.

Activation and Deactivation

Sequence diagrams can show object activation and deactivation.

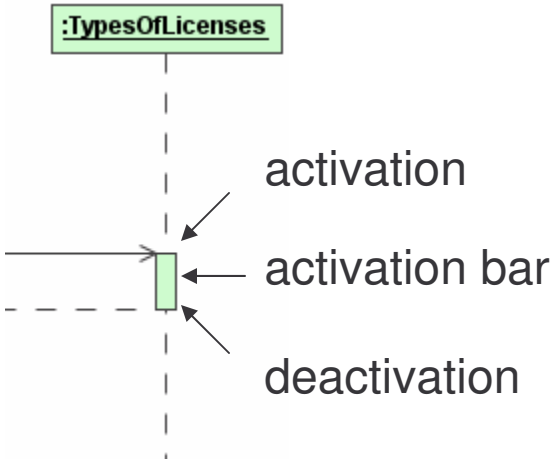
Activation means that an object is occupied performing a task.

Deactivation means that the object is idle, waiting for a message.

Activation and Focus of Control

Activation is shown by widening the vertical object lifeline to a narrow rectangle, called an activation bar or focus of control.

An object becomes active at the top of the rectangle and is deactivated when control reaches the bottom of the rectangle.

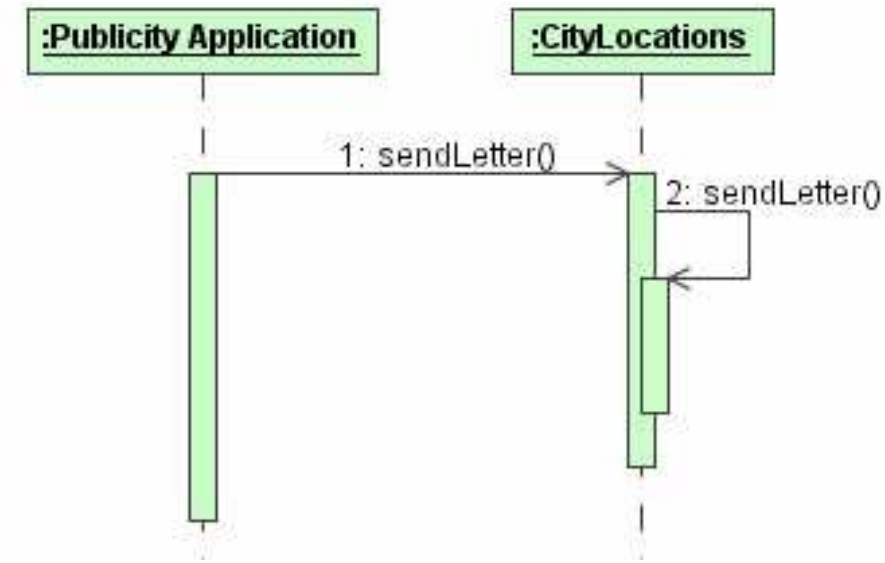


Recursion

An object might also need to call a message recursively, this means to call the same message from within the message.

Suppose that cityLocations is defined in the class diagram as a set of one or more apartments or houses.

You need to send a letter to all locations, so if a location is compound, the letter should be sent to all components.



Creation and Destruction

Object Creation:

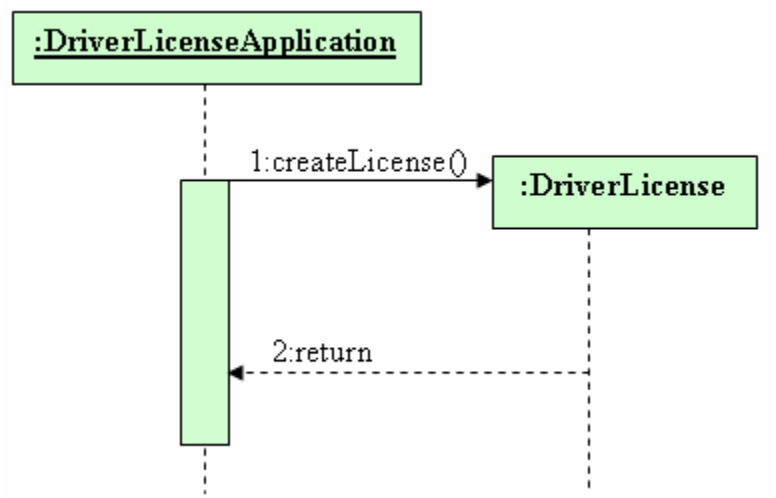
- if the object is created during the sequence execution it should appear somewhere below the top of the diagram.

Object Destruction:

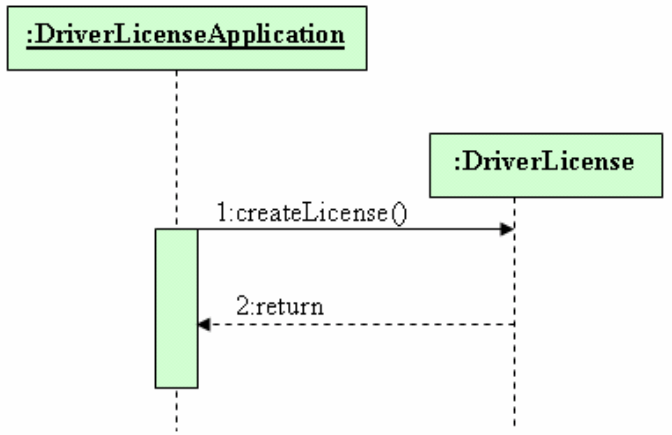
- if the object is deleted during the sequence execution, place an X at the point in the object lifeline when the termination occurs.

Examples: Object Creation

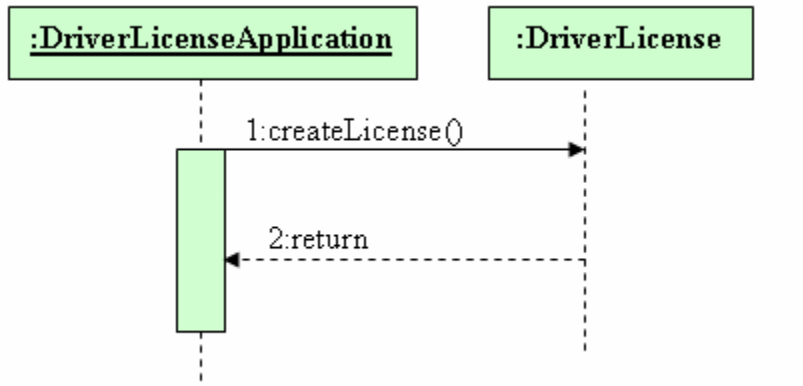
Alternative A



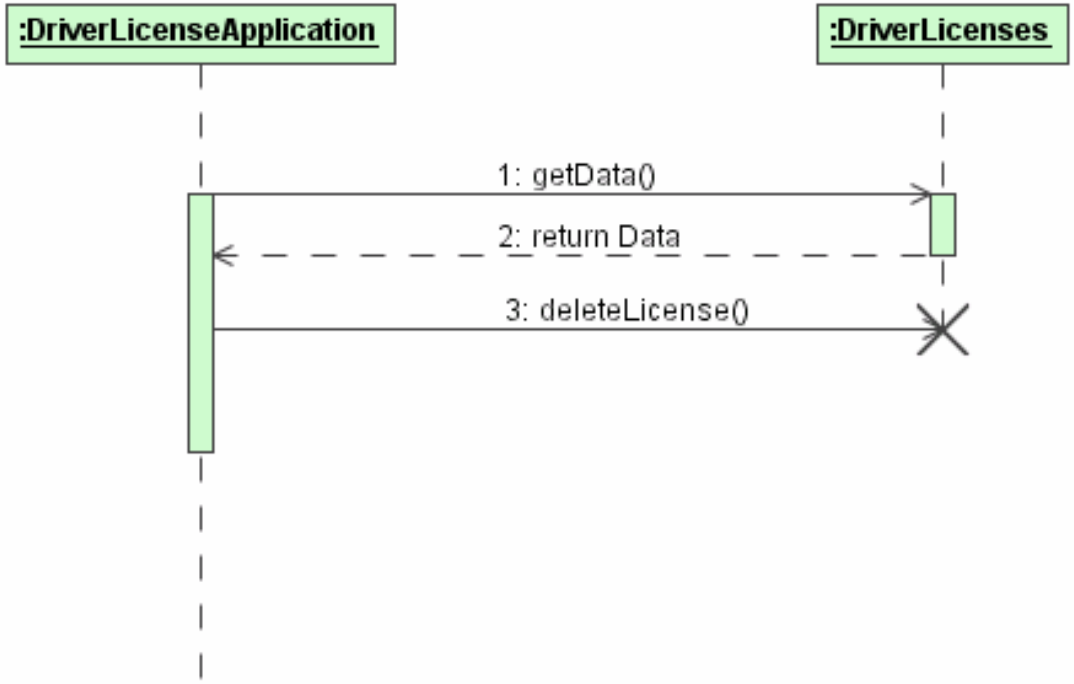
Alternative B



Alternative C

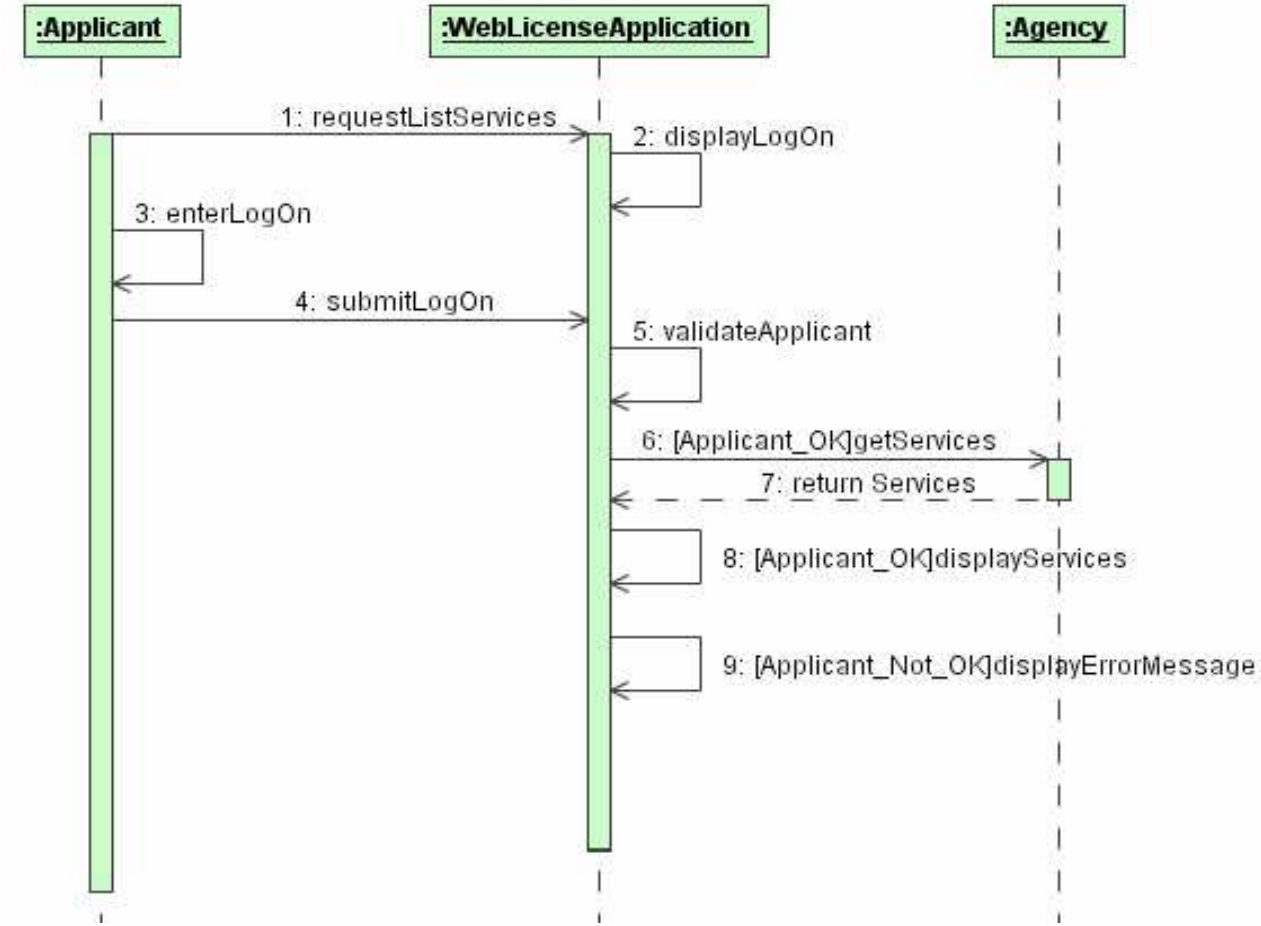


Example: Object Destruction

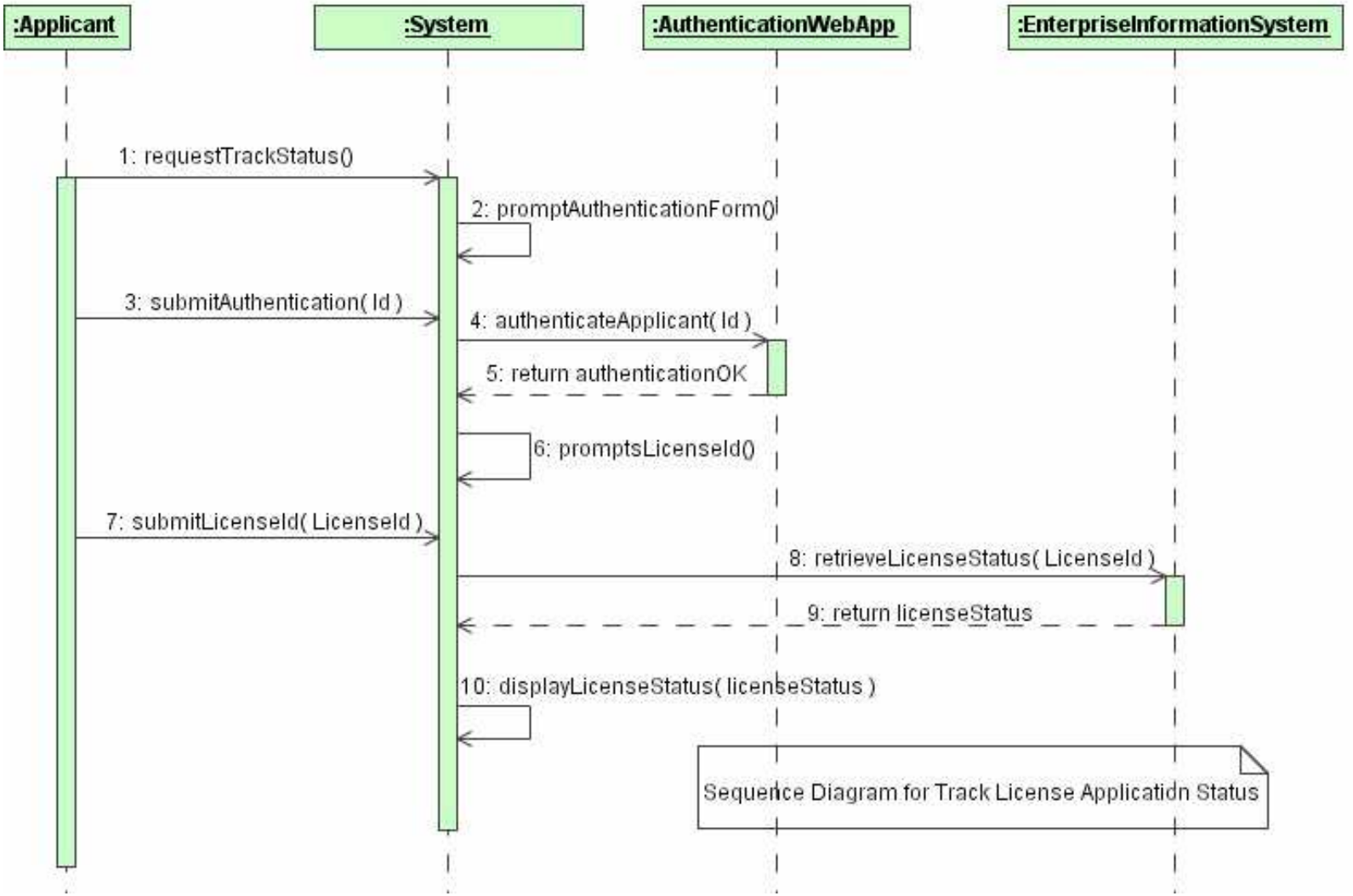


Advanced Features

It is possible to combine several scenarios in one sequence diagram:



Example: Case Study



Activity Diagrams

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

Activity Diagrams 1

1) it is a variation of the state machine in which the states represent the performance of actions or sub-activities and the transitions are triggered by the completion of the actions or sub-activities

2) one of the five diagrams in UML for modelling the dynamic aspect of a system.

   Others: sequence, collaboration, statechart and use cases

3) it is attached through a model to a classifier, such as a use case, or to a package, or to the implementation of an operation

4) its purpose is to focus on flows driven by internal processing (as opposed to external events)

Activity Diagrams 2

5) it involves modelling the dynamic aspects of a system; usually the sequential (but possibly concurrent) steps in a computational process

6) activity diagrams also model the flow of an object as it moves from one state to another at different points in the flow of control

7) they are used in situations where all or most of the events represent the completion of internally-generated actions (procedural flow of controls)

Action State

An action state is a shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action.

There may be several outgoing transitions with guard conditions.

Action states may not have:

a) internal transitions
b) outgoing transitions based on explicit events or exit actions

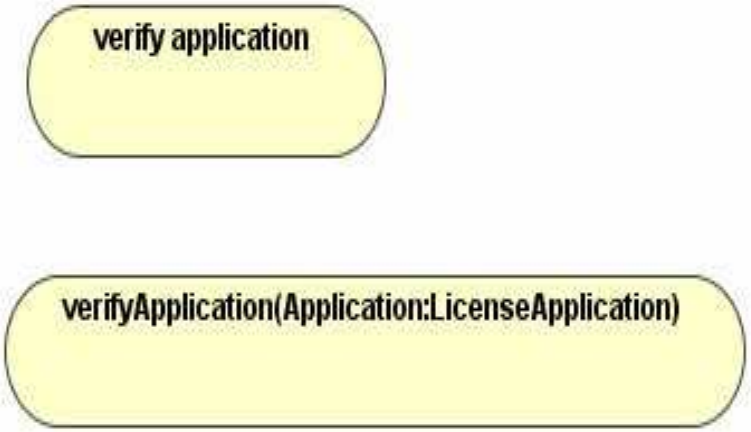
It models a step in the execution of a workflow.

Action State – Notation

It is shown as a shape with straight top and bottom and with convex arcs on the two sides.

Action expression is placed in the action state symbol.

Action expression may be described as natural language, pseudocode, action language or programming language.



Subactivity State

It invokes an activity graph.

When a subactivity state is entered, the activity graph nested in it is executed as any activity graph would be.

The subactivity graph is not exited until the final state of the nested graph is reached.

A single activity graph may be invoked by many subactivity states.

Subactivity State – Notation

It is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram.

The name of the subactivity is placed in the symbol.

Transitions

They specify the flow of control from one action or activity state to another.

Transitions in activity diagrams are triggerless.

Control passes immediately once the work of the source state is done.

Once the action of a given source state completes, the state's exit action (if any) is executed; next without delay control follows the transition and passes on to the next action or activity state.

Decisions

Decisions are expressed as guard conditions.

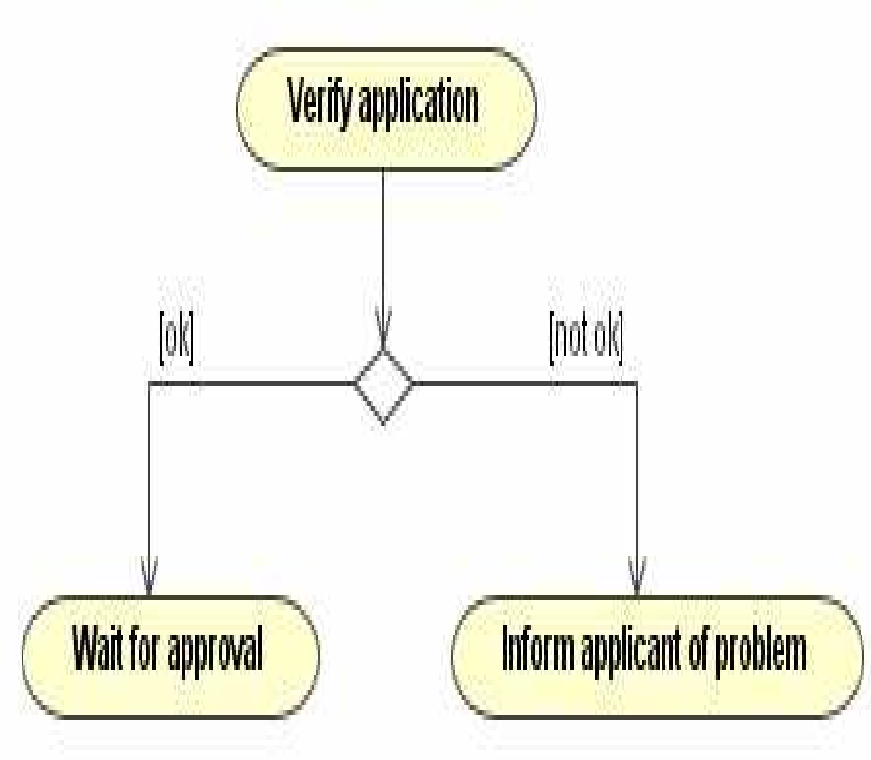
Different guards are used to indicate different possible transitions that depend on boolean conditions of the owning objects.

The predefined guard else may be defined for at most one outgoing transition which is enabled if all the guards labeling the other transitions are false.

Decision - Notation

Decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each with a distinct guard condition with no event trigger.



Forks and Joins 1

They are used for modelling concurrency and synchronization in business processes.

Synchronization bar is used to specify the forking and joining of these parallel flows of control.

A fork represents the splitting of a single flow of control into two or more concurrent flows of control.

Activities of each of these flows are truly concurrent (in the case of a system deployed across multiple nodes) or sequentially interleaved if deployed on a single node.

Forks and Joins 2

A join represents the synchronization of two or more concurrent flows of control.

Joins may have two or more incoming transitions and one outgoing transition.

Above a join, the activities associated with each of these paths continues in parallel.

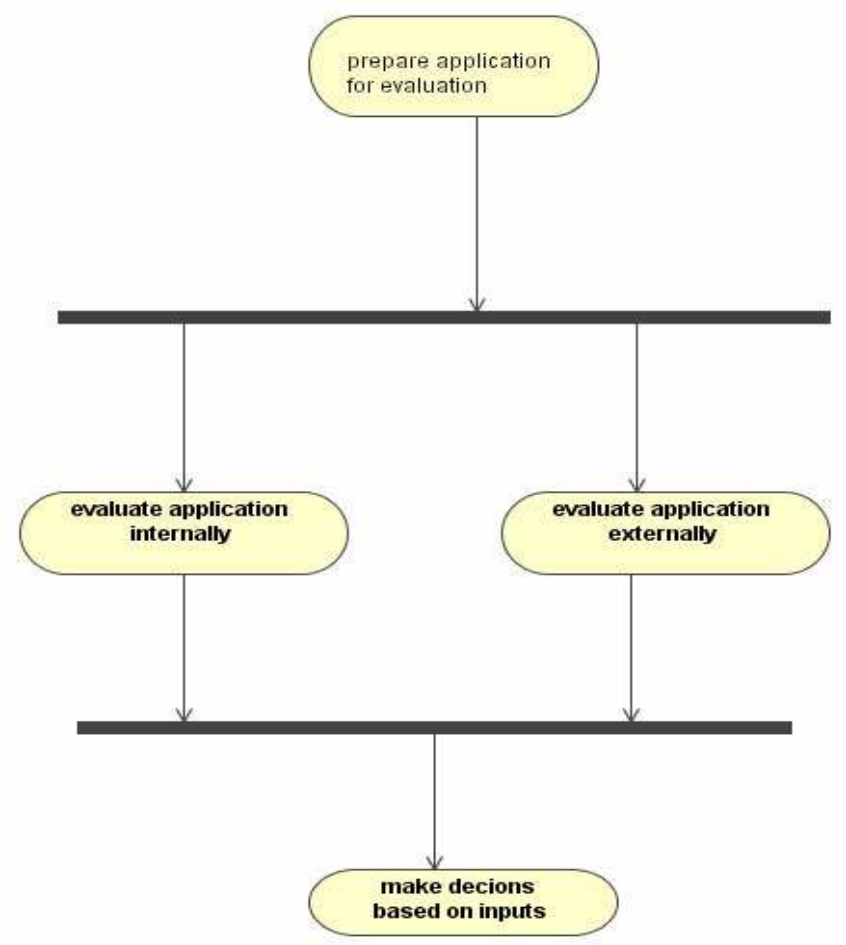
At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join at which point one flow of control continues on below the join.

Forks and Join Example

1) application is prepared for evaluation

2) internal and external evaluation proceeds concurrently

3) decision is made only when the internal and external evaluation are completed



Swimlanes

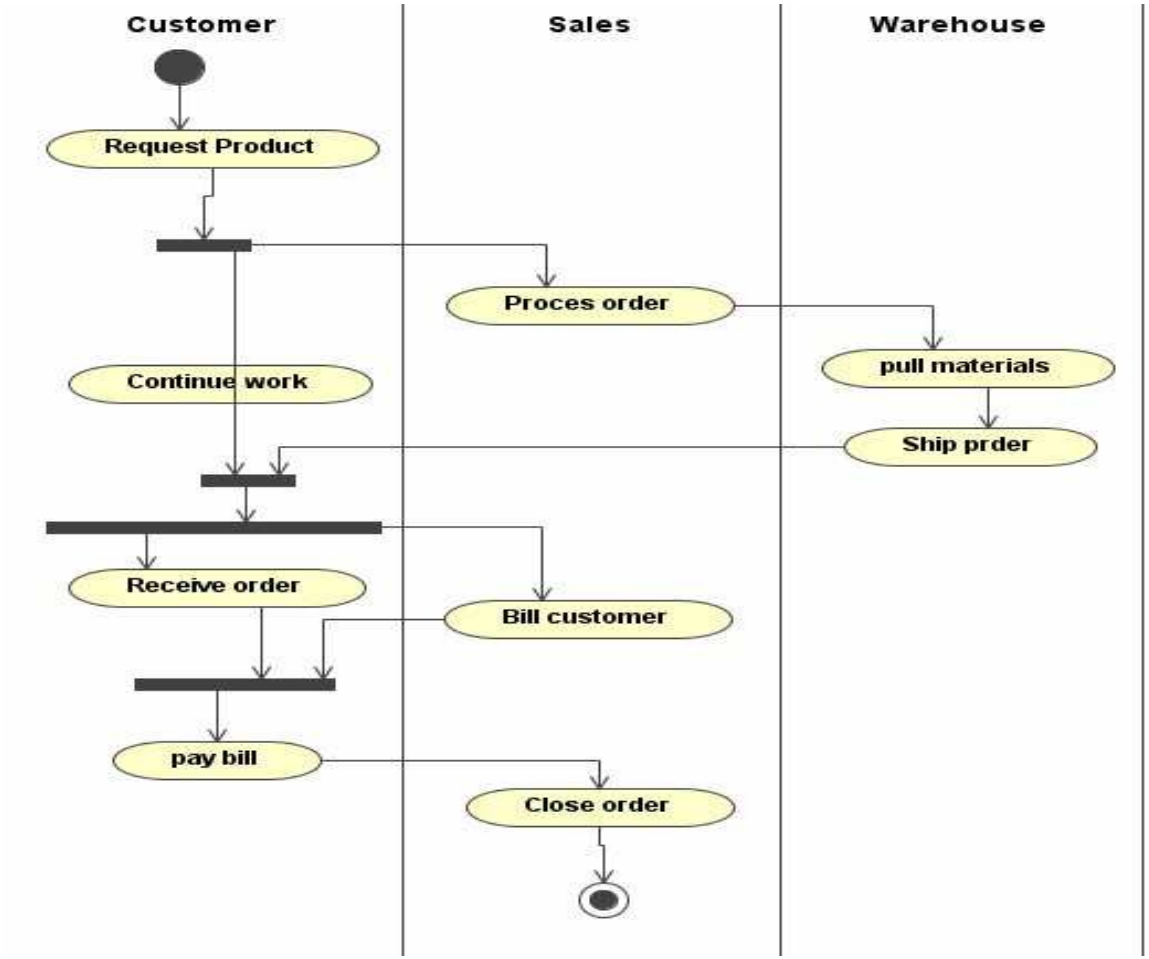
1) we may wish to partition the activity states on an activity diagram into groups

2) these groups may represent the business organization responsible for those activities

3) each of these groups is called a swimlane and it specifies a locus of activities.

4) each swimlane has a unique name within a diagram

Example: Swimlanes



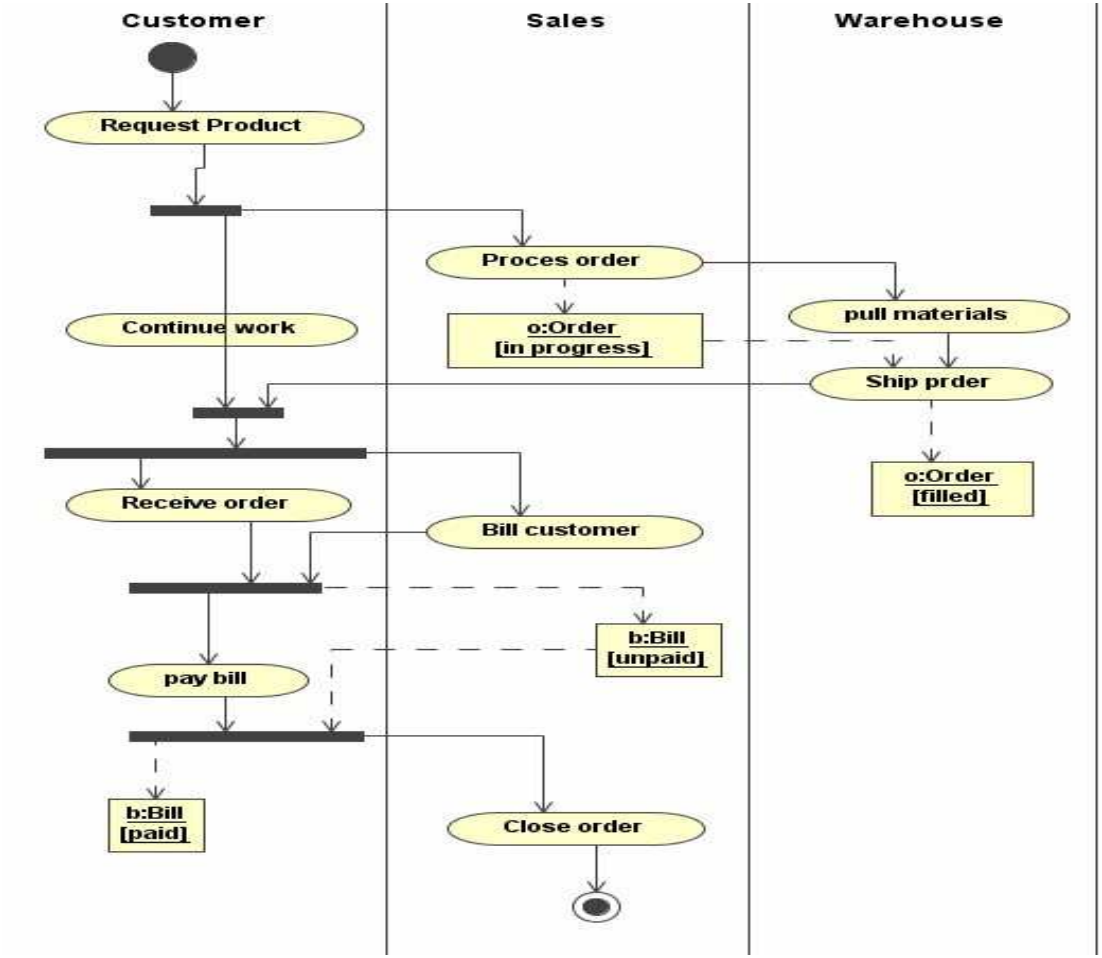
Object Flow

1) objects may be involved in the flow of control associated with an activity diagram

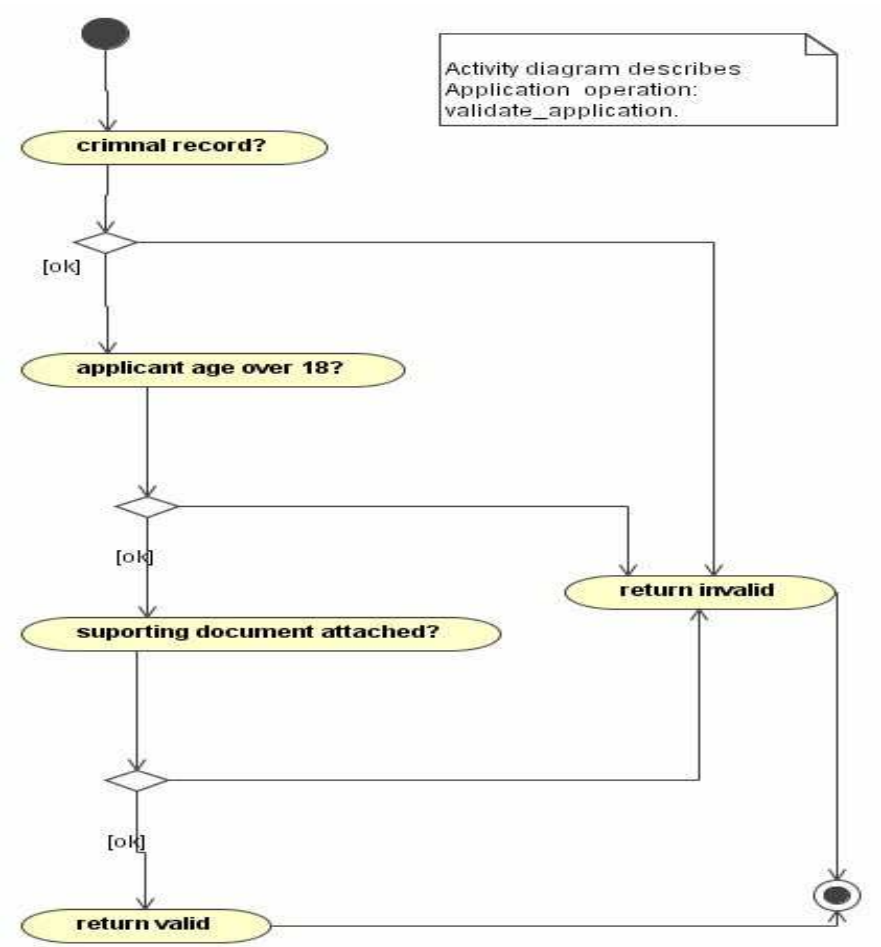
2) for example: in the workflow of processing an order as shown in the last example, we may wish to show how the state of the *Order* and *Bill* object changes.

3) we can use dependency relationship to show how objects are created, modified and destroyed by transitions in the activity diagram

Example: Object Flow



Example: Case Study



Design Statechart Diagrams

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

Design Statechart Diagrams

Collaboration and sequence diagrams are used to understand how the objects collaborate within the system.

Statechart diagrams illustrate how these objects behave internally.

Statecharts diagrams relate events to state transitions and states.

The transitions change the state of the system and are triggered by events.

During the design phase we add more implementation details to the statechart diagrams or we develop new statechart diagrams.

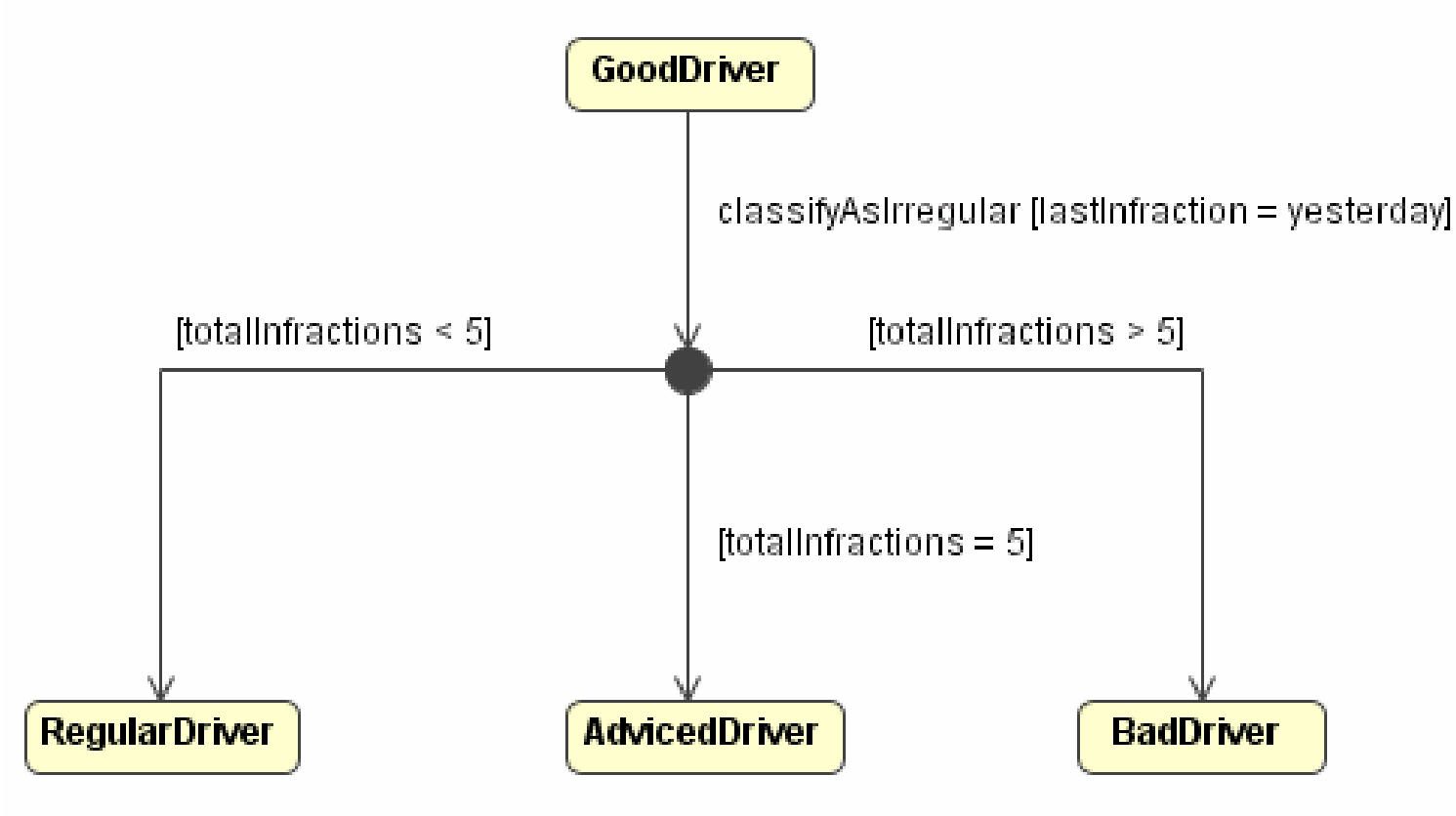
Static Branch Point

Another pseudo state provided by UML is the static branch point that allows a transition to split into two or more paths.

Notation:

It provides a means to simplify compound guard conditions by combining the like portions into a single transition segment, then branching based on the portions of the guard that are unique.

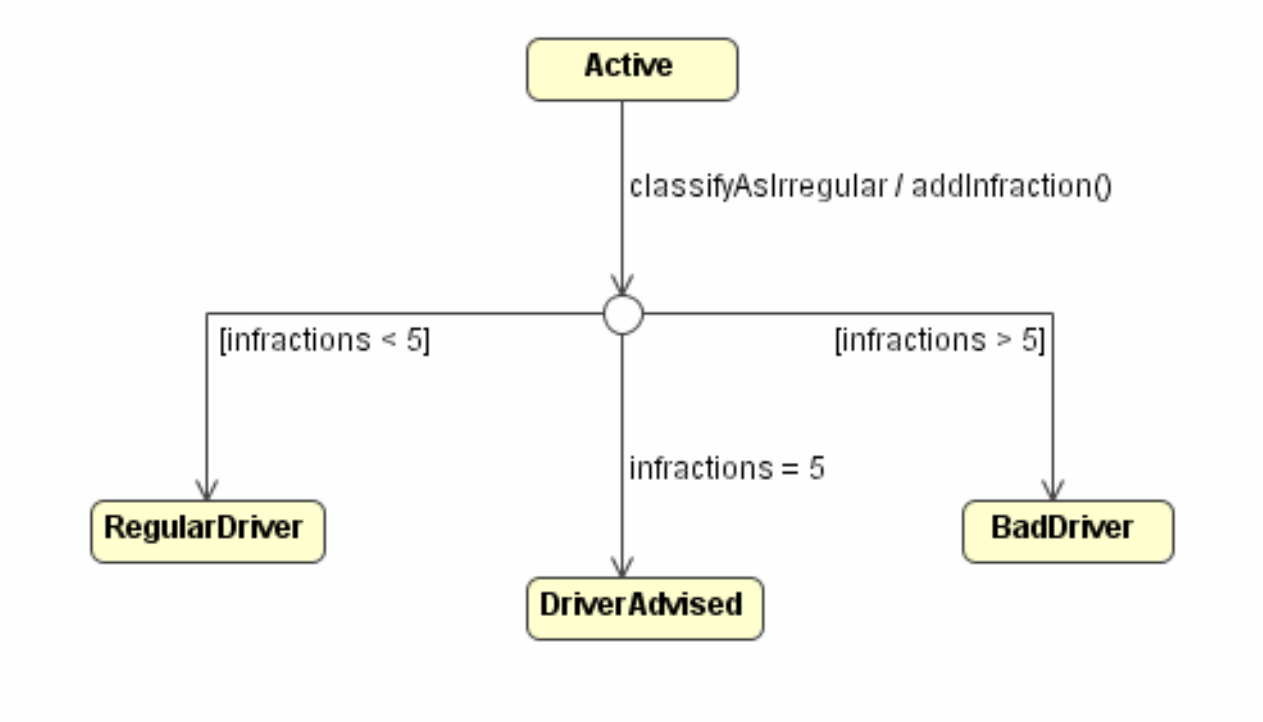
Example: Static Branch Point



Dynamic Branch Point

In other transitions, the destination is not known until the associated action has been completed.

Notation: ○



Modelling Composite States

A composite state is simply a state that contains one or more statechart diagrams.

Composite states may contain either:

1) a set of mutually exclusive states: is literally like embedding a statechart diagram inside a state
2) a set of concurrent states: different states divided into regions, active at the same time

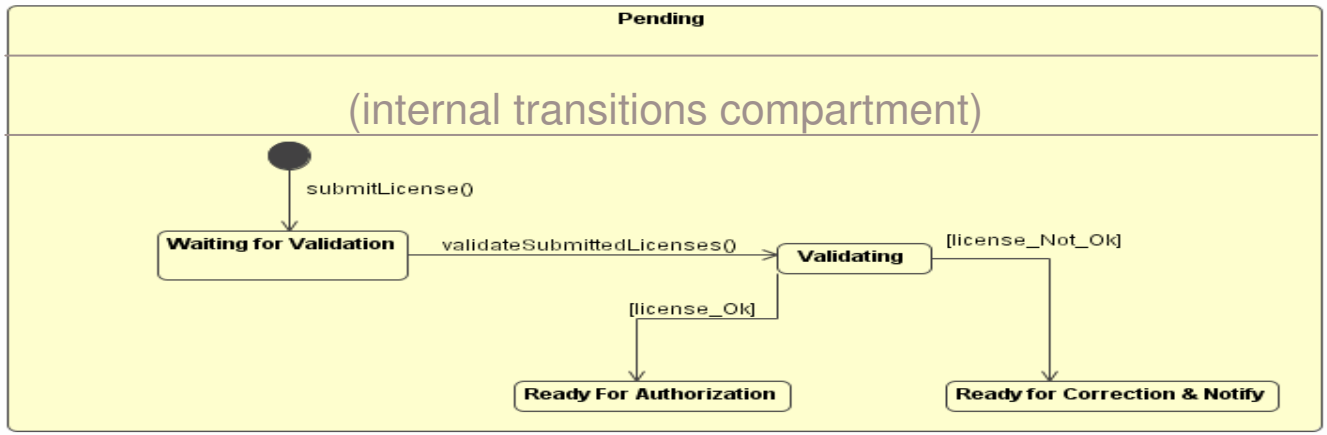
A composite state is also called a super-state, a generalized state that contains a set of specialized states called sub-states.

Sub-States

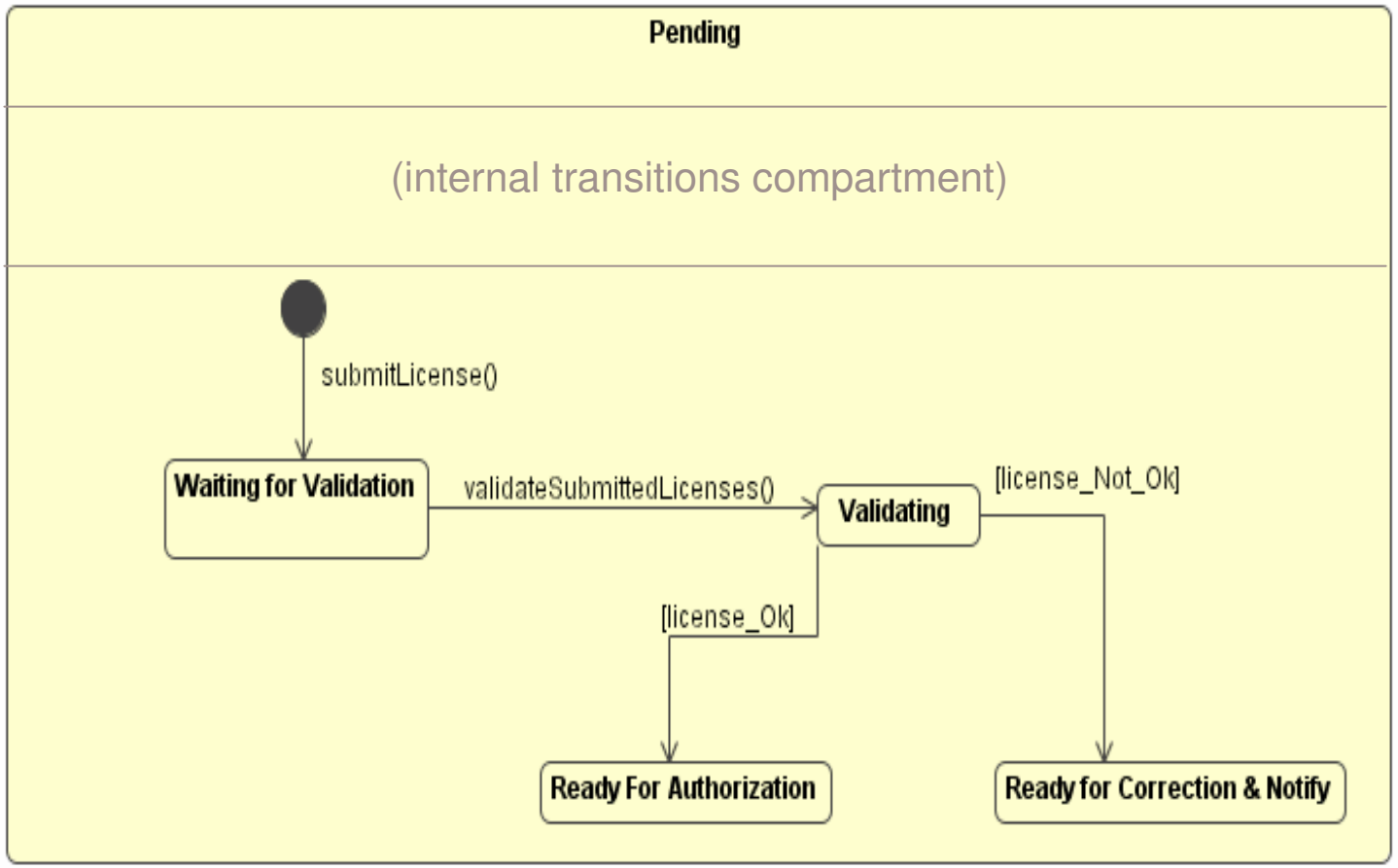
A composite state (super-state) may be decomposed into two or more lower-level states (sub-states).

All the rules and notation are the same for the contained sub-states as for any statechart diagram.

Decomposition may have as many levels as needed.



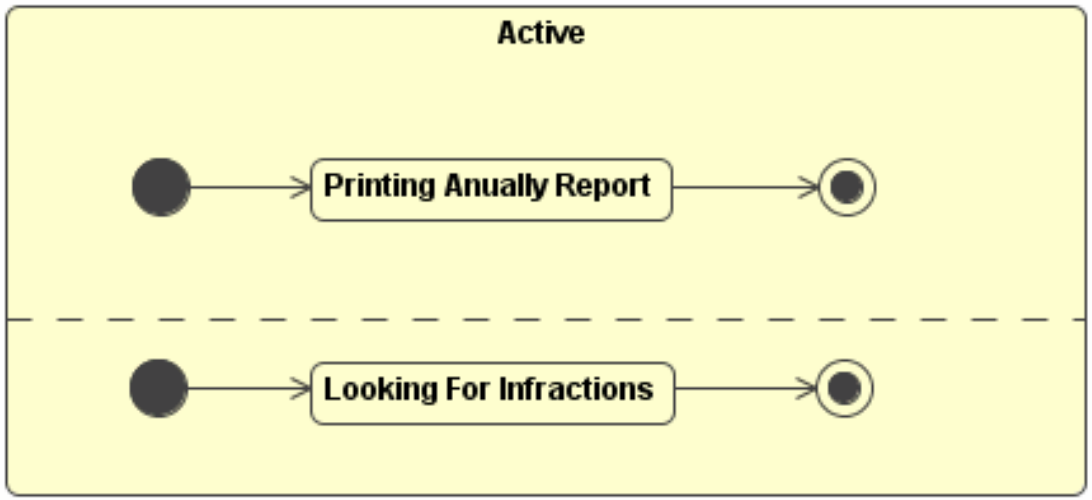
Example: Sub-States



Concurrent Sub-States

Modelling concurrent sub-states implies that you have many things occurring at the same time.

To isolate them, the composite state is divided into regions, and each region contains a distinct statechart diagram.



Sub-Machine States

A sub-machine state is a kind of shorthand for referring to an existing statechart diagram.

Within a composite state, it is possible to reference to a sub-machine state in the same way that a class may call a subroutine or a function of another class.

The composite state containing the sub-machine is called the containing state machine, and the sub-machine is called the referenced state machine.

Access to sub-machines states is through entry and exit points that are specified by stub states.

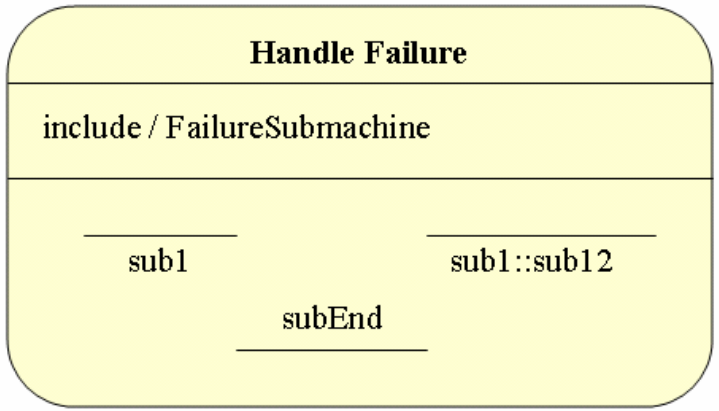
Notation for Sub-Machine

The containing state icon models the reference to the sub-machine adding the keyword include followed by a slash plus the name of the submachine state.

The stub state uses a line with the state name placed near the line.

Entry points are represented with a line and the name under the line.

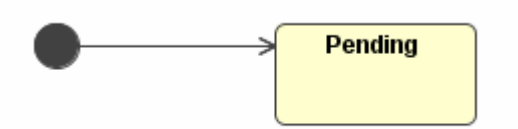
Exit points are represented with a line and the name over the line.



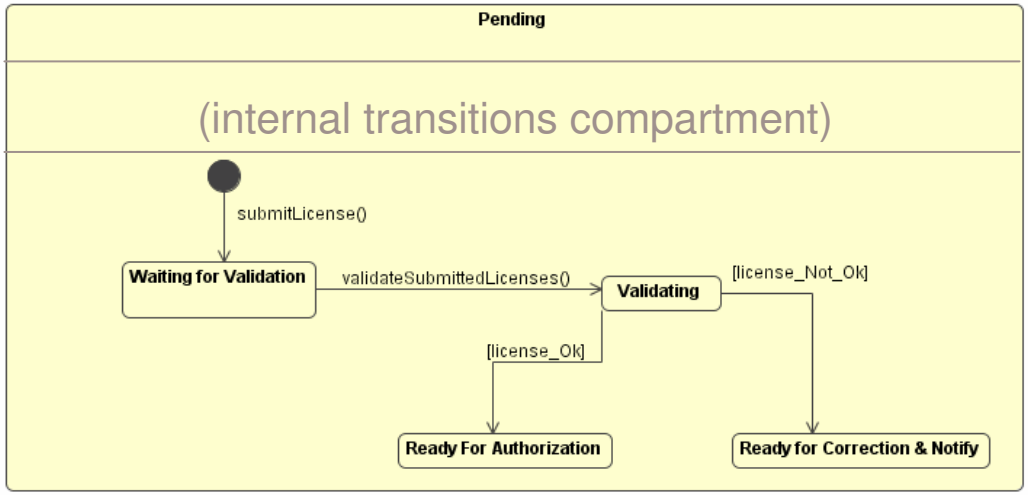
Transitions to Sub-States 1

Alternative A:

- draw a transition pointing to the edge of the composite state icon
- it means that the composite state starts in the default initial state
- the default initial state is the sub-state associated with the initial state icon in the contained statechart diagram



The initial sub-state is Waiting for Validation

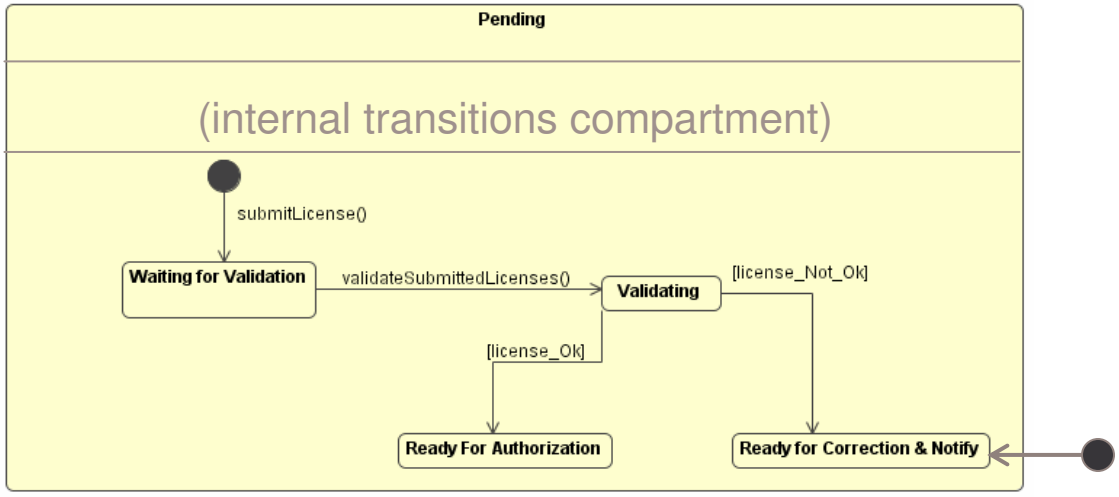


Transitions to Sub-States 2

Alternative B:

- draw a transition through the edge of the super-state to the edge of the specific sub-state
- it means that the composite state starts in that specific sub-state

The initial sub-state is Ready for Correction & Notify

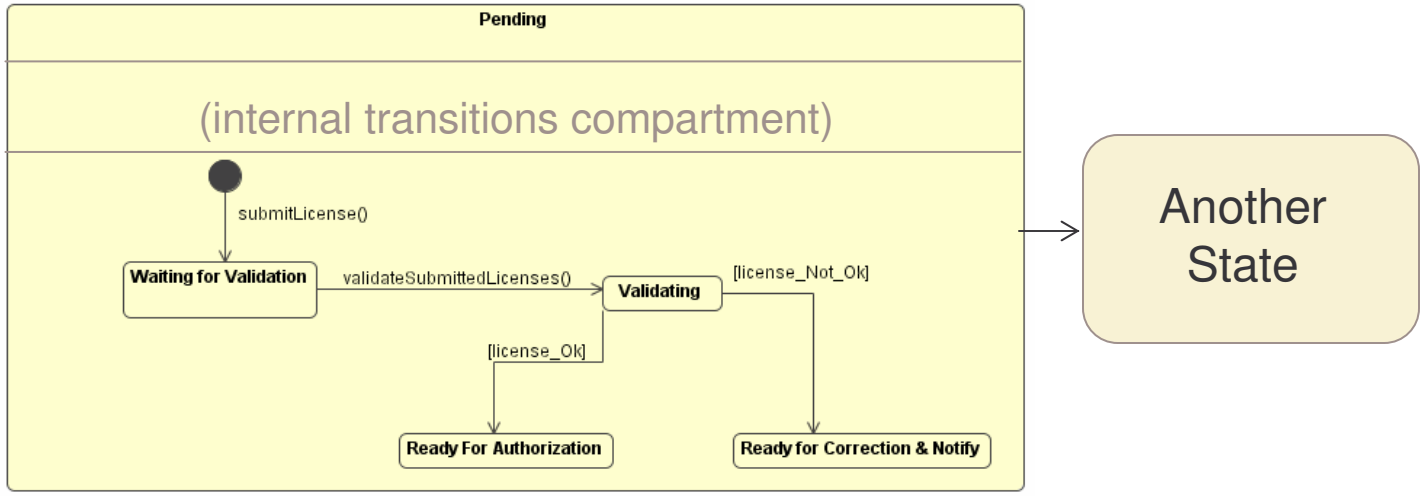


Transitions from Sub-States 1

Alternative A:

1) an event can cause the object to leave the super-state regardless of the current sub-state
2) draw the transition from the edge of the super-state to the new state.

At any sub-state the object can do a transition to AnotherState

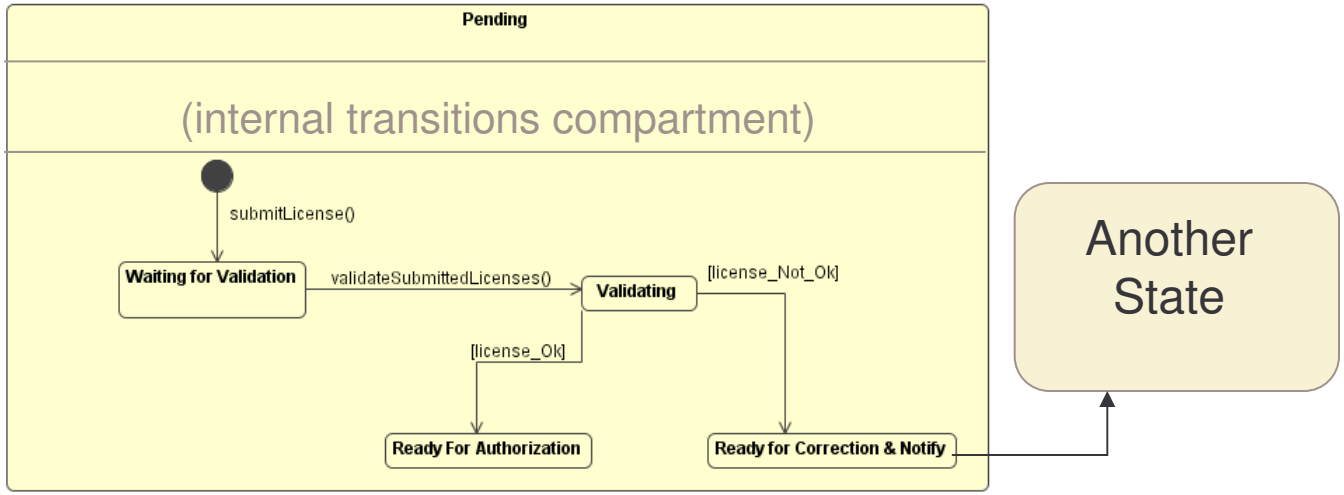


Transitions from Sub-States 2

Alternative B:

1) an event can cause the object to leave the super-state directly from a specific sub-state
2) implies that the exit event may only happen when the object is in the specific sub-state
3) draw the transition from the edge of the specific sub-state through the edge of the super-state.

Only from sub-state Ready for Correction & Notify the object can make a transition to AnotherState

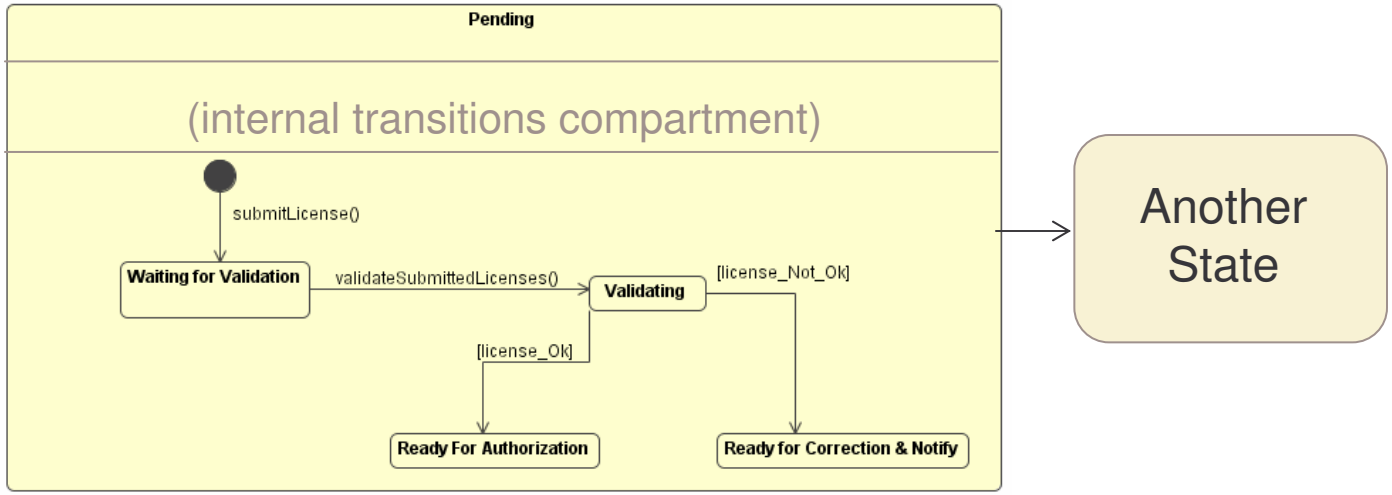


Transitions from Sub-States 3

Alternative C:

1) an object may leave a super-state because all the activities in the state and its sub-state has been completed
2) it is called an automatic transition
3) draw the transition from the edge of the super-state to the new state

When the activities finish an automatic transition changes the state to AnotherState



History Indicator 1

The history indicator is used to represent the ability for doing backtracking to a previous composite state.

Represents a pseudo-state and is a shorthand notation to solve a complex modelling problem.

May refer to:

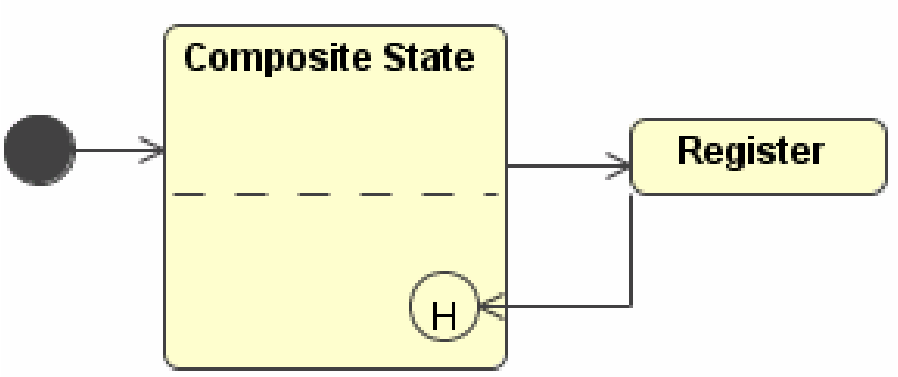
1) shallow history: the object should return to the last sub-state on the top most layer of sub-states
2) deep history: the object needs to return to the exact sub-state from it which left, no matter how many layers down that is

History Indicator 2

Notation:

1) shallow history 

2) deep history 



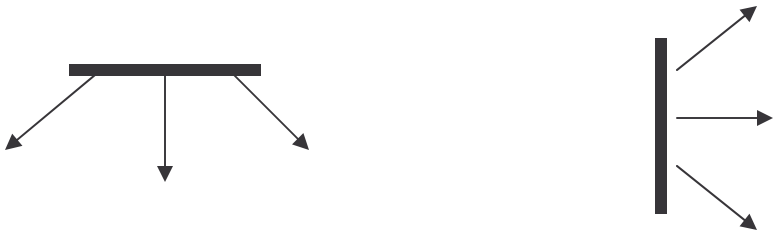
Split of Control

Split of control means that based on a single transition it is necessary to proceed with several tasks concurrently.

Notation:

1) a single transition divided into multiple arrows, each pointing to a different sub-state

2) the division is accomplished by the synchronization bar



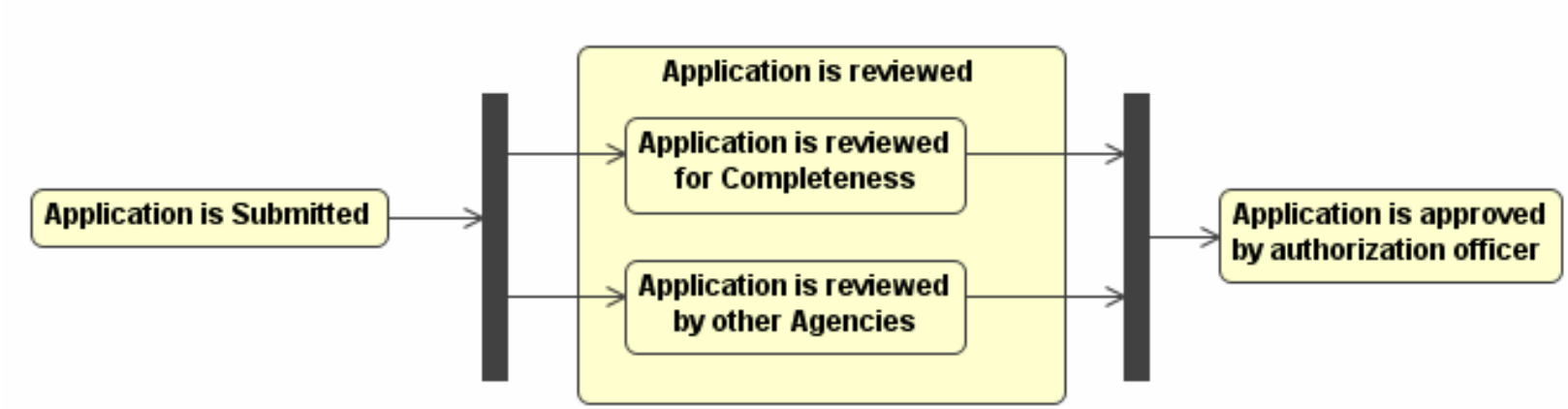
Merge of Control

Merge of control means that based on the completion of a number of transitions it is necessary to proceed with a single task.

Notation:

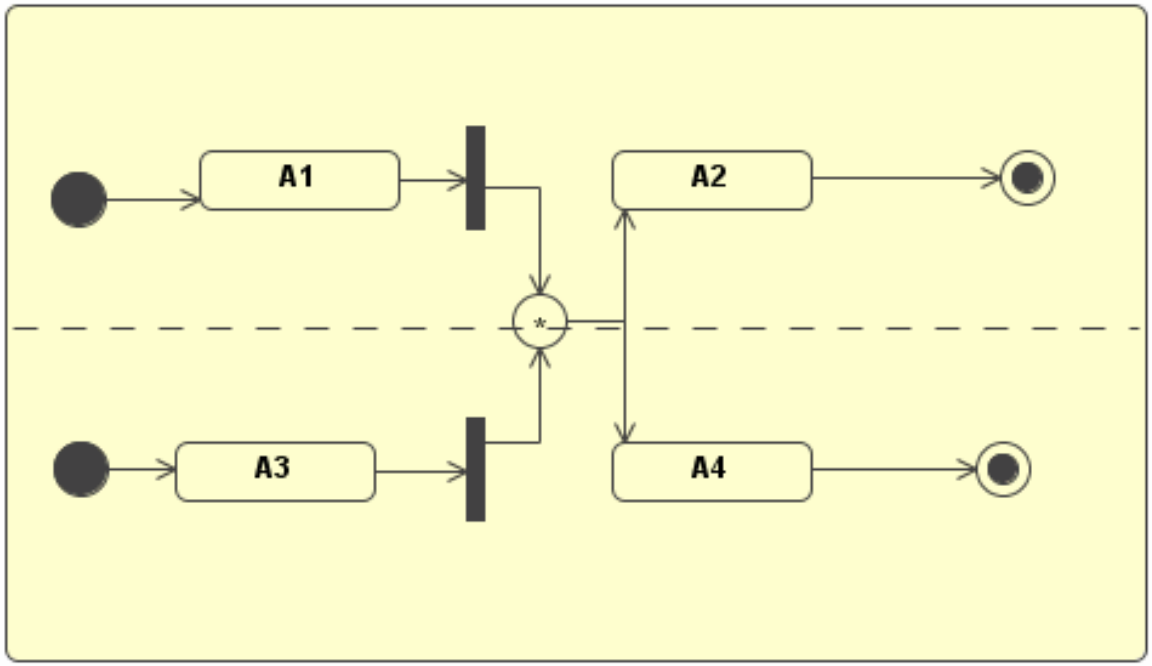
- multiple transitions converge to a synchronization bar and only one transition outputs from the bar

Example: Split/Merge Control

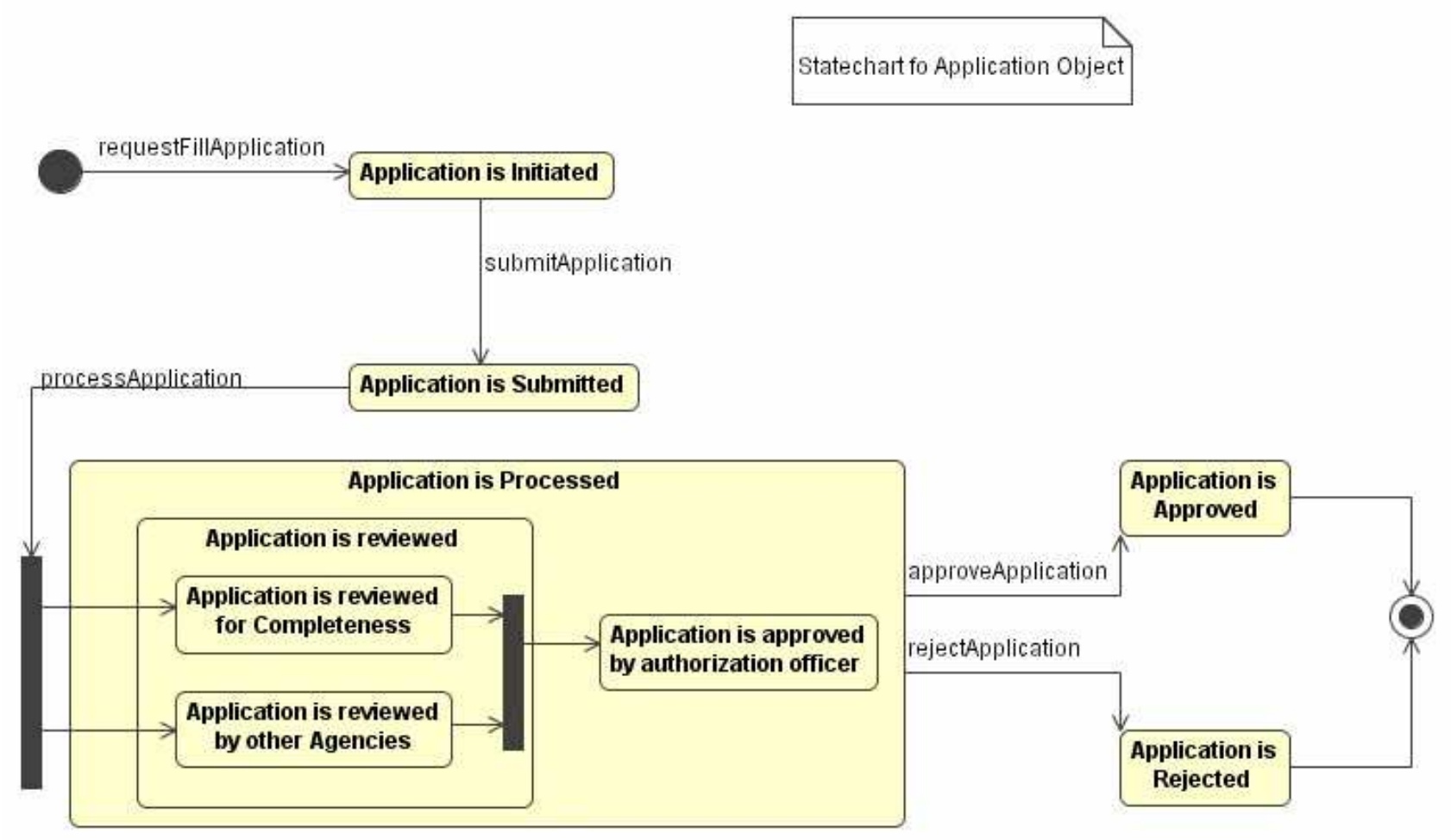


Synchronous States

Synchronous states extend the split and merge of control across regions to coordinate the behaviour of concurrent states.



Example: Case Study



Summary

Design Modelling

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

Summary 1

System design involves the transformation of analysis models of the application domain into design models of the solution space.

Object design includes the service specification for classes, component selection, object model restructuring and object model optimization.

Design patterns are partial solutions to common problems. They name, abstract and identify the key aspects of common design structure that make them useful for creating reusable object oriented design.

Summary 2

There are 23 basic patterns as provided by the GoF (slide 395).

Design class diagrams provide specification for software classes and interfaces in an application.

Typical information contained in a Design Class Diagram include classes, associations, attributes, interfaces, methods, attribute type information, navigability and dependencies.

Design Sequence Diagram present the interactions between objects needed to provide a specific behaviour.

Summary 3

Design sequence diagrams will specify message types (synchronous and asynchronous), temporal constraints, object creation and destruction and recursion.

Activity diagrams models the dynamic aspect of a system by showing the flow from one activity to another.

Activity diagram can be used to describe a workflow or the details of an operation.

An activity diagram may also show the flow of an object as it moves from one state to another at different points in the flow of control.

Summary 4

Design statechart diagrams describes detailed the internal behavior of objects.

Design Statechart diagrams provide more implementation details with features such as sub-states, static branch points, sub-machine states etc.

Exercise 1

1. Identify and list the relevant software classes for your system by analyzing the various interaction diagrams provided as part of your requirement models.

2. By using some of the information contained in the conceptual class diagrams, describe the attributes and the operation of the software classes. Indicate the types for attributes and operations of the classes as well their visibilities.

3. Present your software classes in a design class diagram and indicate navigability on class associations.

4. Add dependencies to your design class diagram.

Exercise 2

5. Provide a design sequence diagram based on your software classes to describe the important interactions in your system (for example those associated with your use cases and scenarios).

6. For each of the messages specified in your interactions, indicate the type of the message (synchronous or asynchronous) and their parameters.

7. Consider the operations specified in the design classes. Provide activity diagrams to describe the details of these operations.

Exercise 3

8) Consider five objects associated with your software or design classes, describe in details the behaviour of these objects using a design statechart.

Implementation Model

The Course: Overview

1) The Course

2) Object Oriented Concepts

3) UML Basics

4) Case Study

5) Requirement Model

6) Architecture Model

7) Design Model

8) Implementation Model

9) Deployment Model

10) Unified Process

11) Tools

12) Summary

Overview

1) Implementation Phase

2) Implementation Model

3) Implementation Component Diagram

4) Elements in the Component Diagram:
   a) Packages
   b) Components and Interfaces

5) Modelling Techniques

6) Summary

Implementation Phase

Involves the implementation of the design models.

It considers non-functional requirements and the deployment of the executable modules onto nodes.

Two models are developed during this phase:

1) the implementation model describing how the design elements will be implemented in terms of software system
2) the deployment model describing how the implemented software will be deployed on the physical hardware

Implementation Model

The implementation model indicates how various aspects of the design map onto the target language.

It describes how components, interfaces, packages and files are related.

The modelling elements are:

1) packages
2) components
3) their relationships

and they are shown in the implementation component diagram.

Packages

A package represents a physical partitioning of the system.

Packages are organized in a hierarchy of layers.

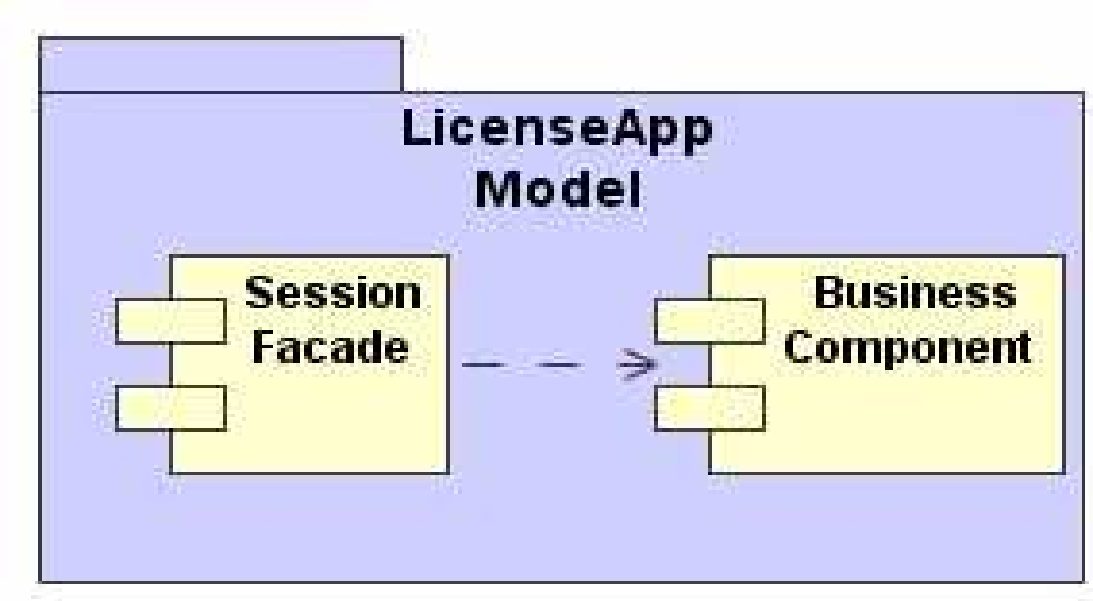
Packages are useful for:

1) associating related components and interfaces
2) resolving naming problems
3) providing some privacy for classes, attributes, and operations that should not be visible outside the package

Example: Package

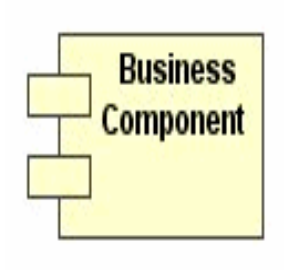
The package LicenseApp Model has two components:

1) SessionFaçade
2) BusinessComponent



Components

A component:



1) is a physical part of a system
2) is replaceable
3) conforms to and provides the realization of a set of interfaces

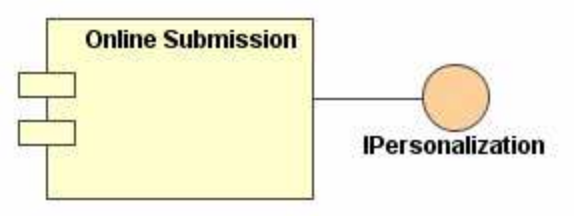
Components are grouped in packages.

Components can be organized by specifying dependency, generalization, association, aggregation, and realization relationships among them.

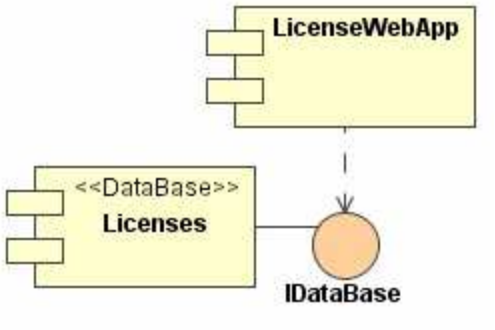
Component modelling is very important to control the versioning and configuration management as system evolves.

Components and Interfaces

The component that realizes an interface is connected to it using a full realization relationship.



The component that accesses the services of other component through the interface is connected to the interface using a dependency relationship.



Modelling Techniques

Component diagrams are used to model the static implementation view of a system.

To model this view, it is possible to use component diagrams in one of four ways:

1) to model source code
2) to model executable releases
3) to model physical databases
4) to model adaptable systems

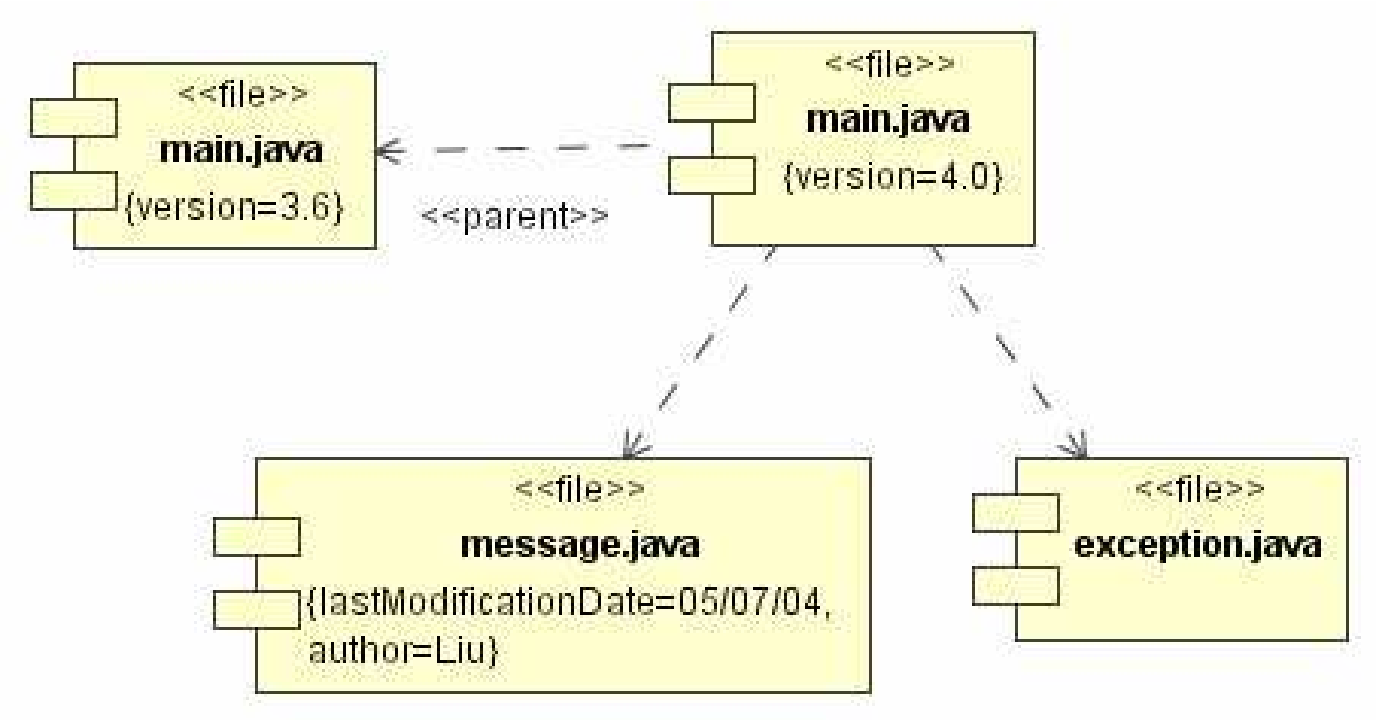
Modelling Source Code

Component diagrams are used to model the configuration management of source files, which represent work-product components.

Modelling procedure:

1) identify the set of source code files of interest, either by forward or reverse engineering, and model them as components stereotyped as files
2) for larger systems, use packages to show groups of source code files
3) consider exposing a tagged value to show interesting information such as author, version number, etc.
4) model the compilation dependencies among these files using dependencies

Example: Source Code Files



Modelling Executable Releases

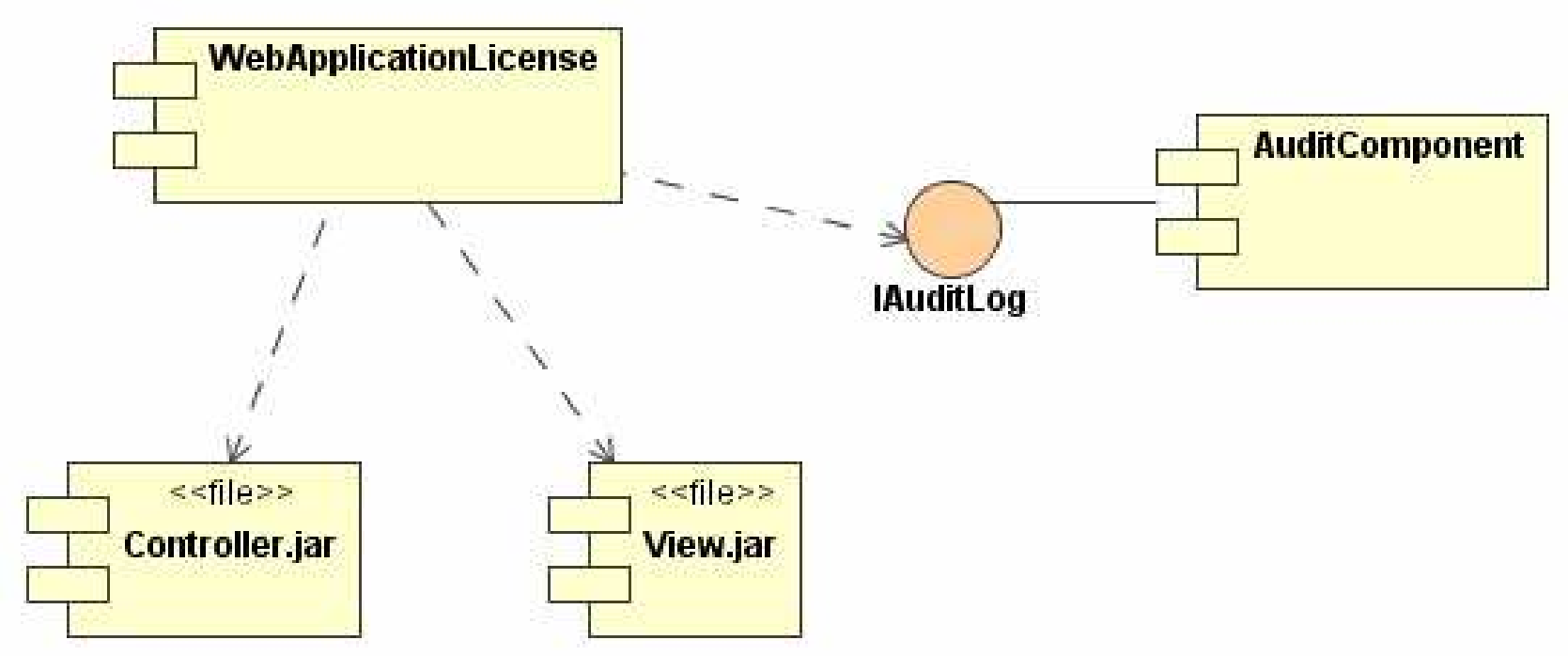
Component diagrams are used to visualize, specify, construct, and document the configuration of the executable releases, including the deployment components that form each release, and the relationships among those components.

Each component diagram should focus on one set of components at a time, such as all components that live on one node.

Modelling procedure:

1) identify the set of components to model
2) consider the stereotype of each component in the set
3) for each component, consider its relationship to its neighbors

Example: Executables



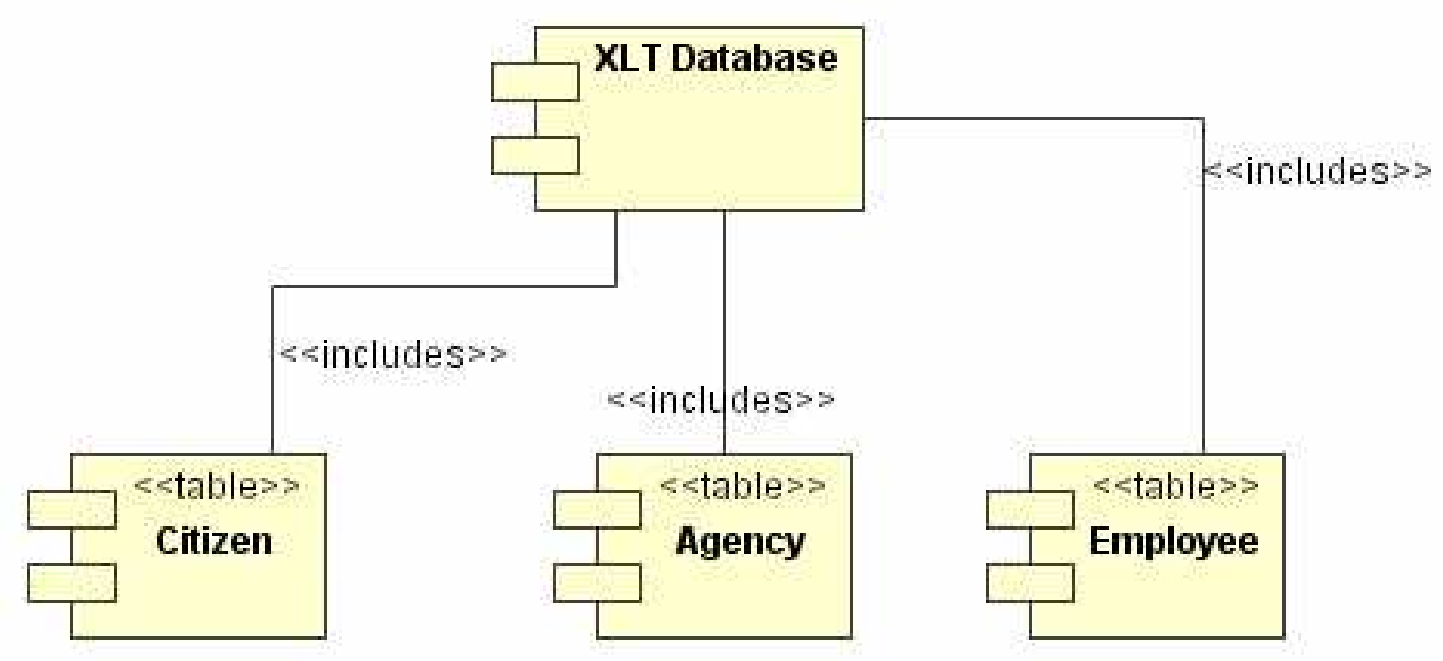
Modelling Physical Databases

Component diagrams can be used to visualize, specify, construct and document the mapping of classes into tables of a database.

Modelling procedure:

1) identify the classes in your model that represent your logical database schema
2) select a strategy for mapping these classes to tables, possibly considering the physical distribution
3) create a component diagram containing components stereotyped as tables to model the mapping

Example: Physical Databases



Modelling Adaptable Systems 1

All the component diagrams shown were used to model static views.

A static view means that its components spend their entire life on one node.

In some distributed systems is necessary to model dynamic views.

A dynamic view models components migrating from one node to another.

To model a dynamic view we need a combination of:

1) component diagrams
2) object diagrams
3) interaction diagrams

Modelling Adaptable Systems 2

Modelling procedure:

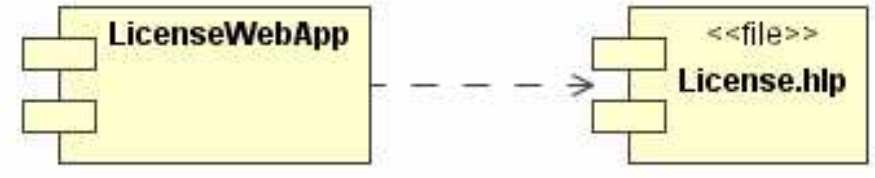
1) consider the physical distribution of the components that may migrate from node to node
2) specify the location of a component instance by marking it with a tagged value, which you can then render in a component diagram
3) model the actions that cause a component to migrate creating a corresponding interaction diagram that contains component instances

Tables, Files and Documents

Implementation might include data files, help documents, scripts, log files, initialization files, and installation/removal files.

Modelling procedure:

1) identify the auxiliary components that are part of the physical implementation
2) model these things as components (introduce new stereotypes as needed)
3) model the relationships among these auxiliary components and other executables.



APIs

An Application Programming Interface (API) is essentially an interface that is realized by one or more components.

One concept, two perspectives:

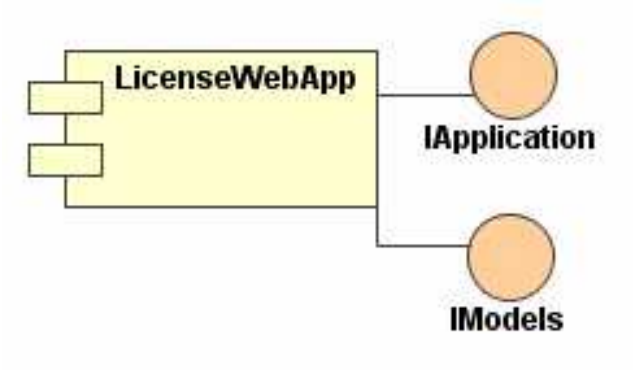
1) as developer: interested only in the interface
2) as system configuration management: interested in which component realizes the interface

The operations associated with any semantically rich API will be fairly extensive, and mostly it is not needed to visualize them. Instead, interfaces grouping these operations are modelled.

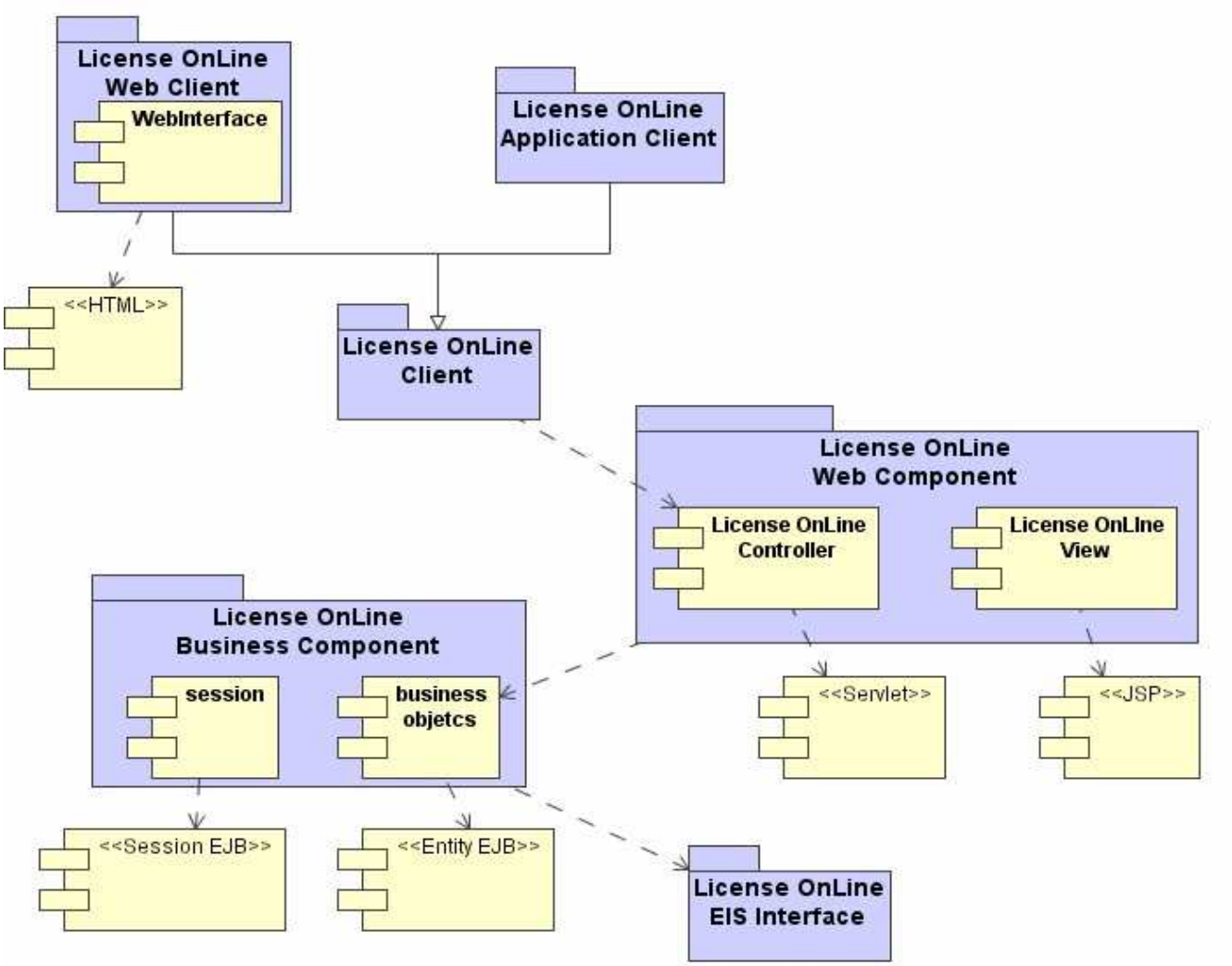
Modelling an API

Procedure:

1) identify the programmatic seams and model each as an interface, collecting the corresponding attributes and operations
2) expose only those properties of the interface that are important to visualize in the given context
3) model the realization of each API only if it is important to show the configuration of a specific implementation



Example: Case Study



Summary

The implementation component diagram describes how components, interfaces, packages and files are related.

Implementation component diagrams may be used to model source code, executable releases, physical databases and adaptable systems.

Deployment Model

The Course: Overview

1) The Course

2) Object Oriented Concepts

3) UML Basics

4) Case Study

5) Requirement Model

6) Architecture Model

7) Design Model

8) Implementation Model

9) Deployment Model

10) Unified Process

11) Tools

12) Summary

Overview

1) deployment diagrams

2) nodes

3) nodes and components

4) common uses of deployment diagrams

Deployment Diagrams

A deployment diagram shows the configuration of run-time processing nodes and the components that live on them.

They are used to model the distribution, delivery, and installation of the parts that make up the physical system.

It involves modelling the topology of the hardware on which the system executes.

Deployment diagrams are essentially class diagrams that focus on a system's nodes, and include:

1) nodes
2) dependencies and associations relationships
3) components
4) packages

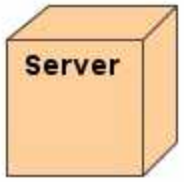
Nodes

Nodes are used to model the topology of the hardware on which the system executes.

The components developed or reused must be deployed on some set of hardware in order to execute.

Nodes represent the hardware on which these components are deployed and executed.

UML notation for nodes allows for visualizing a node independently of any specific hardware.

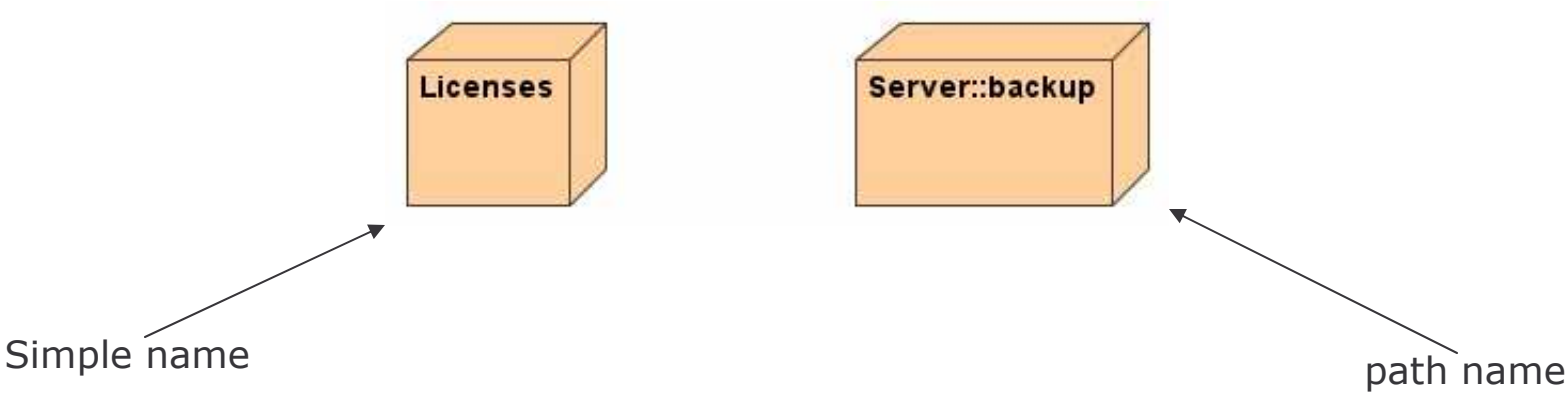


Stereotypes allows to represent specific kinds of processors and devices.

Nodes Names

Every node must have a name that distinguishes it from other nodes.

A name is a textual string which may be written as a simple name or as a path name.



Nodes and Components

Components

1) participate in the execution of a system.

2) represent the physical packaging of otherwise logical elements

Nodes

1) execute components

2) represent the physical deployment of components

The relationship deploys between a node and a component can be shown using a dependency relationship.

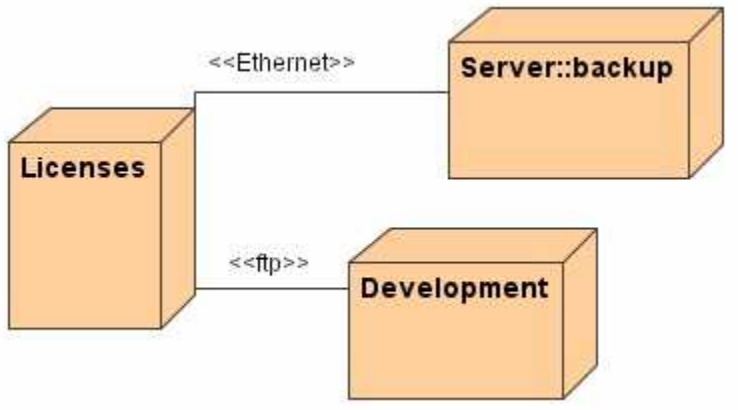
A set of objects or components that are allocated to a node as a group is called a distribution unit.

Organizing Nodes

Nodes can be organized:

1) in the same manner as classes and components
2) by specifying dependency, generalization, association, aggregation, and realization relationships among them.

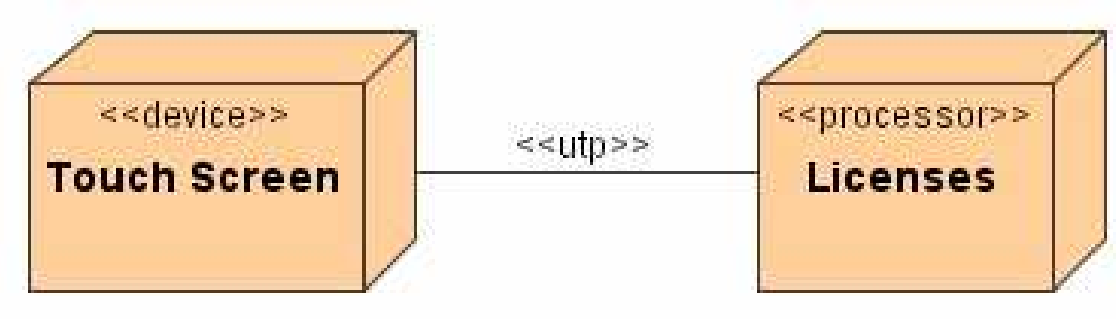
The most common kind of relationship used among nodes is an association representing a physical connection among then.



Processors and Devices

A processor is a node that has processing capability. It can execute a component.

A device is a node that has no processing capability (at least at the level of abstraction showed).



Modelling Nodes

Procedure:

1) identify the computational elements of the system's deployment view and model each as a node
2) add the corresponding stereotype to the nodes
3) consider attributes and operations that might apply to each node.

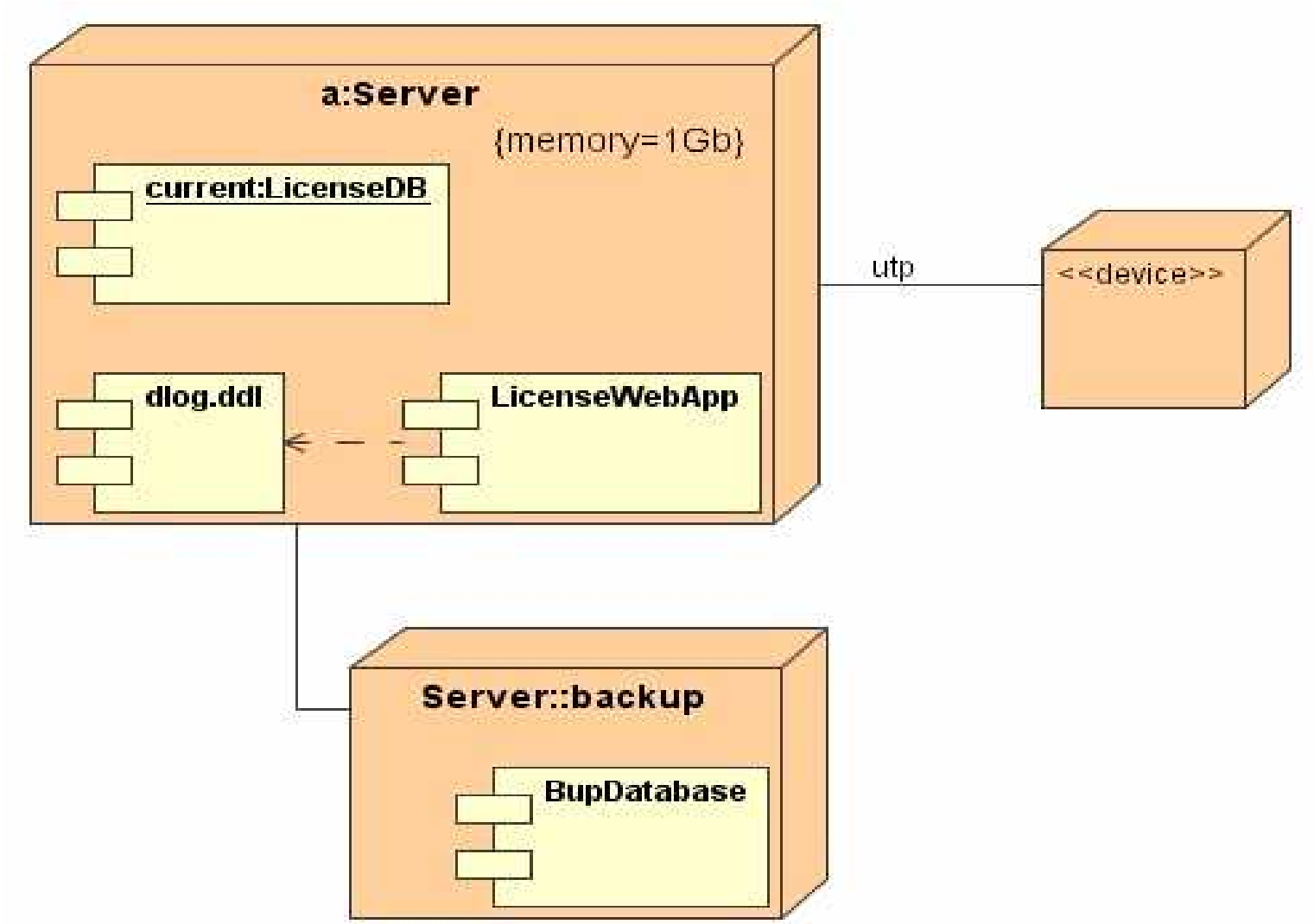
Distribution of Components

To model the topology of a system it is necessary to specify the physical distribution of its components across the processors and devices of the system.

Procedure:

1) allocate each component in a given node
2) consider duplicate locations for components, if it is necessary
3) render the allocation in one of these ways:
    a) don’t make visible the allocation
    b) use dependency relationship between the node and the component it’s deploy
    c) list the components deployed on a node in an additional compartment

Example: Deployment Diagram



Common Uses

Deployment diagrams may be used to model:

1) embedded systems: software-intensive collection of hardware that interfaces with the physical world. Involve software that controls devices, and that in turn is controlled by external stimuli

2) client/server systems: architectures making a clear distinction between system's user interface (client) and system's persistent data (server)

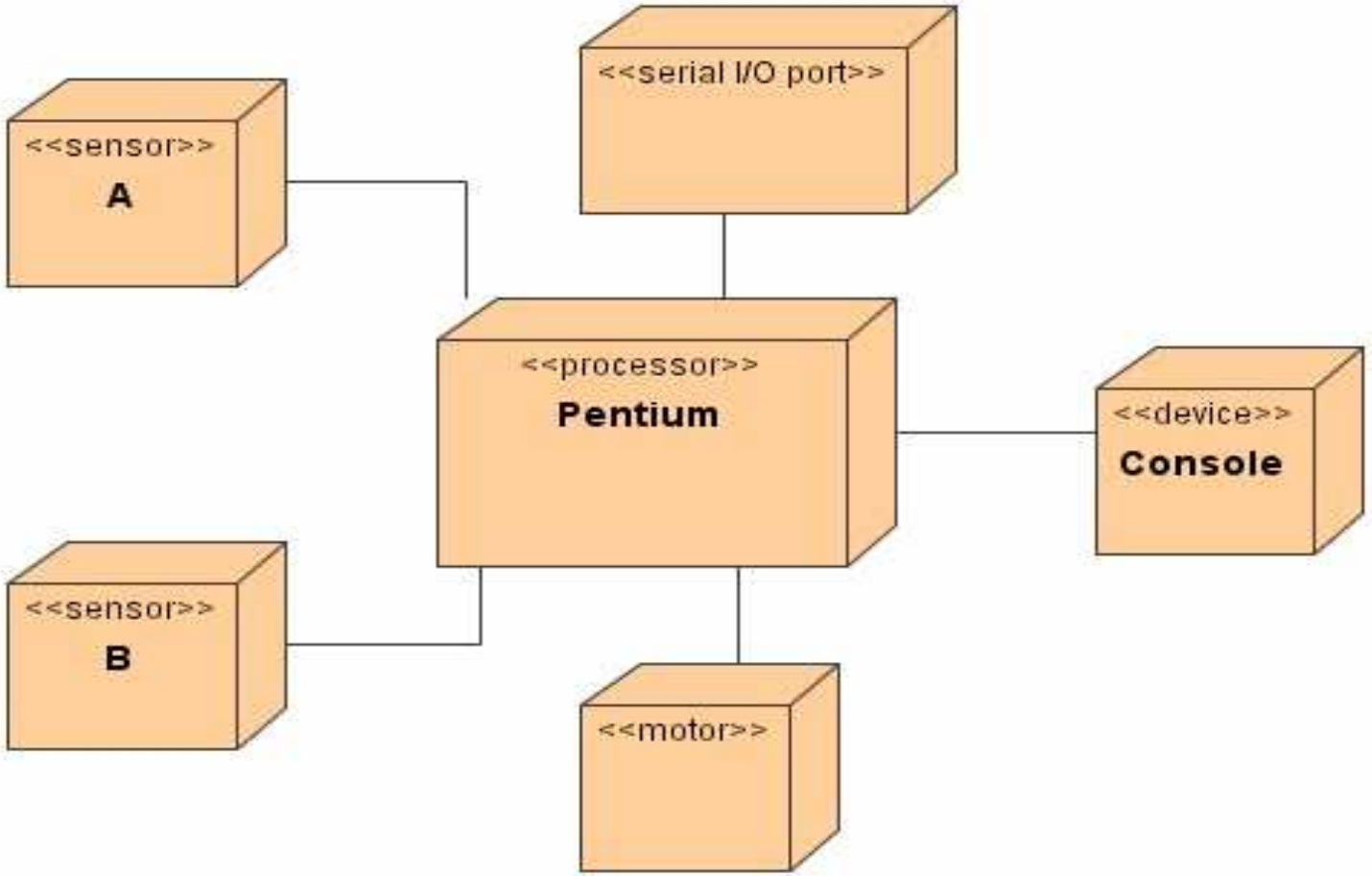
3) distributed systems: encompass multiple levels of servers.

Embedded System

Modelling procedure:

1) identify the devices and nodes that are unique to the system
2) provide visual indication for unusual devices with system-specific stereotypes and appropriate icons, distinguishing processors and devices
3) model the relationships among these processors and devices using a deployment diagram and specify the relationship between the components in the implementation view and the nodes in the deployment view
4) if necessary, expand on any intelligent devices by modelling their structure with a more detailed deployment diagram.

Example: Embedded System

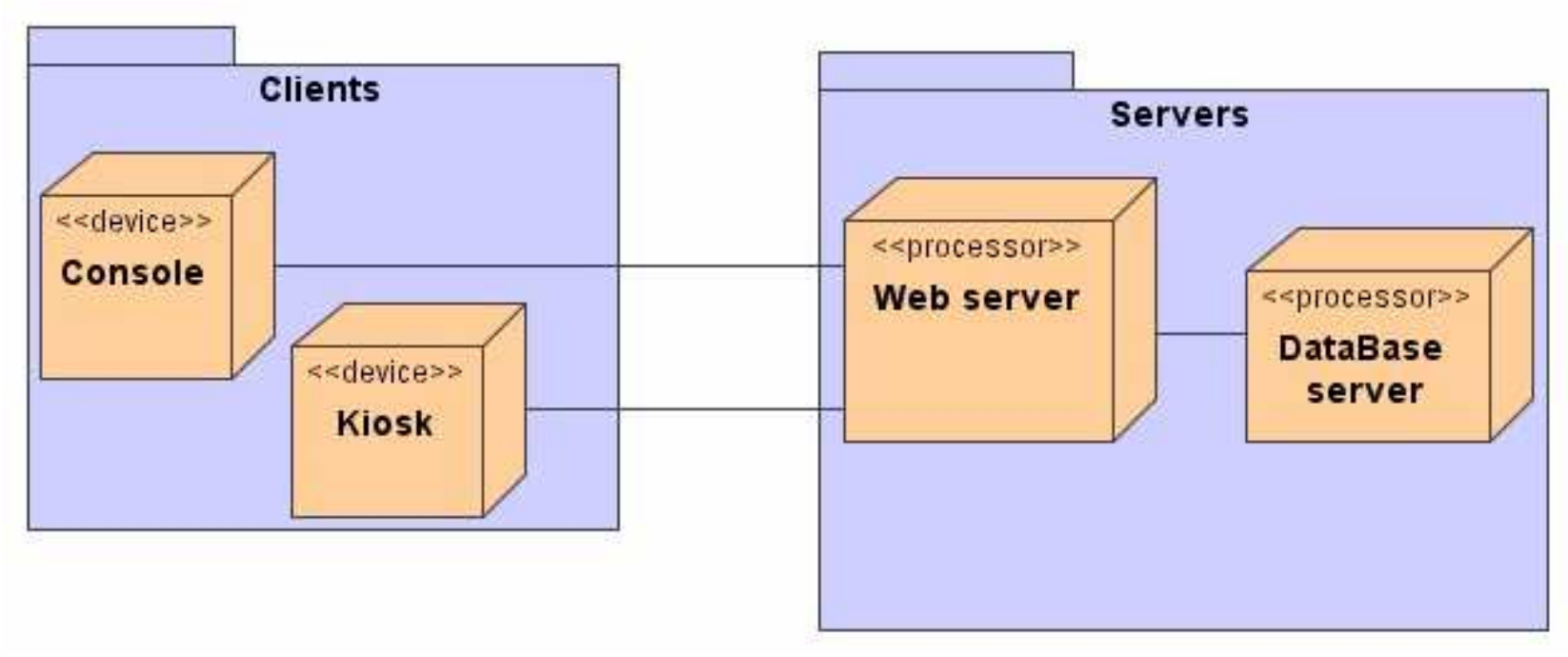


Client/Server Systems

Modelling procedure:

1) identify the nodes that represent the system's client and servers processors
2) highlight those devices that are relevant to the system
3) provide visual indication for these devices with stereotypes
4) model the topology of these nodes in a deployment diagram, and specify the relationship between the components in the implementation view and the nodes in the deployment view.

Example: Client/Server System

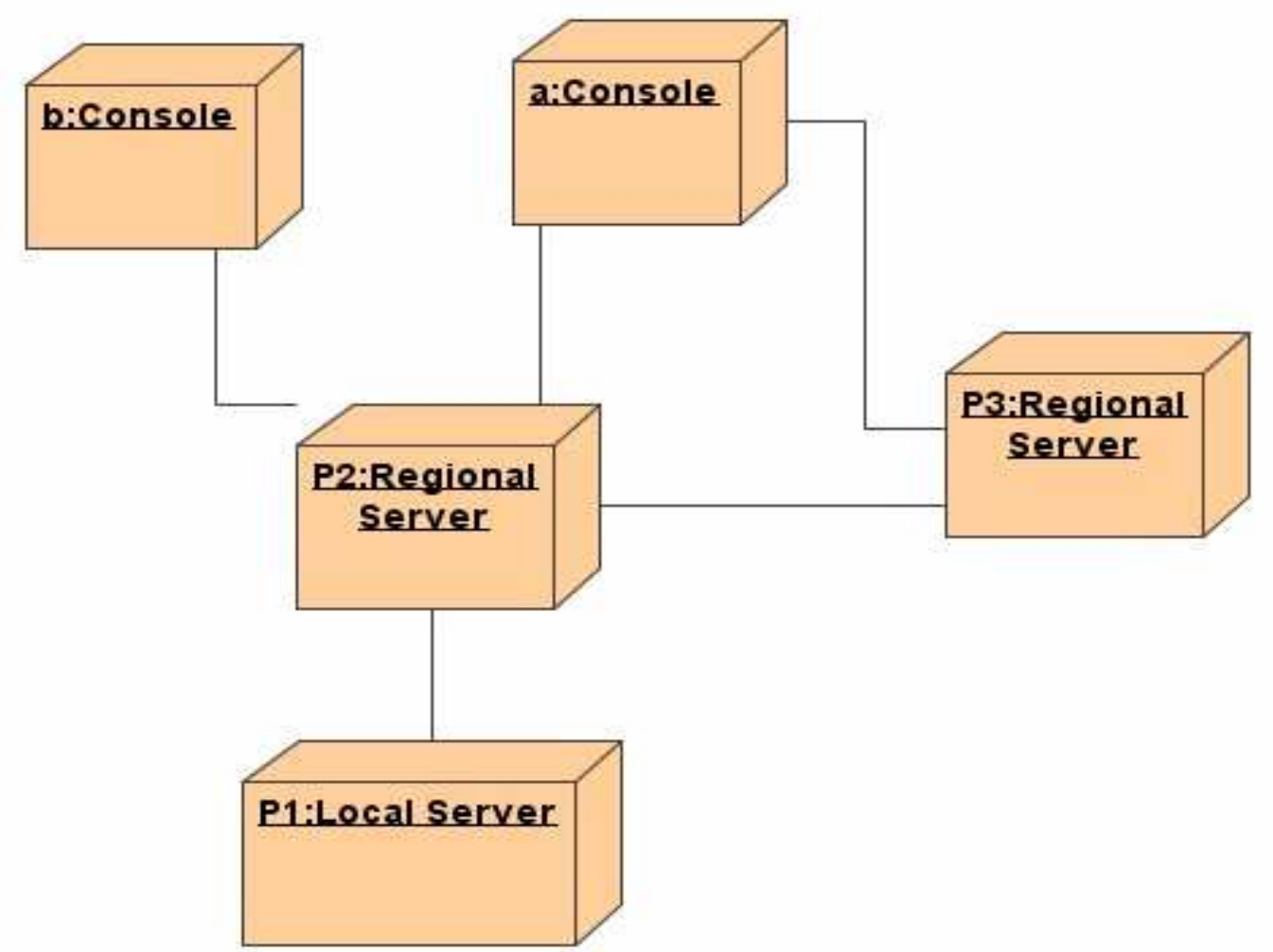


Distributed Systems

Modelling procedure:

1) identify the system's devices and processors
2) model the communication devices including sufficient detail if you need to assess the performance of the system's network or the impact of changes to the network
3) pay close attention to logical groupings of nodes that can be specified by using packages
4) model these devices and processors using deployment diagrams
5) if it is needed to focus on the dynamics of the system, introduce use case diagrams to specify the kinds of behaviour of interest, and expand these diagrams with interaction diagrams.

Example: Distributed System



Summary

Deployment diagrams model the topology of the hardware on which the system executes and the software installed on each of the hardware components.

Deployment diagrams include nodes, packages, components and their relationships.

Deployment diagrams may be used to model different kind of systems such as embedded, client-server or distributed systems.

Exercise 4

1. Consider your architectural design. Describe the components that are likely implement the layers of your architecture.

2. Specify the nature of the work product components or artifacts that are likely to implement these components.

3. Describe the relationship between the components in question 1 and the work products identified in question 2 using an implementation diagram.

4. Describe a simple database schema for your system.

Exercise 5

5) Describe an ideal deployment environment for your system.

6) Specify how the various components of your systems identified in question 3 will be distributed in the environment described in question 5.

Unified Process

The Course: Overview

1) The Course

2) Object Oriented Concepts

3) UML Basics

4) Case Study

5) Requirement Model

6) Architecture Model

7) Design Model

8) Implementation Model

9) Deployment Model

10) Unified Process

11) Tools

12) Summary

Overview

1) Software Development Process

2) Unified Process Overview

3) Unified Process Structure
    1) Building Blocks
    2) Phases
    3) Workflows

Why a process ?

A process defines:

1) who is doing what
2) when to do it
3) how to reach a certain goal
4) the inputs and outputs for each activity

Development Process

Is a framework which guides the tasks, people and define output products of the development process.

It is a framework because:

1) provides the inputs and outputs of each activity
2) does not restrict how each activity must be performed
3) must be tailored for every project

There is no universal process.

Unified Process - Overview

Key elements:

1) iterative and incremental

2) use case-driven

3) architecture-centric

Iterative and Incremental 1

The design process is based on iterations that address different aspects of the design process.

The iterations evolve into the final system (incremental aspect).

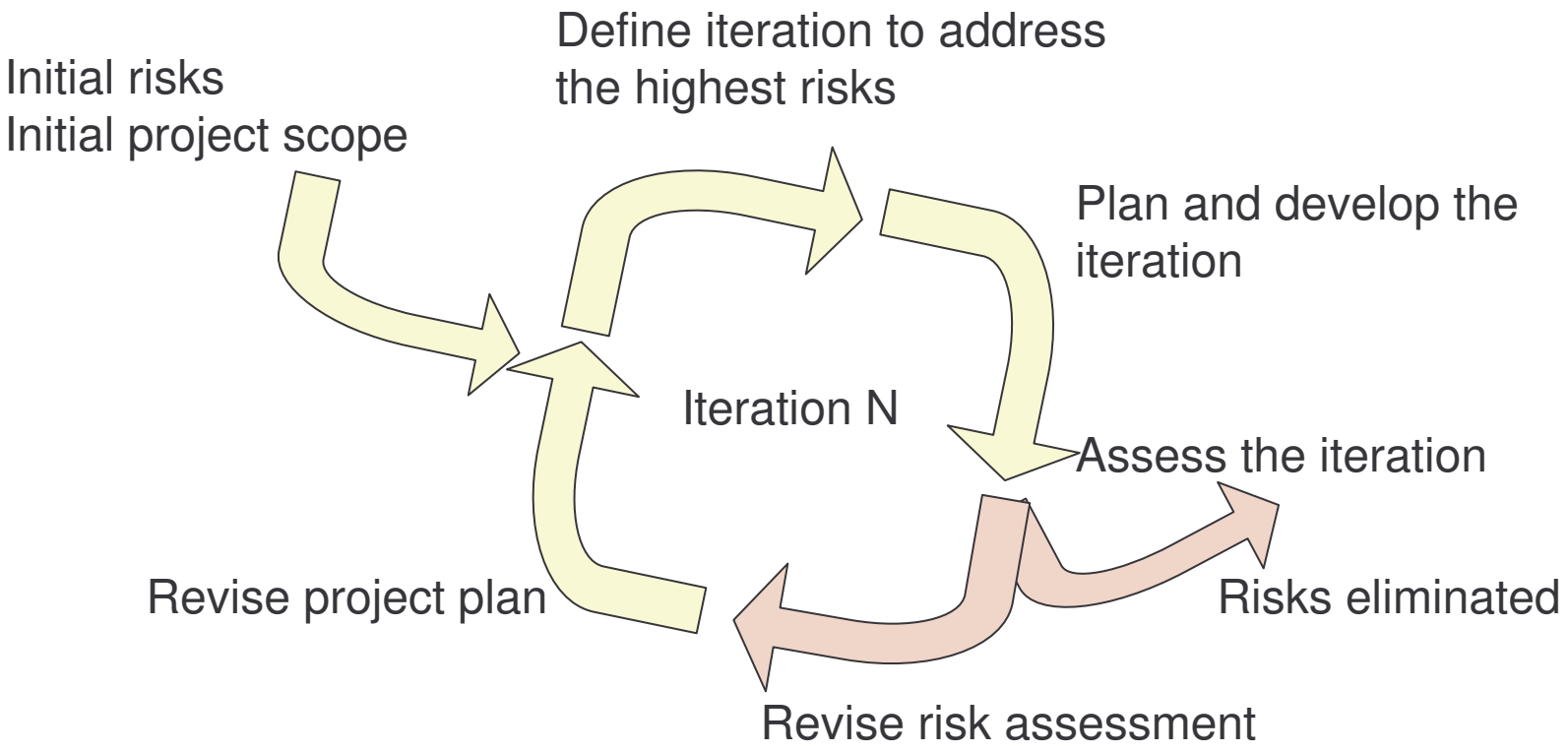
The process does not try to complete the whole design task in one go.

How to do it?

1) plan a little
2) specify, design and implement a little
3) integrate, test and run
4) obtain feedback before next iteration

Iterative and Incremental 2

Technical risks are assessed and prioritized early and are revised during each iteration.



Use Case-Driven

Use cases are used for:

1) identify users and their requirements
2) aid in the creation and validation of the architecture
3) help produce definitions of test cases and procedures
4) direct the planning of iterations
5) drive the creation of user documentation
6) direct the deployment of the system
7) synchronize the content of different models
8) drive traceability throughout models

Architecture-Centric

Problem:

- with the iterative and incremental approach different development activities are done concurrently

Solution:

- the system's architecture ensures that all parts fit together

"An architecture is the skeleton on which the muscles (functionality) and skin (user-interface) of the system will be hung".

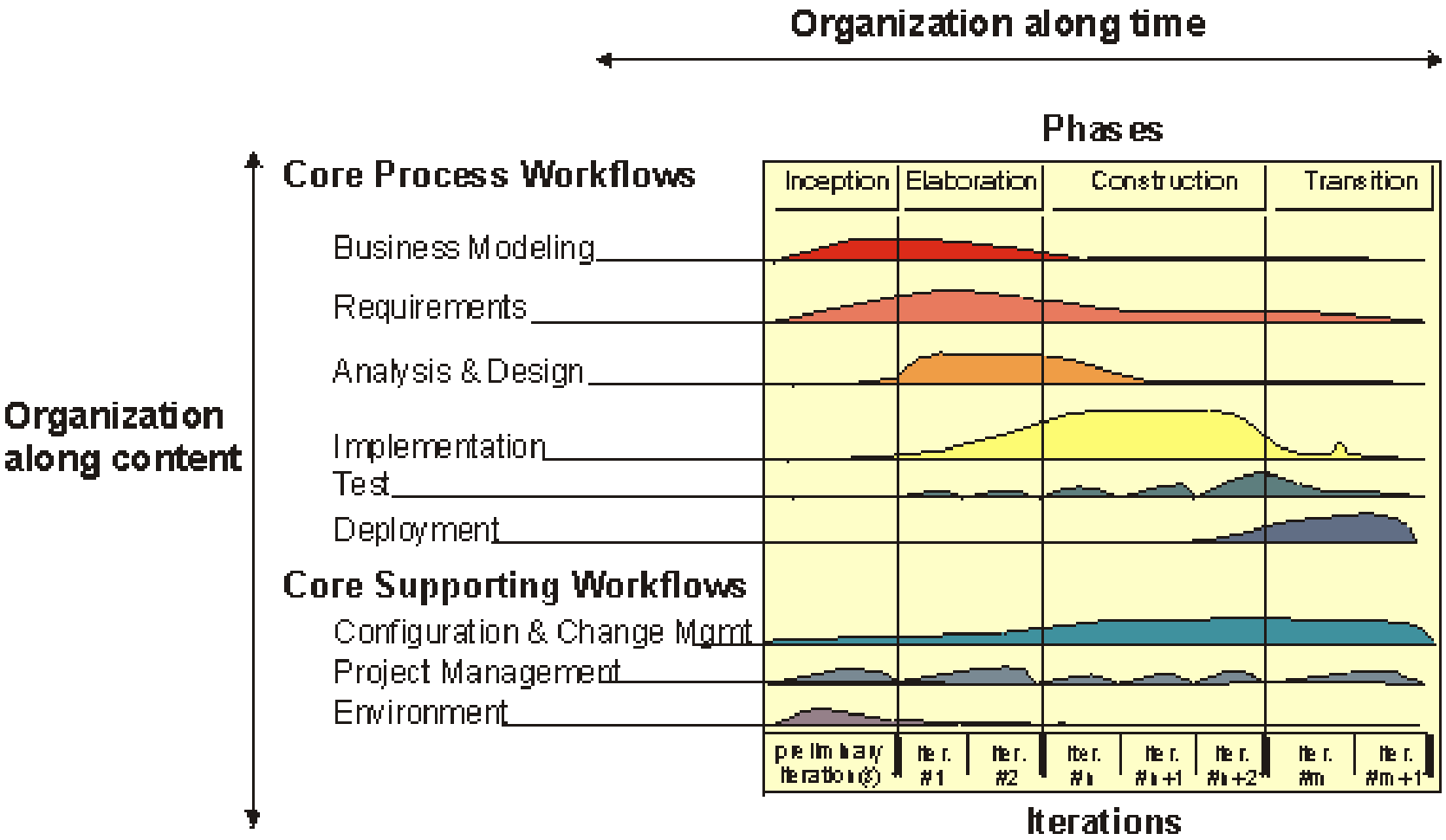
Unified Process - Structure

Two dimensions:

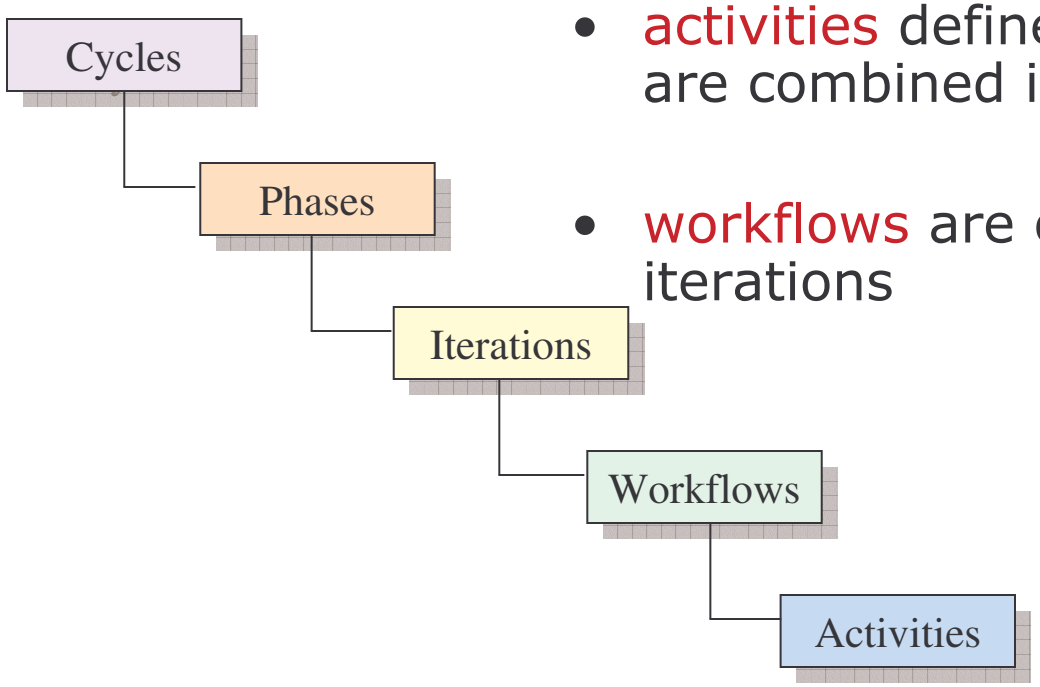
1) time → division on the life cycle into phases and iterations

2) process components → production of a specific set of artifacts with well-defined activities called workflows

The Development Process



Building Blocks

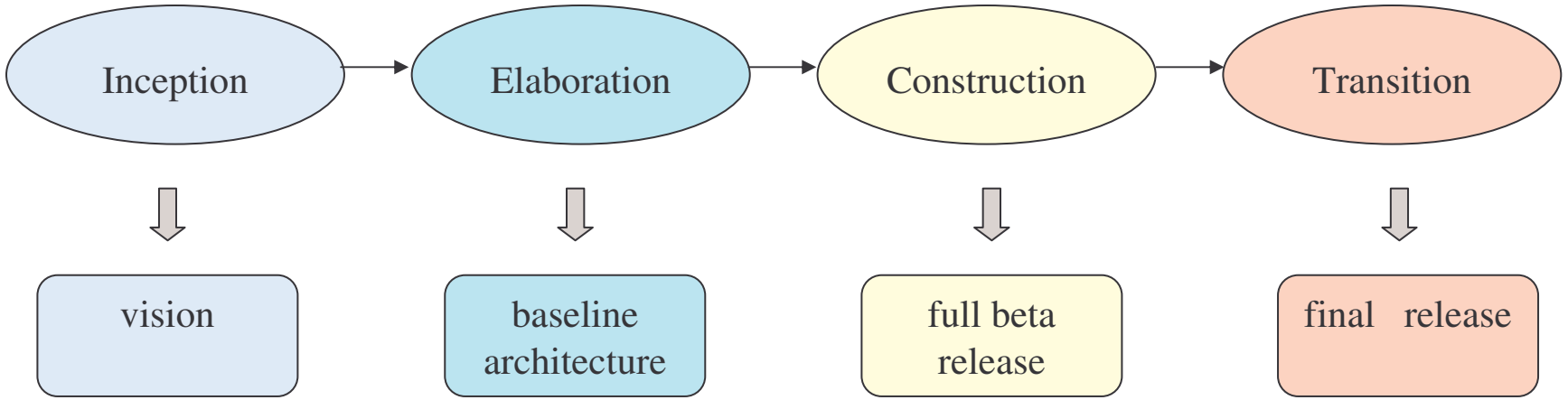


- activities define detailed work and are combined in workflows
- workflows are organized into iterations

- each iteration identifies some aspect of the system and are organized into phases
- phases can be grouped into cycles
- cycles focus on the generation of successive releases

Life Cycle Phases

Phases and major deliverables of the Unified Process



Inception

Focuses on the generation of the business case that involves:

1) identification of core uses cases
2) definition of the actual scope
3) identification of risky difficult parts of the system

The main objectives are:

1) to prove the feasibility of the system to be built
2) to determine the complexity involved in order to provide reasonable estimates

Outputs:

1) the vision of the system
2) very simplified use case model
3) tentative architecture
4) risks identified
5) plan for the elaboration phase

Elaboration

Involves:

1) understanding how requirements are translated into the internals of the system
2) producing the baseline architecture
3) capturing the majority of the use cases
4) exploring further the risks identified earlier and identifying the most significant
5) specifying any non-functional requirements specially those related to reliability and performance

Outputs:

1) the system's architecture
2) detailed use case model
3) set of plans for the construction phase

Construction

Involves:

1) completing the analysis of the system
2) performing the majority of the design and the implementation

Outputs:

1) implemented software product as a full beta release. It may contain some defects
2) associated models

An important aspect for the success of this phase is to monitor the critical aspects of the projects, specially significant risks.

Transition

Involves:

1) deployment of the beta system
2) monitoring user feedback and handling any modifications or updates required

Output:

1) the formal release of the software

UP - Workflows 1

1) requirements: focuses on the activities which allow to identify functional and non-functional requirements

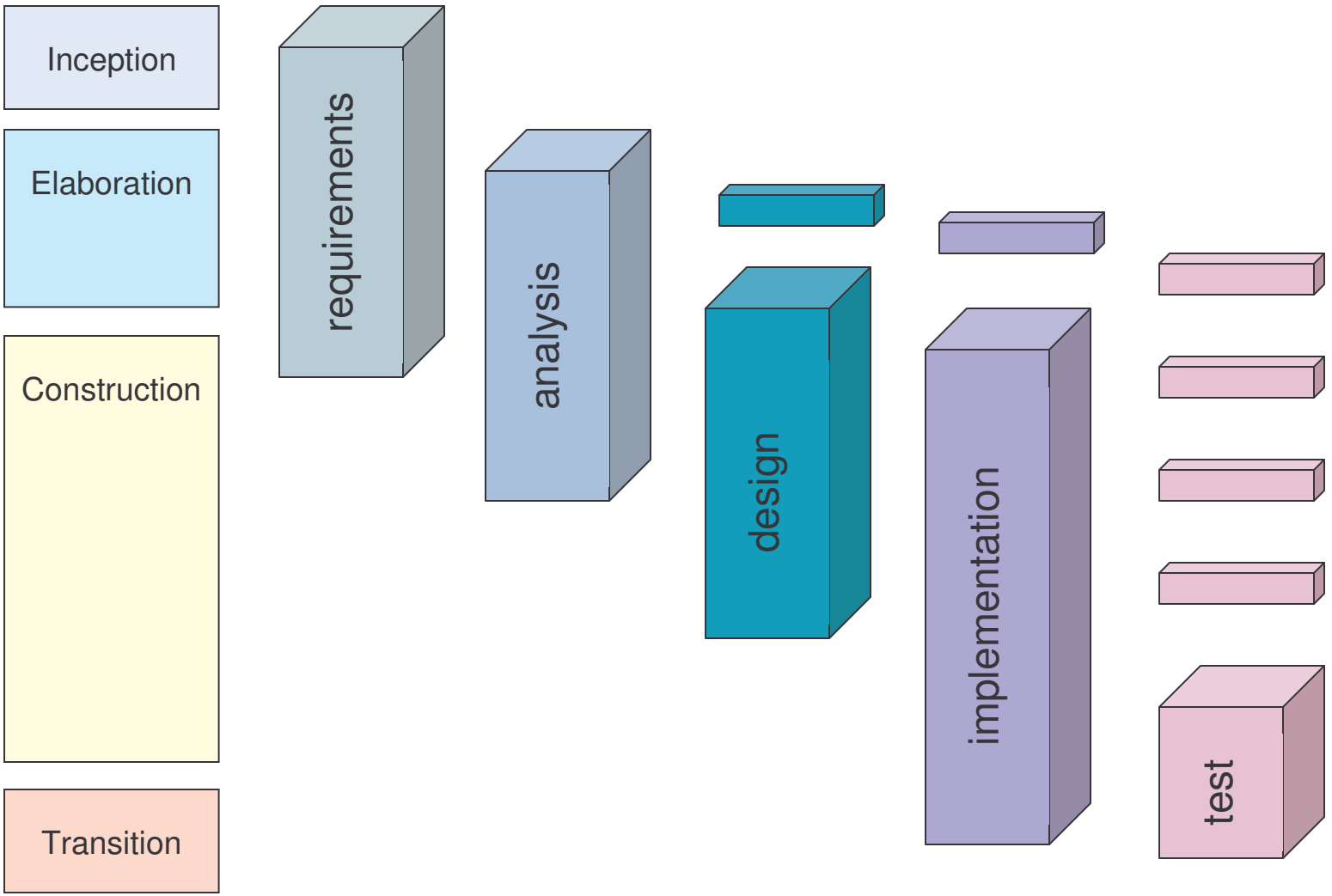
2) analysis: restructures the requirements identified in terms of the software to be built

3) design: produces a detailed design

4) implementation: represents the coding of the design in a programming language, and the compilation, packaging, deployment and documentation of the software

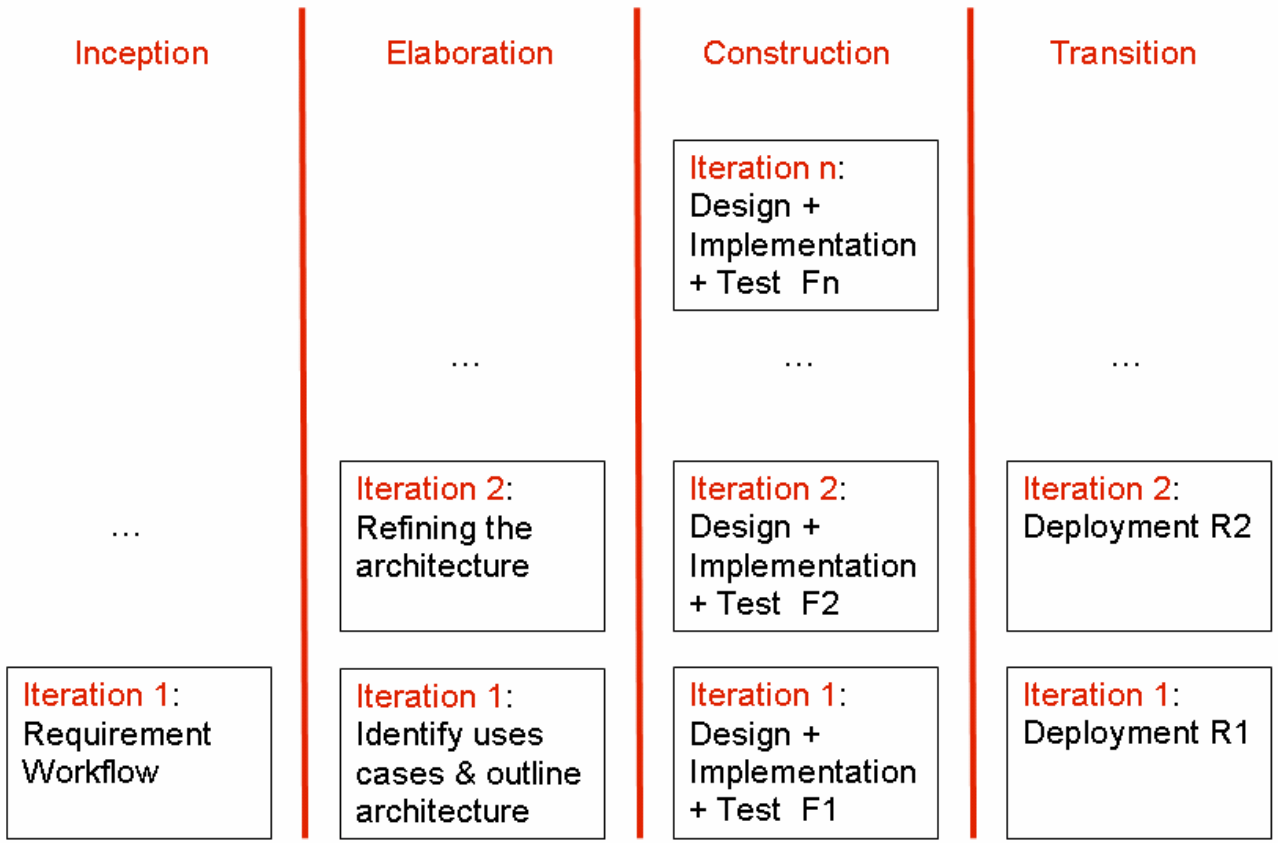
5) test: describes the activities to be carried out for testing

Workflows and Phases



Phases and Iterations

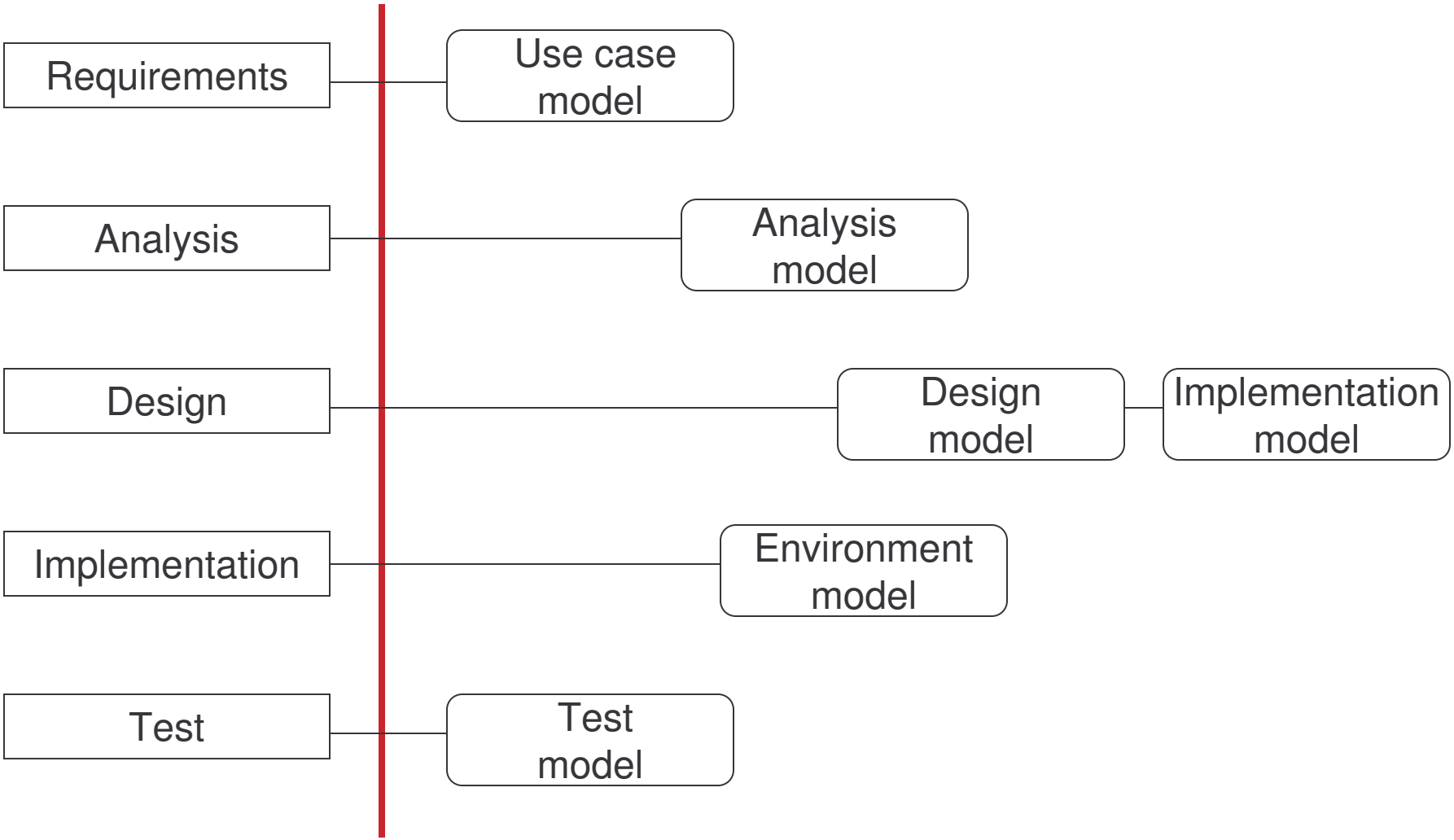
The iterations of workflows occur one or more times during a phase.



Workflows and Activities

Workflows	Activities
Requirements	Find actors and use cases, prioritize use cases, detail use cases, prototype user interface, structure the use case model
Analysis	Architectural analysis, analyze use cases, explore classes, find packages
Design	Architectural design, trace use cases, refine and design classes, design packages
Implementation	Architectural implementation, implement classes and interfaces, implement subsystems, perform unit testing, integrate systems
Test	Plan and design tests, implement tests, perform integration and system tests, evaluate tests

Workflows and Models



Summary 1

A software development process is a framework that provides guidance to carry out the different activities needed to produce a software product.

Every software development process must be parameterized for each individual project.

There is no universal process.

Summary 2

The main features of the Unified Process are:

1) use case-driven
2) architecture-centric
3) iterative and incremental

With respect to time dimension, the lifecycle is divided into phases and iteration.

There are four main phases: inception, elaboration, construction and transition.

A specific set of artifacts are produced with well-defined activities called workflows.

There are five main workflows: requirements, analysis, design, implementation and test.

Tools

The Course: Overview

1) The Course

2) Object Oriented Concepts

3) UML Basics

4) Case Study

5) Requirement Model

6) Architecture Model

7) Design Model

8) Implementation Model

9) Deployment Model

10) UML and UP

11) Tools

12) Summary

Overview

1) Why UML CASE tools?

2) Benefits

3) Different tools

4) Evaluating UML CASE tools

5) Main Features of:
    a) Enterprise Architect
    b) MagicDraw
    c) Poseidon
    d) Rational Rose

Why UML CASE tools?

1) enable to apply an object oriented methodology

2) allow to make abstraction of source code

3) allow to model the architecture and the design in such a way that are easier to understand and modify

4) enable to use models as blueprint for the system

5) enable to manage the project with a time dimension and with high level of abstraction

Benefits of UML CASE Tools

1) the use of these tools offer benefits to everyone involved in a project:
    a) analysts can capture requirements with use case model
    b) designers can produce models that capture interactions between objects
    c) developers can quickly turn the model into a working application

2) UML case tool, plus a methodology, plus empowered resources enable the development of the right software solution, faster and cheaper

Different UML CASE Tools

Tools vary with respect to:

1) UML modelling capabilities
2) project life-cycle support
3) forward and reverse engineering
4) data modelling
5) performance
6) price
7) supportability
8) easy of use
9) …

Evaluating CASE Tools

Different Criteria for evaluating CASE tools:

1) repository support
2) round-trip engineering
3) HTML documentation
4) UML support
5) data modelling integration
6) versioning
7) model navigation
8) printing support
9) diagrams views
10) exporting diagrams
11) platform

Different UML Tools

- Enterprise Architect
  - organization: Sparx Systems
  - web-site: http://www.sparxsystems.com.au/
- MagicDraw
  - organization: No Magic Inc.
  - web-site: http://www.nomagic.com/
- Poseidon
  - organization: Gentleware
  - web-site: http://www.gentleware.com/
- Rational Rose
  - organization: IBM
  - web-site: http://www-306.ibm.com/software/rational/

Enterprise Architect

Criteria	Features
Platform	Windows
UML Compliance	Support for all 13 UML 2.0 diagrams
Usability	Replication capable – Comprehensive and flexible documentation
Development Environment	Multi-user enabled – Allows to replicate and share projects
Outputs & Code Generation	C++, Java, C#, VB, VB.Net, Delphi, PHP – HTML and RTF document generation - Forward and reverse database engineering
Data Repository	Models are stored in a data repository - Checks data integrity in the data repository – Provides a project browser
Other features	Allows scripting to extend functionality – Project estimation tools – User definable patterns

Magic Draw

Criteria	Features
Platform	Any where Java 1.4 is supported
UML Compliance	Support for UML 1.4 notation and semantics
Usability	Replication capable – Customizable views of UML elements – Customizable elements properties
Development Environment	Multi-user enabled – Lock parts of the model to edit – Commit changes – Model versioning and rollback
Outputs & Code Generation	Code generation and reverse engineering to C++, Java, C# - RTF and PDF document generation
Data Repository	Provides a project browser
Other features	Friendly and customizable GUI – Hyperlinks can be added to any model element

Poseidon

Criteria	Features
Platform	Platform independent – Implemented in Java
UML Compliance	Supports all 9 diagrams of UML 1.4
Usability	Replication capable – Internationalization an localization for several languages
Development Environment	Collaborative environment based on client-server architecture – Locking of model parts – Secure transmission of files
Outputs & Code Generation	VB.Net, C#, C++, CORBA IDL, Delphi, Perl, PHP, SQL DDL – Round trip engineering for Java – Diagram export as gif, ps, eps and svg.
Data Repository	Uses MDR (Meta Data Repository) developed by Sun and based on the JMI (Java Metadata Interface) standard
Other features	Allows to import Rational Rose files

Rational Rose

Criteria	Features
Platform	Windows
UML Compliance	Not fully supported UML 1.4
Usability	The add-in feature allows to customize the environment – User configurable support for UML, OMT and Booch 93.
Development Environment	Parallel multi-user development through repository and private support
Outputs & Code Generation	C++, Visual C++, VB6, Java – Documentation generation – Round trip engineering
Data Repository	Maintains consistency between the diagram and the specification, you may change any of them and automatically updates the information.
Other features	Can be integrated with other Rational products such as RequisitePro, Test Manager

Summary

There exist several CASE Tools supporting Object Oriented modelling with UML.

Different criteria should be considered when evaluating a software tool.

Four tools were presented: Enterprise Architect, Magic Draw, Poseidon and Rational Rose.

Acknowledgements

We would like to thank Dr. Tomasz Janowski and all the members of the eMacao team for their valuable comments and help in preparing this material.

e-Macao Course Assessment
Course: Java Workshop, Team E

Name (CAPITALS, English): _____

Please select ALL correct answers for the following questions by ticking the appropriate option(s).

1	What will happen if you try to compile and run the following code? `public class Q {` `  public static void main(String argv[]){` `  int anar[]=new int[]{1,2,3};` `  System.out.println(anar[1]);` `  }` `}`	
	a	1
	b	Error anar is referenced before it is initialized
	c	2
	d	Error: size of array must be defined
2	What will be the result when you try to compile and run the following code? `private class Base{` `  Base(){` `  int i = 100;` `  System.out.println(i);` `  }` `}` `public class Pri extends Base{` `  static int i = 200;` `  public static void main(String argv[]){` `  Pri p = new Pri();` `  System.out.println(i);` `  }` `}`	
	a	Error at compile time
	b	200
	c	100 followed by 200
	d	100
3	Which of the following statements is true?	
	a	`Interface variables must be declared with private modifier.`
	b	`Not all methods defined by an interface must be implemented by an implementation class.`
	c	`An interface can extend more than one interface.`
	d	`A class cannot implement more than one interface.`
4	What will happen when you compile and run the following code? `public class MyClass{` `  static int i;`	

```
 public static void main(String argv[]){
 System.out.println(i);
 }
}
```

	a	Error Variable i may not have been initialized
	b	null
	c	1
	d	0
5	What will be printed out if you attempt to compile and run the following code?	

```
int i=1;
 switch (i) {
 case 0:
 System.out.println("zero");
 break;
 case 1:
 System.out.println("one");
 case 2:
 System.out.println("two");
 default:
 System.out.println("default");
 }
```

	a	one
	b	one, default
	c	one, two, default
	d	default
6	What will be the result of attempting to compile and run the following code?	

```
abstract class MineBase {
 abstract void amethod();
 static int i;
}
public class Mine extends MineBase {
 public static void main(String argv[]){
 int[] ar=new int[5];
 for(i=0;i < ar.length;i++)
 System.out.println(ar[i]);
 }
}
```

	a	a sequence of 5 0's will be printed
	b	Error: ar is used before it is initialized
	c	Error Mine must be declared abstract
	d	IndexOutOfBoundes Error
7	Which of the following statements are true?	
	a	Methods cannot be overriden to be more private
	b	static methods cannot be overloaded

	c	private methods cannot be overloaded
	d	An overloaded method cannot throw exceptions not checked in the base class
8	If you wanted to find out where the position of the letter v (ie return 2) in the string s containing "Java", which of the following could you use?	
	a	mid(2,s);
	b	charAt(2);
	c	s.indexOf('v');
	d	indexOf(s,'v');
9	Which of the following lines of code will compile without error	
	a	`int i=0;` `if(i) {` `System.out.println("Hello");` `}`
	b	`boolean b=true;` `boolean b2=true;` `if(b==b2) {` `System.out.println("So true");` `}`
	c	`int i=1;` `int j=2;` `if(i==1|| j==2)` `System.out.println("OK");`
	d	`int i=1;` `int j=2;` `if(i==1 &| j==2)` `System.out.println("OK");`
10	A byte can be of what size	
	a	-128 to 127
	b	(-2 power 8)-1 to 2 power 8
	c	-255 to 256
	d	depends on the particular implementation of the Java Virtual machine
11	Which of the following are keywords or reserved words in Java?	
	a	`if`
	b	then
	c	`goto`
	d	`case-switch`
12	What will happen if you try to compile and run the following code `public class MyClass {` `public static void main(String arguments[]) {` `amethod(arguments);` `}` `public void amethod(String[] arguments) {`	

```
        System.out.println(arguments);
        System.out.println(arguments[1]);
    }
}
```

	a	error Can't make static reference to void amethod.
	b	error method main not correct
	c	error array must include parameter
	d	amethod must be declared with String

13 What will happen when you attempt to compile and run the following code?.

```
class Background implements Runnable{
        int i=0;
        public int run(){
            while(true){
                i++;
                System.out.println("i="+i);
            } //End while
          return 1;
         }//End run

}//End class
```

	a	It will compile and the run method will print out the increasing value of i.
	b	It will compile and calling start will print out the increasing value of i.
	c	The code will cause an error at compile time.
	d	Compilation will cause an error because while cannot take a parameter of true.

14 Given the following code

```
import java.io.*;
public class Th{
    public static void main(String argv[]){
        Th t = new Th();
        t.amethod();
    }
    public void amethod(){
        try{
            ioCall();
        }catch(IOException ioe){}
    }
}
```

What code would be most likely for the body of the ioCall method?

	a	public void ioCall ()throws IOException{ DataInputStream din = new DataInputStream(System.in); din.readChar(); }
	b	public void ioCall () throw IOException{ DataInputStream din = new DataInputStream(System.in); din.readChar(); }

	c	public void ioCall (){ DataInputStream din = new DataInputStream(System.in); din.readChar(); }
	d	public void ioCall throws IOException(){ DataInputStream din = new DataInputStream(System.in); din.readChar(); }
15	You need to create a class that will store unique object elements. You do not need to sort these elements but they must be unique. What interface might be most suitable to meet this need?	
	a	Set
	b	List
	C	Map
	d	Vector
	Answer:	

Answers

Answer 1)

Back to Question 9)
Objective 4.4)

3) 2

No error will be triggered.

Like in C/C++, arrays are always referenced from 0. Java allows an array to be populated at creation time. The size of array is taken from the number of initializers. If you put a size within any of the square brackets you will get an error.

http://www.jchq.net/tutorial/04_04Tut.htm

Answer 2)

Back to question 6)
Objective 4.3)

1) if
3) goto
4) while
5) case

then is not a Java keyword, though if you are from a VB background you might think it was. Goto is a reserved word in Java.

http://www.jchq.net/tutorial/04_03Tut.htm

Answer 3)

Back to question 10)
Objective 4.4)

3) 0

Arrays are always initialised when they are created. As this is an array of ints it will be initalised with zeros.

http://www.jchq.net/tutorial/04_04Tut.htm

Answer 4)

Back to Question 8)
Objective 4.4)

4) 0

Class level variables are always initialised to default values. In the case of an int this will be 0. Method level variables are not given default values and if you attempt to use one before it has been initialised it will cause the

```
Error Variable i may not have been initialized
```

type of error.

http://www.jchq.net/tutorial/04_04Tut.htm

Answer 5)

Back to Question 12)
Objective 2.1)

3) one, two, default

Code will continue to fall through a case statement until it encounters a break.

http://www.jchq.net/tutorial/02_01Tut.htm

Answer 6)

Back to Question 11)
Objective 1.2

3) Error Mine must be declared abstract

A class that contains an abstract method must itself be declared as abstract. It may however contain non abstract methods. Any class derived from an abstract class must either define all of the abstract methods or be declared abstract itself.

http://www.jchq.net/tutorial/01_021Tut.htm

Answer 7)

Back to Question 16)
Objective 6.2)

1) Methods cannot be overriden to be more private

Static methods cannot be overriden but they can be overloaded. If you have doubts about that statement, please follow and read carefully the link given to the Sun tutorial below. There is no logic or reason why private methods should not be overloaded. Option 4 is a jumbled up version of the limitations of exceptions for overriden methods

http://www.jchq.net/tutorial/06_02Tut.htm
http://java.sun.com/docs/books/tutorial/java/javaOO/override.html

Answer 8)

Back to question 5)
Objective 4.2)

4) Exception raised: "java.lang.ArrayIndexOutOfBoundsException: 2"

Unlike C/C++ java does not start the parameter count with the program name. It does however start from zero. So in this case zero starts with good, morning would be 1 and there is no parameter 2 so an exception is raised.

http://www.jchq.net/tutorial/04_02Tut.htm

Answer 9)

Back to Question 14)
Objective 5.1

2,3

Example 1 will not compile because if must always test a boolean. This can catch out C/C++ programmers who expect the test to be for either 0 or not 0.

http://www.jchq.net/tutorial/05_01Tut.htm

Answer 10)

Back to question 4)
Objective 4.5)

1) A byte is a signed 8 bit integer.

http://www.jchq.net/tutorial/04_05Tut.htm

Answer 11)

back to Question 3)
Objective 4.1)

2 and 3 will compile without error.

1 will not compile because any package declaration must come before any other code. Comments may appear anywhere

http://www.jchq.net/tutorial/04_01Tut.htm .

Answer 12)

Back to question 2)

Objective 4.1

1) Can't make static reference to void amethod.

Because main is defined as static you need to create an instance of the class in order to call any non-static methods. Thus a typical way to do this would be.

MyClass m=new MyClass();

m.amethod();

Answer 2 is an attempt to confuse because the convention is for a main method to be in the form

String argv[]

That argv is just a convention and any acceptable identifier for a string array can be used. Answers 3 and 4 are just nonsense.

http://www.jchq.net/tutorial/04_01Tut.htm

Answer 13)

Back to question 1)
Objective 4.5)

5) int i=10;

explanation:
1) float f=1.3;
Will not compile because the default type of a number with a floating point component is a double. This would compile with a cast as in

float f=(float) 1.3

2) char c="a";

Will not compile because a char (16 bit unsigned integer) must be defined with single quotes. This would compile if it were in the form

char c='a';

3) byte b=257;

Will not compile because a byte is eight bits. Take of one bit for the sign component you can define numbers between

-128 to +127

4) a boolean value can either be true or false, null is not allowed

http://www.jchq.net/tutorial/04_05Tut.htm.

Answer 14)

Back to Question 13)
Objective 4.1)

2) default, zero

Although it is normally placed last the default statement does not have to be the last item as you fall through the case block. Because there is no case label found matching the expression the default label is executed and the code continues to fall through until it encounters a break.

http://www.jchq.net/tutorial/04_01Tut.htm

Answer 15)

Back to Question 7)
Objective 4.1)

2) variable2
3) _whatavariable
4) _3_
5) $anothervar

An identifier can begin with a letter (most common) or a dollar sign($) or an underscore(_). An identifier cannot start with anything else such as a number, a hash, # or a dash -. An identifier cannot have a dash in its body, but it may have an underscore _. Choice 4) _3_ looks strange but it is an acceptable, if unwise form for an identifier.

http://www.jchq.net/tutorial/04_01Tut.htm

Java Workshop

Training Course

Gabriel Oteniya
Tomasz Janowski

e-Macao Report 20

Version 1.0, October 2005

Table of Contents

1. Overview

Java is an Object-Oriented programming language designed and developed by Sun Microsystems. It is a modern language that has become the language of the choice for many programmers, particularly for writing programs that have to run on a variety of different computer systems. The language was designed from the outset to be simple, robust, secure, and machine-independent.

Java has matured immensely in recent years, particularly with the introduction of Java 2. The range of functions provided by the core Application Programming Interfaces (APIs) has grown substantially. For instance, there are APIs to support the development of applications that rely on a Graphical User Interface (GUI), extensive image processing and programmable graphics. Java APIs exist to access relational databases and to communicate with remote computers over a network. Java can also be used to write applets - programs that are embedded inside Internet web pages to provide interactivity. Java also supports the development of multithreaded and distributed applications.

This document presents a refresher course on Java Technology. The main part of the course is the introduction to the language, first the procedural and then the object-oriented part. A number of selected APIs are introduced next - those that are used across the language and those that are designed to perform specific functions.

The duration of the course is 40 hours and the target audience is software practitioners with some basic programming experience.

The rest of this document explains the objectives, prerequisites and methodology for teaching the course in Sections 2, 3 and 4 respectively. The content of the course is introduced in Section 5, followed by assessment and organization in Sections 6 and 7 respectively. Following references, Appendix A includes the complete set of slides and Appendix B contains two sets of assessment questions with answers.

2. Objectives

The course has five main objectives:

1) To refresh and reinforce the knowledge of Java Technology.
2) To review selected Java APIs (libraries).
3) To learn best practices in developing Java applications.
4) To provide solid foundation for learning J2EE Technology.
5) To explain why Java is adequate for implementing government services.

3. Prerequisites

The course assumes that:

1) The students have some knowledge of Object-Oriented Analysis and Design.
2) The students have the ability to develop desktop applications using any Object-Oriented or Object-Based programming language.

4. Methodology

The course has been designed relying on some general didactic principles:

1) Concepts are first defined and illustrated with some concrete examples, before they are used or applied in any context.
2) Visual illustrations are used as much as possible.
3) Assessment is carried out at the end of every section.
4) Most exercises require hands-on participation.
5) The overall assessment is administered at the end of the course to test the general understanding of the material presented.

Generally, an instructor is expected to administer the course in a tutorial style.

5. Content

The course is divided into five sections: Introduction, Language, Object-Orientation, Horizontal Libraries and Vertical Libraries. They are described as follows.

5.1. Introduction

This section, comprising the slides 14 through 54 provides an overview of Java. It starts by presenting the origin and history of Java then explains the components of the Java platform - a Virtual Machine, a set of Application Programming Interfaces and the language itself. The highlights, features and goals of the language are explained as well, showing how to set up the runtime environment and how to use Java documentation. The section concludes by demonstrating how to write, compile and run a simple Java application.

5.2. Language

This section, comprising the slides 55 to 204 introduces the procedural part of the Java language. It explains the different syntactic elements, data types, arrays and operators - assignment, relational, logical and bitwise and others. A number of procedural control structures to express sequencing, selection and iteration are presented as well.

5.3. Object Orientation

This section, covered by the slides 205 through 644, presents the object-oriented aspects of the language. It elaborates on how to encapsulate data and behavior within classes, how to build hierarchies of classes with inheritance, how to determine the behavior of objects at run-time, how to group classes into packages and how to introduce abstraction through interfaces. The section concludes by introducing exception handling and multithreading.

5.4. Horizontal Libraries

This section consists of the slides 645 to 783. It presents three major horizontal APIs provided by Java: String-Handling, Event-Handling and Object Collections. The String Handling section explains how Java manipulates strings. The Event Handling section explains how programs can handle events. The Object Collection section explains how to group objects and how such groups can be manipulated.

5.5. Vertical Libraries

This section, comprising the slides 784 to 907 presents a number of APIs that are designed to perform specific functions. Specifically, it introduces the Swing APIs for building Graphical User Interfaces and applets. An applet allows a Java application to be executed within a web browser, as part of viewing a web page.

6. Assessment

Assessment questions are provided at the end of the course to test the student's understanding of the various topics taught. The assessment is more like a quiz which tests the general understanding of the course. The assessment does not replace the exercises provided across the various sections, nor the tasks students are asked to perform. The tasks, in particular, are more rigorous and aim to test the depth of the understanding in specific areas. The assessment consists of multiple choice questions.

7. Organization

The course is designed to be taught during 42 hours. In the context of the e-Macao Core Team Training Programme, the course was taught over seven days. For the e-Macao Extended Team Training, the same course was taught over four days.

References

1. Java 2 The Complete Reference, Herbert Schildt, Osborne, 5th edition, 2002.
2. The Java Tutorial, Sun Microsystems, http://java.sun.com/docs/books/tutorial/, 2004.
3. Bruce Ecker, Thinking in Java, 3rd edition, Prentice Hall, 2002.
4. Sun Microsystems, Inc, Java™ Look and Feel Design Guidelines: Advanced Topics, Addison Wesley, December 27, 2001.
5. Robert Simmons Jr, Hardcore Java, O'Reilly, March 2004.
6. Ian F. Darwin, Java Cookbook, 2nd Edition, O'Reilly, June 2004.
7. Scott Oaks, Henry Wong, Java Threads, Third Edition, O'Reilly, September 2004.
8. Ivor Horton, Beginning Java 2, SDK 1.4 Edition, Wrox Press, 2003.

Appendix

A. Slides

A.1. Introduction

Java Workshop

Gabriel Oteniya and Tomasz Janowski

UNU-IIST

The Course

1) objectives - what do we intend to achieve?
2) outline - what content will be taught?
3) resources - what teaching resources will be available?
4) organization - duration, major activities, daily schedule

e-Macao-16-2-3

Course Objectives

1) refresh and reinforce the knowledge of Java technology
2) review selected Java APIs (libraries)
3) learn best practices in developing Java applications
4) lay foundation for learning J2EE technology
5) understand why Java technology is adequate for the implementation of e-government services

e-Macao-16-2-4

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) Applet
6) summary

e-Macao-16-2-5

Outline: Introduction

An overview of the Java language and its runtime environment.

Main points:

1) Familiarize participants with the language's features, goals, and documentation.
2) Explains how to set up the Java runtime environment to facilitate the development and execution of Java code.
3) Describes how to use the Java technology documentation.
4) Provides a simple example on how to write, compile and run a Java application.

e-Macao-16-2-6

Outline: Language

Introduces procedural aspects of the Java language.

Main points:

1) Basic data type and variables.
2) How to work with variables and arrays.
3) Operators: arithmetic, bitwise, relational, logical, etc.
4) Control structures: selection, iteration, jumps, etc.

e-Macao-16-2-7

Outline: Object-Orientation

Presents object-oriented aspects of the Java language.

Main points:

1) How to encapsulate data and behavior within classes.
2) How to build hierarchies of classes with inheritance.
3) How to determine the behavior of objects at run-time.
4) How to group classes into packages.
5) How to introduce abstraction thorough interfaces.

e-Macao-16-2-8

Outline: Horizontal APIs

API – Application Programming Interface

Horizontal APIs are used across the language.

Presentation of selected horizontal APIs:

1) string handling
2) event handling
3) Object collections

e-Macao-16-2-9

Outline: Vertical APIs

Vertical APIs are designed to perform specific functions.

Presentation of selected vertical APIs:

1) graphical user interface
2) applets

e-Macao-16-2-10

Outline: Summary

Revision of the material introduced during the course.

How this course provides a foundation for the remaining courses:

1) distributed programming
2) Java XML processing
3) Java Web Services
4) J2EE web components
5) J2EE business components

e-Macao-16-2-11

Course Resources

1) books
   a) Java 2 The Complete Reference, Herbert Schildt, Osborne, 5th edition, 2002
   b) The Java Tutorial, Sun Microsystems, http://java.sun.com/docs/books/tutorial/, 2004
   c) Thinking in Java, Bruce Ecker, 3rd edition, Prentice Hall, 2002

2) articles
   Links available from the website http://www.emacao.gov.mo.

3) tools
   a) JDK 1.5
   b) Eclipse IDE

e-Macao-16-2-12

Course Logistics

1) duration - 42 hours
2) activities - lecture (hands-on), development
3) sessions/day - morning 09:00–13:00 and afternoon 14:30–16:30
4) number of sessions - 6 morning and 6 afternoon
5) style - interactive and tutorial

e-Macao-16-2-13

Course Prerequisite

1) some experience in object-oriented programming:
   a) C++
   b) Delphi
   c) any other object-oriented language
2) basic understanding of TCP/IP networking concepts

Introduction

e-Macao-16-2-15

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) Applets
6) summary

e-Macao-16-2-16

Overview

Topics under this module:

1) Java origin and history
2) Java technology
3) Java language
4) Java platform
5) simple Java program
6) Java documentation
7) setting up Java environment

e-Macao-16-2-17

Java Origins

Computer language innovation and development occurs for two fundamental reasons:

1) to adapt to changing environments and uses
2) to implement improvements in the art of programming

The development of Java was driven by both in equal measures.

Many Java features are inherited from the earlier languages:

B → C → C++ → Java

e-Macao-16-2-18

Before Java: C

Designed by Dennis Ritchie in 1970s.

Before C, there was no language to reconcile: ease-of-use versus power, safety versus efficiency, rigidity versus extensibility.

BASIC, COBOL, FORTRAN, PASCAL optimized one set of traits, but not the other.

C- structured, efficient, high-level language that could replace assembly code when creating systems programs.

Designed, implemented and tested by programmers, not scientists.

e-Macao-16-2-19

Before Java: C++

Designed by Bjarne Stroustrup in 1979.

Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:

1) assembler languages
2) high-level languages
3) structured programming
4) object-oriented programming (OOP)

OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.

C++ extends C by adding object-oriented features.

e-Macao-16-2-20

Java History

Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.

The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.

With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.

Java as an “Internet version of C++”? No.

Java was not designed to replace C++, but to solve a different set of problems. There are significant practical/philosophical differences.

e-Macao-16-2-21

Java Technology

There is more to Java than the language.

Java Technology consists of:

1) Java Programming Language
2) Java Virtual Machine (JVM)
3) Java Application Programming Interfaces (APIs)

e-Macao-16-2-22

Java Language Features

1) simple
2) object-oriented
3) robust
4) multithreaded
5) architecture-neutral
6) interpreted and high-performance
7) distributed
8) dynamic
9) secure

e-Macao-16-2-23

Java Language Features 1

1) simple – Java is designed to be easy for the professional programmer to learn and use.
2) object-oriented – a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
3) robust – restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.

e-Macao-16-2-24

Java Language Features 2

4) multithreaded – supports multi-threaded programming for writing program that perform concurrent computations
5) architecture-neutral – Java Virtual Machine provides a platform-independent environment for the execution of Java bytecode
6) interpreted and high-performance – Java programs are compiled into an intermediate representation – bytecode:

   a) can be later interpreted by any JVM

   b) can be also translated into the native machine code for efficiency.

e-Macao-16-2-25

Java Language Features 3

7) distributed – Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
8) dynamic – substantial amounts of run-time type information to verify and resolve access to objects at run-time.
9) secure – programs are confined to the Java execution environment and cannot access other parts of the computer.

e-Macao-16-2-26

Execution Platform

What is an execution platform?

1) An execution platform is the hardware or software environment in which a program runs, e.g. Windows 2000, Linux, Solaris or MacOS.
2) Most platforms can be described as a combination of the operating system and hardware.

e-Macao-16-2-27

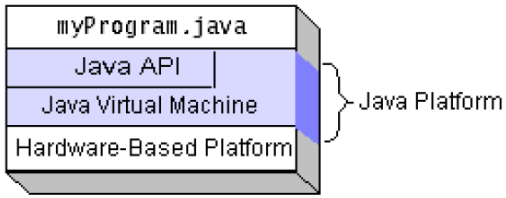
Java Execution Platform

What is Java Platform?

1) A software-only platform that runs on top of other hardware-based platforms.
2) Java Platform has two components:
   a) Java Virtual Machine (JVM) – interpretation for the Java bytecode, ported onto various hardware-based platforms.
   b) The Java Application Programming Interface (Java API)

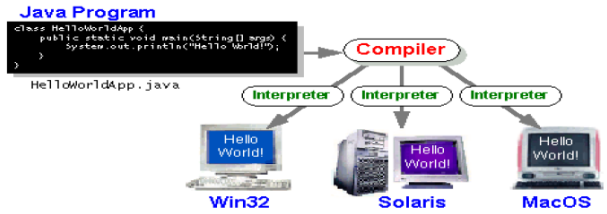
e-Macao-16-2-28

Java Execution Platform



e-Macao-16-2-29

Java Platform Independence



e-Macao-16-2-30

Java Program Execution

Java programs are both compiled and interpreted:

Steps:

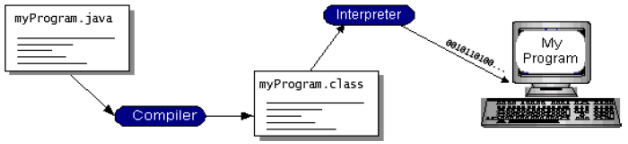
- write the Java program
- compile the program into bytecode
- execute (interpret) the bytecode on the computer through the Java Virtual Machine

Compilation happens once.

Interpretation occurs each time the program is executed.

e-Macao-16-2-31

Java Execution Process



e-Macao-16-2-32

Java API

What is Java API?

1) a large collection of ready-made software components that provide many useful capabilities, e.g. graphical user interface
2) grouped into libraries (packages) of related classes and interfaces
3) together with JVM insulates Java programs from the hardware and operating system variations

e-Macao-16-2-33

Java Program Types

Types of Java programs:

1) applications – standalone (desktop) Java programs, executed from the command line, only need the Java Virtual Machine to run
2) applets – Java program that runs within a Java-enabled browser, invoked through a "applet" reference on a web page, dynamically downloaded to the client computer
3) servlets – Java program running on the web server, capable of responding to HTTP requests made through the network
4) etc.

e-Macao-16-2-34

Java Platform Features 1

1) essentials - objects, strings, threads, numbers, input/output, data structures, system properties, date and time, and others.
2) networking:
   1) Universal Resource Locator (URL)
   2) Transmission Control Protocol (TCP)
   3) User Datagram Protocol (UDP) sockets
   4) Internet Protocol (IP) addresses
3) internationalization - programs that can be localized for users worldwide, automatically adapting to specific locales and appropriate languages.

e-Macao-16-2-35

Java Platform Features 2

4) security – low-level and high-level security, including electronic signatures, public and private key management, access control, and certificates
5) software components – JavaBeans can plug into an existing component architecture
6) object serialization - lightweight persistence and communication, in particular using Remote Method Invocation (RMI)
7) Java Database Connectivity (JDBC) - provides uniform access to a wide range of relational databases

e-Macao-16-2-36

Java Technologies

Different technologies depending on the target applications:

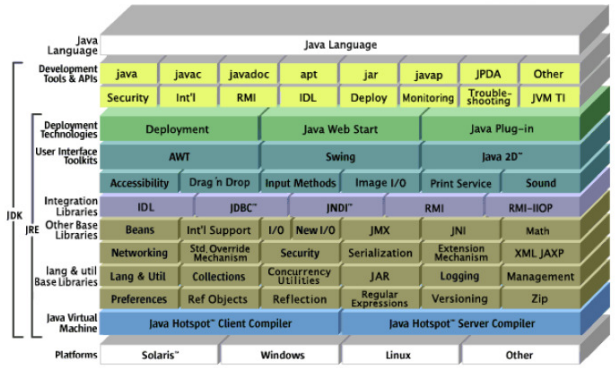
1) desktop applications - Java 2 Standard Edition (J2SE)
2) enterprise applications – Java 2 Enterprise Edition (J2EE)
3) mobile applications – Java 2 Mobile Edition (J2ME)
4) smart card applications – JavaCard
5) etc.

Each edition puts together a large collections of packages offering functionality needed and relevant to a given application.

The Java Virtual Machine remains essentially the same.

e-Macao-16-2-37

Java Technology: SDK



e-Macao-16-2-38

Exercise: Java Technology

1) Explain the statement "There is more to Java than the Language".
2) Enumerate and explain Java design goals.
3) How does Java maintain a balance between Interpretation and High Performance?.
4) Java program is termed "Write once run everywhere". Explain.
5) Why is it difficult to write viruses and malicious programs with Java?

e-Macao-16-2-39

Simple Java Program

A class to display a simple message:

```
class MyProgram {
     public static void main(String[] args) {
           System.out.println("First Java program.");
     }
}
```

e-Macao-16-2-40

Running the Program

Type the program, save as `MyProgram.java`.

In the command line, type:

```
> dir
MyProgram.java
> javac MyProgram.java
> dir
MyProgram.java, MyProgram.class
> java MyProgram
First Java program.
```

e-Macao-16-2-41

Explaining the Process

1) creating a source file - a source file contains text written in the Java programming language, created using any text editor on any system.
2) compiling the source file – Java compiler (javac) reads the source file and translates its text into instructions that the Java interpreter can understand. These instructions are called bytecode.
3) running the compiled program – Java interpreter (java) installed takes as input the bytecode file and carries out its instructions by translating them on the fly into instructions that your computer can understand.

e-Macao-16-2-42

Java Program Explained

`MyProgram` is the name of the class:

```
class MyProgram {
```

`main` is the method with one parameter `args` and no results:

```
   public static void main(String[] args) {
```

`println` is a method in the standard `System` class:

```
      System.out.println("First Java program.");
   }
}
```

e-Macao-16-2-43

Classes and Objects

A class is the basic building block of Java programs.

A class encapsulates:

a) data (attributes) and

b) operations on this data (methods)

and permits to create objects as its instances.

e-Macao-16-2-44

Main Method

The `main` method must be present in every Java application:

1) `public static void main(String[] args)` where:
   a) `public` means that the method can be called by any object
   b) `static` means that the method is shared by all instances
   c) `void` means that the method does not return any value
2) When the interpreter executes an application, it starts by calling its `main` method which in turn invokes other methods in this or other classes.
3) The main method accepts a single argument – a string array, which holds all command-line parameters.

e-Macao-16-2-45

Exercise: Java Program

1) Personalize the `MyProgram` program with your name so that it tells you "`Hello, my name is …`"
2) Write a program that produces the following output:

```
Welcome to e-Macao Java Workshop!
I hope you will benefit from the training.
```

3) Here is a slightly modified version of `MyProgram`:

```
class MyProgram2 {
   public void static Main(String args) {
     system.out.println("First Java Program!);
   }
}
```

The program has some errors. Fix the errors so that the program successfully compiles and runs. What were the errors?

e-Macao-16-2-46

Using API Documentation 1

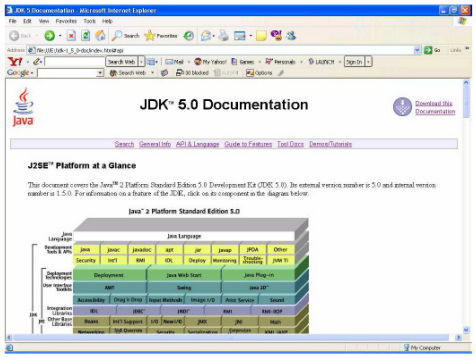
Ability to use Java API Specification is crucial for any practicing programmer. Here are the steps:

1) download the API documentation from http://java.sun.com/j2se/1.5.0/download.jsp
2) Extract the archive file

e-Macao-16-2-47

Using API Documentation 2

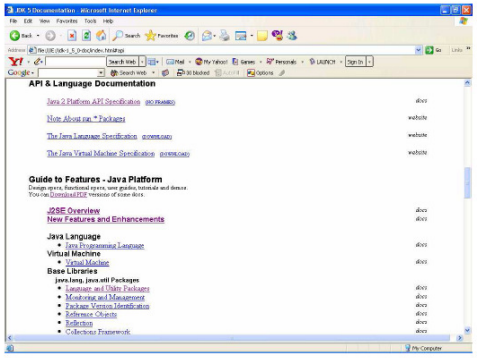
3) open `index.html`



e-Macao-16-2-48

Using API Documentation 3

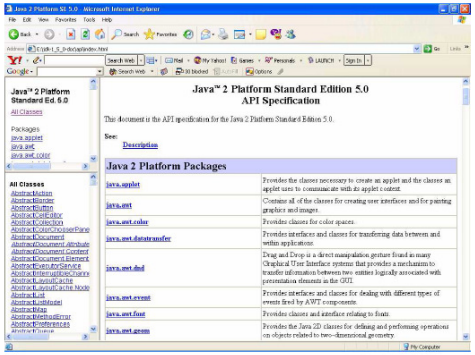
4) click on `API & Language`



e-Macao-16-2-49

Using API Documentation 4

5) click on `Java 2 Platform API Specification`



e-Macao-16-2-50

Java 2 Platform API Specification

The window is divided into three panes:

1) specification pane - click on any package, package pane will display all classes, interfaces, exceptions and errors that belong to that package.
2) package pane - click on any class, class pane will display all information about the class.
3) class pane – contains such information as:
    1) inheritance hierarchy
    2) implemented interfaces
    3) constructors
    4) attributes
    5) methods

e-Macao-16-2-51

Java Environment Setup 1

Setting up `JAVA_HOME` and `PATH` environment variables is necessary for the Java tools and applications to work properly from any directory.

Steps to carry out:

1) `JAVA_HOME` variable should point to the top installation directory of Java environment.
2) `PATH` should point to the `bin` directory where the Java Virtual Machine - interpreter (`java`), compiler (`javac`) and other executables are located.

e-Macao-16-2-52

Java Environment Setup 2

With more than one JVM installed, the one you wish to use must appear as the first one in the PATH variable.

Example:

1) Windows - add `%JAVA_HOME%\bin` to the beginning of the `PATH` variable.
2) Linux - include `PATH="$PATH:$JAVA_HOME/bin:."` in your `/etc/profile` or `.bashrc` files.

To test the environment, open the command window and run:

```
java -version
```

The current version number of the installed JVM should appear.

e-Macao-16-2-53

Summary: Introduction

Material covered:

1) What is the origin and history of Java?
2) What is Java Technology and its components?
3) What are the basic features of the Java language?
4) What are the basic features of the Java execution platform?
5) How to write, compile and execute a simple Java application?
6) How to use Java API documentation?
7) How to set up the Java execution environment?

e-Macao-16-2-54

Exercise: API Specification

1) Download and extract the latest JDK 5.0 Documentation.
2) Using the Documentation
   a) Locate the `java.math` package
   b) How many classes does it have?
   c) List all the classes.
   d) List their inheritance hierarchies.
   e) List their implemented interfaces.
   f) How many constructors does each have?
   g) How many fields does each have?
   h) How many methods does each have?

A.2. Language

Language

e-Macao-16-2-56

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-57

Language

Java is intrinsically an object-oriented language.

However, for presentation reasons explain it in two parts:

a) procedural part
b) object-oriented part

e-Macao-16-2-58

Language Overview

a) syntax – whitespaces, identifiers, comments, separators, keywords
b) types – simple types byte, short, int, long, float double, char, boolean
c) variables – declaration, initialization, scope, conversion and casting
d) arrays – declaration, initialization, one- and multi-dimensional arrays
e) operators – arithmetic, relational, conditional, shift, logical, assignment
f) control flow – branching, selection, iteration, jumps and returns

A.2.1. Syntax

Syntax

e-Macao-16-2-60

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-61

Java Syntax

On the most basic level, Java programs consist of:

a) whitespaces
b) identifiers
c) comments
d) literals
e) separators
f) keywords
g) operators

Each of them will be described in order.

e-Macao-16-2-62

Whitespaces

A whitespace is a space, tab or new line.
Java is a form-free language that does not require special indentation.

A program could be written like this:

```
class MyProgram {
  public static void main(String[] args) {
      System.out.println("First Java program.");
  }
}
```

It could be also written like this:

```
class MyProgram { public static void main(String[] args)
{ System.out.println("First Java program.");  } }
```

e-Macao-16-2-63

Identifiers

Java identifiers:

a) used for class names, method names, variable names
b) an identifier is any sequence of letters, digits, "_" or "$" characters that do not begin with a digit
c) Java is case sensitive, so `value`, `Value` and `VALUE` are all different.

Seven identifiers in this program:

```
class MyProgram {
  public static void main(String[] args) {
      System.out.println("First Java program.");
      }
}
```

e-Macao-16-2-64

Comments 1

Three kinds of comments:

1) Ignore the text between `/*` and `*/`:

```
/* text */
```

2) Documentation comment (`javadoc` tool uses this kind of comment to automatically generate software documentation):

```
/** documentation */
```

3) Ignore all text from `//` to the end of the line:

```
// text
```

e-Macao-16-2-65

Comments 2

```
/**
   *   MyProgram implements application that displays
       *      a simple message on the standard output
   device.
*/
class MyProgram {
   /* The main method of the class.*/
   public static void main(String[] args) {
            //display string
            System.out.println("First Java program.");
   }
}
```

e-Macao-16-2-66

Literals

A literal is a constant value of certain type.

It can be used anywhere values of this type are allowed.

Examples:

a) 100

b) 98.6

c) 'X'

d) "test"

```
class MyProgram {
  public static void main(String[] args) {
      System.out.println("My first Java program.");
      }
}
```

e-Macao-16-2-67

Separators

()	parenthesis	lists of parameters in method definitions and invocations, precedence in expressions, type casts
{}	braces	block of code, class definitions, method definitions, local scope, automatically initialized arrays
[]	brackets	declaring array types, referring to array values
;	semicolon	terminating statements, chain statements inside the "for" statement
,	comma	separating multiple identifiers in a variable declaration
.	period	separate package names from subpackages and classes, separating an object variable from its attribute or method

e-Macao-16-2-68

Keywords

Keywords are reserved words recognized by Java that cannot be used as identifiers. Java defines 49 keywords as follows:

abstract	continue	goto	package	synchronize
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

e-Macao-16-2-69

Exercise: Syntax

1) What's the difference between a *keyword* and an *identifier*?
2) What's the difference between an *identifier* and a *literal*?
3) What's the difference between `/** text */` and `/* text */`.
4) Which of these are valid identifiers?

```
int, anInt, I, i1, 1, thing1, 1thing, one-hundred,
one_hundred, something2do
```

5) Identify the literals, separators and identifiers in the progam below.

```
class MyProgram {
   int i = 30;  int j = i; char c = 'H';
   public static void main(String[] args) {
        System.out.println("i and j " + i + "  " + j);
   } }
```

6) What's the difference between a brace `{}` and a bracket `()`?

A.2.2. Types

Types

e-Macao-16-2-71

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-72

Strong Typing

Java is a strongly-typed language:

a) every variable and expression has a type
b) every type is strictly defined
c) all assignments are checked for type-compatibility
d) no automatic conversion of non-compatible, conflicting types
e) Java compiler type-checks all expressions and parameters
f) any typing errors must be corrected for compilation to succeed

e-Macao-16-2-73

Simple Types

Java defines eight simple types:

1) `byte` – 8-bit integer type
2) `short` – 16-bit integer type
3) `int` – 32-bit integer type
4) `long` – 64-bit integer type
5) `float` – 32-bit floating-point type
6) `double` – 64-bit floating-point type
7) `char` – symbols in a character set
8) `boolean` – logical values `true` and `false`

e-Macao-16-2-74

Simple Type: byte

8-bit integer type.

Range: `-128` to `127`.

Example:

```
byte b = -15;
```

Usage: particularly when working with data streams.

e-Macao-16-2-75

Simple Type: short

16-bit integer type.

Range: `-32768` to `32767`.

Example:

```
short c = 1000;
```

Usage: probably the least used simple type.

e-Macao-16-2-76

Simple Type: int

32-bit integer type.

Range: `-2147483648` to `2147483647`.

Example:

```
int b = -50000;
```

Usage:

1) Most common integer type.
2) Typically used to control loops and to index arrays.
3) Expressions involving the `byte`, `short` and `int` values are promoted to `int` before calculation.

e-Macao-16-2-77

Simple Type: long

64-bit integer type.

Range: `-9223372036854775808` to `9223372036854775807`.

Example:

```
long l = 10000000000000000;
```

Usage:

1) useful when `int` type is not large enough to hold the desired value

e-Macao-16-2-78

Example: long

```
// compute the light travel distance
class Light {
  public static void main(String args[]) {
            int lightspeed = 186000;
            long days = = 1000;
            long seconds = days * 24 * 60 * 60;
            long distance = lightspeed * seconds;
            System.out.print("In " + days);
            System.out.print(" light will travel about
  ");
            System.out.println(distance + " miles.");
  }
}
```

e-Macao-16-2-79

Simple Type: float

32-bit floating-point number.

Range: `1.4e-045` to `3.4e+038`.

Example:

```
float f = 1.5;
```

Usage:

1) fractional part is needed
2) large degree of precision is not required

e-Macao-16-2-80

Simple Type: double

64-bit floating-point number.

Range: `4.9e-324` to `1.8e+308`.

Example:

```
double pi = 3.1416;
```

Usage:

1) accuracy over many iterative calculations
2) manipulation of large-valued numbers

e-Macao-16-2-81

Example: double

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
      double pi = 3.1416;     // approximate pi value
      double r = 10.8;        // radius of circle
      double a = pi * r * r;  // compute area
      System.out.println("Area of circle is " + a);
  }
}
```

e-Macao-16-2-82

Simple Type: char

16-bit data type used to store characters.

Range: `0` to `65536`.

Example:

```
char c = 'a';
```

Usage:

1) Represents both ASCII and Unicode character sets; Unicode defines a character set with characters found in (almost) all human languages.
2) Not the same as in C/C++ where `char` is 8-bit and represents ASCII only.

e-Macao-16-2-83

Example: char

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
      char ch1, ch2;
      ch1 = 88;  // code for X
      ch2 = 'Y';
      System.out.print("ch1 and ch2: ");
      System.out.println(ch1 + " " + ch2);
  }
}
```

e-Macao-16-2-84

Another Example: char

It is possible to operate on `char` values as if they were integers:

```
class CharDemo2 {
     public static void main(String args[]) {
          char c = 'X';
          System.out.println("c contains " + c);
          c++;  // increment c
          System.out.println("c is now " + c);
     }
}
```

e-Macao-16-2-85

Simple Type: boolean

Two-valued type of logical values.
Range: values `true` and `false`.

Example:

```
boolean b = (1<2);
```

Usage:

1) returned by relational operators, such as `1<2`
2) required by branching expressions such as `if` or `for`

e-Macao-16-2-86

Example: boolean

```
class BoolTest {
  public static void main(String args[]) {
     boolean b;
     b = false;
     System.out.println("b is " + b);
     b = true;
     System.out.println("b is " + b);
     if (b) System.out.println("executed");
     b = false;
     if (b) System.out.println("not executed");
     System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

e-Macao-16-2-87

Literals Revisited

Literals express constant values.

The form of a literal depends on its type:

1) integer types
2) floating-point types
3) character type
4) boolean type
5) string type

e-Macao-16-2-88

Literals: Integer Types

Writing numbers with different bases:

1) decimal – `123`
2) octal – `0173`
3) hexadecimal – `0x7B`

Integer literals are of type `int` by default.

Integer literal written with "`L`" (e.g. `123L`) are of type `long`.

e-Macao-16-2-89

Literals: Floating-Point Types

Two notations:

1) standard – `2000.5`
2) scientific – `2.0005E3`

Floating-point literals are of type `double` by default.

Floating-point literal written with "`F`" (e.g. `2.0005E3F`) are of type `float`.

e-Macao-16-2-90

Literals: Boolean

Two literals are allowed only: `true` and `false`.

Those values do not convert to any numerical representation.

In particular:

1) `true` is not equal to `1`
2) `false` is not equal to `0`

e-Macao-16-2-91

Literals: Characters

Character literals belong to the Unicode character set.
Representation:

1) visible characters inside quotes, e.g. `'a'`
2) invisible characters written with escape sequences:
   - a) `\ddd` octal character `ddd`
   - b) `\uxxxx` hexadecimal Unicode character `xxxx`
   - c) `\'` single quote
   - d) `\"` double quote
   - e) `\\` backslash
   - f) `\r` carriage return
   - g) `\n` new line
   - h) `\f` form feed
   - i) `\t` tab
   - j) `\b` backspace

e-Macao-16-2-92

Literals: String

String is not a simple type.

String literals are character-sequences enclosed in double quotes.

Example:

```
"Hello World!"
```

Notes:

1) escape sequences can be used inside string literals
2) string literals must begin and end on the same line
3) unlike in C/C++, in Java `String` is not an array of characters

e-Macao-16-2-93

Exercise: Types

1) What is the value of `b` in the following code snippet?

```
byte b = 0;
b += 1;
```

2) What happens when this code is executed?

```
char c = -1;
```

3) Which of the following lines will compile without warning or error. Explain each line.

```
float f=1.3;
char c="a";
byte b=257;
boolean b=null;
int i=10;
```

4) Write a java statement that will print the following line to the console:

```
To print a ' " ', we use the ' \"  'escape sequence.
```

A.2.3. Variables

Variables

e-Macao-16-2-95

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-96

Outline: Variables

1) declaration – how to assign a type to a variable
2) initialization – how to give an initial value to a variable
3) scope – how the variable is visible to other parts of the program
4) lifetime – how the variable is created, used and destroyed
5) type conversion – how Java handles automatic type conversion
6) type casting – how the type of a variable can be narrowed down
7) type promotion – how the type of a variable can be expanded

e-Macao-16-2-97

Variables

Java uses variables to store data.

To allocate memory space for a variable JVM requires:

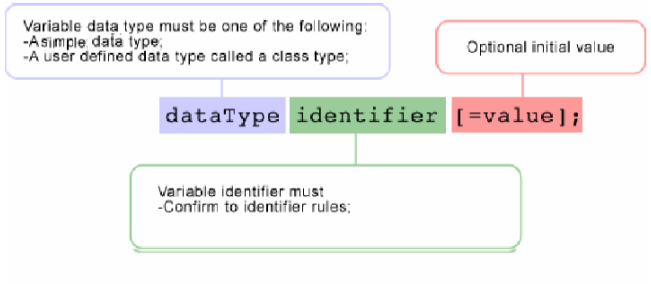
1) to specify the data type of the variable
2) to associate an identifier with the variable
3) optionally, the variable may be assigned an initial value

All done as part of variable declaration.

e-Macao-16-2-98

Basic Variable Declaration

Basic form of variable declaration:



e-Macao-16-2-99

Variable Declaration

We can declare several variables at the same time:

```
type identifier [=value][, identifier [=value] …];
```

Examples:

```
int a, b, c;
int d = 3, e, f = 5;
byte hog = 22;
double pi = 3.14159;
char kat = 'x';
```

e-Macao-16-2-100

Constant Declaration

A variable can be declared as final:

```
final double PI = 3.14;
```

The value of the final variable cannot change after it has been initialized:

```
PI = 3.13;
```

e-Macao-16-2-101

Variable Identifiers

Identifiers are assigned to variables, methods and classes.

An identifier:

1) starts with a letter, underscore _ or dollar $
2) can contain letters, digits, underscore or dollar characters
3) it can be of any length
4) it must not be a keyword (e.g. `class`)
5) it must be unique in its scope

Examples: `identifier`, `userName`, `_sys_var1`, `$change`

The code of Java programs is written in Unicode, rather than ASCII, so letters and digits have considerably wider definitions than just a-z and 0-9.

e-Macao-16-2-102

Naming Conventions

Conventions are not part of the language.

Naming conventions:

1) variable names begin with a lowercase letter
2) class names begin with an uppercase letter
3) constant names are all uppercase

If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter.

The underscore character is used only to separate words in constants, as they are all caps and thus cannot be case-delimited.

e-Macao-16-2-103

Variable Initialization

During declaration, variables may be optionally initialized.

Initialization can be static or dynamic:

1) static – initialize with a literal:

```
int n = 1;
```

2) dynamic – initialize with an expression composed of any literals, variables or method calls available at the time of initialization:

```
int m = n + 1;
```

The types of the expression and variable must be the same.

e-Macao-16-2-104

Example: Variable Initialization

```
class DynamicInit {
   public static void main(String args[]) {
      double a = 3.0, b = 4.0;
      double c = Math.sqrt(a * a + b * b);
      System.out.println("Hypotenuse is " + c);
   }
}
```

e-Macao-16-2-105

Variable Scope

Scope determines the visibility of program elements with respect to other program elements.

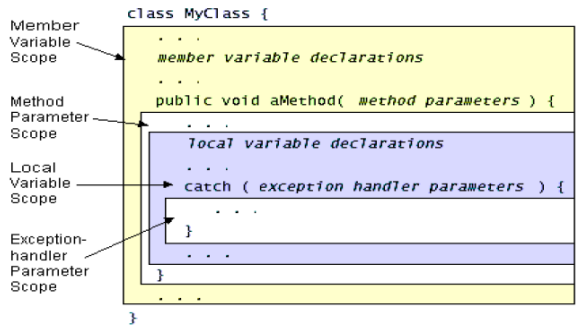
In Java, scope is defined separately for classes and methods:

1) variables defined by a class have a "global" scope
2) variables defined by a method have a "local" scope

We consider the scope of method variables only; class variables will be considered later.

e-Macao-16-2-106

Variable Scope



e-Macao-16-2-107

Scope Definition

A scope is defined by a block:

```
{
      …
}
```

A variable declared inside the scope is not visible outside:

```
{
      int n;
}
n = 1;
```

e-Macao-16-2-108

Scope Nesting

Scopes can be nested:

```
{     …     { … }       …     }
```

Variables declared in the outside scope are visible in the inside scope, but not the other way round:

```
{
      int i;
      {
            int n = i;
      }
      int m = n;
}
```

e-Macao-16-2-109

Example: Variable Scope

```
class Scope {
     public static void main(String args[]) {
           int x;
           x = 10;
           if (x == 10) {
                 int y = 20;
                 System.out.println("x and y: " + x + "
" + y);
                 x = y * 2;
           }
           System.out.println("x is " + x + "y is" + y);
     }
}
```

e-Macao-16-2-110

Declaration Order

Method variables are only valid after their declaration:

```
{
      int n = m;
      int m;
}
```

e-Macao-16-2-111

Variable Lifetime

Variables are created when their scope is entered by control flow and destroyed when their scope is left:

1) A variable declared in a method will not hold its value between different invocations of this method.
2) A variable declared in a block looses its value when the block is left.
3) Initialized in a block, a variable will be re-initialized with every re-entry.

Variable's lifetime is confined to its scope!

e-Macao-16-2-112

Example: Variable Lifetime

```
class LifeTime {
     public static void main(String args[]) {
           int x;
           for (x = 0; x < 3; x++) {
                 int y = -1;
                 System.out.println("y is: " + y);
                 y = 100;
                 System.out.println("y is now: " + y);
           }
     }
}
```

e-Macao-16-2-113

Type Differences

Suppose a value of one type is assigned to a variable of another type.

```
T1 t1;
T2 t2 = t1;
```

What happens? Different situations:

1) types T1 and T2 are incompatible
2) types T1 and T2 are compatible:
   a) T1 and T2 are the same
   b) T1 is larger than T2
   c) T2 is larger than T1

e-Macao-16-2-114

Type Compatibility

When types are compatible:

1) integer types and floating-point types are compatible with each other
2) numeric types are not compatible with `char` or `boolean`
3) `char` and `boolean` are not compatible with each other

Examples:

```
byte b;
int i = b;
char c = b;
```

e-Macao-16-2-115

Widening Type Conversion

Java performs automatic type conversion when:

1) two types are compatible
2) destination type is larger then the source type

Example:

```
int i;
double d = i;
```

e-Macao-16-2-116

Narrowing Type Conversion

When:

1) two types are compatible
2) destination type is smaller then the source type

then Java will not carry out type-conversion:

```
int i;
byte b = i;
```

Instead, we have to rely on manual type-casting:

```
int i;
byte b = (byte)i;
```

e-Macao-16-2-117

Type Casting

General form: `(targetType) value`

Examples:

1) integer value will be reduced module `byte`'s range:

```
int i;
byte b = (byte) i;
```

2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

e-Macao-16-2-118

Example: Type Casting

```
class Conversion {
   public static void main(String args[]) {
      byte b;
      int i = 257;
      double d = 323.142;
      System.out.println("\nConversion of int to byte.");
      b = (byte) i;
      System.out.println("i and b " + i + " " + b);
      System.out.println("\ndouble to int.");
      i = (int) d;
      System.out.println("d and i " + d + " " + i);
   }
}
```

e-Macao-16-2-119

Type Promotion

In an expression, precision required to hold an intermediate value may sometimes exceed the range of either operand:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Java promotes each `byte` operand to `int` when evaluating the expression.

e-Macao-16-2-120

Type Promotion Rules

1) `byte` and `short` are always promoted to `int`
2) if one operand is `long`, the whole expression is promoted to `long`
3) if one operand is `float`, the entire expression is promoted to `float`
4) if any operand is `double`, the result is `double`

Danger of automatic type promotion:

```
byte b = 50;
b = b * 2;
```

What is the problem?

e-Macao-16-2-121

Example: Type Promotion

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println("result = " + result);
  }
}
```

e-Macao-16-2-122

Exercise: Variables

1) What's the difference between widening conversion and casting?
2) What is the output of the program below?.

```
class MyProgram {
  static final double pi = 3.15;
  static double radius = 2.78;
  public static void main(String[] args) {
    pi = 3.14;
    double area = pi * radius * radius;
    System.out.println("The Area is " + area);
  }
}
```

3) What is the value of c[50] after this declaration:

```
char[] c = new char[100];
```

A.2.4. Arrays

Arrays

e-Macao-16-2-124

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-125

Arrays

An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.

Arrays are:
1) declared
2) created
3) initialized
4) used

Also, arrays can have one or several dimensions.

e-Macao-16-2-126

Array Declaration

Array declaration involves:

1) declaring an array identifier
2) declaring the number of dimensions
3) declaring the data type of the array elements

Two styles of array declaration:

```
type array-variable[];
```

or

```
type [] array-variable;
```

e-Macao-16-2-127

Array Creation

After declaration, no array actually exists.

In order to create an array, we use the `new` operator:

```
type array-variable[];
array-variable = new type[size];
```

This creates a new array to hold `size` elements of type `type`, which reference will be kept in the variable `array-variable`.

e-Macao-16-2-128

Array Indexing

Later we can refer to the elements of this array through their indexes:

```
array-variable[index]
```

The array index always starts with zero!

The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

e-Macao-16-2-129

Example: Array Use

```
class Array {
      public static void main(String args[]) {
            int monthDays[];
            monthDays = new int[12];
            monthDays[0] = 31;
            monthDays[1] = 28;
            monthDays[2] = 31;
            monthDays[3] = 30;
            monthDays[4] = 31;
            monthDays[5] = 30;
            …
            monthDays[12] = 31;
            System.out.print("April has ");
            System.out.println(monthDays[3] +" days.");
      }
}
```

e-Macao-16-2-130

Array Initialization

Arrays can be initialized when they are declared:

```
int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Comments:

1) there is no need to use the `new` operator
2) the array is created large enough to hold all specified elements

e-Macao-16-2-131

Example: Array Initialization

```
class Array {
      public static void main(String args[]) {
            int mthDys[]=
                  {31,28,31,30,31,30,31,31,30,31,30,31};

            System.out.print("April ");
            System.out.println(mthDys[3]+ " days.");
      }
}
```

e-Macao-16-2-132

Multidimensional Arrays

Multidimensional arrays are arrays of arrays:

1) declaration

```
int array[][];
```

2) creation

```
int array = new int[2][3];
```

3) initialization

```
int array[][] = { {1, 2, 3}, {4, 5, 6} };
```

e-Macao-16-2-133

Example: Multidimensional Arrays

```
class Array {

   public static void main(String args[]) {

      int array[][] = { {1, 2, 3}, {4, 5, 6} };
      int i, j, k = 0;

      for(i=0; i<2; i++) {
            for(j=0; j<3; j++)
                  System.out.print(array[i][j] + " ");
                  System.out.println();
      }
   }
}
```

e-Macao-16-2-134

Exercise: Arrays

1) Write a program that creates an array of 10 integers with the initial values of 3.
2) Write a Java program to find the average of a sequence of nonnegative numbers entered by the user, where the user enters a negative number to terminate the input. Assume the only method in the class is the main method.
3) What's the index of the first and the last component of a one hundred component array?
4) What will happen if you try to compile and run the following code?

```
public class Q {
  public static void main(String argv[]){
    int var[]=new int[5];
    System.out.println(var[0]);
  } }
```

A.2.5. Operators

Operators

e-Macao-16-2-136

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-137

Operators Types

Java operators are used to build value expressions.

Java provides a rich set of operators:

1) assignment
2) arithmetic
3) relational
4) logical
5) bitwise
6) other

e-Macao-16-2-138

Operators and Operands

Each operator takes one, two or three operands:

1) a unary operator takes one operand

```
j++;
```

2) a binary operator takes two operands

```
i = j++;
```

3) a ternary operator requires three operands

```
i = (i>12) ? 1 : i++;
```

e-Macao-16-2-139

Assignment Operator

A binary operator:

```
variable = expression;
```

It assigns the value of the expression to the variable.

The types of the variable and expression must be compatible.

The value of the whole assignment expression is the value of the expression on the right, so it is possible to chain assignment expressions as follows:

```
int x, y, z;
x = y = z = 2;
```

e-Macao-16-2-140

Arithmetic Operators

Java supports various arithmetic operators for:

1) integer numbers
2) floating-point numbers

There are two kinds of arithmetic operators:

1) basic: addition, subtraction, multiplication, division and modulo
2) shortcut: arithmetic assignment, increment and decrement

e-Macao-16-2-141

Table: Basic Arithmetic Operators

+	op1 + op2	adds op1 and op2
-	op1 - op2	subtracts op2 from op1
*	op1 * op2	multiplies op1 by op2
/	op1 / op2	divides op1 by op2
%	op1 % op2	computes the remainder of dividing op1 by op2

e-Macao-16-2-142

Arithmetic Assignment Operators

Instead of writing

```
variable = variable operator expression;
```

for any arithmetic binary operator, it is possible to write shortly

```
variable operator= expression;
```

Benefits of the assignment operators:
1) save some typing
2) are implemented more efficiently by the Java run-time system

e-Macao-16-2-143

Table: Arithmetic Assignments

+=	v += expr;	v = v + expr;
-=	v -= expr;	v = v - expr;
*=	v *= expr;	v = v * expr;
/=	v /= expr;	v = v / expr;
%=	v %= expr;	v = v % expr;

e-Macao-16-2-144

Increment/Decrement Operators

Two unary operators:

1) ++ increments its operand by 1
2) -- decrements its operand by 1

The operand must be a numerical variable.

Each operation can appear in two versions:

- prefix version evaluates the value of the operand after performing the increment/decrement operation
- postfix version evaluates the value of the operand before performing the increment/decrement operation

e-Macao-16-2-145

Table: Increment/Decrement

++	v++	return value of v, then increment v
++	++v	increment v, then return its value
--	v--	return value of v, then decrement v
--	--v	decrement v, then return its value

e-Macao-16-2-146

Example: Increment/Decrement

```
class IncDec {
   public static void main(String args[]) {
      int a = 1;
      int b = 2;
      int c, d;
      c = ++b;
      d = a++;
      c++;
      System.out.println("a= " + a);
      System.out.println("b= " + b);
      System.out.println("c= " + c);
   }
}
```

e-Macao-16-2-147

Relational Operators

Relational operators determine the relationship that one operand has to the other operand, specifically equality and ordering.

The outcome is always a value of type `boolean`.

They are most often used in branching and loop control statements.

e-Macao-16-2-148

Table: Relational Operators

==	equals to	apply to any type
!=	not equal to	apply to any type
>	greater than	apply to numerical types only
<	less than	apply to numerical types only
>=	greater than or equal	apply to numerical types only
<=	less than or equal	apply to numerical types only

e-Macao-16-2-149

Logical Operators

Logical operators act upon `boolean` operands only.

The outcome is always a value of type `boolean`.

In particular, "and" and "or" logical operators occur in two forms:

1) full – `op1 & op2` and `op1 | op2` where both `op1` and `op2` are evaluated
2) short-circuit - `op1 && op2` and `op1 || op2` where `op2` is only evaluated if the value of `op1` is insufficient to determine the final outcome

e-Macao-16-2-150

Table: Logical Operators

`&`	`op1 & op2`	logical `AND`
`|`	`op1 | op2`	logical `OR`
`&&`	`op1 && op2`	short-circuit `AND`
`||`	`op1 || op2`	short-circuit `OR`
`!`	`! op`	logical `NOT`
`^`	`op1 ^ op2`	logical `XOR`

e-Macao-16-2-151

Example: Logical Operators

```
class LogicalDemo {

   public static void main(String[] args) {
      int n = 2;
      if (n != 0 && n / 0 > 10)
            System.out.println("This is true");
      else
            System.out.println("This is false");
   }

}
```

e-Macao-16-2-152

Bitwise Operators

Bitwise operators apply to integer types only.

They act on individual bits of their operands.

There are three kinds of bitwise operators:

1) basic – bitwise `AND`, `OR`, `NOT` and `XOR`
2) shifts – left, right and right-zero-fill
3) assignments – bitwise assignment for all basic and shift operators

e-Macao-16-2-153

Table: Bitwise Operators

~	~ op	inverts all bits of its operand
&	op1 & op2	produces 1 bit if both operands are 1
|	op1 | op2	produces 1 bit if either operand is 1
^	op1 ^ op2	produces 1 bit if exactly one operand is 1
>>	op1 >> op2	shifts all bits in op1 right by the value of op2
<<	op1 << op2	shifts all bits in op1 left by the value of op2
>>>	op1 >>> op2	shifts op1 right by op2 value, write zero on the left

e-Macao-16-2-154

Example: Bitwise Operators

```
class BitLogic {
   public static void main(String args[]) {
      String binary[] = { "0000","0001","0010", … };
      int a = 3; // 0 + 2 + 1 or 0011 in binary
      int b = 6; // 4 + 2 + 0 or 0110 in binary
      int c = a | b;
      int d = a & b;
      int e = a ^ b;
      System.out.print("a =" + binary[a]);
      System.out.println("and b =" +     binary[b]);
      System.out.println("a|b = " + binary[c]);
      System.out.println("a&b = " + binary[d]);
      System.out.println("a^b = " + binary[e]);
   }
}
```

e-Macao-16-2-155

Other Operators

?:	shortcut if-else statement
[]	used to declare arrays, create arrays, access array elements
.	used to form qualified names
(params)	delimits a comma-separated list of parameters
(type)	casts a value to the specified type
new	creates a new object or a new array
instanceof	determines if its first operand is an instance of the second

e-Macao-16-2-156

Conditional Operator

General form:

```
expr1? expr2 : expr3
```

where:

1) `expr1` is of type boolean
2) `expr2` and `expr3` are of the same type

If `expr1` is true, `expr2` is evaluated, otherwise `expr3` is evaluated.

e-Macao-16-2-157

Example: Conditional Operator

```
class Ternary {
   public static void main(String args[]) {
      int i, k;

      i = 10;
      k = i < 0 ? -i : i;
      System.out.print("Abs value of " + i + " is " + k);

      i = -10;
      k = i < 0 ? -i : i;
      System.out.print("Abs value of " + i + " is " + k);
   }
}
```

e-Macao-16-2-158

Operator Precedence

Java operators are assigned precedence order.

Precedence determines that the expression

```
1 + 2 * 6 / 3 > 4 && 1 < 0
```

if equivalent to

```
(((1 + ((2 * 6) / 3)) > 4) && (1 < 0))
```

When operators have the same precedence, the earlier one binds stronger.

e-Macao-16-2-159

Table: Operator Precedence

highest			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
lowest			

e-Macao-16-2-160

Exercise: Operators

1) What operators do the code snippet below contain?

   ```
   arrayOfInts[j] > arrayOfInts[j+1];
   ```

2) Consider the following code snippet:

   ```
   int i = 10;
   int n = i++%5;
   ```

   a) What are the values of `i` and `n` after the code is executed?

   b) What are the final values of `i` and `n` if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i`))?

3) What is the value of `i` after the following code snippet executes?

   ```
   int i = 8;
   i >>=2;
   ```

4) What's the result of `System.out.println(010| 4);` ?

A.2.6. Control Flow

Control Flow

e-Macao-16-2-162

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-163

Control Flow

Writing a program means typing statements into a file.

Without control flow, the interpreter would execute these statements in the order they appear in the file, left-to-right, top-down.

Control flow statements, when inserted into the text of the program, determine in which order the program should be executed.

e-Macao-16-2-164

Control Flow Statements

Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.

Control statements are divided into three groups:

1) selection statements allow the program to choose different parts of the execution based on the outcome of an expression
2) iteration statements enable program execution to repeat one or more statements
3) jump statements enable your program to execute in a non-linear fashion

e-Macao-16-2-165

Selection Statements

Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.

Java provides four selection statements:

1) `if`
2) `if-else`
3) `if-else-if`
4) `switch`

e-Macao-16-2-166

if Statement

General form:

```
if (expression) statement
```

If `expression` evaluates to `true`, execute `statement`, otherwise do nothing.

The expression must be of type `boolean`.

e-Macao-16-2-167

Simple/Compound Statement

The component statement may be:

1) simple

```
if (expression) statement;
```

2) compound

```
if (expression) {
      statement;
}
```

e-Macao-16-2-168

if-else Statement

Suppose you want to perform two different statements depending on the outcome of a `boolean` expression. `if-else` statement can be used.

General form:

```
if (expression) statement1
else statement2
```

Again, `statement1` and `statement2` may be simple or compound.

e-Macao-16-2-169

if-else-if Statement

General form:

```
if (expression1) statement1
else if (expression2) statement2
else if (expression3) statement3
…
else statement
```

Semantics:
1) statements are executed top-down
2) as soon as one expressions is true, its statement is executed
3) if none of the expressions is true, the last statement is executed

e-Macao-16-2-170

Example: if-else-if

```
class IfElse {
  public static void main(String args[]) {
     int month = 4;
     String season;
     if (month == 12 || month == 1 || month == 2)
        season = "Winter";
     else if(month == 3 || month == 4 || month == 5)
        season = "Spring";
     else if(month == 6 || month == 7 || month == 8)
        season = "Summer";
     else if(month == 9 || month == 10 || month == 11)
        season = "Autumn";
     else season = "Bogus Month";
     System.out.println("April is in the " + season + ".");
  }
}
```

e-Macao-16-2-171

switch Statement

`switch` provides a better alternative than `if-else-if` when the execution follows several branches depending on the value of an expression.

General form:

```
switch (expression) {
   case value1: statement1; break;
   case value2: statement2; break;
   case value3: statement3; break;
   …
   default: statement;
}
```

e-Macao-16-2-172

switch Assumptions/Semantics

Assumptions:

1) `expression` must be of type `byte`, `short`, `int` or `char`
2) each of the `case` values must be a literal of the compatible type
3) `case` values must be unique

Semantics:

1) `expression` is evaluated
2) its value is compared with each of the `case` values
3) if a match is found, the statement following the `case` is executed
4) if no match is found, the statement following `default` is executed

`break` makes sure that only the matching statement is executed.

Both `default` and `break` are optional.

e-Macao-16-2-173

Example: switch 1

```
class Switch {
  public static void main(String args[]) {
     int month = 4;
     String season;
     switch (month) {
        case 12:
        case 1:
        case 2: season = "Winter"; break;
        case 3:
        case 4:
        case 5: season = "Spring"; break;
        case 6:
        case 7:
        case 8: season = "Summer"; break;
```

e-Macao-16-2-174

Example: switch 2

```
        case 9:
        case 10:
        case 11: season = "Autumn"; break;
        default: season = "Bogus Month";
     }
     System.out.println("April is in " + season + ".");
  }
}
```

e-Macao-16-2-175

Nested switch Statement

A switch statement can be nested within another switch statement:

```
switch(count) {
   case 1:
      switch(target) {
         case 0:System.out.println("target is zero");
                break;
         case 1:System.out.println("target is one");
                break;
      }
      break;
   case 2: …
}
```

Since, every switch statement defines its own block, no conflict arises between the case constants in the inner and outer switch statements.

e-Macao-16-2-176

Comparing switch and if

Two main differences:

1) switch can only test for equality, while if can evaluate any kind of boolean expression
2) Java creates a "jump table" for switch expressions, so a switch statement is usually more efficient than a set of nested if statements

e-Macao-16-2-177

Iteration Statements

Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.

Java provides three iteration statements:

1) `while`
2) `do-while`
3) `for`

e-Macao-16-2-178

while Statement

General form:

```
while (expression) statement
```

where `expression` must be of type `boolean`.

Semantics:

1) repeat execution of `statement` until `expression` becomes false
2) `expression` is always evaluated before `statement`
3) if `expression` is false initially, `statement` will never get executed

e-Macao-16-2-179

Simple/Compound Statement

The component statement may be:

1) simple

```
while (expression) statement;
```

2) compound

```
while (expression) {
      statement;
}
```

e-Macao-16-2-180

Example: while

```
class MidPoint {
  public static void main(String args[]) {
     int i, j;
     i = 100;
     j = 200;
     while(++i < --j) {
        System.out.println("i is " + i);
        System.out.println("j is " + j);
     }
     System.out.println("The midpoint is " + i);
  }
}
```

e-Macao-16-2-181

do-while Statement

If a component statement has to be executed at least once, the `do-while` statement is more appropriate than the `while` statement.

General form:

```
do statement
while (expression);
```

where `expression` must be of type `boolean`.

Semantics:

1) repeat execution of `statement` until `expression` becomes false
2) `expression` is always evaluated after `statement`
3) even if `expression` is false initially, `statement` will be executed

e-Macao-16-2-182

Example: do-while

```
class DoWhile {
  public static void main(String args[]) {
     int i;
     i = 0;
     do
        i++;
     while ( 1/i < 0.001);
     System.out.println("i is " + i);
  }
}
```

e-Macao-16-2-183

for Statement

When iterating over a range of values, `for` statement is more suitable to use then `while` or `do-while`.

General form:

```
for (initialization; termination; increment)
   statement
```

where:

1) `initialization` statement is executed once before the first iteration
2) `termination` expression is evaluated before each iteration to determine when the loop should terminate
3) `increment` statement is executed after each iteration

e-Macao-16-2-184

for Statement Semantics

This is how the for statement is executed:

1) `initialization` is executed once
2) `termination` expression is evaluated:
   a) if false, the statement terminates
   b) otherwise, continue to (3)
3) `increment` statement is executed
4) component `statement` is executed
5) control flow continues from (2)

e-Macao-16-2-185

Loop Control Variable

The `for` statement may include declaration of a loop control variable:

```
for (int i = 0; i < 1000; i++) {
   …
}
```

The variable does not exist outside the `for` statement.

e-Macao-16-2-186

Example: for

```
class FindPrime {
  public static void main(String args[]) {
     int num = 14;
     boolean isPrime = true;
     for (int i=2; i < num/2; i++) {
           if ((num % i) == 0) {
                 isPrime = false;
              break;
           }
        }
        if (isPrime) System.out.println("Prime");
        else System.out.println("Not Prime");
  }
}
```

e-Macao-16-2-187

Many Initialization/Iteration Parts

The `for` statement may include several `initialization` and `iteration` parts.

Parts are separated by a comma:

```
int a, b;

for (a = 1, b = 4; a < b; a++, b--) {
   …
}
```

e-Macao-16-2-188

for Statement Variations

The `for` statement need not have all components:

```
class ForVar {
  public static void main(String args[]) {
     int i = 0;
     boolean done = false;
     for( ; !done; ) {
           System.out.println("i is " + i);
           if(i == 10) done = true;
           i++;
        }
  }
}
```

e-Macao-16-2-189

Empty for

In fact, all three components may be omitted:

```
public class EmptyFor {
  public static void main(String[] args) {
     int i = 0;
     for (; ;) {
        System.out.println("Infinite Loop " + i);
     }
  }
}
```

e-Macao-16-2-190

Jump Statements

Java jump statements enable transfer of control to other parts of program.

Java provides three jump statements:

1) `break`
2) `continue`
3) `return`

In addition, Java supports exception handling that can also alter the control flow of a program. Exception handling will be explained in its own section.

e-Macao-16-2-191

break Statement

The break statement has three uses:

1) to terminate a case inside the switch statement
2) to exit an iterative statement
3) to transfer control to another statement

(1) has been described.

We continue with (2) and (3).

e-Macao-16-2-192

Loop Exit with break

When `break` is used inside a loop, the loop terminates and control is transferred to the following instruction.

```
class BreakLoop {
  public static void main(String args[]) {
     for (int i=0; i<100; i++) {
        if (i == 10) break;
        System.out.println("i: " + i);
     }
     System.out.println("Loop complete.");
  }
}
```

e-Macao-16-2-193

break in Nested Loops

Used inside nested loops, `break` will only terminate the innermost loop:

```
class NestedLoopBreak {
  public static void main(String args[]) {
     for (int i=0; i<3; i++) {
        System.out.print("Pass " + i + ": ");
        for (int j=0; j<100; j++) {
           if (j == 10) break;
           System.out.print(j + " ");
        }
        System.out.println();
     }
     System.out.println("Loops complete.");
  }
}
```

e-Macao-16-2-194

Control Transfer with break

Java does not have an unrestricted “goto” statement, which tends to produce code that is hard to understand and maintain.

However, in some places, the use of gotos is well justified. In particular, when breaking out from the deeply nested blocks of code.

`break` occurs in two versions:

1) unlabelled
2) labeled

The labeled `break` statement is a “civilized” replacement for goto.

e-Macao-16-2-195

Labeled break

General form:

```
break label;
```

where `label` is the name of a label that identifies a block of code:

```
label: { … }
```

The effect of executing `break label;` is to transfer control immediately after the block of code identified by `label`.

e-Macao-16-2-196

Example: Labeled break

```
class Break {
  public static void main(String args[]) {
     boolean t = true;
     first: {
        second: {
           third: {
              System.out.println("Before the break.");
              if (t) break second;
              System.out.println("This won't execute");
           }
           System.out.println("This won't execute");
        }
        System.out.println("After second block.");
     }
} }
```

e-Macao-16-2-197

Example: Nested Loop break

```
class NestedLoopBreak {
  public static void main(String args[]) {
     outer: for (int i=0; i<3; i++) {
        System.out.print("Pass " + i + ": ");
        for (int j=0; j<100; j++) {
           if (j == 10) break outer; // exit both loops
           System.out.print(j + " ");
        }
        System.out.println("This will not print");
     }
  System.out.println("Loops complete.");
  }
}
```

e-Macao-16-2-198

break Without Label

It is not possible to break to any label which is not defined for an enclosing block. Trying to do so will result in a compiler error.

```
class BreakError {
  public static void main(String args[]) {
     one: for(int i=0; i<3; i++) {
        System.out.print("Pass " + i + ": ");
     }
     for (int j=0; j<100; j++) {
        if (j == 10) break one;
        System.out.print(j + " ");
     }
  }
}
```

e-Macao-16-2-199

continue Statement

The `break` statement terminates the block of code, in particular it terminates the execution of an iterative statement.

The `continue` statement forces the early termination of the current iteration to begin immediately the next iteration.

Like break, continue has two versions:

1) unlabelled – continue with the next iteration of the current loop
2) labeled – specifies which enclosing loop to continue

e-Macao-16-2-200

Example: Unlabeled continue

```
class Continue {
  public static void main(String args[]) {
     for (int i=0; i<10; i++) {
        System.out.print(i + " ");
        if (i%2 == 0) continue;
        System.out.println("");
     }
  }
}
```

e-Macao-16-2-201

Example: Labeled continue

```
class LabeledContinue {
  public static void main(String args[]) {
     outer: for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++) {
           if (j > i) {
              System.out.println();
              continue outer;
           }
           System.out.print(" " + (i * j));
        }
     }
     System.out.println();
  }
}
```

e-Macao-16-2-202

Return Statement

The return statement is used to return from the current method: it causes program control to transfer back to the caller of the method.

Two forms:

1) return without value

```
return;
```

2) return with value

```
return expression;
```

e-Macao-16-2-203

Example: Return

```
class Return {
  public static void main(String args[]) {
     boolean t = true;
     System.out.println("Before the return.");
     if (t) return; // return to caller
     System.out.println("This won't execute.");
  }
}
```

e-Macao-16-2-204

Exercise: Control Flow

1) Write a program that prints all the prime numbers between 1 and 49 to the console.

2) Write a program that prints the first 20 Fibonacci numbers.

   The Fibonacci numbers are defined as follows:

   The zeroth Fibonacci number is 1.
   The first Fibonacci number is also 1.
   The second Fibonacci number is 1 + 1 = 2.
   The third Fibonacci number is 1 + 2 = 3.

   In other words, except for the first two numbers each Fibonacci number is the sum of the two previous numbers.

A.3. Object Orientation

Object-Orientation

e-Macao-16-2-206

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-207

Data versus Operations

Data (nouns) versus operations (verbs):

```
data1
…                    } nouns
dataN

operation1 { … }
…                    } verbs
operationN { … }
```

e-Macao-16-2-208

Procedural Programming

Procedural programming is verb-oriented:

1) decomposition around *operations*
2) operation are divided into smaller operations

Programming Languages:

1) C
2) Pascal
3) Fortran, etc.

e-Macao-16-2-209

Example: Procedural Program

```
program AddNums {
   Integer a;
   Integer b;

   a = 100;
   b = 200;
   midPoint(a, b);

   procedure midPoint(Integer a, Integer b) {
      while(a < b) {
         println("a is " + a); a = a+1;
         println("b is " + b); b = b-1;
      }
   }
}
```

e-Macao-16-2-210

Drawbacks

1) data is given a second-class status when compared with operations
2) difficult to relate to the real world – there are no functions in real world
3) difficult to create new data types – reduces extensibility
4) programs are difficult to debug – little restriction to data access
5) programs are hard to understand – many variables have global scope
6) programs are hard to reuse – data/functions are mutually dependent
7) little support for developing and comprehending really large programs
8) top-down development approach tends to produce monolithic programs

e-Macao-16-2-211

What is an Object?

Real world objects are things that have:

1) state
2) behavior

Example: your dog:

1) state – name, color, breed, sits?, barks?, wages tail?, runs?
2) behavior – sitting, barking, waging tail, running

A software object is a bundle of variables (state) and methods (operations).

e-Macao-16-2-212

What is a Class?

A class is a blueprint that defines the variables and methods common to all objects of a certain kind.

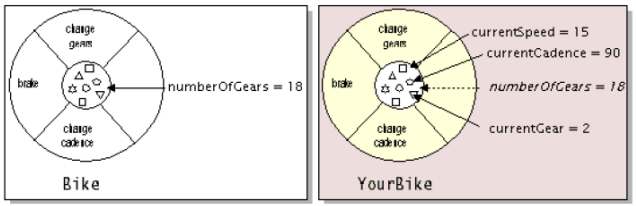
Example: 'your dog' is a object of the class Dog.

An object holds values for the variables defines in the class.

An object is called an instance of the Class

e-Macao-16-2-213

Objects versus Classes



Class **Instance of a Class**

1) operations: `changeGears`, `brake`, `changeCadence`
2) variables: `currentSpeed`, `currentCadence`, `currentGear`
3) static variable: `numberOfGears`
   It holds the same value for all objects of the class.

e-Macao-16-2-214

Object-Oriented Programming

Programming defined in terms:

1) objects (nouns) and
2) relationships between objects

Object-Oriented programming languages:

1) SmallTalk
2) C++
3) C#
4) Java

e-Macao-16-2-215

Object-Oriented Concepts

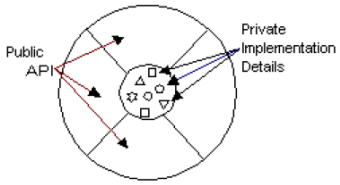
Three main concepts:

1) encapsulation
2) inheritance
3) polymorphism

e-Macao-16-2-216

Encapsulation

Illustration:



- Enclose data inside an object.
- Along with data, include operations on this data.
- Data cannot be accessed from outside except through the operations.
- Provides data security and facilitates code reuse.
- Operations provide the interface - internal state can change without affecting the user as long as the interface does not change.

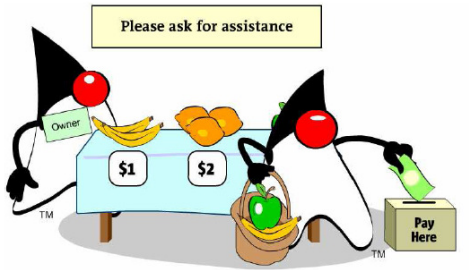
e-Macao-16-2-217

Encapsulation 1

The shopkeeper's fruit stand is designed to serve all customers. However, the design of the fruit stand does not prevent self-service. The fruit is freely accessible to all customers.

e-Macao-16-2-218

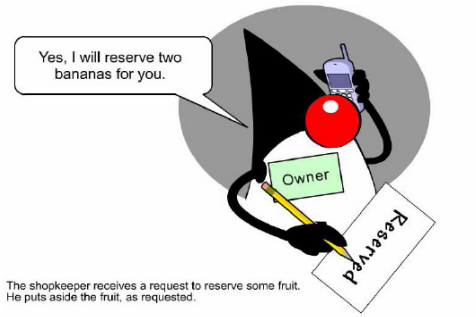
Encapsulation 2



A customer can select some fruit and make payment, all without the shopkeeper's involvement.

e-Macao-16-2-219

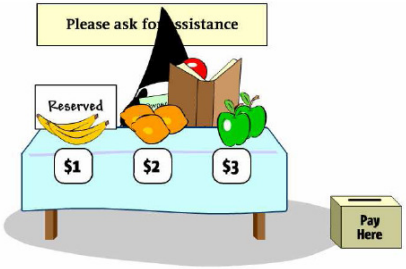
Encapsulation 3



The shopkeeper receives a request to reserve some fruit. He puts aside the fruit, as requested.

e-Macao-16-2-220

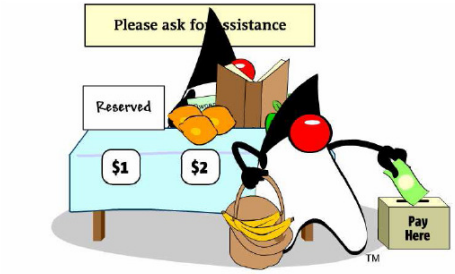
Encapsulation 4



However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

e-Macao-16-2-221

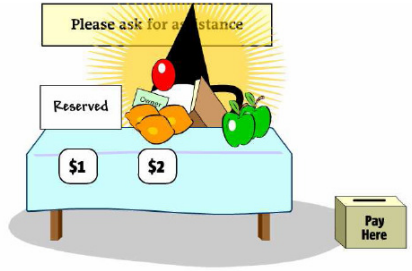
Encapsulation 5



However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

e-Macao-16-2-222

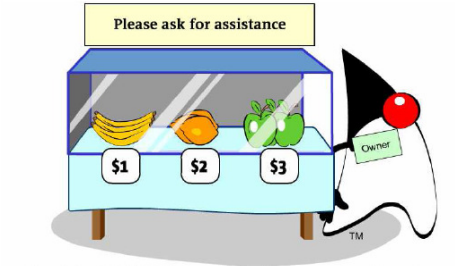
Encapsulation 6



However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

e-Macao-16-2-223

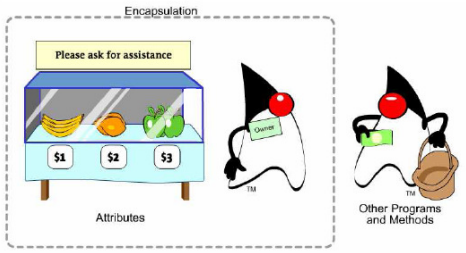
Encapsulation 7



By enclosing the fruit behind glass, and controlling all access by customers, the shopkeeper has encapsulated the process of buying fruit.

e-Macao-16-2-224

Encapsulation 8



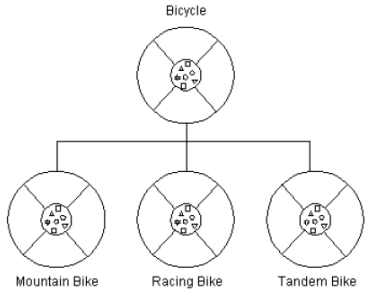
Encapsulation in the Java programming language is protection of the attributes from abritrary access by other programs and methods. You control how access to attributes is accomplished.

e-Macao-16-2-225

Inheritance

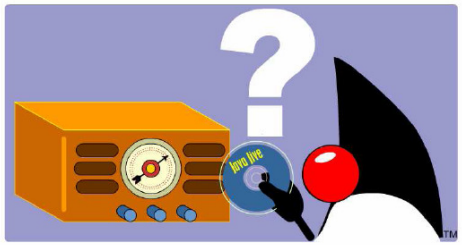
Features:

1) a class obtains variables anc methods from another class
2) the former is called sub-class, the latter super-class
3) a sub-class provides a specialized behavior with respect to its super-class
4) inheritance faciliates code reuse and avoids duplication of data



e-Macao-16-2-226

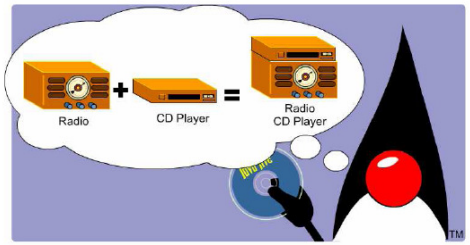
Inheritance 1



Duke would like to listen to his music CD, but his radio is not equipped with a CD player.

e-Macao-16-2-227

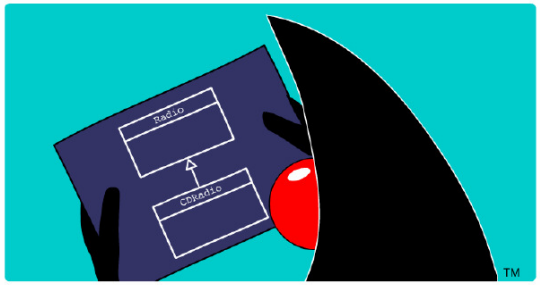
Inheritance 2



Duke decides he will create a specialized radio, one that plays CDs.

e-Macao-16-2-228

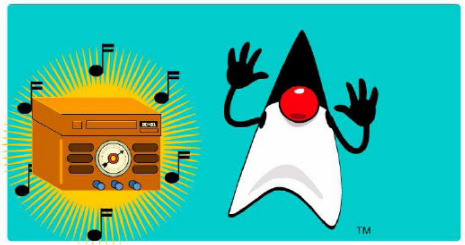
Inheritance 3



Duke creates a blueprint for a radio CD player.

e-Macao-16-2-229

Inheritance 4



Now Duke can enjoy listening to CDs on his radio CD player.

e-Macao-16-2-230

Polymorphism

Polymorphism = many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked.

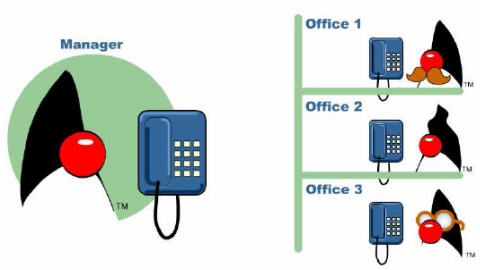
Polymorphism is enabled by inheritance:

1) a super-class defines an interface that all sub-classes must follow
2) it is up to the sub-classes how this interface is implemented; a sub-class may override methods of its super-class

Therefore, objects from the classes related by inheritance may receive the same requests but respond to such requests in their own ways.

e-Macao-16-2-231

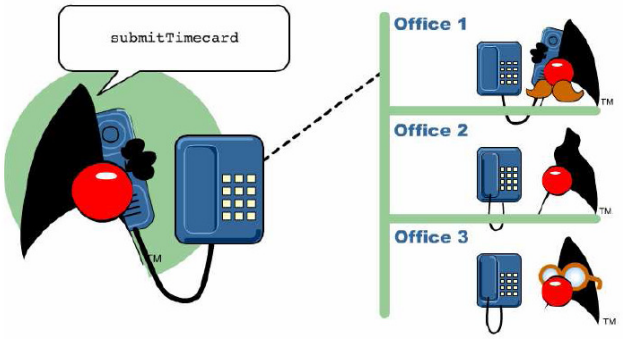
Polymorphism 1



To illustrate polymorphism, imagine a manager with a phone connected to several employee offices.

e-Macao-16-2-232

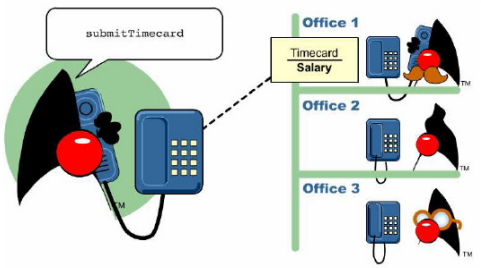
Polymorphism 2



The manager places a call to the first office, and sends the `submitTimecard` method.

e-Macao-16-2-233

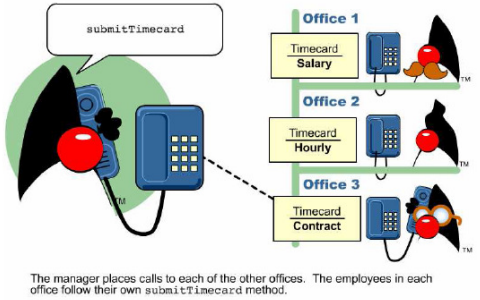
Polymorphism 3



The employee in the first office is on salary. Salaried employees have their own method for submitting a timecard, which this employee follows.

e-Macao-16-2-234

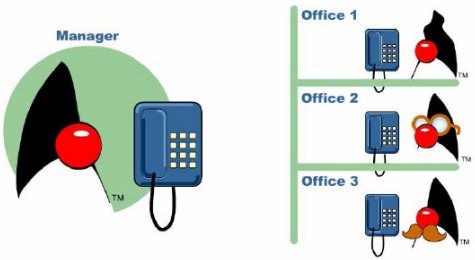
Polymorphism 4



The manager places calls to each of the other offices. The employees in each office follow their own `submitTimecard` method.

e-Macao-16-2-235

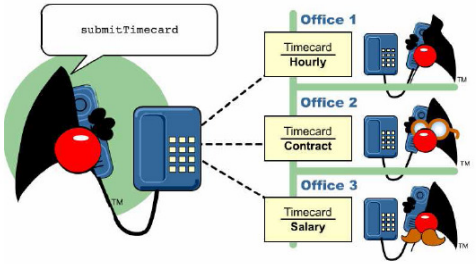
Polymorphism 5



During the next pay period, the employees have switched offices.

e-Macao-16-2-236

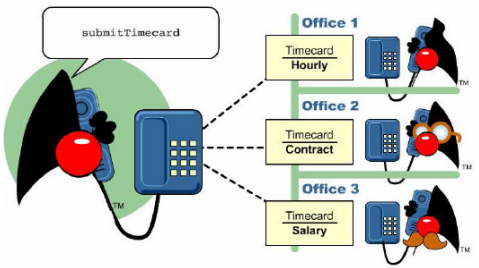
Polymorphism 6



The manager can still place the same call to all the offices, knowing that the employees will follow their own `submitTimcard` method.

e-Macao-16-2-237

Polymorphism 7



Polymorphism is the ability for objects from separate, yet related classes to receive the same message, but act on it in their own way.

e-Macao-16-2-238

Exercise: OOP Concepts

1) Why Object-oriented programming?
2) Explain the three main concepts of Object-orientation.
3) How does Object-orientation enables data security?
4) Explain how to achieve Polymorphism through Inheritance.

A.3.1. Objects

Objects

e-Macao-16-2-240

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-241

Objects

Everything in Java is an object.

Well ... almost.

Object lifecycle:

1) creation
2) usage
3) destruction

e-Macao-16-2-242

Object Creation

A variable s is declared to refer to the objects of type/class `String`:

```
String s;
```

The value of `s` is `null`; it does not yet refer to any object.

A new `String` object is created in memory with initial `"abc"` value:

```
String s = new String("abc");
```

Now `s` contains the address of this new object.

e-Macao-16-2-243

Object Usage

Objects are used mostly through variables.

Four usage scenarios:

1) one variable, one object
2) two variables, one object
3) two variables, two objects
4) one variable, two objects

e-Macao-16-2-244

One Variable, One Object

One variable, one object:

```
String s = new String("abc");
```

What can you do with the object addressed by `s`?

1) Check the length: `s.length() == 3`
2) Return the substring: `s.substring(2)`
3) etc.

Depending what is allowed by the definition of `String`.

e-Macao-16-2-245

Two Variables, One Object

Two variables, one object:

```
String s1 = new String("abc");
String s2;
```

Assignment copies the address, not value:

```
s2 = s1;
```

Now `s1` and `s2` both refer to one object. After

```
s1 = null;
```

`s2` still points to this object.

e-Macao-16-2-246

Two Variables, Two Objects

Two variables, two objects:

```
String s1 = new String("abc");
String s2 = new String("abc");
```

`s1` and `s2` objects have initially the same values:

```
s1.equals(s2) == true
```

But they are not the same objects:

```
(s1 == s2) == false
```

They can be changed independently of each other.

e-Macao-16-2-247

One Variable, Two Objects

One variable, two objects:

```
String s = new String("abc");
s = new String("cba");
```

The `"abc"` object is no longer accessible through any variable.

e-Macao-16-2-248

Object Destruction

A program accumulates memory through its execution.

Two mechanism to free memory that is no longer need by the program:

1) manual – done in C/C++
2) automatic – done in Java

In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.

Garbage collector is parts of the Java Run-Time Environment.

e-Macao-16-2-249

Exercise: Objects

1) Explain in detail the lifecycle of an object, making reference to it's
   a) creation
   b) usage and
   c) destruction
2) What is garbage collection?

A.3.2. Classes

Classes

e-Macao-16-2-251

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-252

Class Outline

1) class definition
   a) attributes
   b) methods
   c) constructors
2) method overloading
3) method communication
   a) passing arguments
   b) returning results
4) recursive methods
5) arrays as object
6) static components
7) nested and internal classes

e-Macao-16-2-253

Class

A basis for the Java language.

Each concept we wish to describe in Java must be included inside a class.

A class defines a new data type, whose values are objects:

1) a class is a template for objects
2) an object is an instance of a class

e-Macao-16-2-254

Class Definition

A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.

General form of a class:

```
class classname {
  type instance-variable-1;
  …
  type instance-variable-n;

  type method-name-1(parameter-list) { … }
  type method-name-2(parameter-list) { … }
  …
  type method-name-m(parameter-list) { … }
}
```

e-Macao-16-2-255

Example: Class

A class with three variable members:

```
class Box {
  double width;
  double height;
  double depth;
}
```

A new `Box` object is created and a new value assigned to its width variable:

```
Box myBox = new Box();
myBox.width = 100;
```

e-Macao-16-2-256

Example: Class Usage

```
class BoxDemo {
  public static void main(String args[]) {
     Box mybox = new Box();
     double vol;

     mybox.width = 10;
     mybox.height = 20;
     mybox.depth = 15;

     vol = mybox.width * mybox.height * mybox.depth;
     System.out.println("Volume is " + vol);
  }
}
```

e-Macao-16-2-257

Compilation and Execution

Place the `Box` class definitions in file `Box.java`:

```
class Box { … }
```

Place the `BoxDemo` class definitions in file `BoxDemo.java`:

```
class BoxDemo {
  public static void main(…) { … }
}
```

Compilation and execution:

```
> javac BoxDemo.java
> java BoxDemo
```

e-Macao-16-2-258

Variable Independence 1

Each object has its own copy of the instance variables: changing the variables of one object has no effect on the variables of another object.

Consider this example:

```
class BoxDemo2 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;

     mybox1.width = 10;
     mybox1.height = 20;
     mybox1.depth = 15;
```

e-Macao-16-2-259

Variable Independence 2

```
     mybox2.width = 3;
     mybox2.height = 6;
     mybox2.depth = 9;

     vol = mybox1.width * mybox1.height * mybox1.depth;
     System.out.println("Volume is " + vol);

     vol = mybox2.width * mybox2.height * mybox2.depth;
     System.out.println("Volume is " + vol);
  }
}
```

What are the printed volumes of both boxes?

e-Macao-16-2-260

Declaring Objects

Obtaining objects of a class is a two-stage process:

1) Declare a variable of the class type:

```
Box myBox;
```

The value of `myBox` is a reference to an object, if one exists, or `null`. At this moment, the value of `myBox` is `null`.

2) Acquire an actual, physical copy of an object and assign its address to the variable. How to do this?

e-Macao-16-2-261

Operator new

Allocates memory for a `Box` object and returns its address:

```
Box myBox = new Box();
```

The address is then stored in the `myBox` reference variable.

`Box()` is a class constructor - a class may declare its own constructor or rely on the default constructor provided by the Java environment.

e-Macao-16-2-262

Memory Allocation

Memory is allocated for objects dynamically.

This has both advantages and disadvantages:

1) as many objects are created as needed
2) allocation is uncertain – memory may be insufficient

Variables of simple types do not require `new`:

```
int n = 1;
```

In the interest of efficiency, Java does not implement simple types as objects. Variables of simple types hold values, not references.

e-Macao-16-2-263

Assigning Reference Variables

Assignment copies address, not the actual value:

```
Box b1 = new Box();
Box b2 = b1;
```

Both variables point to the same object.

Variables are not in any way connected. After

```
b1 = null;
```

`b2` still refers to the original object.

e-Macao-16-2-264

Methods

General form of a method definition:

```
type name(parameter-list) {
  … return value; …
}
```

Components:

1) `type` - type of values returned by the method. If a method does not return any value, its return type must be `void`.
2) `name` is the name of the method
3) `parameter-list` is a sequence of type-identifier lists separated by commas
4) `return value` indicates what value is returned by the method.

e-Macao-16-2-265

Example: Method 1

Classes declare methods to hide their internal data structures, as well as for their own internal use:

Within a class, we can refer directly to its member variables:

```
class Box {
  double width, height, depth;
  void volume() {
     System.out.print("Volume is ");
     System.out.println(width * height * depth);
  }
}
```

e-Macao-16-2-266

Example: Method 2

When an instance variable is accessed by code that is not part of the class in which that variable is defined, access must be done through an object:

```
class BoxDemo3 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     mybox1.width = 10;    mybox2.width = 3;
     mybox1.height = 20;   mybox2.height = 6;
     mybox1.depth = 15;    mybox2.depth = 9;

     mybox1.volume();
     mybox2.volume();
  }
}
```

e-Macao-16-2-267

Value-Returning Method 1

The type of an expression returning value from a method must agree with the return type of this method:

```
class Box {
  double width;
  double height;
  double depth;

  double volume() {
     return width * height * depth;
  }
}
```

e-Macao-16-2-268

Value-Returning Method 2

```
class BoxDemo4 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;
     mybox1.width = 10;
     mybox2.width = 3;
     mybox1.height = 20;
     mybox2.height = 6;
     mybox1.depth = 15;
     mybox2.depth = 9;
```

e-Macao-16-2-269

Value-Returning Method 3

The type of a variable assigned the value returned by a method must agree with the return type of this method:

```
     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

e-Macao-16-2-270

Parameterized Method

Parameters increase generality and applicability of a method:

1) method without parameters

```
int square() { return 10*10; }
```

2) method with parameters

```
int square(int i) { return i*i; }
```

Parameter: a variable receiving value at the time the method is invoked.

Argument: a value passed to the method when it is invoked.

e-Macao-16-2-271

Example: Parameterized Method 1

```
class Box {
  double width;
  double height;
  double depth;

  double volume() {
     return width * height * depth;
  }

  void setDim(double w, double h, double d) {
     width = w; height = h; depth = d;
  }
}
```

e-Macao-16-2-272

Example: Parameterized Method 2

```
class BoxDemo5 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;

     mybox1.setDim(10, 20, 15);
     mybox2.setDim(3, 6, 9);

     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

e-Macao-16-2-273

Constructor

A constructor initializes the instance variables of an object.

It is called immediately after the object is created but before the `new` operator completes.

1) it is syntactically similar to a method:
2) it has the same name as the name of its class
3) it is written without return type; the default return type of a class constructor is the same class

When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

e-Macao-16-2-274

Example: Constructor 1

```
class Box {
  double width;
  double height;
  double depth;

  Box() {
     System.out.println("Constructing Box");
     width = 10; height = 10; depth = 10;
  }

  double volume() {
     return width * height * depth;
  }
}
```

e-Macao-16-2-275

Example: Constructor 2

```
class BoxDemo6 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

e-Macao-16-2-276

Parameterized Constructor 1

So far, all boxes have the same dimensions.

We need a constructor able to create boxes with different dimensions:

```
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
     width = w; height = h; depth = d;
  }

  double volume() { return width * height * depth; }
}
```

e-Macao-16-2-277

Parameterized Constructor 2

```
class BoxDemo7 {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(3, 6, 9);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

e-Macao-16-2-278

finalize() Method

A constructor helps to initialize an object just after it has been created.

In contrast, the `finalize` method is invoked just before the object is destroyed:

1) implemented inside a class as:

   ```
   protected void finalize() { … }
   ```

2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out

How is the `finalize` method invoked?

e-Macao-16-2-279

Garbage Collection

Garbage collection is a mechanism to remove objects from memory when they are no longer needed.

Garbage collection is carried out by the garbage collector:

1) The garbage collector keeps track of how many references an object has.
2) It removes an object from memory when it has no longer any references.
3) Thereafter, the memory occupied by the object can be allocated again.
4) The garbage collector invokes the `finalize` method.

e-Macao-16-2-280

Keyword this

Keyword `this` allows a method to refer to the object that invoked it.

It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
   this.width = width;
   this.height = height;
   this.depth = depth;
}
```

The above use of `this` is redundant but correct.

When is `this` really needed?

e-Macao-16-2-281

Instance Variable Hiding

Variables with the same names:

1) it is illegal to declare two local variables with the same name inside the same or enclosing scopes
2) it is legal to declare local variables or parameters with the same name as the instance variables of the class.

As the same-named local variables/parameters will hide the instance variables, using `this` is necessary to regain access to them:

```
Box(double width, double height, double depth) {
   this.width = width;
   this.height = height;
   this.depth = depth;
}
```

e-Macao-16-2-282

Example: Stack 1

A stack hold data in the first-in-last-out order.

Here is the implementation of a 10-element stack:

```
class Stack {
  int stck[] = new int[10];
  int tos;

  Stack() {
     tos = -1;
  }
```

e-Macao-16-2-283

Example: Stack 2

```
  void push(int item) {
     if (tos==9) System.out.println("Stack is full.");
     else stck[++tos] = item;
  }

  int pop() {
     if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else return stck[tos--];
  }
}
```

e-Macao-16-2-284

Example: Stack 3

```
class TestStack {
  public static void main(String args[]) {
     Stack mystack1 = new Stack();
     Stack mystack2 = new Stack();
     for (int i=0; i<10; i++) mystack1.push(i);
     for (int i=10; i<20; i++) mystack2.push(i);

     System.out.println("Stack in mystack1:");
     for(int i=0; i<10; i++)
        System.out.println(mystack1.pop());
     System.out.println("Stack in mystack2:");
     for(int i=0; i<10; i++)
        System.out.println(mystack2.pop());
  }
}
```

e-Macao-16-2-285

Method Overloading

It is legal for a class to have two or more methods with the same name.

However, Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.

Therefore the same-named methods must be distinguished:

1) by the number of arguments, or
2) by the types of arguments

Overloading and inheritance are two ways to implement polymorphism.

e-Macao-16-2-286

Example: Overloading 1

```
class OverloadDemo {
  void test() {
     System.out.println("No parameters");
  }
  void test(int a) {
     System.out.println("a: " + a);
  }
  void test(int a, int b) {
     System.out.println("a and b: " + a + " " + b);
  }
  double test(double a) {
     System.out.println("double a: " + a); return a*a;
  }
}
```

e-Macao-16-2-287

Example: Overloading 2

```
class Overload {
  public static void main(String args[]) {
     OverloadDemo ob = new OverloadDemo();
     double result;
     ob.test();
     ob.test(10);
     ob.test(10, 20);
     result = ob.test(123.2);
     System.out.println("ob.test(123.2): " + result);
  }
}
```

e-Macao-16-2-288

Different Result Types

Different result types are insufficient.

The following will not compile:

```
double test(double a) {
  System.out.println("double a: " + a);
  return a*a;
}

int test(double a) {
  System.out.println("double a: " + a);
  return (int) a*a;
}
```

e-Macao-16-2-289

Overloading and Conversion 1

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

When no exact match can be found, Java's automatic type conversion can aid overload resolution:

```
class OverloadDemo {
  void test() {
     System.out.println("No parameters");
  }
  void test(int a, int b) {
     System.out.println("a and b: " + a + " " + b);
  }
```

e-Macao-16-2-290

Overloading and Conversion 2

```
  void test(double a) {
     System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
     OverloadDemo ob = new OverloadDemo();
     int i = 88;
     ob.test();
     ob.test(10, 20);
     ob.test(i);
     ob.test(123.2);
  }
}
```

e-Macao-16-2-291

Overloading and Polymorphism

In the languages without overloading, methods must have a unique names:

```
int abs(int i)
long labs(int i)
float fabs(int i)
```

Java enables logically-related methods to occur under the same name:

```
static int abs(int i)
static long abs(long i)
static float abs(float i)
```

e-Macao-16-2-292

Constructor Overloading

Why overload constructors? Consider this:

```
class Box {
  double width, height, depth;

  Box(double w, double h, double d) {
     width = w; height = h; depth = d;
  }
  double volume() {
     return width * height * depth;
  }
}
```

All `Box` objects can be created in one way: passing all three dimensions.

e-Macao-16-2-293

Example: Overloading 1

Three constructors: 3-parameter, 1-parameter, parameter-less.

```
class Box {
  double width, height, depth;
  Box(double w, double h, double d) {
     width = w; height = h; depth = d;
  }
  Box() {
     width = -1;  height = -1;  depth = -1;
  }
  Box(double len) {
     width = height = depth = len;
  }
  double volume() { return width * height * depth; }
}
```

e-Macao-16-2-294

Example: Overloading 2

```
class OverloadCons {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box();
     Box mycube = new Box(7);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     vol = mycube.volume();
     System.out.println("Volume of mycube is " + vol);
  }
}
```

e-Macao-16-2-295

Object Argument 1

So far, all method received arguments of simple types.

They may also receive an object as an argument. Here is a method to check if a parameter object is equal to the invoking object:

```
class Test {
  int a, b;
  Test(int i, int j) {
     a = i; b = j;
  }
  boolean equals(Test o) {
     if (o.a == a && o.b == b) return true;
     else return false;
  }
}
```

e-Macao-16-2-296

Object Argument 2

```
class PassOb {
  public static void main(String args[]) {
     Test ob1 = new Test(100, 22);
     Test ob2 = new Test(100, 22);
     Test ob3 = new Test(-1, -1);
     System.out.println("ob1==ob2: " + ob1.equals(ob2));
     System.out.println("ob1==ob3: " + ob1.equals(ob3));
  }
}
```

e-Macao-16-2-297

Passing Object to Constructor 1

A special case of object-passing is passing an object to the constructor.

This is to initialize one object with another object:

```
class Box {
  double width, height, depth;

  Box(Box ob) {
     width = ob.width;
     height = ob.height;
     depth = ob.depth;
  }
```

e-Macao-16-2-298

Passing Object to Constructor 2

```
  Box(double w, double h, double d) {
     width = w;
     height = h;
     depth = d;
  }

  double volume() {
     return width * height * depth;
  }
}
```

e-Macao-16-2-299

Passing Object to Constructor 3

```
class OverloadCons2 {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(mybox1);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
  }
}
```

e-Macao-16-2-300

Argument-Passing

Two types of variables:

1) simple types
2) class types

Two corresponding ways of how the arguments are passed to methods:

1) by value – a method receives a cope of the original value; parameters of simple types
2) by reference – a method receives the memory address of the original value, not the value itself; parameters of class types

e-Macao-16-2-301

Simple Type Argument-Passing 1

Passing arguments of simple types takes place by value:

```
class Test {
  void meth(int i, int j) {
     i *= 2;
     j /= 2;
  }
}
```

e-Macao-16-2-302

Simple Type Argument-Passing 2

With by-value argument-passing what occurs to the parameter that receives the argument has no effect outside the method:

```
class CallByValue {
  public static void main(String args[]) {
     Test ob = new Test();
     int a = 15, b = 20;
     System.out.print("a and b before call: ");
     System.out.println(a + " " + b);
     ob.meth(a, b);
     System.out.print("a and b after call: ");
     System.out.println(a + " " + b);
  }
}
```

e-Macao-16-2-303

Class Type Argument-Passing 1

Objects are passed to the methods by reference: a parameter obtains the same address as the corresponding argument:

```
class Test {
  int a, b;

  Test(int i, int j) {
     a = i; b = j;
  }

  void meth(Test o) {
     o.a *=  2; o.b /= 2;
  }
}
```

e-Macao-16-2-304

Class Type Argument-Passing 2

As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:

```
class CallByRef {
  public static void main(String args[]) {
     Test ob = new Test(15, 20);
     System.out.print("ob.a and ob.b before call: ");
     System.out.println(ob.a + " " + ob.b);
     ob.meth(ob);
     System.out.print("ob.a and ob.b after call: ");
     System.out.println(ob.a + " " + ob.b);
  }
}
```

e-Macao-16-2-305

Returning Objects 1

So far, all methods returned no values or values of simple types.

Methods may also return objects:

```
class Test {
  int a;
  Test(int i) {
     a = i;
  }
  Test incrByTen() {
     Test temp = new Test(a+10);
     return temp;
  }
}
```

e-Macao-16-2-306

Returning Objects 2

Each time a method `incrByTen` is invoked a new object is created and a reference to it is returned:

```
class RetOb {
  public static void main(String args[]) {
     Test ob1 = new Test(2);
     Test ob2;
     ob2 = ob1.incrByTen();
     System.out.println("ob1.a: " + ob1.a);
     System.out.println("ob2.a: " + ob2.a);
     ob2 = ob2.incrByTen();
     System.out.print("ob2.a after second increase: ");
     System.out.println(ob2.a);
  }
}
```

e-Macao-16-2-307

Recursion

A recursive method is a method that calls itself:

What happens then?

1) all method parameters and local variables are allocated on the stack
2) arguments are prepared in the corresponding parameter positions
3) the method code is executed for the new arguments
4) upon return, all parameters and variables are removed from the stack
5) the execution continues immediately after the invocation point

e-Macao-16-2-308

Example: Recursion

```
class Factorial {
  int fact(int n) {
     if (n==1) return 1;
     return fact(n-1) * n;
  }
}

class Recursion {
  public static void main(String args[]) {
     Factorial f = new Factorial();
     System.out.print("Factorial of 5 is ");
     System.out.println(f.fact(5));
  }
}
```

e-Macao-16-2-309

Example: Recursion and Arrays 1

A method that recursively prints out a given number of elements in a table:

```
class RecTest {
  int values[];

  RecTest(int i) {
     values = new int[i];
  }
  void printArray(int i) {
     if (i==0) return;
     else printArray(i-1);
     System.out.print("[" + (i-1) + "] ");
     System.out.println(values[i-1]);
  }
}
```

e-Macao-16-2-310

Example: Recursion and Arrays 2

```
class Recursion2 {
  public static void main(String args[]) {
     RecTest ob = new RecTest(10);
     int i;
     for(i=0; i<10; i++)
        ob.values[i] = i;
     ob.printArray(10);
  }
}
```

e-Macao-16-2-311

Arrays Revisited

All arrays are implemented as objects.

In particular, all arrays have the `length` attribute to return the number of elements that an array may hold:

```
class Length {
  public static void main(String args[]) {
     int a1[] = new int[10];
     int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
     int a3[] = {4, 3, 2, 1};
     System.out.println("length of a1 is " + a1.length);
     System.out.println("length of a2 is " + a2.length);
     System.out.println("length of a3 is " + a3.length);
  }
}
```

e-Macao-16-2-312

Example: Arrays as Objects 1

An improved `Stack` example, able to handle stacks of any size:

```
class Stack {
  private int stck[];
  private int tos;

  Stack(int size) {
     stck = new int[size]; tos = -1;
  }

  void push(int item) {
     if (tos==stck.length-1)
        System.out.println("Stack is full.");
     else stck[++tos] = item;
  }
```

e-Macao-16-2-313

Example: Arrays as Objects 2

```
  int pop() {
     if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else
        return stck[tos--];
  }
}
```

e-Macao-16-2-314

Example: Arrays as Objects 3

```
class TestStack2 {
  public static void main(String args[]) {
     Stack mystack1 = new Stack(5);
     Stack mystack2 = new Stack(8);
     for (int i=0; i<5; i++) mystack1.push(i);
     for (int i=0; i<8; i++) mystack2.push(i);

     System.out.println("Stack in mystack1:");
     for (int i=0; i<5; i++)
        System.out.println(mystack1.pop());
     System.out.println("Stack in mystack2:");
     for (int i=0; i<8; i++)
        System.out.println(mystack2.pop());
  }
}
```

e-Macao-16-2-315

Static Class Members

Normally, the members of a class (its variables and methods) may be only used through the objects of this class.

Static members are independent of the objects:

1) variables
2) methods
3) initialization block

All declared with the `static` keyword.

e-Macao-16-2-316

Static Variables

Static variable:

```
static int a;
```

Essentially, it a global variable shared by all instances of the class.

It cannot be used within a non-static method.

e-Macao-16-2-317

Static Methods

Static method:

```
static void meth() { … }
```

Several restrictions apply:

- can only call static methods
- must only access static variables
- cannot refer to `this` or `super` in any way

e-Macao-16-2-318

Static Block

Static block:

```
static { … }
```

This is where the static variables are initialized.

The block is executed exactly once, when the class is first loaded.

e-Macao-16-2-319

Example: Static 1

```
class UseStatic {
  static int a = 3;
  static int b;
  static void meth(int x) {
     System.out.print("x = " + x + " a = " + a);
     System.out.println(" b = " + b);
  }
  static {
     System.out.println("Static block initialized.");
     b = a * 4;
  }
  public static void main(String args[]) {
     meth(42);
  }
}
```

e-Macao-16-2-320

Static Member Usage 1

How to use static members outside their class?

Consider this class:

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void callme() {
     System.out.println("a = " + a);
  }
}
```

e-Macao-16-2-321

Static Member Usage 2

Static variables/method are used through the class name:

```
StaticDemo.a
StaticDemo.callme()
```

Example:

```
class StaticByName {
  public static void main(String args[]) {
     StaticDemo.callme();
     System.out.println("b = " + StaticDemo.b);
  }
}
```

e-Macao-16-2-322

Nested Classes

It is possible to define a class within a class – nested class.

The scope of the nested class is its enclosing class: if class `B` is defined within class `A` then `B` is known to `A` but not outside.

Access rights:

1) a nested class has access to all members of its enclosing class, including its private members
2) the enclosing class does not have access to the members of the nested class

e-Macao-16-2-323

Types of Nested Classes

There are two types of nested classes:

1) static – cannot access the members of its enclosing class directly, but through an object; defined with the `static` keyword
2) non-static – has direct access to all members of the enclosing class in the same way as other non-static member of this class so

A static nested class is seldom used.

A non-static nested class is also called an inner class.

e-Macao-16-2-324

Example: Inner Classes 1

`Outer` has a variable `out_x`, an inner class `Inner` and a method `test` which creates an object of the `Inner` class and calls its `display` method:

```
class Outer {
  int outer_x = 100;
  void test() {
     Inner inner = new Inner();
     inner.display();
  }
  class Inner {
     void display() {
        System.out.println("outer_x = " + outer_x);
     }
  }
}
```

e-Macao-16-2-325

Example: Inner Classes 2

A demonstration class to create an object of the `Outer` class and invoke the `test` method on this object:

```
class InnerClassDemo {
  public static void main(String args[]) {
     Outer outer = new Outer();
     outer.test();
  }
}
```

The `Inner` class is only known within the `Outer` class. Any reference to `Inner` outside `Outer` will create a compile-time error.

e-Macao-16-2-326

Inner Members Visibility 1

Inner class has access to all member of the outer class.

The reverse is not true: members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

This is the `Outer` class with a variable, two methods and `Inner` class.

The first method refers to the `Inner` class correctly through an object:

```
class Outer {
  int outer_x = 100;
  void test() {
     Inner inner = new Inner();
     inner.display();
  }
```

e-Macao-16-2-327

Inner Members Visibility 2

`Inner` class declares variable `y` and refers to the `Outer` class variable:

```
  class Inner {
     int y = 10;
     void display() {
        System.out.println("outer_x = " + outer_x);
     }
  }
```

`showy` method refers incorrectly to the `Inner` class's `y` variable:

```
 void showy() {
    System.out.println(y);
  }
}
```

e-Macao-16-2-328

Inner Members Visibility 3

As a result, this program will not compile:

```
class InnerClassDemo {
  public static void main(String args[]) {
     Outer outer = new Outer();
     outer.test();
  }
}
```

e-Macao-16-2-329

Inner Class Declaration

So far, all inner classes were defined within the outer class scope.

In fact, an inner class may be defined within any block scope.

The following is an example of an inner class define within a for loop.

e-Macao-16-2-330

Example: Inner Class Declaration 1

```
class Outer {
  int outer_x = 100;
  void test() {
     for (int i=0; i<10; i++) {
        class Inner {
           void display() {
              System.out.println("outer_x= " + outer_x);
           }
        }
        Inner inner = new Inner();
        inner.display();
     }
  }
}
```

e-Macao-16-2-331

Example: Inner Class Declaration 2

A demonstration creates an `Outer` object and invokes a `test` method on it:

```
class InnerClassDemo {
  public static void main(String args[]) {
     Outer outer = new Outer();
     outer.test();
  }
}
```

e-Macao-16-2-332

Exercise: Classes 1

1) What's the difference between an object and a class?
2) Explain why constructors don't have return types.
3) Can *this* keyword be used within the context of a static method?. Why?.
4) What's Overloading?
5) Explain the mechanism of Argument passing in Java. Is it by value or by reference?
6) Explain the mechanism that Java uses to remove objects from memory when they are no longer needed.
7) An object has a handle to some non-Java resources such as file or window character font. How would you ensure that these resources are freed before the object is destroyed?

e-Macao-16-2-333

Exercise: Classes 2

1) Create a class with a default constructor (no arguments) that prints a message. Create an object of this class.
2) Add an overloaded constructor to the class in 1 which takes a String argument and prints it along with your message.
3) Make a two dimensional array of objects of the class you created in 2.
4) Create a class Counter with an adequate data member. Write four methods, one to get the value of the counter, one to increment the counter, one to decrement the counter, and one to reset the counter to zero. Make sure that the value of the counter never gets less than zero.
5) Create an object of your class Counter and assign it to two different variables. Change the state of the object, calling your method on one variable. See what happens if you call the get-value-method on the other variable.

A.3.3. Inheritance

Inheritance

e-Macao-16-2-335

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-336

Inheritance

One of the pillars of object-orientation.

A new class is derived from an existing class:

1) existing class is called super-class
2) derived class is called sub-class

A sub-class is a specialized version of its super-class:

1) has all non-private members of its super-class
2) may provide its own implementation of super-class methods

Objects of a sub-class are a special kind of objects of a super-class.

e-Macao-16-2-337

Inheritance Syntax

Syntax:

```
class sub-class extends super-class {
  …
}
```

Each class has at most one super-class; no multi-inheritance in Java.

No class is a sub-class of itself.

e-Macao-16-2-338

Example: Super-Class

```
class A {
  int i;

  void showi() {
     System.out.println("i: " + i);
  }

}
```

e-Macao-16-2-339

Example: Sub-Class

```
class B extends A {

  int j;

  void showj() {
     System.out.println("j: " + j);
  }

  void sum() {
     System.out.println("i+j: " + (i+j));
  }

}
```

e-Macao-16-2-340

Example: Testing Class

```
class SimpleInheritance {
  public static void main(String args[]) {
     A a = new A();
     B b = new B();
     a.i = 10;
     System.out.println("Contents of a: ");
     a.showi();
     b.i = 7; b.j = 8;
     System.out.println("Contents of b: ");
     subOb.showi(); subOb.showj();
     System.out.println("Sum of I and j in b:");
     b.sum();
  }
}
```

e-Macao-16-2-341

Inheritance and Private Members 1

A class may declare some of its members private.

A sub-class has no access to the private members of its super-class:

```
class A {
  int i;
  private int j;
  void setij(int x, int y) {
     i = x; j = y;
  }
}
```

e-Macao-16-2-342

Inheritance and Private Members 2

Class `B` has no access to the `A`'s private variable `j`.

This program will not compile:

```
class B extends A {
  int total;
  void sum() {
     total = i + j;
  }
}
```

e-Macao-16-2-343

Example: Box Class

The basic `Box` class with width, height and depth:

```
class Box {
  double width, height, depth;
  Box(double w, double h, double d) {
     width = w; height = h; depth = d;
  }
  Box(Box b) {
     width = b.width;
     height = b.height; depth = b.depth;
  }
  double volume() {
     return width * height * depth;
  }
}
```

e-Macao-16-2-344

Example: BoxWeight Sub-Class

`BoxWeight` class extends `Box` with the new `weight` variable:

```
class BoxWeight extends Box {
  double weight;
  BoxWeight(double w, double h, double d, double m) {
     width = w; height = h; depth = d; weight = m;
  }
  BoxWeight(Box b, double w) {
     Box(b); weight = w;
  }
}
```

`Box` is a super-class, `BoxWeight` is a sub-class.

e-Macao-16-2-345

Example: BoxWeight Demo

```
class DemoBoxWeight {
  public static void main(String args[]) {
     BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
     BoxWeight mybox2 = new BoxWeight(mybox1);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     System.out.print("Weight of mybox1 is ");
     System.out.println(mybox1.weight);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     System.out.print("Weight of mybox2 is ");
     System.out.println(mybox2.weight);
  }
}
```

e-Macao-16-2-346

Another Sub-Class

Once a super-class exists that defines the attributes common to a set of objects, it can be used to create any number of more specific sub-classes.

The following sub-class of `Box` adds the `color` attribute instead of `weight`:

```
class ColorBox extends Box {
  int color;

  ColorBox(double w, double h, double d, int c) {
     width = w; height = h; depth = d;
     color = c;
  }
}
```

e-Macao-16-2-347

Referencing Sub-Class Objects

A variable of a super-class type may refer to any of its sub-class objects:

```
class SuperClass { … }
class SubClass extends SuperClass { … }

SuperClass o1;
SubClass o2 =  new SubClass();

o1 = o2;
```

However, the inverse is illegal:

```
o2 = o1
```

e-Macao-16-2-348

Example: Sub-Class Objects 1

```
class RefDemo {
  public static void main(String args[]) {
     BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
     Box plainbox = new Box(5, 5, 5);
     double vol;
     vol = weightbox.volume();
     System.out.print("Volume of weightbox is ");
     System.out.println(vol);
     System.out.print("Weight of weightbox is ");
     System.out.println(weightbox.weight);
     plainbox = weightbox;
     vol = plainbox.volume();
     System.out.println("Volume of plainbox is " + vol);
  }
}
```

e-Macao-16-2-349

Super-Class Variable Access

`plainbox` variable now refers to the `WeightBox` object.

Can we then access this object's `weight` variable through `plainbox`?

No. The type of a variable, not the object this variable refers to, determines which members we can access!

This is illegal:

```
System.out.print("Weight of plainbox is ");
System.out.println(plainbox.weight);
```

e-Macao-16-2-350

Super as a Constructor

Calling a constructor of a super-class from the constructor of a sub-class:

```
super(parameter-list);
```

Must occur as the very first instructor in the sub-class constructor:

```
class SuperClass { … }

class SubClass extends SuperClass {
  SubClass(…) {
     super(…);
     …
  }
  …
}
```

e-Macao-16-2-351

Example: Super Constructor 1

`BoxWeight` need not initialize the variable for the `Box` super-class, only the added `weight` variable:

```
class BoxWeight extends Box {
  double weight;

  BoxWeight(double w, double h, double d, double m) {
     super(w, h, d); weight = m;
  }

  BoxWeight(Box b, double w) {
     super(b); weight = w;
  }
}
```

e-Macao-16-2-352

Example: Super Constructor 2

```
class DemoSuper {
  public static void main(String args[]) {
     BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
     BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     System.out.print("Weight of mybox1 is ");
     System.out.println(mybox1.weight);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     System.out.print("Weight of mybox2 is ");
     System.out.println(mybox2.weight);
  }
}
```

e-Macao-16-2-353

Referencing Sub-Class Objects

Sending a sub-class object:

```
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
```

to the constructor expecting a super-class object:

```
BoxWeight(Box b, double w) {
   super(b); weight = w;
}
```

e-Macao-16-2-354

Uses of Super

Two uses of `super`:

1) to invoke the super-class constructor

```
super();
```

2) to access super-class members

```
super.variable;
super.method(…);
```

(1) was discussed, consider (2).

e-Macao-16-2-355

Super and Hiding

Why is `super` needed to access super-class members?

When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {
  int i = 1;
}

class B extends A {
  int i = 2;
  System.out.println("i is " + i);
}
```

The re-declared variables/methods hide those of the super-class.

e-Macao-16-2-356

Example: Super and Hiding 1

```
class A {
  int i;
}

class B extends A {
  int i;

  B(int a, int b) {
     super.i = a; i = b;
  }
  void show() {
     System.out.println("i in superclass: " + super.i);
     System.out.println("i in subclass: " + i);
  }
}
```

e-Macao-16-2-357

Example: Super and Hiding 2

Although the `i` variable in `B` hides the `i` variable in `A`, `super` allows access to the hidden variable of the super-class:

```
class UseSuper {
  public static void main(String args[]) {
     B subOb = new B(1, 2);
     subOb.show();
  }
}
```

e-Macao-16-2-358

Multi-Level Class Hierarchy 1

The basic `Box` class:

```
class Box {
  private double width, height, depth;
  Box(double w, double h, double d) {
     width = w; height = h; depth = d;
  }
  Box(Box ob) {
     width = ob.width;
     height = ob.height; depth = ob.depth;
  }
  double volume() {
     return width * height * depth;
  }
}
```

e-Macao-16-2-359

Multi-Level Class Hierarchy 2

Adding the `weight` variable to the `Box` class:

```
class BoxWeight extends Box {
  double weight;

  BoxWeight(BoxWeight ob) {
     super(ob); weight = ob.weight;
  }

  BoxWeight(double w, double h, double d, double m) {
     super(w, h, d); weight = m;
  }

}
```

e-Macao-16-2-360

Multi-Level Class Hierarchy 3

Adding the `cost` variable to the `BoxWeight` class:

```
class Ship extends BoxWeight {
  double cost;

  Ship(Ship ob) {
     super(ob); cost = ob.cost;
  }

  Ship(double w, double h,
     double d, double m, double c) {
     super(w, h, d, m); cost = c;
  }
}
```

e-Macao-16-2-361

Multi-Level Class Hierarchy 4

```
class DemoShip {
  public static void main(String args[]) {
     Ship ship1 = new Ship(10, 20, 15, 10, 3.41);
     Ship ship2 = new Ship(2, 3, 4, 0.76, 1.28);
     double vol;

     vol = ship1.volume();
     System.out.println("Volume of ship1 is " + vol);
     System.out.print("Weight of ship1 is”);
     System.out.println(ship1.weight);
     System.out.print("Shipping cost: $");
     System.out.println(ship1.cost);
```

e-Macao-16-2-362

Multi-Level Class Hierarchy 5

```
     vol = ship2.volume();
     System.out.println("Volume of ship2 is " + vol);
     System.out.print("Weight of ship2 is ");
     System.out.println(ship2.weight);
     System.out.print("Shipping cost: $");
     System.out.println(ship2.cost);
  }
}
```

e-Macao-16-2-363

Constructor Call-Order

Constructor call-order:

1) first call super-class constructors
2) then call sub-class constructors

In the sub-class constructor, if `super(…)` is not used explicitly, Java calls the default, parameter-less super-class constructor.

e-Macao-16-2-364

Example: Constructor Call-Order 1

`A` is the super-class:

```
class A {
  A() {
     System.out.println("Inside A's constructor.");
  }
}
```

`B` and `C` are sub-classes of `A`:

```
class B extends A {
  B() {
     System.out.println("Inside B's constructor.");
  }
}
```

e-Macao-16-2-365

Example: Constructor Call-Order 2

```
class C extends B {
  C() {
     System.out.println("Inside C's constructor.");
  }
}
```

`CallingCons` creates a single object of the class `C`:

```
class CallingCons {
  public static void main(String args[]) {
     C c = new C();
  }
}
```

e-Macao-16-2-366

Exercise: Inheritance 1

1) Define a Class `Building` for building objects. Each building has a door as one of its components.
   a) In the class `Door`, model the fact that a door has a color and three states, "`open`" , "`closed`", "`locked`" and "`unlocked`". To avoid illegal state changes, make the state private, write a method (`getState`) that inspects the state and four methods (open, close, lock and unlock) that change the state. Initialize the state to "closed" in the constructor. Look for an alternative place for this initialization.
   b) Write a method `enter` that visualizes the process of entering the building (unlock door, open door, enter, ...) by printing adequate messages, e.g. to show the state of the door.
   c) Write a corresponding method `quit` that visualizes the process of leaving the house. Don't forget to close and lock the door.
   d) Test your class by defining an object of type `Building` and visualizing the state changes when entering and leaving the building.

e-Macao-16-2-367

Exercise: Inheritance 2

3) Extend question 1 by introducing a subclass `HighBuilding` that contains an elevator and the height of the building in addition to the components of `Building`. Override the method `enter` to reflect the use of the elevator. Define a constructor that takes the height of the building as a parameter.
4) Define a subclass `Skyscraper` of `HighBuilding`, where the number of floors is stored with each skyscraper.

   What happens, if you don't define a constructor for class `Skyscraper` (Try it)?

   Write a constructor that takes the number of floors and the height as a parameter. Test the class by creating a skyscraper with 40 floors and using the inherited method `enter`.

A.3.4. Polymorphism

Polymorphism

e-Macao-16-2-369

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-370

Inheritance and Reuse

Reuse of code: every time a new sub-class is defined, programmers are reusing the code in a super-class.

All non-private members of a super-class are inherited by its sub-class:

1) an attribute in a super-class is inherited as-such in a sub-class

2) a method in a super-class is inherited in a sub-class:
   a) as-such, or
   b) is substituted with the method which has the same name and parameters (overriding) but a different implementation

e-Macao-16-2-371

Polymorphism: Definition

Polymorphism is one of three pillars of object-orientation.

Polymorphism: many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:

1) a super-class defines the common interface
2) sub-classes have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)

A sub-class provides a specialized behaviors relying on the common elements defined by its super-class.

e-Macao-16-2-372

Polymorphism: Behavior

Suppose we have a hierarchy of classes:

1) The top class in the hierarchy represents a common interface to all classes below. This class is the base class.
2) All classes below the base represent a number of forms of objects, all referred to by a variable of the base class type.

What happens when a method is invoked using the base class reference?
The object responds in accordance with its true type.

What if the user pointed to a different form of object using the same reference? The user would observe different behavior.

This is polymorphism.

e-Macao-16-2-373

Method Overriding

When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is overridden.

When an overridden method is called from within the sub-class:

1) it will always refer to the sub-class method
2) super-class method is hidden

e-Macao-16-2-374

Example: Hiding with Overriding 1

```
class A {
  int i, j;

  A(int a, int b) {
     i = a; j = b;
  }

  void show() {
     System.out.println("i and j: " + i + " " + j);
  }
}
```

e-Macao-16-2-375

Example: Hiding with Overriding 2

```
class B extends A {
  int k;

  B(int a, int b, int c) {
     super(a, b);
     k = c;
  }

  void show() {
     System.out.println("k: " + k);
  }
}
```

e-Macao-16-2-376

Example: Hiding with Overriding 3

When `show()` is invoked on an object of type `B`, the version of `show()` defined in `B` is used:

```
class Override {
  public static void main(String args[]) {
     B subOb = new B(1, 2, 3);
     subOb.show();
  }
}
```

The version of `show()` in `A` is hidden through overriding.

e-Macao-16-2-377

Super and Method Overriding

The hidden super-class method may be invoked using `super`:

```
class B extends A {
  int k;
  B(int a, int b, int c) {
     super(a, b);
     k = c;
  }
  void show() {
     super.show();
     System.out.println("k: " + k);
  }
}
```

The super-class version of `show()` is called within the sub-class's version.

e-Macao-16-2-378

Overriding versus Overloading 1

Method overriding occurs only when the names and types of the two methods (super-class and sub-class methods) are identical.

If not identical, the two methods are simply overloaded:

```
class A {
  int i, j;

  A(int a, int b) {
     i = a; j = b;
  }
  void show() {
     System.out.println("i and j: " + i + " " + j);
  }
}
```

e-Macao-16-2-379

Overriding versus Overloading 2

The `show()` method in `B` takes a `String` parameter, while the `show()` method in `A` takes no parameters:

```
class B extends A {
  int k;

  B(int a, int b, int c) {
     super(a, b); k = c;
  }

  void show(String msg) {
     System.out.println(msg + k);
  }
}
```

e-Macao-16-2-380

Overriding versus Overloading 3

The two invocations of `show()` are resolved through the number of arguments (zero versus one):

```
class Override {
  public static void main(String args[]) {
     B subOb = new B(1, 2, 3);

     subOb.show("This is k: ");
     subOb.show();
  }
}
```

e-Macao-16-2-381

Dynamic Method Invocation

Overriding is a lot more than the namespace convention.

Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.

Method overriding allows for dynamic method invocation:

1) an overridden method is called through the super-class variable
2) Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
3) when different types of objects are referred, different versions of the overridden method will be called.

e-Macao-16-2-382

Example: Dynamic Invocation 1

A super-class `A`:

```
class A {
  void callme() {
     System.out.println("Inside A's callme method");
  }
}
```

e-Macao-16-2-383

Example: Dynamic Invocation 2

Two sub-classes `B` and `C`:

```
class B extends A {
  void callme() {
     System.out.println("Inside B's callme method");
  }
}
class C extends A {
  void callme() {
     System.out.println("Inside C's callme method");
  }
}
```

`B` and `C` override the `A`'s `callme()` method.

e-Macao-16-2-384

Example: Dynamic Invocation 3

Overridden method is invoked through the variable of the super-class type. Each time, the version of the `callme()` method executed depends on the type of the object being referred to at the time of the call:

```
class Dispatch {
  public static void main(String args[]) {
     A a = new A();
     B b = new B();
     C c = new C();
     A r;
     r = a; r.callme();
     r = b; r.callme();
     r = c; r.callme();
  }
}
```

e-Macao-16-2-385

Polymorphism Again

One interface, many behaviors:

1) super-class defines common methods for sub-classes
2) sub-class provides specific implementations for some of the methods of the super-class

A combination of inheritance and overriding – sub-classes retain flexibility to define their own methods, yet they still have to follow a consistent interface.

e-Macao-16-2-386

Example: Polymorphism 1

A class that stores the dimensions of various 2-dimensional objects:

```
class Figure {
  double dim1;
  double dim2;
  Figure(double a, double b) {
     dim1 = a; dim2 = b;
  }
  double area() {
     System.out.println("Area is undefined.");
     return 0;
  }
}
```

e-Macao-16-2-387

Example: Polymorphism 2

`Rectangle` is a sub-class of `Figure`:

```
class Rectangle extends Figure {

  Rectangle(double a, double b) {
     super(a, b);
  }

  double area() {
     System.out.println("Inside Area for Rectangle.");
     return dim1 * dim2;
  }
}
```

e-Macao-16-2-388

Example: Polymorphism 3

`Triangle` is a sub-class of `Figure`:

```
class Triangle extends Figure {

  Triangle(double a, double b) {
     super(a, b);
  }

  double area() {
     System.out.println("Inside Area for Triangle.");
     return dim1 * dim2 / 2;
  }
}
```

e-Macao-16-2-389

Example: Polymorphism 4

Invoked through the `Figure` variable and overridden in their respective sub-classes, the `area()` method returns the area of the invoking object:

```
class FindAreas {
  public static void main(String args[]) {
     Figure f = new Figure(10, 10);
     Rectangle r = new Rectangle(9, 5);
     Triangle t = new Triangle(10, 8);
     Figure figref;
     figref = r; System.out.println(figref.area());
     figref = t; System.out.println(figref.area());
     figref = f; System.out.println(figref.area());
  }
}
```

e-Macao-16-2-390

Abstract Method

Inheritance allows a sub-class to override the methods of its super-class.

In fact, a super-class may altogether leave the implementation details of a method and declare such a method abstract:

```
abstract type name(parameter-list);
```

Two kinds of methods:

1) concrete – may be overridden by sub-classes
2) abstract – must be overridden by sub-classes

It is illegal to define abstract constructors or static methods.

e-Macao-16-2-391

Example: Abstract Method

The `area` method cannot compute the area of an arbitrary figure:

```
double area() {
   System.out.println("Area is undefined.");
   return 0;
}
```

Instead, `area` should be defined abstract in `Figure`:

```
abstract double area() ;
```

e-Macao-16-2-392

Abstract Class

A class that contains an abstract method must be itself declared abstract:

```
abstract class abstractClassName {
   abstract type methodName(parameter-list) {
      …
   }
   …
}
```

An abstract class has no instances - it is illegal to use the `new` operator:

```
abstractClassName a = new abstractClassName();
```

It is legal to define variables of the abstract class type.

e-Macao-16-2-393

Abstract Sub-Class

A sub-class of an abstract class:

1) implements all abstract methods of its super-class, or
2) is also declared as an abstract class

```
abstract class A {
  abstract void callMe();
}

abstract class B extends A {
  int checkMe;
}
```

e-Macao-16-2-394

Abstract and Concrete Classes 1

Abstract super-class, concrete sub-class:

```
abstract class A {
  abstract void callme();
  void callmetoo() {
     System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
     System.out.println("B's implementation.");
  }
}
```

e-Macao-16-2-395

Abstract and Concrete Classes 2

Calling concrete and overridden abstract methods:

```
class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

e-Macao-16-2-396

Example: Abstract Class 1

`Figure` is an abstract class; it contains an abstract `area` method:

```
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
     dim1 = a; dim2 = b;
  }

  abstract double area();
}
```

e-Macao-16-2-397

Example: Abstract Class 2

`Rectangle` is concrete – it provides a concrete implementation for `area`:

```
class Rectangle extends Figure {
  Rectangle(double a, double b) {
     super(a, b);
  }

  double area() {
     System.out.println("Inside Area for Rectangle.");
     return dim1 * dim2;
  }
}
```

e-Macao-16-2-398

Example: Abstract Class 3

Triangle is concrete – it provides a concrete implementation for `area`:

```
class Triangle extends Figure {
  Triangle(double a, double b) {
     super(a, b);
  }

  double area() {
     System.out.println("Inside Area for Triangle.");
     return dim1 * dim2 / 2;
  }
}
```

e-Macao-16-2-399

Example: Abstract Class 4

Invoked through the `Figure` variable and overridden in their respective sub-classes, the `area()` method returns the area of the invoking object:

```
class AbstractAreas {
  public static void main(String args[]) {
     Rectangle r = new Rectangle(9, 5);
     Triangle t = new Triangle(10, 8);

     Figure figref;

     figref = r; System.out.println(figref.area());
     figref = t; System.out.println(figref.area());
  }
}
```

e-Macao-16-2-400

Abstract Class References

It is illegal to create objects of the abstract class:

```
Figure f = new Figure(10, 10);
```

It is legal to create a variable with the abstract class type:

```
Figure figref;
```

Later, `figref` may be used to assign references to any object of a concrete sub-class of `Figure` (e.g. `Rectangle`) and to invoke methods of this class:

```
Rectangle r = new Rectangle(9, 5);
figref = r; System.out.println(figref.area());
```

e-Macao-16-2-401

Uses of final

The final keyword has three uses:

1) declare a variable which value cannot change after initialization
2) declare a method which cannot be overridden in sub-classes
3) declare a class which cannot have any sub-classes

(1) has been discussed before.

Now is time for (2) and (3).

e-Macao-16-2-402

Preventing Overriding with final

A method declared `final` cannot be overridden in any sub-class:

```
class A {
  final void meth() {
     System.out.println("This is a final method.");
  }
}
```

This class declaration is illegal:

```
class B extends A {
  void meth() {
     System.out.println("Illegal!");
  }
}
```

e-Macao-16-2-403

final and Early Binding

Two types of method invocation:

1) early binding – method call is decided at compile-time
2) late binding – method call is decided at run-time

By default, method calls are resolved at run-time.

As a final method cannot be overridden, their invocations are resolved at compile-time. This is one way to improve performance of a method call.

e-Macao-16-2-404

Preventing Inheritance with final

A class declared `final` cannot be inherited – has no sub-classes.

```
final class A { … }
```

This class declaration is considered illegal:

```
class B extends A { … }
```

Declaring a class `final` implicitly declares all its methods `final`.

It is illegal to declare a class as both `abstract` and `final`.

e-Macao-16-2-405

Object Class

`Object` class is a super-class of all Java classes:

1) `Object` is the root of the Java inheritance hierarchy.
2) A variable of the `Object` type may refer to objects of any class.
3) As arrays are implemented as objects, it may also refer to any array.

e-Macao-16-2-406

Object Class Methods 1

Methods declared in the `Object` class:

1) `Object clone()` - creates an object which is an ideal copy of the invoking object.
2) `boolean equals(Object object)` - determines if the invoking object and the argument object are the same.
3) `void finalize()` – called before an unused object is recycled
4) `Class getClass()` – obtains the class description of an object at run-time
5) `int hashCode()` – returns the hash code for the invoking object

e-Macao-16-2-407

Object Class Methods 2

6) `void notify()` – resumes execution of a thread waiting on the invoking object
7) `void notifyAll()` – resumes execution of all threads waiting on the invoking object
8) `String toString()` – returns the string that describes the invoking object
9) three methods to wait on another thread of execution:
   a) `void wait()`
   b) `void wait(long milliseconds)`
   c) `void wait(long milliseconds, int nanoseconds)`

e-Macao-16-2-408

Overriding Object Class Methods

All methods except `getClass`, `notify`, `notifyAll` and `wait` can be overridden.

Two methods are frequently overridden:

1) `equals()`
2) `toString()`

This way, classes can tailor the equality and the textual description of objects to their own specific structure and needs.

e-Macao-16-2-409

Exercise: Polymorphism

1) Define a relationship among the following `Building`, `HighBuilding` and `Skyscraper` classes.
2) Define a class `Visits` that stores an array of 10 buildings (representing a street).
3) Define a method that enters all the buildings in the street using the method `enter`, one after another.
4) Fill the array with mixed objects from the classes `Building`, `HighBuilding` and `Skyscraper`.
   Make sure, that the output of your program visualizes the fact that different method implementations are used depending on the type of the actual object.

A.3.5. Access

Access

e-Macao-16-2-411

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-412

Name Space Management

Classes written so far all belong to a single name space: a unique name has to be chosen for each class to avoid name collision.

Some way to manage the name space is needed to:

1) ensure that the names are unique
2) provide a continuous supply of convenient, descriptive names
3) ensure that the names chosen by one programmer will not collide with those chosen by another programmers

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is a package.

e-Macao-16-2-413

Package

A package is both a naming and a visibility control mechanism:

1) divides the name space into disjoint subsets

   It is possible to define classes within a package that are not accessible by code outside the package.

2) controls the visibility of classes and their members

   It is possible to define class members that are only exposed to other members of the same package.

Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages.

e-Macao-16-2-414

Package Definition

A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

means that all classes in this file belong to the `myPackage` package. The package statement creates a name space where such classes are stored.

When the package statement is omitted, class names are put into the default package which has no name.

e-Macao-16-2-415

Multiple Source Files

Other files may include the same package instruction:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

```
package myPackage;
class MyClass3{ … }
```

A package may be distributed through several source files.

e-Macao-16-2-416

Packages and Directories

Java uses file system directories to store packages.

Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

The bytecode files `MyClass1.class` and `MyClass2.class` must be stored in a directory `myPackage`.

Case is significant! Directory names must match package names exactly.

e-Macao-16-2-417

Package Hierarchy

To create a package hierarchy, separate each package name with a dot:

```
package myPackage1.myPackage2.myPackage3;
```

A package hierarchy must be stored accordingly in the file system:

1) Unix `myPackage1/myPackage2/myPackage3`
2) Windows `myPackage1\myPackage2\myPackage3`
3) Macintosh `myPackage1:myPackage2:myPackage3`

You cannot rename a package without renaming its directory!

e-Macao-16-2-418

Finding Packages

As packages are stored in directories, how does the Java run-time system know where to look for packages?

Two ways:

1) The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.
2) Specify a directory path or paths by setting the `CLASSPATH` environment variable.

e-Macao-16-2-419

CLASSPATH Variable

`CLASSPATH` - environment variable that points to the root directory of the system's package hierarchy.

Several root directories may be specified in `CLASSPATH`, e.g. the current directory and the `C:\myJava` directory:

```
.;C:\myJava
```

Java will search for the required packages by looking up subsequent directories described in the `CLASSPATH` variable.

e-Macao-16-2-420

Finding Packages

Consider this package statement:

```
package myPackage;
```

In order for a program to find `myPackage`, one of the following must be true:

1) program is executed from the directory immediately above `myPackage` (the parent of `myPackage` directory)
2) `CLASSPATH` must be set to include the path to `myPackage`

e-Macao-16-2-421

Example: Package 1

```
package MyPack;

class Balance {
  String name;
  double bal;
  Balance(String n, double b) {
     name = n; bal = b;
  }
  void show() {
     if (bal<0) System.out.print("-->> ");
     System.out.println(name + ": $" + bal);
  }
}
```

e-Macao-16-2-422

Example: Package 2

```
class AccountBalance {
  public static void main(String args[]) {
     Balance current[] = new Balance[3];

     current[0] = new Balance("K. J. Fielding", 123.23);
     current[1] = new Balance("Will Tell", 157.02);
     current[2] = new Balance("Tom Jackson", -12.33);

     for (int i=0; i<3; i++) current[i].show();
  }
}
```

e-Macao-16-2-423

Example: Package 3

Save, compile and execute:

1) call the file `AccountBalance.java`
2) save the file in the directory `MyPack`
3) compile; `AccountBalance.class` should be also in `MyPack`
4) set access to `MyPack` in `CLASSPATH` variable, or make the parent of `MyPack` your current directory
5) run:

   ```
   java MyPack.AccountBalance
   ```

   Make sure to use the package-qualified class name.

e-Macao-16-2-424

Importing of Packages

Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.

The import statement allows to use classes or whole packages directly.

Importing of a concrete class:

```
import myPackage1.myPackage2.myClass;
```

Importing of all classes within a package:

```
import myPackage1.myPackage2.*;
```

e-Macao-16-2-425

Access Control

Classes and packages are both means of encapsulating and containing the name space and scope of classes, variables and methods:

1) packages act as a container for classes and other packages
2) classes act as a container for data and code

Access control is set separately for classes and class members.

e-Macao-16-2-426

Access Control: Classes

Two levels of access:

1) A class available in the whole program:

```
public class MyClass { … }
```

2) A class available within the same package only:

```
class MyClass { … }
```

e-Macao-16-2-427

Access Control: Members

Four levels of access:

1) a member is available in the whole program:

```
public int variable;
public int method(…) { … }
```

2) a member is only available within the same class:

```
private int variable;
private int method(…) { … }
```

e-Macao-16-2-428

Access Control: Members

3) a member is available within the same package (default access):

```
int variable;
int method(…) { … }
```

4) a member is available within the same package as the current class, or within its sub-classes:

```
protected int variable;
protected int method(…) { … }
```

The sub-class may be located inside or outside the current package.

e-Macao-16-2-429

Access Control Summary

Complicated?

Any member declared `public` can be accessed from anywhere.

Any member declared `private` cannot be seen outside its class.

When a member does not have any access specification (default access), it is visible to all classes within the same package.

To make a member visible outside the current package, but only to sub-classes of the current class, declare this member `protected`.

e-Macao-16-2-430

Table: Access Control

	`private`	default	`protected`	`public`
same class	yes	yes	yes	yes
same package subclass	no	yes	yes	yes
same package non-sub-class	no	yes	yes	yes
different package sub-class	no	no	yes	yes
different package non-sub-class	no	no	no	yes

e-Macao-16-2-431

Example: Access 1

Access example with two packages `p1` and `p2` and five classes.

A public `Protection` class is in the package `p1`.

It has four variables with four possible access rights:

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;
```

e-Macao-16-2-432

Example: Access 2

```
  public Protection() {
     System.out.println("base constructor");
     System.out.println("n = " + n);
     System.out.println("n_pri = " + n_pri);
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
  }
}
```

The rest of the example tests the access to those variables.

e-Macao-16-2-433

Example: Access 3

`Derived` class is in the same `p1` package and is the sub-class of `Protection`.

It has access to all variables of `Protection` except the private `n_pri`:

```
package p1;

class Derived extends Protection {
  Derived() {
     System.out.println("derived constructor");
     System.out.println("n = " + n);
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
  }
}
```

e-Macao-16-2-434

Example: Access 4

`SamePackage` is in the `p1` package but is not a sub-class of `Protection`.

It has access to all variables of `Protection` except the private `n_pri`:

```
package p1;

class SamePackage {
  SamePackage() {
     Protection p = new Protection();
     System.out.println("same package constructor");
     System.out.println("n = " + p.n);
     System.out.println("n_pro = " + p.n_pro);
     System.out.println("n_pub = " + p.n_pub);
  }
}
```

e-Macao-16-2-435

Example: Access 5

`Protection2` is a sub-class of `p1.Protection`, but is located in a different package – package `p2`.

`Protection2` has access to the public and protected variables of `Protection`. It has no access to its private and default-access variables:

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
     System.out.println("derived other package");
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
  }
}
```

e-Macao-16-2-436

Example: Access 6

`OtherPackage` is in the `p2` package and is not a sub-class of `p1.Protection`.

`OtherPackage` has access to the public variable of `Protection` only. It has no access to its private, protected or default-access variables:

```
class OtherPackage {
  OtherPackage() {
     p1.Protection p = new p1.Protection();
     System.out.println("other package constructor");
     System.out.println("n_pub = " + p.n_pub);
  }
}
```

e-Macao-16-2-437

Example: Access 7

A demonstration to use classes of the `p1` package:

```
package p1;

public class Demo {
  public static void main(String args[]) {
     Protection ob1 = new Protection();
     Derived ob2 = new Derived();
     SamePackage ob3 = new SamePackage();
  }
}
```

e-Macao-16-2-438

Example: Access 8

A demonstration to use classes of the `p2` package:

```
package p2;

public class Demo {
  public static void main(String args[]) {
     Protection2 ob1 = new Protection2();
     OtherPackage ob2 = new OtherPackage();
  }
}
```

e-Macao-16-2-439

Import Statement

The import statement occurs immediately after the package statement and before the class statement:

```
package myPackage;
import otherPackage1;otherPackage2.otherClass;
class myClass { … }
```

The Java system accepts this import statement by default:

```
import java.lang.*;
```

This package includes the basic language functions. Without such functions, Java is of no much use.

e-Macao-16-2-440

Name Conflict 1

Suppose a same-named class occurs in two different imported packages:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass { … otherClass … }
```

```
package otherPackage1;
class otherClass { … }
```

```
package otherPackage2;
class otherClass { … }
```

e-Macao-16-2-441

Name Conflict 2

Compiler will remain silent, unless we try to use `otherClass`.
Then it will display an error message.

In this situation we should use the full name:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass {
  …
  otherPackage1.otherClass
  …
  otherPackage2.otherClass
  …
}
```

e-Macao-16-2-442

Short versus Full References

Short reference:

```
import java.util.*;
class MyClass extends Date { … }
```

Full reference:

```
class MyClass extends java.util.Date { … }
```

Only the `public` components in imported package are accessible for non-sub-classes in the importing code!

e-Macao-16-2-443

Example: Packages 1

A package `MyPack` with one `public` class `Balance`. The class has two same-package variables: `public` constructor and a `public show` method.

```
package MyPack;
public class Balance {
  String name;
  double bal;
  public Balance(String n, double b) {
     name = n; bal = b;
  }
  public void show() {
     if (bal<0) System.out.print("-->> ");
     System.out.println(name + ": $" + bal);
  }
}
```

e-Macao-16-2-444

Example: Packages 2

The importing code has access to the `public` class `Balance` of the `MyPack` package and its two public members:

```
import MyPack.*;

class TestBalance {
  public static void main(String args[]) {
     Balance test = new Balance("J. J. Jaspers", 99.88);
     test.show();
  }

}
```

e-Macao-16-2-445

Java Source File

Finally, a Java source file consists of:

1) a single package instruction (optional)
2) several import statements (optional)
3) a single public class declaration (required)
4) several classes private to the package (optional)

At the minimum, a file contains a single public class declaration.

e-Macao-16-2-446

Exercise: Access

1) Create a package `emacao`. Don't forget to insert your package into a directory of the same name. Insert a class `AccessTest` into this packge. Define `public`, `default` and `private` data members and methods in your class `AccessTest`.
2) Define a second class `Accessor1` in your package that accesses the different kinds of data members of methods (`private`, `public`, `default`).

   See what compiler messages you get.

3) Define class `Accessor2` outside the package. Again try to access all methods and data members of the class `AccessTest`.

   See what compiler messages you get.

4) Where are the differences between `Accessor1` and `Accessor2` ?

A.3.6. Interfaces

Interfaces

e-Macao-16-2-448

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-449

Interface

Using interface, we specify what a class must do, but not how it does this.

An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.

An interface is defined with an `interface` keyword.

e-Macao-16-2-450

Interface Format

General format:

```
access interface name {
  type method-name1(parameter-list);
  type method-name2(parameter-list);
  …
  type var-name1 = value1;
  type var-nameM = valueM;
  …
}
```

e-Macao-16-2-451

Interface Comments

Two types of access:

1) `public` – interface may be used anywhere in a program
2) default – interface may be used in the current package only

Interface methods have no bodies – they end with the semicolon after the parameter list. They are essentially abstract methods.

An interface may include variables, but they must be `final`, `static` and initialized with a constant value.

In a `public` interface, all members are implicitly `public`.

e-Macao-16-2-452

Interface Implementation

A class implements an interface if it provides a complete set of methods defined by this interface.

1) any number of classes may implement an interface
2) one class may implement any number of interfaces

Each class is free to determine the details of its implementation.

Implementation relation is written with the `implements` keyword.

e-Macao-16-2-453

Implementation Format

General format of a class that includes the `implements` clause:

```
access class name
  extends super-class
  implements interface1, interface2, …, interfaceN {
  …
}
```

Access is `public` or default.

e-Macao-16-2-454

Implementation Comments

If a class implements several interfaces, they are separated with a comma.

If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.

The methods that implement an interface must be declared `public`.

The type signature of the implementing method must match exactly the type signature specified in the interface definition.

e-Macao-16-2-455

Example: Interface

Declaration of the `Callback` interface:

```
interface Callback {
  void callback(int param);
}
```

`Client` class implements the `Callback` interface:

```
class Client implements Callback {
  public void callback(int p) {
     System.out.println("callback called with " + p);
  }
}
```

e-Macao-16-2-456

More Methods in Implementation

An implementing class may also declare its own methods:

```
class Client implements Callback {
  public void callback(int p) {
     System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
     System.out.println("Classes that implement " +
        "interfaces may also define " +
        "other members, too.");
  }
}
```

e-Macao-16-2-457

Interface as a Type

Variable may be declared with interface as its type:

```
interface MyInterface { … }
…
MyInterface mi;
```

The variable of an interface type may reference an object of any class that implements this interface.

```
class MyClass implements MyInterface { … }

MyInterface mi = new MyClass();
```

e-Macao-16-2-458

Call Through Interface Variable

Using the interface type variable, we can call any method in the interface:

```
interface MyInterface {
   void myMethod(…) ;
   …
}
class MyClass implements MyInterface { … }
…
MyInterface mi = new MyClass();
…
mi.myMethod();
```

The correct version of the method will be called based on the actual instance of the interface being referred to.

e-Macao-16-2-459

Example: Call Through Interface 1

Declaration of the `Callback` interface:

```
interface Callback {
  void callback(int param);
}
```

`Client` class implements the `Callback` interface:

```
class Client implements Callback {
  public void callback(int p) {
     System.out.println("callback called with " + p);
  }
}
```

e-Macao-16-2-460

Example: Call Through Interface 2

`TestIface` declares the `Callback` interface variable, initializes it with the new `Client` object, and calls the `callback` method through this variable:

```
class TestIface {
  public static void main(String args[]) {
     Callback c = new Client();
     c.callback(42);
  }
}
```

e-Macao-16-2-461

Call Through Interface Variable 2

Call through an interface variable is one of the key features of interfaces:

1) the method to be executed is looked up dynamically at run-time
2) the calling code can dispatch through an interface without having to know anything about the callee

Allows classes to be created later than the code that calls methods on them.

e-Macao-16-2-462

Example: Interface Call 1

Another implementation of the `Callback` interface:

```
class AnotherClient implements Callback {
  public void callback(int p) {
     System.out.println("Another version of callback");
     System.out.println("p squared is " + (p*p));
  }
}
```

e-Macao-16-2-463

Example: Interface Call 2

`Callback` variable `c` is assigned `Client` and later `AnotherClient` objects and the corresponding `callback` is invoked depending on its value:

```
class TestIface2 {
  public static void main(String args[]) {
     Callback c = new Client();
     c.callback(42);
     AnotherClient ob = new AnotherClient();
     c = ob;
     c.callback(42);
  }
}
```

e-Macao-16-2-464

Compile-Time Method Binding

Normally, in order for a method to be called from one class to another, both classes must be present at compile time.

This implies:

1) a static, non-extensible classing environment
2) functionality gets pushed higher and higher in the class hierarchy to make them available to more sub-classes

e-Macao-16-2-465

Run-Time Method Binding

Interfaces support dynamic method binding.

Interface disconnects the method definition from the inheritance hierarchy:

1) interfaces are in a different hierarchy from classes
2) it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface

e-Macao-16-2-466

Interface and Abstract Class

A class that claims to implement an interface but does not implement all its methods must be declared abstract.

`Incomplete` class implements the `Callback` interface but not its `callback` method, so the class is declared `abstract`:

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
     System.out.println(a + " " + b);
  }
}
```

e-Macao-16-2-467

Example: Stack Interface

Many ways to implement a stack but one interface:

```
interface IntStack {
  void push(int item);
  int pop();
}
```

Lets look at two implementations of this interface:

1) `FixedStack` – a fixed-length version of the integer stack
2) `DynStack` – a dynamic-length version of the integer stack

e-Macao-16-2-468

Example: FixedStack 1

A fixed-length stack implements the `IntStack` interface with two private variables, a constructor and two public methods:

```
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  FixedStack(int size) {
     stck = new int[size]; tos = -1;
  }
```

e-Macao-16-2-469

Example: FixedStack 2

```
  public void push(int item) {
     if (tos==stck.length-1)
        System.out.println("Stack is full.");
     else stck[++tos] = item;
  }

  public int pop() {
     if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else return stck[tos--];
  }
}
```

e-Macao-16-2-470

Example: FixedStack 3

A testing class creates two stacks:

```
class IFTest {
  public static void main(String args[]) {
     FixedStack mystack1 = new FixedStack(5);
     FixedStack mystack2 = new FixedStack(8);
```

e-Macao-16-2-471

Example: FixedStack 4

It pushes and them pops off some values from those stacks:

```
     for (int i=0; i<5; i++) mystack1.push(i);
     for (int i=0; i<8; i++) mystack2.push(i);

     System.out.println("Stack in mystack1:");
     for (int i=0; i<5; i++)
        System.out.println(mystack1.pop());

     System.out.println("Stack in mystack2:");
     for (int i=0; i<8; i++)
        System.out.println(mystack2.pop());
  }
}
```

e-Macao-16-2-472

Example: DynStack 1

Another implementation of an integer stack.

A dynamic-length stack is first created with an initial length. The stack is doubled in size every time this initial length is exceeded.

```
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  DynStack(int size) {
     stck = new int[size];
     tos = -1;
  }
```

e-Macao-16-2-473

Example: DynStack 2

If stack if full, `push` creates a new stack with double the size of the old stack:

```
  public void push(int item) {
     if (tos==stck.length-1) {
        int temp[] = new int[stck.length * 2];
        for (int i=0; i<stck.length; i++)
           temp[i] = stck[i];
        stck = temp;
        stck[++tos] = item;
     }
     else stck[++tos] = item;
  }
```

e-Macao-16-2-474

Example: DynStack 3

If the stack is empty, `pop` returns the zero value:

```
  public int pop() {
     if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else return stck[tos--];
  }
}
```

e-Macao-16-2-475

Example: DynStack 4

The testing class creates two dynamic-length stacks:

```
class IFTest2 {
  public static void main(String args[]) {
     DynStack mystack1 = new DynStack(5);
     DynStack mystack2 = new DynStack(8);
```

e-Macao-16-2-476

Example: DynStack 5

It then pushes some numbers onto those stacks, dynamically increasing their size, then pops those numbers off:

```
     for (int i=0; i<12; i++) mystack1.push(i);
     for (int i=0; i<20; i++) mystack2.push(i);

     System.out.println("Stack in mystack1:");
     for (int i=0; i<12; i++)
        System.out.println(mystack1.pop());

     System.out.println("Stack in mystack2:");
     for (int i=0; i<20; i++)
        System.out.println(mystack2.pop());
  }
}
```

e-Macao-16-2-477

Example: Two Stacks 1

Testing two stack implementations through an interface variable.

First, some numbers are pushed onto both stacks:

```
class IFTest3 {
  public static void main(String args[]) {
     IntStack mystack;
     DynStack ds = new DynStack(5);
     FixedStack fs = new FixedStack(8);

     mystack = ds;
     for (int i=0; i<12; i++) mystack.push(i);
     mystack = fs;
     for (int i=0; i<8; i++) mystack.push(i);
```

e-Macao-16-2-478

Example: Two Stacks 2

Then, those numbers are popped off:

```
     mystack = ds;
     System.out.println("Values in dynamic stack:");
     for (int i=0; i<12; i++)
        System.out.println(mystack.pop());
     mystack = fs;
     System.out.println("Values in fixed stack:");
     for (int i=0; i<8; i++)
        System.out.println(mystack.pop());
  }
}
```

Which stack implementation is the value of the `mystack` variable, therefore which version of `push` and `pop` are used, is determined at run-time.

e-Macao-16-2-479

Interface Variables

Variables declared in an interface must be constants.

A technique to import shared constants into multiple classes:

1) declare an interface with variables initialized to the desired values
2) include that interface in a class through implementation

As no methods are included in the interface, the class does not implement anything except importing the variables as constants.

e-Macao-16-2-480

Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}
```

e-Macao-16-2-481

Example: Interface Variables 2

`Question` implements `SharedConstants`, including all its constants.

Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
     int prob = (int) (100 * rand.nextDouble());
     if (prob < 30)     return NO;
     else if (prob < 60) return YES;
     else if (prob < 75) return LATER;
     else if (prob < 98) return SOON;
     else return NEVER;
  }
}
```

e-Macao-16-2-482

Example: Interface Variables 3

`AskMe` includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {
  static void answer(int result) {
     switch(result) {
        case NO:    System.out.println("No"); break;
        case YES:   System.out.println("Yes"); break;
        case MAYBE: System.out.println("Maybe"); break;
        case LATER: System.out.println("Later"); break;
        case SOON:  System.out.println("Soon"); break;
        case NEVER: System.out.println("Never"); break;
     }
  }
```

e-Macao-16-2-483

Example: Interface Variables 4

The testing function relies on the fact that both `ask` and `answer` methods, defined in different classes, rely on the same constants:

```
  public static void main(String args[]) {
     Question q = new Question();
     answer(q.ask());
     answer(q.ask());
     answer(q.ask());
     answer(q.ask());
  }
}
```

e-Macao-16-2-484

Interface Inheritance

One interface may inherit another interface.

The inheritance syntax is the same for classes and interfaces.

```
  interface MyInterface1 {
     void myMethod1(…) ;
  }
  interface MyInterface2 extends MyInterface1 {
     void myMethod2(…) ;
  }
```

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

e-Macao-16-2-485

Inheritance and Implementation

When a class implements an interface that inherits another interface, it must provide implementations for all inherited methods:

```
class MyClass implements MyInterface2 {
   void myMethod1(…)  { … }
   void myMethod1(…)  { … }
   …
}
```

e-Macao-16-2-486

Example: Interface Inheritance 1

Consider interfaces `A` and `B`.

```
interface A {
  void meth1();
  void meth2();
}
```

`B` extends `A`:

```
interface B extends A {
  void meth3();
}
```

e-Macao-16-2-487

Example: Interface Inheritance 2

`MyClass` must implement all of `A` and `B` methods:

```
class MyClass implements B {
  public void meth1() {
     System.out.println("Implement meth1().");
  }
  public void meth2() {
     System.out.println("Implement meth2().");
  }
  public void meth3() {
     System.out.println("Implement meth3().");
  }
}
```

e-Macao-16-2-488

Example: Interface Inheritance 3

Create a new `MyClass` object, then invoke all interface methods on it:

```
class IFExtend {
  public static void main(String arg[]) {
     MyClass ob = new MyClass();
     ob.meth1();
     ob.meth2();
     ob.meth3();
  }
}
```

e-Macao-16-2-489

Exercise: Interface

1) Define two interfaces:
   a) an interface `CardUse` for card use with methods `read` to read the state and `reduceBy` with a parameter amount to change the state of the card. The user has to identify himself by a PIN for this operation;
   b) an interface `CardChange` for the administration of the card by authorized people, that need a method `reset` to reset the card state, a method `fill` to fill the card with an amount of money and a method `changePIN` to change the PIN for the card.

2) Define a third interface `CardAll` that includes all card operation (Use inheritance).

3) Change the interface `CardAll` into an abstract class that implements the balance of the card and a basic solution for the methods `fill` and `reduceBy` leaving the rest of the methods abstract. Choose the correct access specifier to make the balance accessible to subclasses but not to the public; Check this;

A.3.7. Exception Handling

Exception-Handling

e-Macao-16-2-491

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-492

Exceptions

Exception is an abnormal condition that arises when executing a program.

In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.

In contrast, Java:

1) provides syntactic mechanisms to signal, detect and handle errors
2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
3) brings run-time error management into object-oriented programming

e-Macao-16-2-493

Exception Handling

An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.

Exception handling involves the following:

1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
2) that method may choose to handle the exception itself or pass it on
3) either way, at some point, the exception is caught and processed

e-Macao-16-2-494

Exception Sources

Exceptions can be:

1) generated by the Java run-time system

   Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

2) manually generated by programmer's code

   Such exceptions are typically used to report some error conditions to the caller of a method.

e-Macao-16-2-495

Exception Constructs

Five constructs are used in exception handling:

1) `try` – a block surrounding program statements to monitor for exceptions
2) `catch` – together with try, catches specific kinds of exceptions and handles them in some way
3) `finally` – specifies any code that absolutely must be executed whether or not an exception occurs
4) `throw` – used to throw a specific exception from the program
5) `throws` – specifies which exceptions a given method can throw

e-Macao-16-2-496

Exception-Handling Block

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…
finally { … }
```

where:

1) `try { … }` is the block of code to monitor for exceptions
2) `catch(Exception ex) { … }` is exception handler for the exception `Exception`
3) `finally { … }` is the block of code to execute before the `try` block ends

e-Macao-16-2-497

Exception Hierarchy

All exceptions are sub-classes of the build-in class `Throwable`.

`Throwable` contains two immediate sub-classes:

1) `Exception` – exceptional conditions that programs should catch

   The class includes:

   a) `RuntimeException` – defined automatically for user programs to include: division by zero, invalid array indexing, etc.
   b) use-defined exception classes

2) `Error` – exceptions used by Java to indicate errors with the run-time environment; user programs are not supposed to catch them

e-Macao-16-2-498

Uncaught Exception

What happens when exceptions are not handled?

```
class Exc0 {
  public static void main(String args[]) {
     int d = 0;
     int a = 42 / d;
  }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and throws this object.

This will cause the execution of `Exc0` to stop – once an exception has been thrown it must be caught by an exception handler and dealt with.

e-Macao-16-2-499

Default Exception Handler

As we have not provided any exception handler, the exception is caught by the default handler provided by the Java run-time system.

This default handler:

1) displays a string describing the exception,
2) prints the stack trace from the point where the exception occurred
3) terminates the program

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Any exception not caught by the user program is ultimately processed by the default handler.

e-Macao-16-2-500

Stack Trace Display

The stack trace displayed by the default error handler shows the sequence of method invocations that led up to the error.

Here the exception is raised in `subroutine()` which is called by `main()`:

```
class Exc1 {
  static void subroutine() {
     int d = 0;
     int a = 10 / d;
  }
  public static void main(String args[]) {
     Exc1.subroutine();
  }
}
```

e-Macao-16-2-501

Own Exception Handling

Default exception handling is basically useful for debugging.

Normally, we want to handle exceptions ourselves because:

1) if we detected the error, we can try to fix it
2) we prevent the program from automatically terminating

Exception handling is done through the try and catch block.

e-Macao-16-2-502

Try and Catch 1

Try and catch:

1) `try` surrounds any code we want to monitor for exceptions
2) `catch` specifies which exception we want to handle and how.

When an exception is thrown in the try block:

```
try {
   d = 0;
   a = 42 / d;
   System.out.println("This will not be printed.");
}
```

e-Macao-16-2-503

Try and Catch 2

control moves immediately to the catch block:

```
catch (ArithmeticException e) {
   System.out.println("Division by zero.");
}
```

The exception is handled and the execution resumes.

The scope of catch is restricted to the immediately preceding try statement - it cannot catch exceptions thrown by another try statements.

e-Macao-16-2-504

Try and Catch 3

Resumption occurs with the next statement after the try/catch block:

```
try { … }
catch (ArithmeticException e) { … }
System.out.println("After catch statement.");
```

Not with the next statement after `a = 42/d;` which caused the exception!

```
a = 42 / d;
System.out.println("This will not be printed.");
```

e-Macao-16-2-505

Catch and Continue 1

The purpose of catch should be to resolve the exception and then continue as if the error had never happened.

Try/catch block inside a loop:

```
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
     int a=0, b=0, c=0;
     Random r = new Random();
```

e-Macao-16-2-506

Catch and Continue 2

After exception-handling, the program continues with the next iteration:

```
    for (int i=0; i<32000; i++) {
       try {
          b = r.nextInt();
          c = r.nextInt();
          a = 12345 / (b/c);
       } catch (ArithmeticException e) {
          System.out.println("Division by zero.");
          a = 0; // set a to zero and continue
       }
       System.out.println("a: " + a);
    }
  }
}
```

e-Macao-16-2-507

Exception Display

All exception classes inherit from the `Throwable` class.

`Throwable` overrides `toString()` to describe the exception textually:

```
try { … }
catch (ArithmeticException e) {
   System.out.println("Exception: " + e);
}
```

The following text will be displayed:

```
Exception: java.lang.ArithmeticException: / by zero
```

e-Macao-16-2-508

Multiple Catch Clauses

When more than one exception can be raised by a single piece of code, several `catch` clauses can be used with one `try` block:

1) each `catch` catches a different kind of exception
2) when an exception is thrown, the first one whose type matches that of the exception is executed
3) after one `catch` executes, the other are bypassed and the execution continues after the try/catch block

e-Macao-16-2-509

Example: Multiple Catch 1

Two different exception types are possible in the following code: division by zero and array index out of bound:

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    }
```

e-Macao-16-2-510

Example: Multiple Catch 2

Both exceptions can be caught by the following catch clauses:

```
      catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

e-Macao-16-2-511

Order of Multiple Catch Clauses

Order is important:

1) catch clauses are inspected top-down
2) a clause using a super-class will catch all sub-class exceptions

Therefore, specific exceptions should appear before more general ones.

In particular, exception sub-classes must appear before super-classes.

e-Macao-16-2-512

Example: Multiple Catch Order 1

A try block with two catch clauses:

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
       int a = 0;
       int b = 42 / a;
```

This exception is more general but occurs first:

```
    } catch(Exception e) {
       System.out.println("Generic Exception catch.");
    }
```

e-Macao-16-2-513

Example: Multiple Catch Order 2

This exception is more specific but occurs last:

```
    catch(ArithmeticException e) {
       System.out.println("This is never reached.");
    }
  }
}
```

The second clause will never get executed. A compile-time error (unreachable code) will be raised.

e-Macao-16-2-514

Nested try Statements

The try statements can be nested:

1) if an inner try does not catch a particular exception
2) the exception is inspected by the outer try block
3) this continues until:
   a) one of the catch statements succeeds or
   b) all the nested try statements are exhausted
4) in the latter case, the Java run-time system will handle the exception

e-Macao-16-2-515

Example: Nested try 1

An example with two nested try statements:

```
class NestTry {
  public static void main(String args[]) {
```

Outer try statement:

```
    try {
       int a = args.length;
```

Division by zero when no command-line argument is present:

```
       int b = 42 / a;
       System.out.println("a = " + a);
```

e-Macao-16-2-516

Example: Nested try 2

Inner try statement:

```
try {
```

Division by zero when one command-line argument is present:

```
if (a==1) a = a/(a-a);
```

Array index out of bound when two command-line arguments are present:

```
if (a==2) {
   int c[] = { 1 };
   c[42] = 99;
}
```

e-Macao-16-2-517

Example: Nested try 3

Catch statement for the inner try statement, catches the array index out of bound exception:

```
} catch(ArrayIndexOutOfBoundsException e) {
   System.out.println(e);
}
```

Catch statement for the outer try statement, catches both division-by-zero exceptions for the inner and outer try statements:

```
    } catch(ArithmeticException e) {
       System.out.println("Divide by 0: " + e);
    }
  }
}
```

e-Macao-16-2-518

Method Calls and Nested try 1

Nesting of try statements with method calls:

1) method call is enclosed within one try statement
2) the method includes another try statement

Still, the try blocks are considered nested within each other.

```
class MethNestTry {
```

e-Macao-16-2-519

Method Calls and Nested try 2

Method `nesttry` contains the try statement:

```
  static void nesttry(int a) {
     try {
        if (a==1) a = a/(a-a);
        if (a==2) {
           int c[] = { 1 }; c[42] = 99;
        }
     } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println(e);
     }
  }
```

e-Macao-16-2-520

Method Calls and Nested try 3

Method `main` contains another try block which includes the call to `nesttry`:

```
  public static void main(String args[]) {
     try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        nesttry(a);
     }
     } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
     }
  }
}
```

e-Macao-16-2-521

Throwing Exceptions

So far, we were only catching the exceptions thrown by the Java system.

In fact, a user program may throw an exception explicitly:

```
  throw ThrowableInstance;
```

`ThrowableInstance` must be an object of type `Throwable` or its subclass.

e-Macao-16-2-522

throw Follow-up

Once an exception is thrown by:

```
throw ThrowableInstance;
```

1) the flow of control stops immediately
2) the nearest enclosing `try` statement is inspected if it has a `catch` statement that matches the type of exception:
   1) if one exists, control is transferred to that statement
   2) otherwise, the next enclosing `try` statement is examined
3) if no enclosing `try` statement has a corresponding `catch` clause, the default exception handler halts the program and prints the stack

e-Macao-16-2-523

Creating Exceptions

Two ways to obtain a `Throwable` instance:

1) creating one with the `new` operator

   All Java built-in exceptions have at least two constructors: one without parameters and another with one String parameter:

   ```
   throw new NullPointerException("demo");
   ```

2) using a parameter of the `catch` clause

   ```
   try { … } catch(Throwable e) { … e … }
   ```

e-Macao-16-2-524

Example: throw 1

```
class ThrowDemo {
```

The method `demoproc` throws a `NullPointerException` exception which is immediately caught in the try block and re-thrown:

```
  static void demoproc() {
     try {
        throw new NullPointerException("demo");
     } catch(NullPointerException e) {
        System.out.println("Caught inside demoproc.");
        throw e;
     }
  }
```

e-Macao-16-2-525

Example: throw 2

The main method calls `demoproc` within the try block which catches and handles the `NullPointerException` exception:

```
  public static void main(String args[]) {
     try {
        demoproc();
     } catch(NullPointerException e) {
        System.out.println("Recaught: " + e);
     }
  }
}
```

e-Macao-16-2-526

throws Declaration

If a method is capable of causing an exception that it does not handle, it must specify this behavior by the `throws` clause in its declaration:

```
type name(parameter-list) throws exception-list {
  …
}
```

where `exception-list` is a comma-separated list of all types of exceptions that a method might throw.

All exceptions must be listed except `Error` and `RuntimeException` or any of their subclasses, otherwise a compile-time error occurs.

e-Macao-16-2-527

Example: throws 1

The `throwOne` method throws an exception that it does not catch, nor declares it within the `throws` clause.

```
class ThrowsDemo {
  static void throwOne() {
     System.out.println("Inside throwOne.");
     throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
     throwOne();
  }
}
```

Therefore this program does not compile.

e-Macao-16-2-528

Example: throws 2

Corrected program: `throwOne` lists exception, `main` catches it:

```
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
     System.out.println("Inside throwOne.");
     throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
     try {
        throwOne();
     } catch (IllegalAccessException e) {
        System.out.println("Caught " + e);
     }
  }
}
```

e-Macao-16-2-529

Motivating finally

When an exception is thrown:

1) the execution of a method is changed
2) the method may even return prematurely.

This may be a problem is many situations.

For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.

The `finally` block is used to address this problem.

e-Macao-16-2-530

finally Clause

The `try/catch` statement requires at least one `catch` or `finally` clause, although both are optional:

```
try { … }
catch(Exception1 ex1) { … } …
finally { … }
```

Executed after try/catch whether of not the exception is thrown.

Any time a method is to return to a caller from inside the try/catch block via:

1) uncaught exception or
2) explicit return

the finally clause is executed just before the method returns.

e-Macao-16-2-531

Example: finally 1

Three methods to exit in various ways.

```
class FinallyDemo {
```

`procA` prematurely breaks out of the `try` by throwing an exception, the `finally` clause is executed on the way out:

```
  static void procA() {
     try {
        System.out.println("inside procA");
        throw new RuntimeException("demo");
     } finally {
        System.out.println("procA's finally");
     }
  }
```

e-Macao-16-2-532

Example: finally 2

`procB`'s `try` statement is exited via a `return` statement, the `finally` clause is executed before `procB` returns:

```
static void procB() {
   try {
      System.out.println("inside procB");
      return;
   } finally {
      System.out.println("procB's finally");
   }
}
```

e-Macao-16-2-533

Example: finally 3

In `procC`, the `try` statement executes normally without error, however the `finally` clause is still executed:

```
static void procC() {
   try {
      System.out.println("inside procC");
   } finally {
      System.out.println("procC's finally");
   }
}
```

e-Macao-16-2-534

Example: finally 4

Demonstration of the three methods:

```
public static void main(String args[]) {
   try {
      procA();
   } catch (Exception e) {
      System.out.println("Exception caught");
   }
   procB();
   procC();
}
}
```

e-Macao-16-2-535

Java Built-In Exceptions

The default `java.lang` package provides several exception classes, all sub-classing the `RuntimeException` class.

Two sets of build-in exception classes:

1) unchecked exceptions – the compiler does not check if a method handles or throws there exceptions
2) checked exceptions – must be included in the method's `throws` clause if the method generates but does not handle them

e-Macao-16-2-536

Unchecked Built-In Exceptions 1

Methods that generate but do not handle those exceptions need not declare them in the `throws` clause:

`ArithmeticException`	arithmetic error such as divide-by-zero
`ArrayIndexOutOfBoundsException`	array index out of bounds
`ArrayStoreException`	assignment to an array element of the wrong type
`ClassCastException`	invalid cast
`IllegalArgumentException`	illegal argument used to invoke a method
`IllegalMonitorStateException`	illegal monitor behavior, e.g. waiting on an unlocked thread
`IllegalStateException`	environment of application is in incorrect state

e-Macao-16-2-537

Unchecked Built-In Exceptions 2

`IllegalThreadStateException`	requested operation not compatible with current thread state
`IndexOutOfBoundsException`	some type of index is out-of-bounds
`NegativeArraySizeException`	array created with a negative size
`NullPointerException`	invalid use of null reference
`NumberFormatException`	invalid conversion of a string to a numeric format
`SecurityException`	attempt to violate security
`StringIndexOutOfBounds`	attempt to index outside the the bounds of a string
`UnsupportedOperationException`	an unsupported operation was encountered

e-Macao-16-2-538

Checked Built-In Exceptions

Methods that generate but do not handle those exceptions must declare them in the `throws` clause:

`ClassNotFoundException`	class not found
`CloneNotSupportedException`	attempt to clone an object that does not implement the Cloneable interface
`IllegalAccessException`	access to a class is denied
`InstantiationException`	attempt to create an object of an abstract class or interface
`InterruptedException`	one thread has been interrupted by another thread
`NoSuchFieldException`	a requested field does not exist
`NoSuchMethodException`	a requested method does not exist

e-Macao-16-2-539

Creating Own Exception Classes

Build-in exception classes handle some generic errors.

For application-specific errors define your own exception classes.

How? Define a subclass of `Exception`:

```
class MyException extends Exception { … }
```

`MyException` need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

e-Macao-16-2-540

Throwable Class 1

`Exception` itself is a sub-class of `Throwable`.

All user exceptions have the methods defined by the `Throwable` class:

1) `Throwable fillInStackTrace()` – returns a `Throwable` object that contains a completed stack trace; the object can be rethrown.
2) `Throwable getCause()` – returns the exception that underlies the current exception. If no underlying exception exists, null is returned.
3) `String getLocalizedMessage()` – returns a localized description of the exception.
4) `String getMessage()` – returns a description of the exception
5) `StackTraceElement[] getStackTrace()` – returns an array that contains the stack trace; the method at the top is the last one called before exception.

e-Macao-16-2-541

Throwable Class 2

More methods defined by the `Throwable` class:

6) `Throwable initCause(Throwable causeExc)` – associates `causeExc` with the invoking exception as its cause, returns the exception reference
7) `void printStackTrace()` – displays the stack trace
8) `void printStackTrace(PringStream stream)` – sends the stack trace to the specified stream
9) `void setStackTrace(StackTraceElement elements[])` – sets the stack trace to the elements passed in elements; for specialized applications only
10) `String toString()` – returns a `String` object containing a description of the exception; called by `print()` when displaying a `Throwable` object.

e-Macao-16-2-542

Example: Own Exceptions 1

A new exception class is defined, with a private `detail` variable, a one-parameter constructor and an overridden `toString` method:

```
class MyException extends Exception {
  private int detail;

  MyException(int a) {
     detail = a;
  }

  public String toString() {
     return "MyException[" + detail + "]";
  }
}
```

e-Macao-16-2-543

Example: Own Exceptions 2

```
class ExceptionDemo {
```

The static `compute` method throws the `MyException` exception whenever its `a` argument is greater than `10`:

```
  static void compute(int a) throws MyException {
     System.out.println("Called compute(" + a + ")");
     if (a > 10) throw new MyException(a);
     System.out.println("Normal exit");
  }
```

e-Macao-16-2-544

Example: Own Exceptions 3

The `main` method calls `compute` with two arguments within a `try` block that catches the `MyException` exception:

```
public static void main(String args[]) {
   try {
      compute(1);
      compute(20);
   } catch (MyException e) {
      System.out.println("Caught " + e);
   }
}
}
```

e-Macao-16-2-545

Chained Exceptions 1

The chained exception allows to associate with a given exception another exception that describes its cause.

`Throwable` class includes two constructors to handle chained exceptions:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

They both create an exception with `causeExc` being its underlying reason and optional `msg` providing the textual description.

e-Macao-16-2-546

Chained Exceptions 2

`Throwable` class also includes two methods to handle chained exceptions:

1) `Throwable getCause()` – returns an exception that is the cause of the current exception, or null if there is no underlying exception.
2) `Throwable initCause(Throwable causeExc)` – associates `causeExc` with the invoking exception and returns a reference to the exception:
   a) `initCause` allows to associate a cause with an existing exception
   b) the cause exception can be set only once
   c) if the cause exception was set by a constructor, it is not possible to set it again with `initCause`

e-Macao-16-2-547

Example: Chained Exceptions 1

```
class ChainExcDemo {
```

The `demoproc` method creates a new `NullPointerException` exception `e`, associates `ArithmethicException` as its cause, then throws `e`:

```
static void demoproc() {
   NullPointerException e =
      new NullPointerException("top layer");
   e.initCause(new ArithmeticException("cause"));
   throw e;
}
```

e-Macao-16-2-548

Example: Chained Exceptions 2

The `main` method calls `demoproc` within the `try` block that catches `NullPointerException`, then displays the exception and its cause:

```
public static void main(String args[]) {
   try {
      demoproc();
   } catch(NullPointerException e) {
      System.out.println("Caught: " + e);
      System.out.println("Cause: " + e.getCause());
   }
}
}
```

A cause exception may itself have a cause. In fact, the cause-chain of exceptions may be arbitrarily long.

e-Macao-16-2-549

Exceptions Usage

New Java programmers should break the habit of returning error codes to signal abnormal exit from a method.

Java provides a clean and powerful way to handle errors and unusual boundary conditions through its:

1) `try`
2) `catch`
3) `finally`
4) `throw` and
5) `throws` statements.

However, Java's exception-handling should not be considered a general mechanism for non-local branching!

e-Macao-16-2-550

Exercise: Exception-Handling

1) Create exception classes called `Even` and `Odd`
2) Generate numbers within an endless loop. Print the generated numbers.
3) If the number is even, throw the `Even` exception with the message "`The number thrown an even number`" along with the number.
4) If the number is odd, throw the `Odd` exception with the message "`The number thrown an odd number`" along with the number.
5) Catch the `Even` exception within the endless loop and print the message.
6) Catch the `Odd` exception outside of the loop and print the message.

A.3.8. Multi-Threading

Multi-Threading

e-Macao-16-2-552

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-553

Multi-Tasking

Two kinds of multi-tasking:

1) process-based multi-tasking
2) thread-based multi-tasking

Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.

Processes are heavyweight tasks:

1) that require their own address space
2) inter-process communication is expensive and limited
3) context-switching from one process to another is expensive and limited

e-Macao-16-2-554

Thread-Based Multi-Tasking

Thread-based multi-tasking is about a single program executing concurrently several tasks e.g. a text editor printing and spell-checking text.

Threads are lightweight tasks:

1) they share the same address space
2) they cooperatively share the same process
3) inter-thread communication is inexpensive
4) context-switching from one thread to another is low-cost

Java multi-tasking is thread-based.

e-Macao-16-2-555

Reasons for Multi-Threading

Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.

There is plenty of idle time for interactive, networked applications:

1) the transmission rate of data over a network is much slower than the rate at which the computer can process it
2) local file system resources can be read and written at a much slower rate than can be processed by the CPU
3) of course, user input is much slower than the computer

e-Macao-16-2-556

Single-Threading

In a single-threaded environment, the program has to wait for each of these tasks to finish before it can proceed to the next.

Single-threaded systems use event loop with pooling:

1) a single thread of control runs in an infinite loop
2) the loop pools a single event queue to decide what to do next
3) the pooling mechanism returns an event
4) control is dispatched to the appropriate event handler
5) until this event handler returns, nothing else can happen

e-Macao-16-2-557

Threads: Model

Thread exist in several states:

1) ready to run
2) running
3) a running thread can be suspended
4) a suspended thread can be resumed
5) a thread can be blocked when waiting for a resource
6) a thread can be terminated

Once terminated, a thread cannot be resumed.

e-Macao-16-2-558

Threads: Priorities

Every thread is assigned priority – an integer number to decide when to switch from one running thread to the next (context-switching).

Rules for context switching:

1) a thread can voluntarily relinquish control (sleeping, blocking on I/O, etc.), then the highest-priority ready to run thread is given the CPU.
2) a thread can be preempted by a higher-priority thread – a lower-priority thread is suspended

When two equal-priority threads are competing for CPU time, which one is chosen depends on the operating system.

e-Macao-16-2-559

Threads: Synchronization

Multi-threading introduces asynchronous behavior to a program. How to ensure synchronous behavior when we need it?

For instance, how to prevent two threads from simultaneously writing and reading the same object?

Java implementation of monitors:

1) classes can define so-called synchronized methods
2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

e-Macao-16-2-560

Thread Class

To create a new thread a program will:

1) extend the `Thread` class, or
2) implement the `Runnable` interface

`Thread` class encapsulates a thread of execution.

The whole Java multithreading environment is based on the `Thread` class.

e-Macao-16-2-561

Thread Methods

`getName`	obtain a thread's name
`getPriority`	obtain a thread's priority
`isAlive`	determine if a thread is still running
`join`	wait for a thread to terminate
`run`	entry-point for a thread
`sleep`	suspend a thread for a period of time
`start`	start a thread by calling its run method

e-Macao-16-2-562

The Main Thread

The main thread is a thread that begins as soon as a program starts.

The main thread:

1) is invoked automatically
2) is the first to start and the last to finish
3) is the thread from which other "child" threads will be spawned

It can be obtained through the

```
public static Thread currentThread()
```

method of `Thread`.

e-Macao-16-2-563

Example: Main Thread 1

```
class CurrentThreadDemo {
  public static void main(String args[]) {
```

The main thread is obtained, displayed, its name changed and re-displayed:

```
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    t.setName("My Thread");
    System.out.println("After name change: " + t);
```

e-Macao-16-2-564

Example: Main Thread 2

A loop performs five iterations pausing for a second between the iterations.

It is performed within the `try/catch` block – the `sleep` method may throw `InterruptedException` if some other thread wanted to interrupt:

```
    try {
      for (int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

e-Macao-16-2-565

Example: Thread Methods

`Thread` methods used by the example:

1) `static void sleep(long milliseconds) throws InterruptedException`
   Causes the thread from which it is executed to suspend execution for the specified number of milliseconds.

2) `final String getName()`
   Allows to obtain the name of the current thread.

3) `final void setName(String threadName)`
   Sets the name of the current thread.

e-Macao-16-2-566

Creating a Thread

Two methods to create a new thread:

1) by implementing the `Runnable` interface
2) by extending the `Thread` class

We look at each method in order.

e-Macao-16-2-567

New Thread: Runnable

To create a new thread by implementing the `Runnable` interface:

1) create a class that implements the `run` method (inside this method, we define the code that constitutes the new thread):

```
public void run()
```

2) instantiate a `Thread` object within that class, a possible constructor is:

```
Thread(Runnable threadOb, String threadName)
```

3) call the `start` method on this object (`start` calls `run`):

```
void start()
```

e-Macao-16-2-568

Example: New Thread 1

A class `NewThread` that implements `Runnable`:

```
class NewThread implements Runnable {
  Thread t;
```

Creating and starting a new thread. Passing `this` to the `Thread` constructor – the new thread will call this object's `run` method:

```
  NewThread() {
     t = new Thread(this, "Demo Thread");
     System.out.println("Child thread: " + t);
     t.start();
  }
```

e-Macao-16-2-569

Example: New Thread 2

This is the entry point for the newly created thread – a five-iterations loop with a half-second pause between the iterations all within try/catch:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println("Child Thread: " + i);
           Thread.sleep(500);
        }
     } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
     }
     System.out.println("Exiting child thread.");
  }
}
```

e-Macao-16-2-570

Example: New Thread 3

```
class ThreadDemo {
  public static void main(String args[]) {
```

A new thread is created as an object of `NewThread`:

```
     new NewThread();
```

After calling the `NewThread` `start` method, control returns here.

e-Macao-16-2-571

Example: New Thread 4

Both threads (new and main) continue concurrently.

Here is the loop for the main thread:

```
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000);
        }
     } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
     }
     System.out.println("Main thread exiting.");
  }
}
```

e-Macao-16-2-572

New Thread: Extend Thread

The second way to create a new thread:

1) create a new class that extends `Thread`
2) create an instance of that class

Thread provides both `run` and `start` methods:

1) the extending class must override `run`
2) it must also call the `start` method

e-Macao-16-2-573

Example: New Thread 1

The new thread class extends `Thread`:

```
class NewThread extends Thread {
```

Create a new thread by calling the `Thread`'s constructor and `start` method:

```
  NewThread() {
     super("Demo Thread");
     System.out.println("Child thread: " + this);
     start();
  }
```

e-Macao-16-2-574

Example: New Thread 2

`NewThread` overrides the `Thread`'s `run` method:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println("Child Thread: " + i);
           Thread.sleep(500);
        }
     } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
     }
     System.out.println("Exiting child thread.");
  }
}
```

e-Macao-16-2-575

Example: New Thread 3

```
class ExtendThread {
  public static void main(String args[]) {
```

After a new thread is created:

```
     new NewThread();
```

the new and main threads continue concurrently…

e-Macao-16-2-576

Example: New Thread 4

This is the loop of the main thread:

```
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000);
        }
     } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
     }
     System.out.println("Main thread exiting.");
  }
}
```

e-Macao-16-2-577

New Thread: Which Approach?

The `Thread` class defines several methods that can be overriden.

Of these methods, only `run` must be overriden.

Creating a new thread:

1) implement `Runnable` if only `run` is overriden
2) extend `Thread` if other methods are also overriden

e-Macao-16-2-578

Example: Multiple Threads 1

So far, we were using only two threads - main and new, but in fact a program may spawn as many threads as it needs.

`NewThread` class implements the `Runnable` interface:

```
class NewThread implements Runnable {
  String name;
  Thread t;

  NewThread(String threadname) {
     name = threadname;
     t = new Thread(this, name);
     System.out.println("New thread: " + t);
     t.start();
  }
```

e-Macao-16-2-579

Example: Multiple Threads 2

Here is the implementation of the `run` method:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println(name + ": " + i);
           Thread.sleep(1000);
        }
     } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
     }
     System.out.println(name + " exiting.");
  }
}
```

e-Macao-16-2-580

Example: Multiple Threads 3

The demonstration class creates three threads then waits until they all finish:

```
class MultiThreadDemo {
  public static void main(String args[]) {
     new NewThread("One");
     new NewThread("Two");
     new NewThread("Three");
     try {
        Thread.sleep(10000);
     } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
     }
     System.out.println("Main thread exiting.");
  }
}
```

e-Macao-16-2-581

Using isAlive and join Methods

How can one thread know when another thread has ended?

Two methods are useful:

1) `final boolean isAlive()` - returns `true` if the thread upon which it is called is still running and `false` otherwise
2) `final void join() throws InterruptedException` – waits until the thread on which it is called terminates

e-Macao-16-2-582

Example: isAlive and join 1

Previous example improved to use `isAlive` and `join` methods.

New thread implements the `Runnable` interface:

```
class NewThread implements Runnable {
  String name;
  Thread t;

  NewThread(String threadname) {
     name = threadname;
     t = new Thread(this, name);
     System.out.println("New thread: " + t);
     t.start();
  }
```

e-Macao-16-2-583

Example: isAlive and join 2

Here is the new thread's run method:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println(name + ": " + i);
           Thread.sleep(1000);
        }
     } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
     }
     System.out.println(name + " exiting.");
  }
}
```

e-Macao-16-2-584

Example: isAlive and join 3

```
class DemoJoin {
  public static void main(String args[]) {
```

Creating three new threads:

```
     NewThread ob1 = new NewThread("One");
     NewThread ob2 = new NewThread("Two");
     NewThread ob3 = new NewThread("Three");
```

Checking if those threads are still alive:

```
     System.out.println(ob1.t.isAlive());
     System.out.println(ob2.t.isAlive());
     System.out.println(ob3.t.isAlive());
```

e-Macao-16-2-585

Example: isAlive and join 4

Waiting until all three threads have finished:

```
try {
   System.out.println("Waiting to finish.");
   ob1.t.join();
   ob2.t.join();
   ob3.t.join();
} catch (InterruptedException e) {
   System.out.println("Main thread Interrupted");
}
```

e-Macao-16-2-586

Example: isAlive and join 5

Testing again if the new threads are still alive:

```
      System.out.println(ob1.t.isAlive());
      System.out.println(ob2.t.isAlive());
      System.out.println(ob3.t.isAlive());

      System.out.println("Main thread exiting.");
   }
}
```

e-Macao-16-2-587

Thread Priorities

Priority is used by the scheduler to decide when each thread should run.

In theory, higher-priority thread gets more CPU than lower-priority thread and threads of equal priority should get equal access to the CPU.

In practice, the amount of CPU time that a thread gets depends on several factors besides its priority.

e-Macao-16-2-588

Setting and Checking Priorities

Setting thread's priority:

```
final void setPriority(int level)
```

where level specifies the new priority setting between:

1) `MIN_PRIORITY` (1)
2) `MAX_PRIORITY` (10)
3) `NORM_PRIORITY` (5)

Obtain the current priority setting:

```
final int getPriority()
```

e-Macao-16-2-589

Example: Priorities 1

A new thread class with `click` and `running` variables:

```
class Clicker implements Runnable {
  int click = 0;
  Thread t;
  private volatile boolean running = true;
```

A new thread is created, its priority initialised:

```
  public Clicker(int p) {
     t = new Thread(this);
     t.setPriority(p);
  }
```

e-Macao-16-2-590

Example: Priorities 2

When running, `click` is incremented. When stopped, `running` is `false`:

```
  public void run() {
     while (running) {
        click++;
     }
  }
  public void stop() {
     running = false;
  }

  public void start() {
     t.start();
  }
}
```

e-Macao-16-2-591

Example: Priorities 3

```
class HiLoPri {
  public static void main(String args[]) {
```

The main thread is set at the highest priority, the new threads at two above and two below the normal priority:

```
     Thread.currentThread().
        setPriority(Thread.MAX_PRIORITY);
     clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
     clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
```

e-Macao-16-2-592

Example: Priorities 4

The threads are started and allowed to run for 10 seconds:

```
     lo.start();
     hi.start();
     try {
        Thread.sleep(10000);
     } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
     }
```

e-Macao-16-2-593

Example: Priorities 5

After 10 seconds, both threads are stopped and click variables printed:

```
      lo.stop();
      hi.stop();
      try {
         hi.t.join();
         lo.t.join();
      } catch (InterruptedException e) {
         System.out.println("InterruptedException");
      }

      System.out.println("Low-priority: " + lo.click);
      System.out.println("High-priority: " + hi.click);
   }
}
```

e-Macao-16-2-594

Volatile Variable

The `volatile` keyword is used to declare the `running` variable:

```
  private volatile boolean running = true;
```

This is to ensure that the value of `running` is examined at each iteration of:

```
  while (running) {
     click++;
  }
```

Otherwise, Java is free to optimize the loop in such a way that a local copy of `running` is created. The use of `volatile` prevents this optimization.

e-Macao-16-2-595

Synchronization

When several threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

This way is called synchronization.

Synchronization uses the concept of monitors:

1) only one thread can enter a monitor at any one time
2) other threads have to wait until the threat exits the monitor

Java implements synchronization in two ways: through the synchronized methods and through the synchronized statement.

e-Macao-16-2-596

Synchronized Method

All objects have their own implicit monitor associated with them.

To enter an object's monitor, call this object's `synchronized` method.

While a thread is inside a monitor, all threads that try to call this or any other `synchronized` method on this object have to wait.

To exit the monitor, it is enough to return from the `synchronized` method.

Consider first an example without synchronization…

e-Macao-16-2-597

Example: No Synchronization 1

The `call` method tries to print the message string inside brackets, pausing the current thread for one second in the middle:

```
class Callme {
  void call(String msg) {
     System.out.print("[" + msg);
     try {
        Thread.sleep(1000);
     } catch(InterruptedException e) {
        System.out.println("Interrupted");
     }
     System.out.println("]");
  }
}
```

e-Macao-16-2-598

Example: No Synchronization 2

`Caller` constructor obtains references to the `Callme` object and `String`, stores them in the `target` and `msg` variables, then creates a new thread:

```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
     target = targ;
     msg = s;
     t = new Thread(this);
     t.start();
  }
```

e-Macao-16-2-599

Example: No Synchronization 3

The Caller's `run` method calls the `call` method on the `target` instance of `Calllme`, passing in the `msg` string:

```
  public void run() {
     target.call(msg);
  }
}
```

e-Macao-16-2-600

Example: No Synchronization 4

`Synch` class creates a single instance of `Callme` and three of `Caller`, each with a message. The `Callme` instance is passed to each `Caller`:

```
class Synch {
  public static void main(String args[]) {
     Callme target = new Callme();
     Caller ob1 = new Caller(target, "Hello");
     Caller ob2 = new Caller(target, "Synchronized");
     Caller ob3 = new Caller(target, "World");
```

e-Macao-16-2-601

Example: No Synchronization 5

Waiting for all three threads to finish:

```
      try {
         ob1.t.join();
         ob2.t.join();
         ob3.t.join();
      } catch(InterruptedException e) {
         System.out.println("Interrupted");
      }
   }
}
```

e-Macao-16-2-602

No Synchronization

Output from the earlier program:

```
[Hello[Synchronized[World]
]
]
```

By pausing for one second, the call method allows execution to switch to another thread. A mix-up of the outputs from of the three message strings.

In this program, nothing exists to stop all three threads from calling the same method on the same object at the same time.

e-Macao-16-2-603

Synchronized Method

To fix the earlier program, we must serialize the access to `call`:

```
class Callme {
   synchronized void call(String msg) {
      …
   }
}
```

This prevents other threads from entering call while another thread is using it. The output result of the program is now as follows:

```
[Hello]
[Synchronized]
[World]
```

e-Macao-16-2-604

Synchronized Statement

How to synchronize access to instances of a class that was not designed for multithreading and we have no access to its source code?

Put calls to the methods of this class inside the `synchronized` block:

```
synchronized(object) {
   …
}
```

This ensures that a call to a method that is a member of the `object` occurs only after the current thread has successfully entered the `object`'s monitor.

e-Macao-16-2-605

Example: Synchronized 1

Now the `call` method is not modified by `synchronized`:

```
class Callme {
  void call(String msg) {
     System.out.print("[" + msg);
     try {
        Thread.sleep(1000);
     } catch (InterruptedException e) {
        System.out.println("Interrupted");
     }
     System.out.println("]");
  }
}
```

e-Macao-16-2-606

Example: Synchronized 2

```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
     target = targ;
     msg = s;
     t = new Thread(this);
     t.start();
  }
```

e-Macao-16-2-607

Example: Synchronized 3

The `Caller`'s `run` method uses the `synchronized` statement to include the call the `target`'s `call` method:

```
  public void run() {
     synchronized(target) {
        target.call(msg);
     }
  }
}
```

e-Macao-16-2-608

Example: Synchronized 4

```
class Synch1 {
  public static void main(String args[]) {
     Callme target = new Callme();
     Caller ob1 = new Caller(target, "Hello");
     Caller ob2 = new Caller(target, "Synchronized");
     Caller ob3 = new Caller(target, "World");
     try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
     } catch(InterruptedException e) {
        System.out.println("Interrupted");
     }
  }
}
```

e-Macao-16-2-609

Inter-Thread Communication

Inter-thread communication relies on three methods in the `Object` class:

1) `final void wait() throws InterruptedException`
   tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.

2) `final void notify()`
   wakes up the first thread that called `wait()` on the same object

3) `final void notifyAll()`
   wakes up all the threads that called `wait()` on the same object; the highest-priority thread will run first.

All three must be called from within a `synchronized` context.

e-Macao-16-2-610

Queuing Problem

Consider the classic queuing problem where one thread (producer) is producing some data and another (consumer) is consuming this data:

1) producer should not overrun the consumer with data
2) consumer should not consume the same data many times

We consider two solutions:

1) incorrect with `synchronized` only
2) correct with synchronized and `wait`/`notify`

e-Macao-16-2-611

Example: Incorrect Queue 1

The one-place queue class `Q`, with the variable `n` and methods `get` and `put`. Both methods are `synchronized`:

```
class Q {
  int n;

  synchronized int get() {
     System.out.println("Got: " + n);
     return n;
  }
  synchronized void put(int n) {
     this.n = n;
     System.out.println("Put: " + n);
  }
}
```

e-Macao-16-2-612

Example: Incorrect Queue 2

`Producer` creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
     this.q = q;
     new Thread(this, "Producer").start();
  }
  public void run() {
     int i = 0;
     while(true) {
        q.put(i++);
     }
  }
}
```

e-Macao-16-2-613

Example: Incorrect Queue 3

`Consumer` creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
     this.q = q;
     new Thread(this, "Consumer").start();
  }
  public void run() {
     while(true) {
        q.get();
     }
  }
}
```

e-Macao-16-2-614

Example: Incorrect Queue 4

The `PC` class first creates a single `Queue` instance `q`, then creates a `Producer` and `Consumer` that share this `q`:

```
class PC {
  public static void main(String args[]) {
     Q q = new Q();
     new Producer(q);
     new Consumer(q);
     System.out.println("Press Control-C to stop.");
  }
}
```

e-Macao-16-2-615

Why Incorrect?

Here is the output:

```
Put: 1
Got: 1
Got: 1
Put: 2
Put: 3
Get: 3
…
```

Nothing stops the producer from overrunning the consumer, nor the consumer from consuming the same data twice.

e-Macao-16-2-616

Example: Corrected Queue 1

The correct producer-consumer system uses `wait` and `notify` to synchronize the behavior of the producer and consumer.

The queue class introduces the additional `boolean` variable `valueSet` used by the `get` and `put` methods:

```
class Q {
  int n;
  boolean valueSet = false;
```

e-Macao-16-2-617

Example: Corrected Queue 2

Inside `get`, `wait` is called to suspend the execution of `Consumer` until `Producer` notifies that some data is ready:

```
synchronized int get() {
   if (!valueSet)
      try {
         wait();
      }
      catch(InterruptedException e) {
         System.out.println("InterruptedException");
      }
```

e-Macao-16-2-618

Example: Corrected Queue 3

After the data has been obtained, `get` calls `notify` to tell `Producer` that it can put more data on the queue:

```
      System.out.println("Got: " + n);
      valueSet = false;
      notify();
      return n;
   }
```

e-Macao-16-2-619

Example: Corrected Queue 4

Inside `put`, `wait` is called to suspend the execution of `Producer` until `Consumer` has removed the item from the queue:

```
synchronized void put(int n) {
   if (valueSet)
      try {
         wait();
      } catch(InterruptedException e) {
         System.out.println("InterruptedException");
      }
```

e-Macao-16-2-620

Example: Corrected Queue 5

After the next item of data is put in the queue, `put` calls `notify` to tell `Consumer` that it can remove this item:

```
      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
   }
}
```

e-Macao-16-2-621

Example: Corrected Queue 6

`Producer` creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
     this.q = q;
     new Thread(this, "Producer").start();
  }
  public void run() {
     int i = 0;
     while(true) {
        q.put(i++);
     }
  }
}
```

e-Macao-16-2-622

Example: Corrected Queue 7

`Consumer` creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
     this.q = q;
     new Thread(this, "Consumer").start();
  }
  public void run() {
     while(true) {
        q.get();
     }
  }
}
```

e-Macao-16-2-623

Example: Corrected Queue 8

The `PCFixed` class first creates a single `Queue` instance `q`, then creates a `Producer` and `Consumer` that share this `q`:

```
class PCFixed {
  public static void main(String args[]) {
     Q q = new Q();
     new Producer(q);
     new Consumer(q);
     System.out.println("Press Control-C to stop.");
  }
}
```

e-Macao-16-2-624

Deadlock

Multi-threading and synchronization create the danger of deadlock.

Deadlock: a circular dependency on a pair of synchronized objects.

Suppose that:
1) one thread enters the monitor on object `X`
2) another thread enters the monitor on object `Y`
3) the first thread tries to call a synchronized method on object `Y`
4) the second thread tries to call a synchronized method on object `X`

The result: the threads wait forever – deadlock.

e-Macao-16-2-625

Example: Deadlock 1

Class `A` contains the `foo` method which takes an instance `b` of class `B` as a parameter. It pauses briefly before trying to call the `b`'s `last` method:

```
class A {
  synchronized void foo(B b) {
     String name = Thread.currentThread().getName();
     System.out.println(name + " entered A.foo");
     try {
        Thread.sleep(1000);
     } catch(Exception e) {
        System.out.println("A Interrupted");
     }
     System.out.println(name + " trying B.last()");
     b.last();
  }
```

e-Macao-16-2-626

Example: Deadlock 2

Class `A` also contains the `synchronized` method `last`:

```
  synchronized void last() {
     System.out.println("Inside A.last");
  }
}
```

e-Macao-16-2-627

Example: Deadlock 3

Class `B` contains the `bar` method which takes an instance `a` of class `A` as a parameter. It pauses briefly before trying to call the `a`'s `last` method:

```
class B {
  synchronized void bar(A a) {
     String name = Thread.currentThread().getName();
     System.out.println(name + " entered B.bar");
     try {
        Thread.sleep(1000);
     } catch(Exception e) {
        System.out.println("B Interrupted");
     }
     System.out.println(name + " trying A.last()");
     a.last();
  }
```

e-Macao-16-2-628

Example: Deadlock 4

Class `B` also contains the `synchronized` method `last`:

```
  synchronized void last() {
     System.out.println("Inside A.last");
  }
}
```

e-Macao-16-2-629

Example: Deadlock 5

The main `Deadlock` class creates the instances `a` of `A` and `b` of `B`:

```
class Deadlock implements Runnable {

  A a = new A();
  B b = new B();
```

e-Macao-16-2-630

Example: Deadlock 6

The constructor creates and starts a new thread, and creates a lock on the `a` object in the `main` thread (running `foo` on `a`) with `b` passed as a parameter:

```
  Deadlock() {
     Thread.currentThread().setName("MainThread");
     Thread t = new Thread(this, "RacingThread");
     t.start();
     a.foo(b);
     System.out.println("Back in main thread");
  }
```

e-Macao-16-2-631

Example: Deadlock 7

The run method creates a lock on the `b` object in the new thread (running `bar` on `b`) with `a` passed as a parameter:

```
  public void run() {
     b.bar(a);
     System.out.println("Back in other thread");
  }
```

Create a new `Deadlock` instance:

```
  public static void main(String args[]) {
     new Deadlock();
  }
}
```

e-Macao-16-2-632

Deadlock Reached

Program output:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

`RacingThread` owns the monitor on `b` while waiting for the monitor on `a`.
`MainThread` owns the monitor on `a` while it is waiting for the monitor on `b`.

The program deadlocks!

e-Macao-16-2-633

Suspending/Resuming Threads

Thread management should use the `run` method to check periodically whether the thread should suspend, resume or stop its own execution.

This is usually accomplished through a flag variable that indicates the execution state of a thread, e.g.

1) `running` – the thread should continue executing
2) `suspend` – the thread must pause
3) `stop` – the thread must terminate

e-Macao-16-2-634

Example: Suspending/Resuming 1

`NewThread` class contains the `boolean` variable `suspendFlag` to control the execution of a thread, initialized to `false`:

```
class NewThread implements Runnable {
  String name;
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
     name = threadname;
     t = new Thread(this, name);
     System.out.println("New thread: " + t);
     suspendFlag = false;
     t.start();
  }
```

e-Macao-16-2-635

Example: Suspending/Resuming 2

The `run` method contains the `synchronized` statement that checks `suspendFlag`. If `true`, the `wait` method is called.

```
  public void run() {
     try {
        for (int i = 15; i > 0; i--) {
           System.out.println(name + ": " + i);
           Thread.sleep(200);
           synchronized(this) {
              while(suspendFlag)    wait();
           }
        }
     }
```

e-Macao-16-2-636

Example: Suspending/Resuming 3

```
       catch (InterruptedException e) {
         System.out.println(name + " interrupted.");
       }
       System.out.println(name + " exiting.");
  }
```

e-Macao-16-2-637

Example: Suspending/Resuming 4

The `mysuspend` method sets `suspendFlag` to true:

```
  void mysuspend() {
     suspendFlag = true;
  }
```

The `myresume` method sets `suspendFlag` to `false` and invokes `notify` to wake up the thread:

```
  synchronized void myresume() {
     suspendFlag = false;
     notify();
  }
}
```

e-Macao-16-2-638

Example: Suspending/Resuming 5

`SuspendResume` class creates two instances `ob1` and `ob2` of `NewThread`, therefore two new threads, through its `main` method:

```
class SuspendResume {

  public static void main(String args[]) {
     NewThread ob1 = new NewThread("One");
     NewThread ob2 = new NewThread("Two");
```

The two threads are kept running, then suspended, then resumed from the main thread:

e-Macao-16-2-639

Example: Suspending/Resuming 6

```
     try {
        Thread.sleep(1000);
        ob1.mysuspend();
        System.out.println("Suspending thread One");
        Thread.sleep(1000);
        ob1.myresume();
        System.out.println("Resuming thread One");
        ob2.mysuspend();
        System.out.println("Suspending thread Two");
        Thread.sleep(1000);
        ob2.myresume();
        System.out.println("Resuming thread Two");
     } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
     }
```

e-Macao-16-2-640

Example: Suspending/Resuming 7

The main thread waits for the two child threads to finish, then finishes itself:

```
     try {
        System.out.println("Waiting to finish.");
        ob1.t.join();
        ob2.t.join();
     } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
     }
     System.out.println("Main thread exiting.");
  }
}
```

e-Macao-16-2-641

The Last Word on Multi-Threading

Multi-threading is a powerful tool to writing efficient programs.

When you have two subsystems within a program that can execute concurrently, make them individual threads.

However, creating too many threads can actually degrade the performance of your program because of the cost of context switching.

e-Macao-16-2-642

Exercise: Multi-Threading 1

1) Create a new main class called `MultiThread`
2) Create a new class called `MemoryThread`. This class should implement the interface `Runnable` so that it can be run as a thread. This Thread will monitor the memory usage on the system.
   a) Add `void start()`, `stop()` and `run()` methods.
   b) The `run()` method should print to the screen every 5 seconds the amount of memory presently being used. This can be done using these two lines of code:
```
Runtime r = Runtime.getRuntime();
long memoryUsed = r.totalMemory()- r.freeMemory();
```
3) Create a new class called `ClockThread`. This class should implement the interface `Runnable` so that it can be run as a thread. This Thread will monitor what the time is.
   a) The constructor for `ClockThread` should take an argument that sets the time (in seconds) that this thread will run for.
   b) Add `void start()`, `stop()` and `run()` methods.

e-Macao-16-2-643

Exercise: Multi-Threading 2

   c) The `run()` method should print the current time to the screen every 5 seconds. This can be done using the built in `Date` class.
```
import java.util.Date;
Date timeNow = new Date();
System.out.println(timeNow.toString());
```
   The `run()` method should stop when the elapsed time set in the constructor has been reached.
4) In the `main()` method of `MultiThread` create the two thread objects and start them; using the *`start()`* method.
5) Add a `while` loop, that will print the number of active Threads every one second. A one second pause can be implemented as follows:
```
while (true) {
    Thread.sleep(1000);
}
```
6) The number of threads can be monitored using the `ThreadGroup` class as follows:
```
int numThreads =
Thread.currentThread().getThreadGroup().activeCount();
```

e-Macao-16-2-644

Exercise: Multi-Threading 3

7) This while loop should terminate after one minute and the two threads stopped by invoking their `stop()` methods.
8) When printing to the screen - the Threads names should be appended so that it is clear from which process the data is originating.
9) The `MemoryThread` thread should be assigned a maximum priority and the `ClockThread` thread a minimum priority.

A.4. Horizontal Libraries

Horizontal Libraries

e-Macao-16-2-646

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-647

Horizontal Libraries

Horizontal libraries are APIs that are used across the language.

Java provides a rich set of horizontal libraries:

a) String handling – for handling sequence of characters
b) event handling – helps to handle how programs respond to actions generated by the user.
c) Object collection – handling a group of objects

A.4.1. String Handling

String Handling

e-Macao-16-2-649

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
6) summary

e-Macao-16-2-650

String Object

String is a sequence of characters.

Unlike many other programming languages that implements string as character arrays, Java implements strings as object of type `String`.

This provides a full compliment of features that make string handling convenient. For example, Java String has methods to:

1) compare two strings
2) Search for a substring
3) Concatenate two strings and
4) Change the case of letters within a string
5) Can be constructed a number of ways making it easy to obtain a string when needed

e-Macao-16-2-651

String is Immutable

Once a String Object has been created, you cannot change the characters that comprise that string.

This is not a restriction. It means each time you need an altered version of an existing string, a new string object is created that contains the modification.

It is more efficient to implement immutable strings than changeable ones.

To solve this, Java provides a companion class to `String` called `StringBuffer.`

`StringBuffer` objects can be modified after they are created.

e-Macao-16-2-652

String Constructors 1

String supports several constrictors:

1) to create an empty String

```
String s = new String();
```

2) to create a string that have initial values

```
String(char chars[])
```

Example:

```
char chars[] = {'a','b','c'};
String s = new String(chars);
```

e-Macao-16-2-653

String Constructors 2

3) to create a string as a subrange of a character array

```
String(char chars[], int startindex, int numchars)
```

Here, `startindex` specifies the index at which the subrange begins, and `numChars` specifies the number of characters to use.

Example:

```
char chars[] = {'a','b','c','d','e','f'};
String s = new String(chars,2,3);
```

This initializes `s` with the characters `cde`.

e-Macao-16-2-654

String Constructors 3

4) to construct a String object that contains the same character sequence as another String object

```
String(String obj)
```

Example

```
class MakeString {
  public static void main(String args[]) {
     char c[] = {'J', 'a', 'v', 'a'};
     String s1 = new String(c);
     String s2 = new String(s1);
     System.out.println(s1);
     System.out.println(s2);
  }
}
```

e-Macao-16-2-655

String Length

The length of a string is the number of characters that it contains.

To obtain this value call the `length()` method:

```
int length()
```

The following fragment prints "3", since there are three characters in the string `s`.

```
char chars[] = {'a','b','c'};
String s = new String(chars);
System.out.println(s.lenght());
```

e-Macao-16-2-656

String Operations

Strings are a common and important part of programming.

Java provides several string operations within the syntax of the language.

These operations include:

1) automatic creation of new `String` instances from literals
2) concatenation of multiple `String` objects using the `+` operator
3) conversion of other data types to a string representation

There are explicit methods to perform all these functions, but Java does them automatically for the convenience of the programmer and to add clarity.

e-Macao-16-2-657

String Literals

Using String literals is an easier way of creating Strings Objects.

For each String literal, Java automatically constructs a `String` object.
You can use String literal to initialize a `String` object.

Example:

```
char chars[] = {'a','b','c'};
String s1 = new String(chars);
```

Using String literals

```
String s2 = "abc";
```

e-Macao-16-2-658

String Concatenation

Java does not allow operations to be applied to a `String` object.

The one exception to this rule is the `+` operator, which concatenates two strings producing a string object as a result.

With this you ca chain together a series of `+` operations.

Example:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

e-Macao-16-2-659

Concatenation Usage

One practical use is found when you are creating very long strings.

Instead of letting long strings wrap around your source code, you can break them into smaller pieces, using the `+` to concatenate them.

Example:

```
class ConCat {
      public static void main(String args[]) {
            String longStr = "This could have been " +
            "a very long line that would have " +
            "wrapped around.  But string concatenation "
            + "prevents this.";
            System.out.println(longStr);
  }
}
```

e-Macao-16-2-660

Concatenation & Other Data Type

You can concatenate Strings with other data types.

Example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

The compiler will convert an operand to its string equivalent whenever the other operand of the + ia an instance of `String`.

Be careful:

```
String s = "Four:" + 2 + 2;
System.out.println(s);
```

Prints `Four:22` rather than `Four: 4`.

To achieve the desired result, bracket has to be used.

```
String s = "Four:" + (2 + 2);
```

e-Macao-16-2-661

Conversion and toString() Method

When Java converts data into its string representation during concatenation, it does so by calling one of its overloaded `valueOf()` method defined by `String`.

`valueOf()` is overloaded for

1) simple types – which returns a string that contains the human – readable equivalent of the value with which it is called.
2) object types – which calls the toString() method of the object.

e-Macao-16-2-662

Example: toString() Method 1

Override toString() for Box class

```
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

e-Macao-16-2-663

Example: toString() Method 2

```
    public String toString() {
        return "Dimensions are " + width + " by " +
                depth + " by " + height + ".";
  }
}
```

e-Macao-16-2-664

Example: toString() Method 3

```
class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

`Box's toString()` method is automatically invoked when a `Box` object is used in a concatenation expression or in a Call to `println()`.

e-Macao-16-2-665

Character Extraction

String class provides a number of ways in which characters can be extracted from a String object.

String index begin at zero.

These extraction methods are:

1) charAt()
2) getChars()
3) getBytes()
4) toCharArray()

Each will considered.

e-Macao-16-2-666

charAt()

To extract a single character from a String.

General form:

```
char charAt(int where)
```

`where` is the index of the character you want to obtain. The value of `where` must be nonnegative and specify alocation within the string.

Example:

```
char ch;
ch = "abc".charAt(1);
```

Assigns a value of "`b`" to `ch`.

e-Macao-16-2-667

getChars()

Used to extract more than one character at a time.

General form:

```
void getChars(int sourceStart, int sourceEnd, char[]
                                target, int targetStart)
```

`sourceStart` – specifies the index of the beginning of the substring

`sourceEnd` – specifies an index that is one past the end of the desired subString

`target` – is the array that will receive the characters

`targetStart` – is the index within target at which the subString will be copied is passed in this parameter

e-Macao-16-2-668

getChars()

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "This is a demo of the getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];
    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

e-Macao-16-2-669

getBytes()

Alternative to getChars() that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform.

General form:

```
byte[] getBytes()
```

Usage:

Most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

For example, most internet protocols and text file formats use 8-bit ASCII for all text interchange.

e-Macao-16-2-670

toCharArray()

To convert all the characters in a String object into character array.

It returns an array of characters for the entire string.

General form:

```
char[] toCharArray()
```

It is provided as a convenience, since it is possible to use `getChars()` to achieve the same result.

e-Macao-16-2-671

String Comparison

The String class includes several methods that compare strings or substrings within strings.

They are:

1) `equals() and equalsIgnoreCase()`
2) `regionMatches()`
3) `startWith() and endsWith()`
4) `equals() Versus ==`
5) `comapreTo()`

Each will be considered.

e-Macao-16-2-672

equals()

To compare two Strings for equality, use equals()

General form:

```
boolean equals(Object str)
```

`str` is the `String` object being compared with the invoking String object.

It returns true if the string contain the same character in the same order, and false otherwise.

The comparison is case-sensitive.

e-Macao-16-2-673

equalsIgnoreCase()

To perform operations that ignores case differences.

When it compares two strings, it considers `A-Z` as the same as `a-z`.

General form:

```
boolean equalsIgnoreCase(Object str)
```

`str` is the `String` object being compared with the invoking String object.

It returns `true` if the string contain the same character in the same order, and `false` otherwise.

The comparison is case-sensitive.

e-Macao-16-2-674

equals and equalsIgnoreCase() 1

Example:

```
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                                         s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                                         s1.equals(s3));
```

e-Macao-16-2-675

equals and equalsIgnoreCase() 2

```
    System.out.println(s1 + " equals " + s4 + " -> " +
                                         s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 +
                        " -> " + s1.equalsIgnoreCase(s4));
  }
}
```

e-Macao-16-2-676

regionMatches() 1

Compares a specific region inside a string with another specific region in another string.

There is an overloaded form that allows you to ignore case in such comparison.

General form:

```
boolean regionMatches(int startindex, String str2,
            int str2StartIndex, int numChars)


boolean regionMatches(boolean ignoreCase, int
      startindex, String str2, int str2StartIndex,
                                    int numChars)
```

e-Macao-16-2-677

regionMatches() 2

In both versions, `startIndex` specifies the index at which the region begins within the invoking String object.

The string object being compared is specified as `str`.

The index at which the comparison will start within `str2` is specified by `str2StartIndex`.

The length of the substring being comapred is passed in `numChars`.

In the second version, if the `ignoreCase` is `true`, the case of the characters is ignored. Otherwise case is significant.

e-Macao-16-2-678

startsWith() and endsWith() 1

`String` defines two routines that are more or less the specialised forms of regionMatches().

The `startsWith()` method determines whether a given string begins with a specified string.

Conversely, `endsWith()` method determines whether the string in question ends with a specified string.

General form:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

`str` is the `String` being tested. If the string matches, true is returned, otherwise false is returned.

e-Macao-16-2-679

startsWith() and endsWith() 2

Example:

```
"Foobar".endsWith("bar");
```

and

```
"Foobar".startsWith("Foo");
```

are both `true`.

e-Macao-16-2-680

startsWith() and endsWith() 3

A second form of `startsWith()`, let you specify a starting point:

General form:

```
boolean startWith(String str, int startIndex)
```

Where `startIndex` specifies the index into the invoking string at which point the search will begin.

Example:

```
"Foobar".startsWith("bar", 3);
```

returns `true`.

e-Macao-16-2-681

equals() Versus ==

It is important to understand that the two metyhod performs different functions.

1) `equals()` method compares the characters inside a `String` object.
2) `==` operator compares two object references to see whether they refer to the same instance.

e-Macao-16-2-682

Example: equals() Versus ==

```
class EqualsNotEqualTo {
  public static void main(String args[]) {
      String s1 = "Hello";
      String s2 = new String(s1);
      System.out.print(s1 + " equals " + s2 + " -> ");
      System.out.println(s1.equals(s2));
      System.out.print(s1 + " == " + s2 + " -> ")
      System.out.println((s1 == s2));
  }
}
```

e-Macao-16-2-683

compareTo() 1

It is not enough to know that two Strings are identical. You need to know which is less than, equal to, or greater than the next.

A string is less than the another if it comes before the other in the dictionary order.

A string is greater than the another if it comes after the other in the dictionary order.

The `String` method `compareTo()` serves this purpose.

e-Macao-16-2-684

compareTo() 2

General form:

```
int compareTo(String str)
```

`str` is the string that is being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

Less than zero	The invoking string is less than `str`
Greater than zero	The invoking string is greater than `str`
Zero	The two strings are equal

e-Macao-16-2-685

Example: compareTo()

```
class SortString {
      static String arr[] =
      {"Now", "is", "the", "time", "for", "all", "good,"
"men","to", "come", "to", "the", "aid", "of","their",
"country"};
      public static void main(String args[]) {
            for(int j = 0; j < arr.length; j++) {
                  for(int i = j + 1; i < arr.length;
i++) {
                        if(arr[i].compareTo(arr[j]) < 0)
{
                              String t = arr[j];
                              arr[j] = arr[i];
                              arr[i] = t;
```

e-Macao-16-2-686

Searching String 1

String class provides two methods that allows you search a string for a specified character or substring:

1) `indexOf()` – Searches for the first occurrence of a character or substring.
2) `lastIndexOf()` – Searches for the last occurrence of a charater or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or `-1` on failure.

e-Macao-16-2-687

Searching String 2

To seach for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

To search for the first and the last occurence of a substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

Here `str` specifies the substring.

e-Macao-16-2-688

Searching String 3

You can specify a starting point for the serach using these forms:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

`startIndex` – specifies the index at which point the search begins.

For `indexOf()`, the search runs from startIndex to the end of the string.
For `lastIndexOf()`, the search runs from startIndex to zero.

e-Macao-16-2-689

Example: Searching String 1

```
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";
    System.out.println(s);
    System.out.println("indexOf(t) = " +
                                    s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                                  s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                                     s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                                   s.lastIndexOf("the"));
```

e-Macao-16-2-690

Example: Searching String 2

```
    System.out.println("indexOf(t, 10) = " +
                                    s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                                s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                               s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                               s.lastIndexOf("the", 60));
  }
}
```

e-Macao-16-2-691

Modifying a String

String object are immutable.

Whenever you want to modify a String, you must either copy it into a StringBuffer or use the following String methods,, which will construct a new copy of the string with your modification complete.

They are:

1) `subString()`
2) `concat()`
3) `replace()`
4) `trim()`

Each will be discussed.

e-Macao-16-2-692

subString() 1

You can extract a substring using subString().

It has two forms:

```
String substring(int startIndex)
```

`startIndex` specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

e-Macao-16-2-693

subString() 2

The second form allows you to specify both the beginning and ending index of the substring.

```
String substring(int startIndex, int ensIndex)
```

`startIndex` specifies the index beginning index, and

`endIndex` specifies the stopping point.

The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

Java Workshop

Gabriel Oteniya and Tomasz Janowski

UNU-IIST

The Course

1) objectives - what do we intend to achieve?
2) outline - what content will be taught?
3) resources - what teaching resources will be available?
4) organization - duration, major activities, daily schedule

Course Objectives

1) refresh and reinforce the knowledge of Java technology
2) review selected Java APIs (libraries)
3) learn best practices in developing Java applications
4) lay foundation for learning J2EE technology
5) understand why Java technology is adequate for the implementation of e-government services

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Outline: Introduction

An overview of the Java language and its runtime environment.

Main points:

1) Familiarize participants with the language's features, goals, and documentation.
2) Explains how to set up the Java runtime environment to facilitate the development and execution of Java code.
3) Describes how to use the Java technology documentation.
4) Provides a simple example on how to write, compile and run a Java application.

Outline: Language

Introduces procedural aspects of the Java language.

Main points:

1) Basic data type and variables.
2) How to work with variables and arrays.
3) Operators: arithmetic, bitwise, relational, logical, etc.
4) Control structures: selection, iteration, jumps, etc.

Outline: Object-Orientation

Presents object-oriented aspects of the Java language.

Main points:

1) How to encapsulate data and behavior within classes.
2) How to build hierarchies of classes with inheritance.
3) How to determine the behavior of objects at run-time.
4) How to group classes into packages.
5) How to introduce abstraction thorough interfaces.

Outline: Horizontal APIs

API – Application Programming Interface

Horizontal APIs are used across the language.

Presentation of selected horizontal APIs:

1) string handling
2) event handling
3) Object collections

Outline: Vertical APIs

Vertical APIs are designed to perform specific functions.

Presentation of selected vertical APIs:

1) graphical user interface
2) applets
3) input/output
4) networking

Outline: Summary

Revision of the material introduced during the course.

How this course provides a foundation for the remaining courses:

1) distributed programming
2) Java XML processing
3) Java Web Services
4) J2EE web components
5) J2EE business components

Course Resources

1) books
   a) Java 2 The Complete Reference, Herbert Schildt, Osborne, 5th edition, 2002
   b) The Java Tutorial, Sun Microsystems, http://java.sun.com/docs/books/tutorial/, 2004
   c) Thinking in Java, Bruce Ecker, 3rd edition, Prentice Hall, 2002

2) articles

   Links available from the website http://www.emacao.gov.mo.

3) tools
   a) JDK 1.5
   b) Eclipse IDE

Course Logistics

1) duration - 36 hours
2) activities - lecture (hands-on), development
3) sessions/day - morning 09:00–13:00 and afternoon 14:30–16:30
4) number of sessions - 6 morning and 6 afternoon
5) style - interactive and tutorial

Course Prerequisite

1) some experience in object-oriented programming:
   a) C++
   b) Delphi
   c) any other object-oriented language
2) basic understanding of TCP/IP networking concepts

Introduction

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Overview

Topics under this module:

1) Java origin and history
2) Java technology
3) Java language
4) Java platform
5) simple Java program
6) Java documentation
7) setting up Java environment

Java Origins

Computer language innovation and development occurs for two fundamental reasons:

1) to adapt to changing environments and uses
2) to implement improvements in the art of programming

The development of Java was driven by both in equal measures.

Many Java features are inherited from the earlier languages:

B → C → C++ → Java

Before Java: C

Designed by Dennis Ritchie in 1970s.

Before C, there was no language to reconcile: ease-of-use versus power, safety versus efficiency, rigidity versus extensibility.

BASIC, COBOL, FORTRAN, PASCAL optimized one set of traits, but not the other.

C- structured, efficient, high-level language that could replace assembly code when creating systems programs.

Designed, implemented and tested by programmers, not scientists.

Before Java: C++

Designed by Bjarne Stroustrup in 1979.

Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:

1) assembler languages
2) high-level languages
3) structured programming
4) object-oriented programming (OOP)

OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.

C++ extends C by adding object-oriented features.

Java History

Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.

The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.

With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.

Java as an “Internet version of C++”? No.

Java was not designed to replace C++, but to solve a different set of problems. There are significant practical/philosophical differences.

Java Technology

There is more to Java than the language.

Java Technology consists of:

1) Java Programming Language
2) Java Virtual Machine (JVM)
3) Java Application Programming Interfaces (APIs)

Java Language Features

1) simple
2) object-oriented
3) robust
4) multithreaded
5) architecture-neutral
6) interpreted and high-performance
7) distributed
8) dynamic
9) secure

Java Language Features 1

1) simple – Java is designed to be easy for the professional programmer to learn and use.
2) object-oriented – a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
3) robust – restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.

Java Language Features 2

4) multithreaded – supports multi-threaded programming for writing program that perform concurrent computations
5) architecture-neutral – Java Virtual Machine provides a platform-independent environment for the execution of Java bytecode
6) interpreted and high-performance – Java programs are compiled into an intermediate representation – bytecode:

   a) can be later interpreted by any JVM

   b) can be also translated into the native machine code for efficiency.

Java Language Features 3

7) distributed – Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
8) dynamic – substantial amounts of run-time type information to verify and resolve access to objects at run-time.
9) secure – programs are confined to the Java execution environment and cannot access other parts of the computer.

Execution Platform

What is an execution platform?

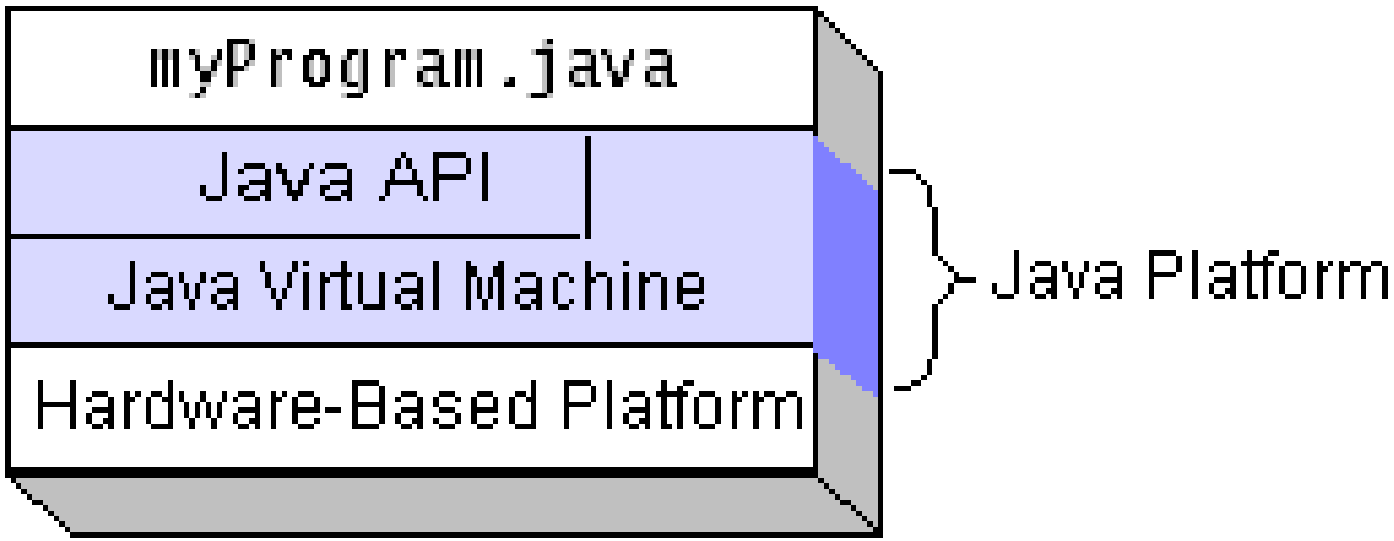
1) An execution platform is the hardware or software environment in which a program runs, e.g. Windows 2000, Linux, Solaris or MacOS.
2) Most platforms can be described as a combination of the operating system and hardware.

Java Execution Platform

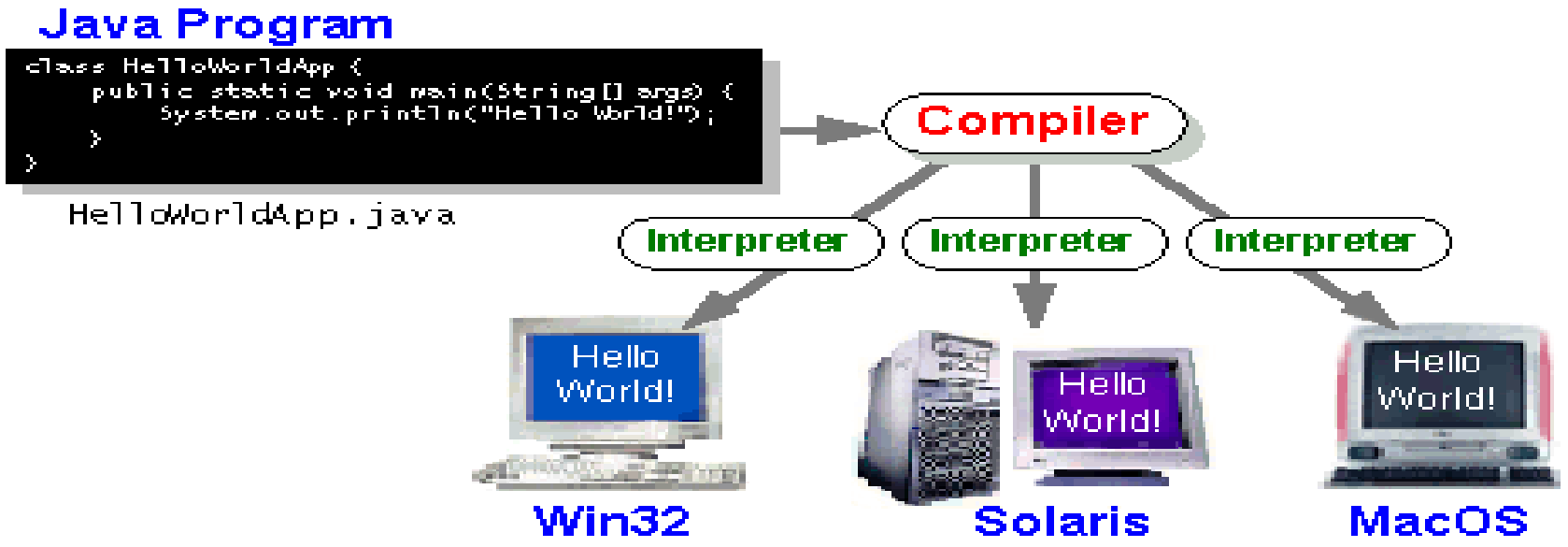
What is Java Platform?

1) A software-only platform that runs on top of other hardware-based platforms.
2) Java Platform has two components:
    a) Java Virtual Machine (JVM) – interpretation for the Java bytecode, ported onto various hardware-based platforms.
    b) The Java Application Programming Interface (Java API)

Java Execution Platform



Java Platform Independence



Java Program Execution

Java programs are both compiled and interpreted:

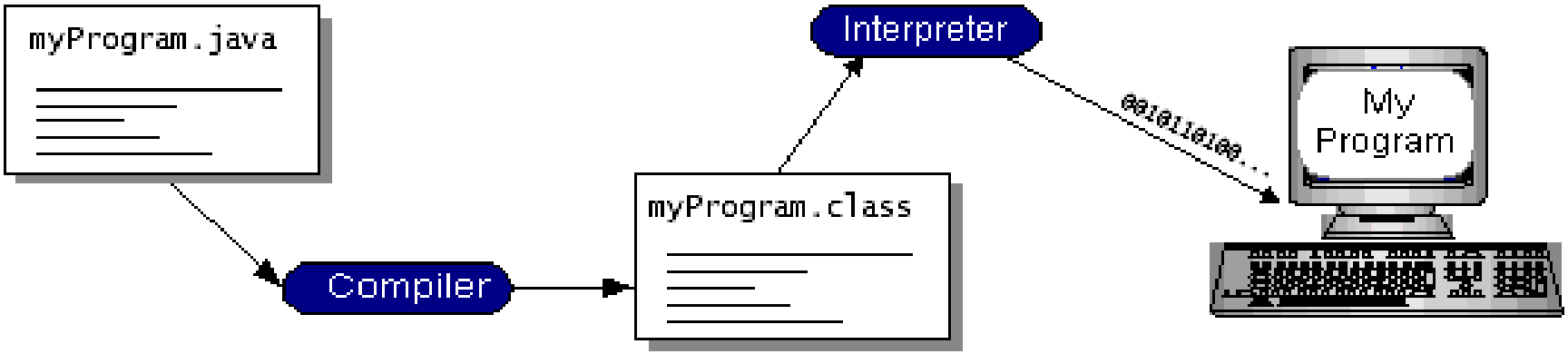
Steps:

- write the Java program
- compile the program into bytecode
- execute (interpret) the bytecode on the computer through the Java Virtual Machine

Compilation happens once.

Interpretation occurs each time the program is executed.

Java Execution Process



Java API

What is Java API?

1) a large collection of ready-made software components that provide many useful capabilities, e.g. graphical user interface
2) grouped into libraries (packages) of related classes and interfaces
3) together with JVM insulates Java programs from the hardware and operating system variations

Java Program Types

Types of Java programs:

1) applications – standalone (desktop) Java programs, executed from the command line, only need the Java Virtual Machine to run
2) applets – Java program that runs within a Java-enabled browser, invoked through a “applet” reference on a web page, dynamically downloaded to the client computer
3) servlets – Java program running on the web server, capable of responding to HTTP requests made through the network
4) etc.

Java Platform Features 1

1) essentials - objects, strings, threads, numbers, input/output, data structures, system properties, date and time, and others.
2) networking:
    1) Universal Resource Locator (URL)
    2) Transmission Control Protocol (TCP)
    3) User Datagram Protocol (UDP) sockets
    4) Internet Protocol (IP) addresses
3) internationalization - programs that can be localized for users worldwide, automatically adapting to specific locales and appropriate languages.

Java Platform Features 2

4) security – low-level and high-level security, including electronic signatures, public and private key management, access control, and certificates
5) software components – JavaBeans can plug into an existing component architecture
6) object serialization - lightweight persistence and communication, in particular using Remote Method Invocation (RMI)
7) Java Database Connectivity (JDBC) - provides uniform access to a wide range of relational databases

Java Technologies

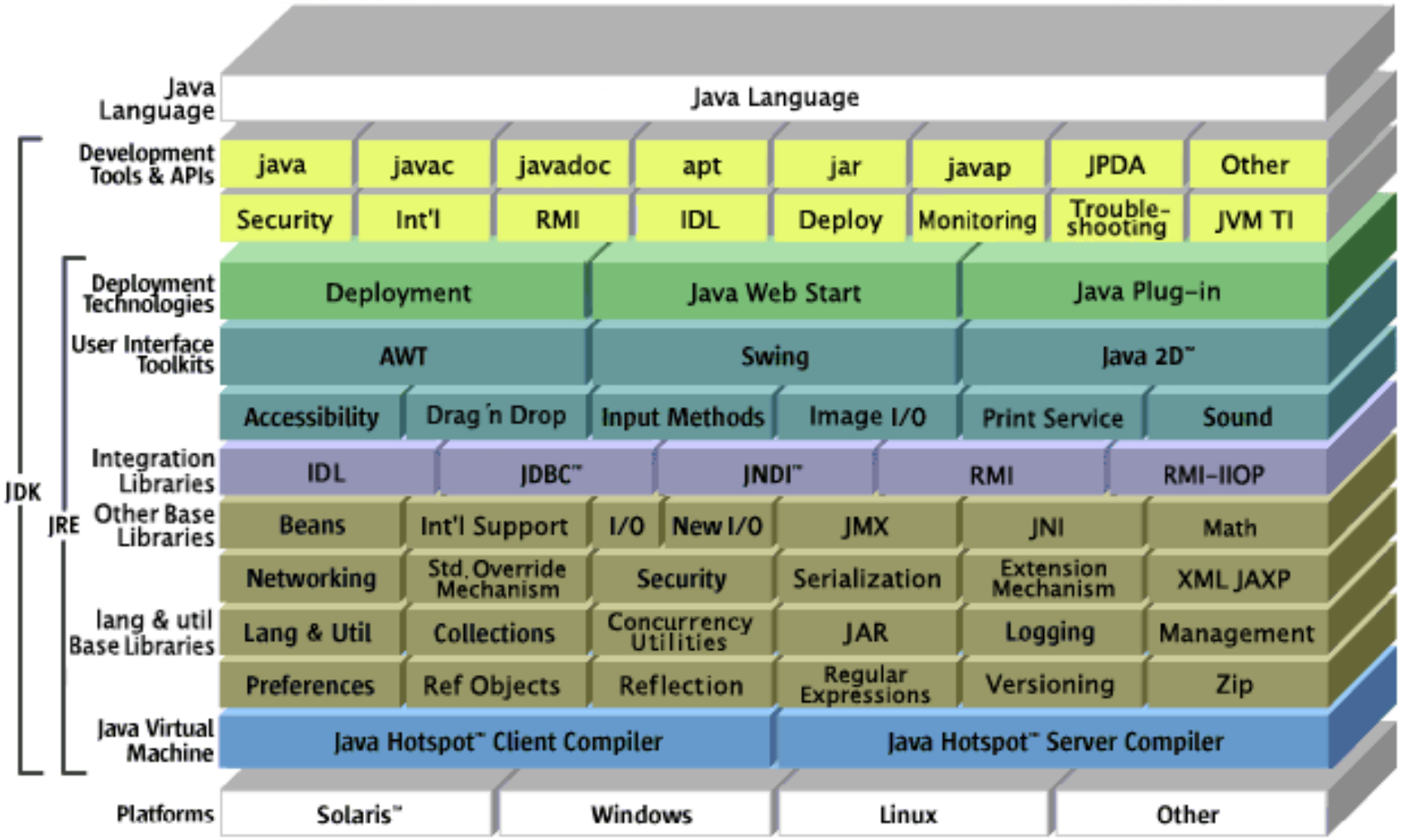
Different technologies depending on the target applications:

1) desktop applications - Java 2 Standard Edition (J2SE)
2) enterprise applications – Java 2 Enterprise Edition (J2EE)
3) mobile applications – Java 2 Mobile Edition (J2ME)
4) smart card applications – JavaCard
5) etc.

Each edition puts together a large collections of packages offering functionality needed and relevant to a given application.

The Java Virtual Machine remains essentially the same.

Java Technology: SDK



Exercise: Java Technology

1) Explain the statement “There is more to Java than the Language”.
2) Enumerate and explain Java design goals.
3) How does Java maintain a balance between Interpretation and High Performance?.
4) Java program is termed “Write once run everywhere”. Explain.
5) Why is it difficult to write viruses and malicious programs with Java?

Simple Java Program

A class to display a simple message:

```
class MyProgram {
     public static void main(String[] args) {
          System.out.println(“First Java program.");
     }
}
```

Running the Program

Type the program, save as `MyProgram.java`.

In the command line, type:

```
> dir
MyProgram.java
> javac MyProgram.java
> dir
MyProgram.java, MyProgram.class
> java MyProgram
First Java program.
```

Explaining the Process

1) creating a source file - a source file contains text written in the Java programming language, created using any text editor on any system.

2) compiling the source file – Java compiler (javac) reads the source file and translates its text into instructions that the Java interpreter can understand. These instructions are called bytecode.

3) running the compiled program – Java interpreter (java) installed takes as input the bytecode file and carries out its instructions by translating them on the fly into instructions that your computer can understand.

Java Program Explained

`MyProgram` is the name of the class:

```
class MyProgram {
```

`main` is the method with one parameter `args` and no results:

```
   public static void main(String[] args) {
```

`println` is a method in the standard `System` class:

```
      System.out.println(“First Java program.");
   }
}
```

Classes and Objects

A class is the basic building block of Java programs.

A class encapsulates:

a) data (attributes) and

b) operations on this data (methods)

and permits to create objects as its instances.

Main Method

The `main` method must be present in every Java application:

1) `public static void main(String[] args)` where:
   a) `public` means that the method can be called by any object
   b) `static` means that the method is shared by all instances
   c) `void` means that the method does not return any value
2) When the interpreter executes an application, it starts by calling its `main` method which in turn invokes other methods in this or other classes.
3) The main method accepts a single argument – a string array, which holds all command-line parameters.

Exercise: Java Program

1) Personalize the `MyProgram` program with your name so that it tells you "`Hello, my name is …`"
2) Write a program that produces the following output:

```
Welcome to e-Macao Java Workshop!
I hope you will benefit from the training.
```

3) Here is a slightly modified version of `MyProgram`:

```
class MyProgram2 {
  public void static Main(String args) {
    system.out.println("First Java Program!);
  }
}
```

The program has some errors. Fix the errors so that the program successfully compiles and runs. What were the errors?

Using API Documentation 1

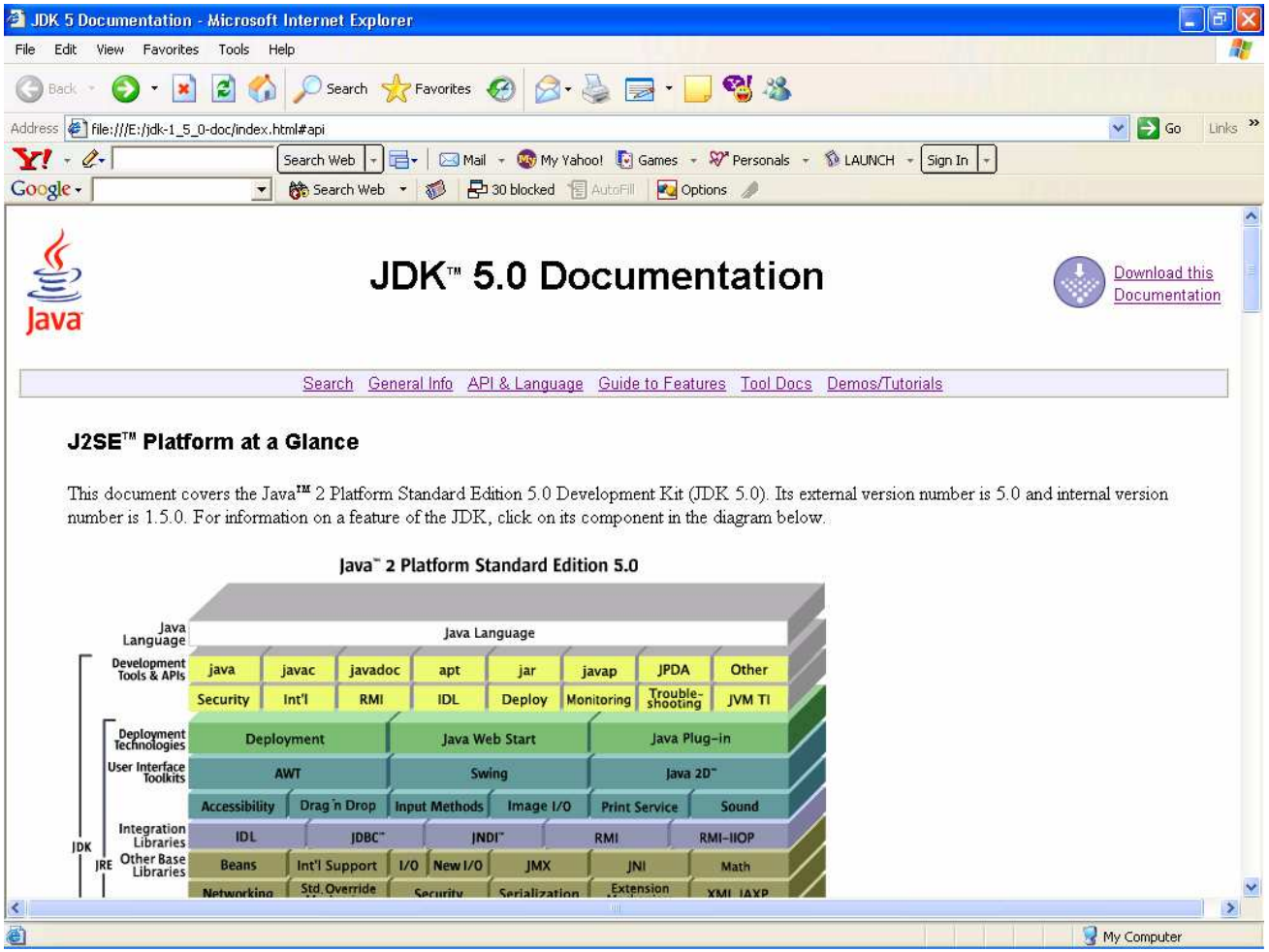
Ability to use Java API Specification is crucial for any practicing programmer. Here are the steps:

1) download the API documentation from
   http://java.sun.com/j2se/1.5.0/download.jsp

2) Extract the archive file

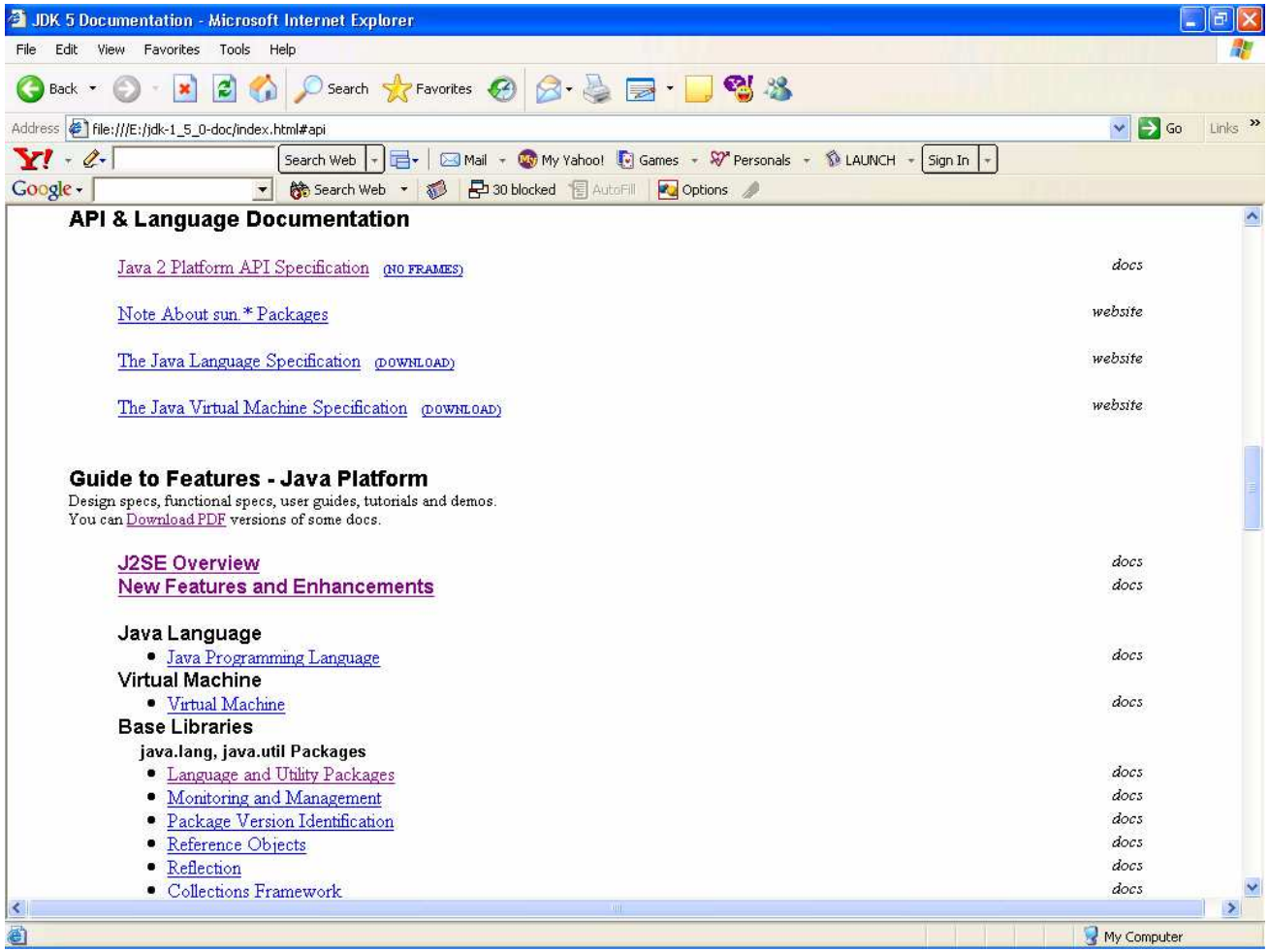
Using API Documentation 2

3) open `index.html`



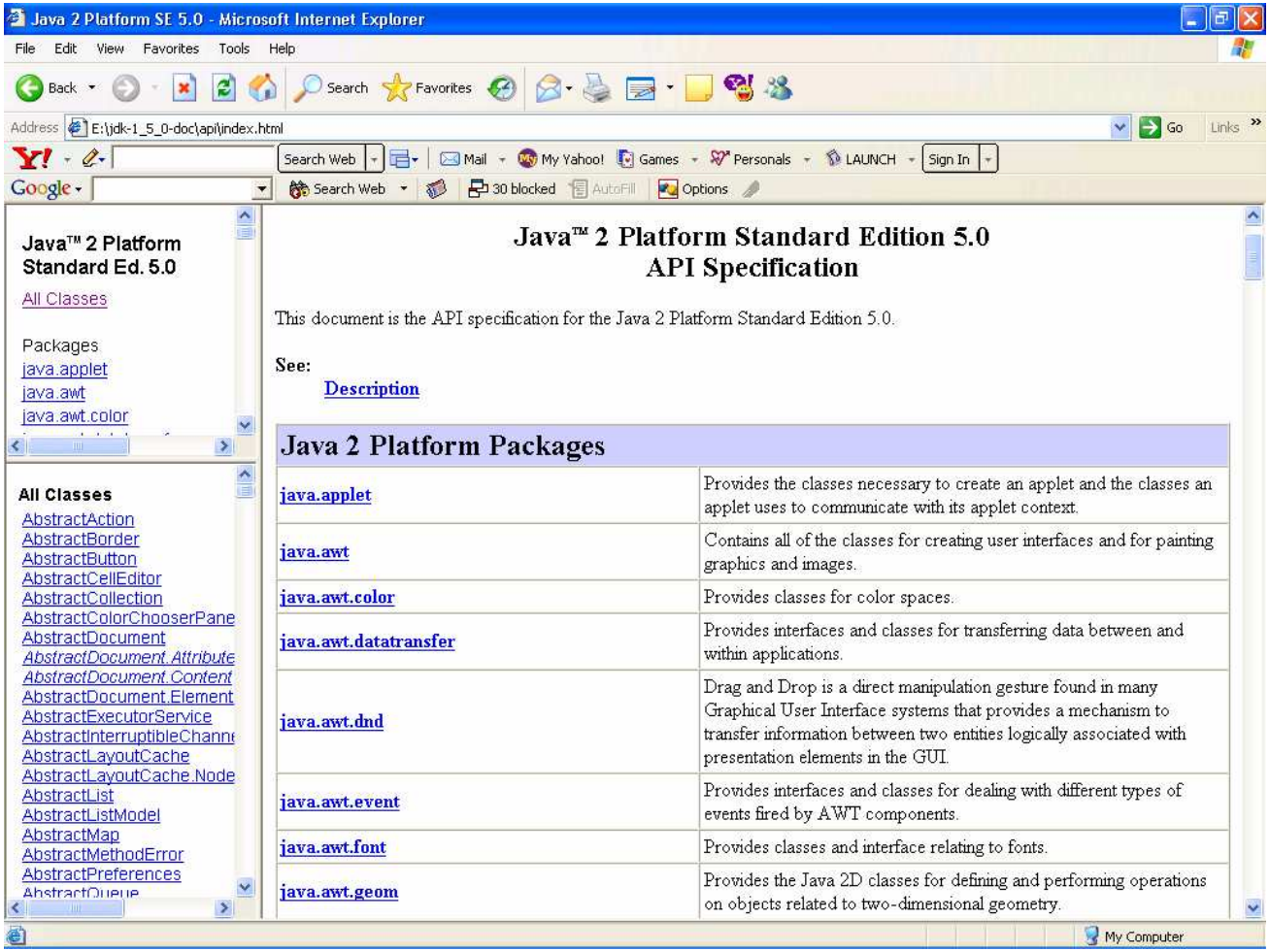
Using API Documentation 3

4) click on `API & Language`



Using API Documentation 4

5) click on `Java 2 Platform API Specification`



Java 2 Platform API Specification

The window is divided into three panes:

1) specification pane - click on any package, package pane will display all classes, interfaces, exceptions and errors that belong to that package.
2) package pane - click on any class, class pane will display all information about the class.
3) class pane – contains such information as:
    1) inheritance hierarchy
    2) implemented interfaces
    3) constructors
    4) attributes
    5) methods

Java Environment Setup 1

Setting up `JAVA_HOME` and `PATH` environment variables is necessary for the Java tools and applications to work properly from any directory.

Steps to carry out:

1) `JAVA_HOME` variable should point to the top installation directory of Java environment.
2) `PATH` should point to the `bin` directory where the Java Virtual Machine - interpreter (`java`), compiler (`javac`) and other executables are located.

Java Environment Setup 2

With more than one JVM installed, the one you wish to use must appear as the first one in the PATH variable.

Example:

1) Windows - add `%JAVA_HOME%\bin` to the beginning of the `PATH` variable.
2) Linux - include `PATH="$PATH:$JAVA_HOME/bin:."` in your `/etc/profile` or `.bashrc` files.

To test the environment, open the command window and run:

```
java -version
```

The current version number of the installed JVM should appear.

Summary: Introduction

Material covered:

1) What is the origin and history of Java?
2) What is Java Technology and its components?
3) What are the basic features of the Java language?
4) What are the basic features of the Java execution platform?
5) How to write, compile and execute a simple Java application?
6) How to use Java API documentation?
7) How to set up the Java execution environment?

Exercise: API Specification

1) Download and extract the latest JDK 5.0 Documentation.
2) Using the Documentation
   a) Locate the `java.math` package
   b) How many classes does it have?
   c) List all the classes.
   d) List their inheritance hierarchies.
   e) List their implemented interfaces.
   f) How many constructors does each have?
   g) How many fields does each have?
   h) How many methods does each have?

Language

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Language

Java is intrinsically an object-oriented language.

However, for presentation reasons explain it in two parts:

a) procedural part
b) object-oriented part

Language Overview

a) syntax – whitespaces, identifiers, comments, separators, keywords
b) types – simple types byte, short, int, long, float double, char, boolean
c) variables – declaration, initialization, scope, conversion and casting
d) arrays – declaration, initialization, one- and multi-dimensional arrays
e) operators – arithmetic, relational, conditional, shift, logical, assignment
f) control flow – branching, selection, iteration, jumps and returns

Syntax

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Java Syntax

On the most basic level, Java programs consist of:

a) whitespaces
b) identifiers
c) comments
d) literals
e) separators
f) keywords
g) operators

Each of them will be described in order.

Whitespaces

A whitespace is a space, tab or new line.

Java is a form-free language that does not require special indentation.

A program could be written like this:

```
class MyProgram {
  public static void main(String[] args) {
      System.out.println(“First Java program.");
  }
}
```

It could be also written like this:

```
class MyProgram { public static void main(String[] args)
{ System.out.println(“First Java program.");  } }
```

Identifiers

Java identifiers:

a) used for class names, method names, variable names

b) an identifier is any sequence of letters, digits, "_" or "$" characters that do not begin with a digit

c) Java is case sensitive, so `value`, `Value` and `VALUE` are all different.

Seven identifiers in this program:

```
class MyProgram {
   public static void main(String[] args) {
        System.out.println("First Java program.");
        }
}
```

Comments 1

Three kinds of comments:

1) Ignore the text between `/*` and `*/`:

```
/* text */
```

2) Documentation comment (`javadoc` tool uses this kind of comment to automatically generate software documentation):

```
/** documentation */
```

3) Ignore all text from `//` to the end of the line:

```
// text
```

Comments 2

```
/**
   *   MyProgram implements application that displays
      *      a simple message on the standard output
   device.
*/
class MyProgram {
   /* The main method of the class.*/
   public static void main(String[] args) {
            //display string
            System.out.println(“First Java program.");
   }
}
```

Literals

A literal is a constant value of certain type.

It can be used anywhere values of this type are allowed.

Examples:

a) 100

b) 98.6

c) 'X'

d) "test"

```
class MyProgram {
  public static void main(String[] args) {
      System.out.println("My first Java program.");
      }
}
```

Separators

`()`	parenthesis	lists of parameters in method definitions and invocations, precedence in expressions, type casts
`{}`	braces	block of code, class definitions, method definitions, local scope, automatically initialized arrays
`[]`	brackets	declaring array types, referring to array values
`;`	semicolon	terminating statements, chain statements inside the “for” statement
`,`	comma	separating multiple identifiers in a variable declaration
`.`	period	separate package names from subpackages and classes, separating an object variable from its attribute or method

Keywords

Keywords are reserved words recognized by Java that cannot be used as identifiers. Java defines 49 keywords as follows:

abstract	continue	goto	package	synchronize
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

Exercise: Syntax

1) What's the difference between a *keyword* and an *identifier*?
2) What's the difference between an *identifier* and a *literal*?
3) What's the difference between `/** text */` and `/* text */`.
4) Which of these are valid identifiers?

```
int, anInt, I, i1, 1, thing1, 1thing, one-hundred,
one_hundred, something2do
```

5) Identify the literals, separators and identifiers in the progam below.

```
class MyProgram {
   int i = 30;  int j = i; char c = 'H';
   public static void main(String[] args) {
        System.out.println("i and j " + i + "  " + j);
   } }
```

6) What's the difference between a brace `{}` and a bracket `()`?

Types

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Strong Typing

Java is a strongly-typed language:

a) every variable and expression has a type

b) every type is strictly defined

c) all assignments are checked for type-compatibility

d) no automatic conversion of non-compatible, conflicting types

e) Java compiler type-checks all expressions and parameters

f) any typing errors must be corrected for compilation to succeed

Simple Types

Java defines eight simple types:

1) `byte` – 8-bit integer type
2) `short` – 16-bit integer type
3) `int` – 32-bit integer type
4) `long` – 64-bit integer type
5) `float` – 32-bit floating-point type
6) `double` – 64-bit floating-point type
7) `char` – symbols in a character set
8) `boolean` – logical values `true` and `false`

Simple Type: byte

8-bit integer type.

Range: `-128` to `127`.

Example:

```
byte b = -15;
```

Usage: particularly when working with data streams.

Simple Type: short

16-bit integer type.

Range: `-32768` to `32767`.

Example:

```
short c = 1000;
```

Usage: probably the least used simple type.

Simple Type: int

32-bit integer type.

Range: `-2147483648` to `2147483647`.

Example:

```
int b = -50000;
```

Usage:

1) Most common integer type.
2) Typically used to control loops and to index arrays.
3) Expressions involving the `byte`, `short` and `int` values are promoted to `int` before calculation.

Simple Type: long

64-bit integer type.

Range: `-9223372036854775808` to `9223372036854775807`.

Example:

```
long l = 10000000000000000;
```

Usage:

1) useful when `int` type is not large enough to hold the desired value

Example: long

```
// compute the light travel distance
class Light {
  public static void main(String args[]) {
            int lightspeed = 186000;
            long days = = 1000;
            long seconds = days * 24 * 60 * 60;
            long distance = lightspeed * seconds;
            System.out.print("In " + days);
            System.out.print(" light will travel about
  ");
            System.out.println(distance + " miles.");
  }
}
```

Simple Type: float

32-bit floating-point number.

Range: `1.4e-045` to `3.4e+038`.

Example:

```
float f = 1.5;
```

Usage:

1) fractional part is needed
2) large degree of precision is not required

Simple Type: double

64-bit floating-point number.

Range: `4.9e-324` to `1.8e+308`.

Example:

```
double pi = 3.1416;
```

Usage:

1) accuracy over many iterative calculations
2) manipulation of large-valued numbers

Example: double

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
      double pi = 3.1416;     // approximate pi value
      double r = 10.8;        // radius of circle
      double a = pi * r * r;  // compute area
      System.out.println("Area of circle is " + a);
  }
}
```

Simple Type: char

16-bit data type used to store characters.

Range: `0` to `65536`.

Example:

```
char c = 'a';
```

Usage:

1) Represents both ASCII and Unicode character sets; Unicode defines a character set with characters found in (almost) all human languages.
2) Not the same as in C/C++ where `char` is 8-bit and represents ASCII only.

Example: char

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
      char ch1, ch2;
      ch1 = 88;  // code for X
      ch2 = 'Y';
      System.out.print("ch1 and ch2: ");
      System.out.println(ch1 + " " + ch2);
  }
}
```

Another Example: char

It is possible to operate on `char` values as if they were integers:

```
class CharDemo2 {
      public static void main(String args[]) {
            char c = 'X';
            System.out.println("c contains " + c);
            c++;  // increment c
            System.out.println("c is now " + c);
      }
}
```

Simple Type: boolean

Two-valued type of logical values.

Range: values `true` and `false`.

Example:

```
boolean b = (1<2);
```

Usage:

1) returned by relational operators, such as `1<2`
2) required by branching expressions such as `if` or `for`

Example: boolean

```
class BoolTest {
   public static void main(String args[]) {
      boolean b;
      b = false;
      System.out.println("b is " + b);
      b = true;
      System.out.println("b is " + b);
      if (b) System.out.println("executed");
      b = false;
      if (b) System.out.println("not executed");
      System.out.println("10 > 9 is " + (10 > 9));
   }
}
```

Literals Revisited

Literals express constant values.

The form of a literal depends on its type:

1) integer types
2) floating-point types
3) character type
4) boolean type
5) string type

Literals: Integer Types

Writing numbers with different bases:

1) decimal – `123`
2) octal – `0173`
3) hexadecimal – `0x7B`

Integer literals are of type `int` by default.

Integer literal written with "`L`" (e.g. `123L`) are of type `long`.

Literals: Floating-Point Types

Two notations:

1) standard – `2000.5`
2) scientific – `2.0005E3`

Floating-point literals are of type `double` by default.

Floating-point literal written with "`F`" (e.g. `2.0005E3F`) are of type `float`.

Literals: Boolean

Two literals are allowed only: `true` and `false`.

Those values do not convert to any numerical representation.

In particular:

1) `true` is not equal to `1`
2) `false` is not equal to `0`

Literals: Characters

Character literals belong to the Unicode character set.

Representation:

1) visible characters inside quotes, e.g. `'a'`
2) invisible characters written with escape sequences:
   - a) `\ddd` — octal character `ddd`
   - b) `\uxxxx` — hexadecimal Unicode character `xxxx`
   - c) `\'` — single quote
   - d) `\"` — double quote
   - e) `\\` — backslash
   - f) `\r` — carriage return
   - g) `\n` — new line
   - h) `\f` — form feed
   - i) `\t` — tab
   - j) `\b` — backspace

Literals: String

String is not a simple type.

String literals are character-sequences enclosed in double quotes.

Example:

```
"Hello World!"
```

Notes:

1) escape sequences can be used inside string literals
2) string literals must begin and end on the same line
3) unlike in C/C++, in Java `String` is not an array of characters

Exercise: Types

1) What is the value of `b` in the following code snippet?

```
byte b = 0;
b += 1;
```

2) What happens when this code is executed?

```
char c = -1;
```

3) Which of the following lines will compile without warning or error. Explain each line.

```
float f=1.3;
char c="a";
byte b=257;
boolean b=null;
int i=10;
```

4) Write a java statement that will print the following line to the console:

```
To print a ' " ', we use the ' \"  'escape sequence.
```

Variables

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Outline: Variables

1) declaration – how to assign a type to a variable
2) initialization – how to give an initial value to a variable
3) scope – how the variable is visible to other parts of the program
4) lifetime – how the variable is created, used and destroyed
5) type conversion – how Java handles automatic type conversion
6) type casting – how the type of a variable can be narrowed down
7) type promotion – how the type of a variable can be expanded

Variables

Java uses variables to store data.

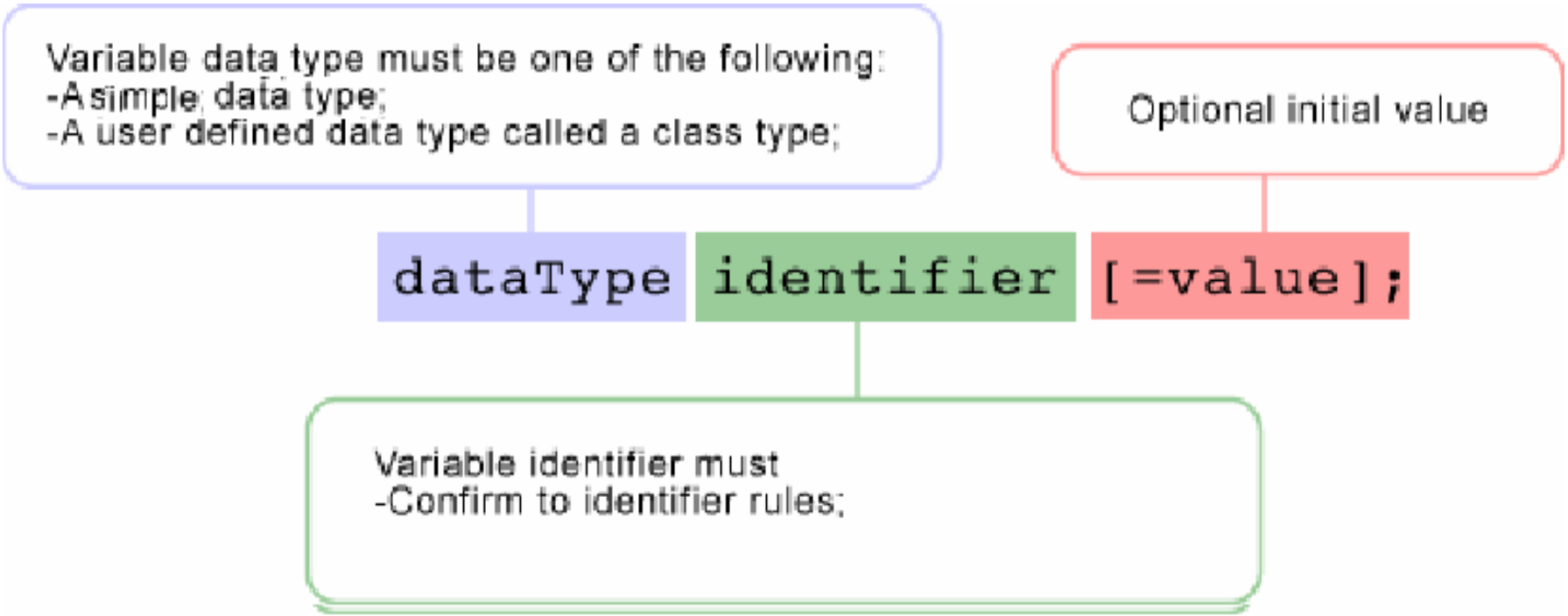
To allocate memory space for a variable JVM requires:

1) to specify the data type of the variable
2) to associate an identifier with the variable
3) optionally, the variable may be assigned an initial value

All done as part of variable declaration.

Basic Variable Declaration

Basic form of variable declaration:



Variable Declaration

We can declare several variables at the same time:

```
type identifier [=value][, identifier [=value] …];
```

Examples:

```
int a, b, c;
int d = 3, e, f = 5;
byte hog = 22;
double pi = 3.14159;
char kat = 'x';
```

Constant Declaration

A variable can be declared as final:

```
final double PI = 3.14;
```

The value of the final variable cannot change after it has been initialized:

```
PI = 3.13;
```

Variable Identifiers

Identifiers are assigned to variables, methods and classes.

An identifier:

1) starts with a letter, underscore _ or dollar `$`
2) can contain letters, digits, underscore or dollar characters
3) it can be of any length
4) it must not be a keyword (e.g. `class`)
5) it must be unique in its scope

Examples: `identifier, userName, _sys_var1, $change`

The code of Java programs is written in Unicode, rather than ASCII, so letters and digits have considerably wider definitions than just a-z and 0-9.

Naming Conventions

Conventions are not part of the language.

Naming conventions:

1) variable names begin with a lowercase letter
2) class names begin with an uppercase letter
3) constant names are all uppercase

If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter.

The underscore character is used only to separate words in constants, as they are all caps and thus cannot be case-delimited.

Variable Initialization

During declaration, variables may be optionally initialized.

Initialization can be static or dynamic:

1) static – initialize with a literal:

```
int n = 1;
```

2) dynamic – initialize with an expression composed of any literals, variables or method calls available at the time of initialization:

```
int m = n + 1;
```

The types of the expression and variable must be the same.

Example: Variable Initialization

```
class DynamicInit {
   public static void main(String args[]) {
      double a = 3.0, b = 4.0;
      double c = Math.sqrt(a * a + b * b);
      System.out.println("Hypotenuse is " + c);
   }
}
```

Variable Scope

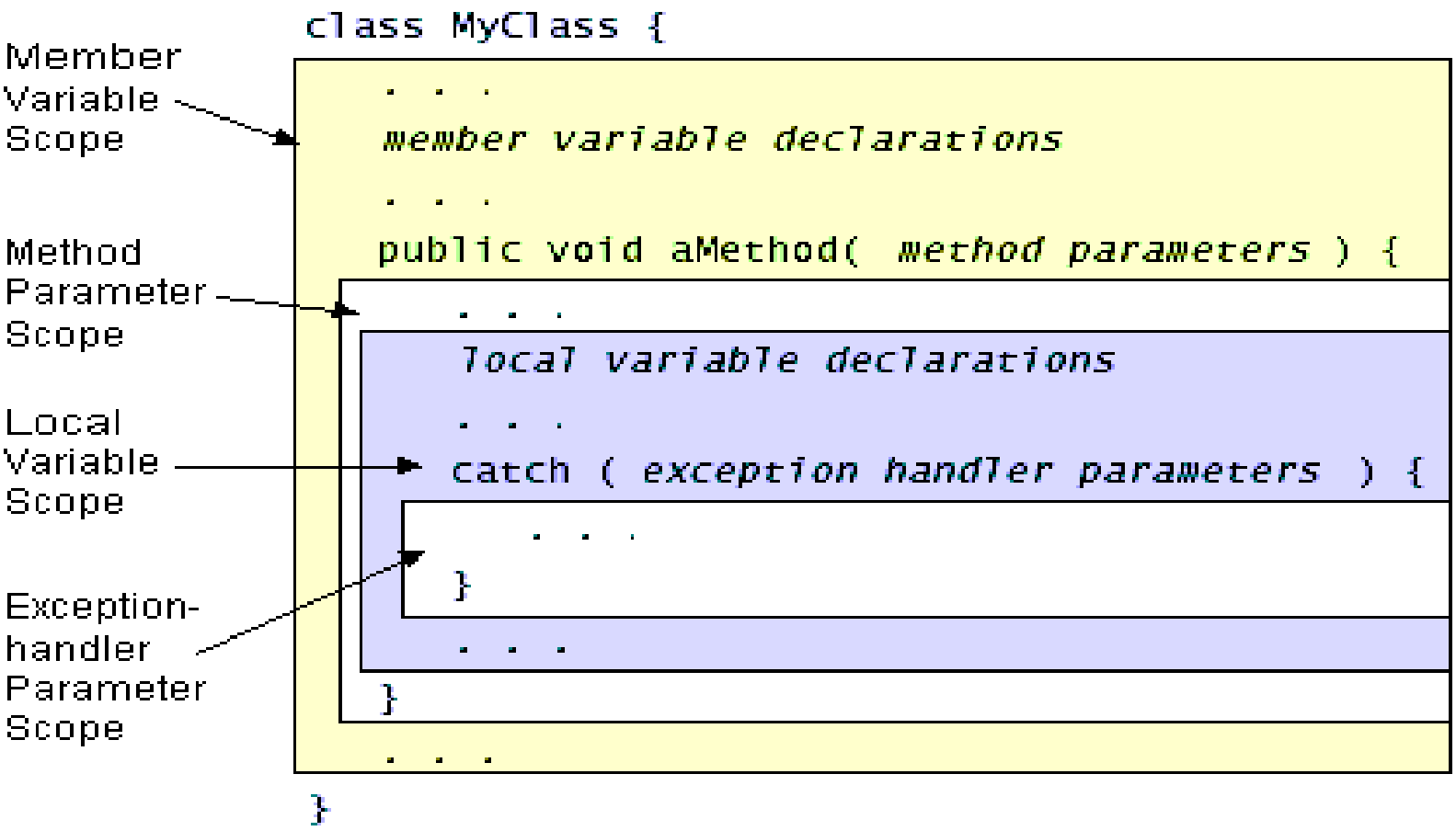
Scope determines the visibility of program elements with respect to other program elements.

In Java, scope is defined separately for classes and methods:

1) variables defined by a class have a "global" scope
2) variables defined by a method have a "local" scope

We consider the scope of method variables only; class variables will be considered later.

Variable Scope



Scope Definition

A scope is defined by a block:

```
{
        …
}
```

A variable declared inside the scope is not visible outside:

```
{
        int n;
}
n = 1;
```

Scope Nesting

Scopes can be nested:

```
{       …       { … }           …       }
```

Variables declared in the outside scope are visible in the inside scope, but not the other way round:

```
{
        int i;
        {
                int n = i;
        }
        int m = n;
}
```

Example: Variable Scope

```
class Scope {
      public static void main(String args[]) {
            int x;
            x = 10;
            if (x == 10) {
                  int y = 20;
                  System.out.println("x and y: " + x + "
" + y);
                  x = y * 2;
            }
            System.out.println("x is " + x + “y is” + y);
      }
}
```

Declaration Order

Method variables are only valid after their declaration:

```
{
        int n = m;
        int m;
}
```

Variable Lifetime

Variables are created when their scope is entered by control flow and destroyed when their scope is left:

1) A variable declared in a method will not hold its value between different invocations of this method.
2) A variable declared in a block looses its value when the block is left.
3) Initialized in a block, a variable will be re-initialized with every re-entry.

Variable's lifetime is confined to its scope!

Example: Variable Lifetime

```
class LifeTime {
      public static void main(String args[]) {
            int x;
            for (x = 0; x < 3; x++) {
                  int y = -1;
                  System.out.println("y is: " + y);
                  y = 100;
                  System.out.println("y is now: " + y);
            }
      }
}
```

Type Differences

Suppose a value of one type is assigned to a variable of another type.

```
T1 t1;
T2 t2 = t1;
```

What happens? Different situations:

1) types T1 and T2 are incompatible
2) types T1 and T2 are compatible:
    a) T1 and T2 are the same
    b) T1 is larger than T2
    c) T2 is larger than T1

Type Compatibility

When types are compatible:

1) integer types and floating-point types are compatible with each other
2) numeric types are not compatible with `char` or `boolean`
3) `char` and `boolean` are not compatible with each other

Examples:

```
byte b;
int i = b;
char c = b;
```

Widening Type Conversion

Java performs automatic type conversion when:

1) two types are compatible
2) destination type is larger then the source type

Example:

```
int i;
double d = i;
```

Narrowing Type Conversion

When:

1) two types are compatible
2) destination type is smaller then the source type

then Java will not carry out type-conversion:

```
int i;
byte b = i;
```

Instead, we have to rely on manual type-casting:

```
int i;
byte b = (byte)i;
```

Type Casting

General form: `(targetType) value`

Examples:

1) integer value will be reduced module `byte`'s range:

```
int i;
byte b = (byte) i;
```

2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

Example: Type Casting

```
class Conversion {
   public static void main(String args[]) {
      byte b;
      int i = 257;
      double d = 323.142;
      System.out.println("\nConversion of int to byte.");
      b = (byte) i;
      System.out.println("i and b " + i + " " + b);
      System.out.println("\ndouble to int.");
      i = (int) d;
      System.out.println("d and i " + d + " " + i);
   }
}
```

Type Promotion

In an expression, precision required to hold an intermediate value may sometimes exceed the range of either operand:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Java promotes each `byte` operand to `int` when evaluating the expression.

Type Promotion Rules

1) `byte` and `short` are always promoted to `int`
2) if one operand is `long`, the whole expression is promoted to `long`
3) if one operand is `float`, the entire expression is promoted to `float`
4) if any operand is `double`, the result is `double`

Danger of automatic type promotion:

```
byte b = 50;
b = b * 2;
```

What is the problem?

Example: Type Promotion

```
class Promote {
   public static void main(String args[]) {
      byte b = 42;
            char c = 'a';
            short s = 1024;
            int i = 50000;
            float f = 5.67f;
            double d = .1234;
            double result = (f * b) + (i / c) - (d * s);
            System.out.println("result = " + result);
   }
}
```

Exercise: Variables

1) What's the difference between widening conversion and casting?
2) What is the output of the program below?.

```
class MyProgram {
   static final double pi = 3.15;
   static double radius = 2.78;
   public static void main(String[] args) {
     pi = 3.14;
     double area = pi * radius * radius;
     System.out.println("The Area is " + area);
   }
}
```

3) What is the value of c[50] after this declaration:

```
char[] c = new char[100];
```

Arrays

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Arrays

An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.

Arrays are:

1) declared
2) created
3) initialized
4) used

Also, arrays can have one or several dimensions.

Array Declaration

Array declaration involves:

1) declaring an array identifier
2) declaring the number of dimensions
3) declaring the data type of the array elements

Two styles of array declaration:

```
type array-variable[];
```

or

```
type [] array-variable;
```

Array Creation

After declaration, no array actually exists.

In order to create an array, we use the `new` operator:

```
type array-variable[];
array-variable = new type[size];
```

This creates a new array to hold `size` elements of type `type`, which reference will be kept in the variable `array-variable`.

Array Indexing

Later we can refer to the elements of this array through their indexes:

```
array-variable[index]
```

The array index always starts with zero!

The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

Example: Array Use

```
class Array {
      public static void main(String args[]) {
            int monthDays[];
            monthDays = new int[12];
            monthDays[0] = 31;
            monthDays[1] = 28;
            monthDays[2] = 31;
            monthDays[3] = 30;
            monthDays[4] = 31;
            monthDays[5] = 30;
            …
            monthDays[12] = 31;
            System.out.print(“April has ”);
            System.out.println(monthDays[3] +“ days.”);
      }
}
```

Array Initialization

Arrays can be initialized when they are declared:

```
int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Comments:

1) there is no need to use the `new` operator
2) the array is created large enough to hold all specified elements

Example: Array Initialization

```
class Array {
      public static void main(String args[]) {
            int mthDys[]=
                  {31,28,31,30,31,30,31,31,30,31,30,31};

            System.out.print("April ");
            System.out.println(mthDys[3]+ " days.");
      }
}
```

Multidimensional Arrays

Multidimensional arrays are arrays of arrays:

1) declaration

```
int array[][];
```

2) creation

```
int array = new int[2][3];
```

3) initialization

```
int array[][] = { {1, 2, 3}, {4, 5, 6} };
```

Example: Multidimensional Arrays

```
class Array {

   public static void main(String args[]) {

      int array[][] = { {1, 2, 3}, {4, 5, 6} };
      int i, j, k = 0;

      for(i=0; i<2; i++) {
            for(j=0; j<3; j++)
                  System.out.print(array[i][j] + " ");
                  System.out.println();
      }
   }
}
```

Exercise: Arrays

1) Write a program that creates an array of 10 integers with the initial values of 3.
2) Write a Java program to find the average of a sequence of nonnegative numbers entered by the user, where the user enters a negative number to terminate the input. Assume the only method in the class is the main method.
3) What's the index of the first and the last component of a one hundred component array?
4) What will happen if you try to compile and run the following code?

```
public class Q {
   public static void main(String argv[]){
     int var[]=new int[5];
     System.out.println(var[0]);
   } }
```

Operators

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Operators Types

Java operators are used to build value expressions.

Java provides a rich set of operators:

1) assignment
2) arithmetic
3) relational
4) logical
5) bitwise
6) other

Operators and Operands

Each operator takes one, two or three operands:

1) a unary operator takes one operand

```
j++;
```

2) a binary operator takes two operands

```
i = j++;
```

3) a ternary operator requires three operands

```
i = (i>12) ? 1 : i++;
```

Assignment Operator

A binary operator:

```
variable = expression;
```

It assigns the value of the expression to the variable.

The types of the variable and expression must be compatible.

The value of the whole assignment expression is the value of the expression on the right, so it is possible to chain assignment expressions as follows:

```
int x, y, z;
x = y = z = 2;
```

Arithmetic Operators

Java supports various arithmetic operators for:

1) integer numbers
2) floating-point numbers

There are two kinds of arithmetic operators:

1) basic: addition, subtraction, multiplication, division and modulo
2) shortcut: arithmetic assignment, increment and decrement

Table: Basic Arithmetic Operators

+	op1 + op2	adds op1 and op2
-	op1 - op2	subtracts op2 from op1
*	op1 * op2	multiplies op1 by op2
/	op1 / op2	divides op1 by op2
%	op1 % op2	computes the remainder of dividing op1 by op2

Arithmetic Assignment Operators

Instead of writing

```
variable = variable operator expression;
```

for any arithmetic binary operator, it is possible to write shortly

```
variable operator= expression;
```

Benefits of the assignment operators:

1) save some typing
2) are implemented more efficiently by the Java run-time system

Table: Arithmetic Assignments

+=	`v += expr;`	`v = v + expr;`
-=	`v -= expr;`	`v = v - expr;`
*=	`v *= expr;`	`v = v * expr;`
/=	`v /= expr;`	`v = v / expr;`
%=	`v %= expr;`	`v = v % expr;`

Increment/Decrement Operators

Two unary operators:

1) `++` increments its operand by `1`
2) `--` decrements its operand by `1`

The operand must be a numerical variable.

Each operation can appear in two versions:

- prefix version evaluates the value of the operand after performing the increment/decrement operation
- postfix version evaluates the value of the operand before performing the increment/decrement operation

Table: Increment/Decrement

`++`	`v++`	return value of `v`, then increment `v`
`++`	`++v`	increment `v`, then return its value
`--`	`v--`	return value of `v`, then decrement `v`
`--`	`--v`	decrement `v`, then return its value

Example: Increment/Decrement

```
class IncDec {
   public static void main(String args[]) {
      int a = 1;
      int b = 2;
      int c, d;
      c = ++b;
      d = a++;
      c++;
      System.out.println("a= " + a);
      System.out.println("b= " + b);
      System.out.println("c= " + c);
   }
}
```

Relational Operators

Relational operators determine the relationship that one operand has to the other operand, specifically equality and ordering.

The outcome is always a value of type `boolean`.

They are most often used in branching and loop control statements.

Table: Relational Operators

==	equals to	apply to any type
!=	not equal to	apply to any type
>	greater than	apply to numerical types only
<	less than	apply to numerical types only
>=	greater than or equal	apply to numerical types only
<=	less than or equal	apply to numerical types only

Logical Operators

Logical operators act upon `boolean` operands only.

The outcome is always a value of type `boolean`.

In particular, "and" and "or" logical operators occur in two forms:

1) full – `op1 & op2` and `op1 | op2` where both `op1` and `op2` are evaluated
2) short-circuit - `op1 && op2` and `op1 || op2` where `op2` is only evaluated if the value of `op1` is insufficient to determine the final outcome

Table: Logical Operators

`&`	`op1 & op2`	logical `AND`
`|`	`op1 | op2`	logical `OR`
`&&`	`op1 && op2`	short-circuit `AND`
`||`	`op1 || op2`	short-circuit `OR`
`!`	`! op`	logical `NOT`
`^`	`op1 ^ op2`	logical `XOR`

Example: Logical Operators

```
class LogicalDemo {

   public static void main(String[] args) {
      int n = 2;
      if (n != 0 && n / 0 > 10)
            System.out.println("This is true");
      else
            System.out.println("This is false");
   }

}
```

Bitwise Operators

Bitwise operators apply to integer types only.

They act on individual bits of their operands.

There are three kinds of bitwise operators:

1) basic – bitwise `AND`, `OR`, `NOT` and `XOR`
2) shifts – left, right and right-zero-fill
3) assignments – bitwise assignment for all basic and shift operators

Table: Bitwise Operators

~	~ op	inverts all bits of its operand
&	op1 & op2	produces 1 bit if both operands are 1
|	op1 | op2	produces 1 bit if either operand is 1
^	op1 ^ op2	produces 1 bit if exactly one operand is 1
>>	op1 >> op2	shifts all bits in op1 right by the value of op2
<<	op1 << op2	shifts all bits in op1 left by the value of op2
>>>	op1 >>> op2	shifts op1 right by op2 value, write zero on the left

Example: Bitwise Operators

```
class BitLogic {
   public static void main(String args[]) {
      String binary[] = { "0000","0001","0010", … };
      int a = 3; // 0 + 2 + 1 or 0011 in binary
      int b = 6; // 4 + 2 + 0 or 0110 in binary
      int c = a | b;
      int d = a & b;
      int e = a ^ b;
      System.out.print("a =" + binary[a]);
      System.out.println("and b =" +     binary[b]);
      System.out.println("a|b = " + binary[c]);
      System.out.println("a&b = " + binary[d]);
      System.out.println("a^b = " + binary[e]);
   }
}
```

Other Operators

`?:`	shortcut if-else statement
`[]`	used to declare arrays, create arrays, access array elements
`.`	used to form qualified names
`(params)`	delimits a comma-separated list of parameters
`(type)`	casts a value to the specified type
`new`	creates a new object or a new array
`instanceof`	determines if its first operand is an instance of the second

Conditional Operator

General form:

```
expr1? expr2 : expr3
```

where:

1) `expr1` is of type boolean
2) `expr2` and `expr3` are of the same type

If `expr1` is true, `expr2` is evaluated, otherwise `expr3` is evaluated.

Example: Conditional Operator

```
class Ternary {
   public static void main(String args[]) {
      int i, k;

      i = 10;
      k = i < 0 ? -i : i;
      System.out.print("Abs value of " + i + " is " + k);

      i = -10;
      k = i < 0 ? -i : i;
      System.out.print("Abs value of " + i + " is " + k);
   }
}
```

Operator Precedence

Java operators are assigned precedence order.

Precedence determines that the expression

```
1 + 2 * 6 / 3 > 4 && 1 < 0
```

if equivalent to

```
(((1 + ((2 * 6) / 3)) > 4) && (1 < 0))
```

When operators have the same precedence, the earlier one binds stronger.

Table: Operator Precedence

highest			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
lowest			

Exercise: Operators

1) What operators do the code snippet below contain?

```
arrayOfInts[j] > arrayOfInts[j+1];
```

2) Consider the following code snippet:

```
int i = 10;

int n = i++%5;
```

   a) What are the values of *i* and *n* after the code is executed?

   b) What are the final values of *i* and *n* if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i`))?

3) What is the value of *i* after the following code snippet executes?

```
int i = 8;

i >>=2;
```

4) What's the result of *`System.out.println(010| 4);`* ?

Control Flow

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Control Flow

Writing a program means typing statements into a file.

Without control flow, the interpreter would execute these statements in the order they appear in the file, left-to-right, top-down.

Control flow statements, when inserted into the text of the program, determine in which order the program should be executed.

Control Flow Statements

Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.

Control statements are divided into three groups:

1) selection statements allow the program to choose different parts of the execution based on the outcome of an expression
2) iteration statements enable program execution to repeat one or more statements
3) jump statements enable your program to execute in a non-linear fashion

Selection Statements

Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.

Java provides four selection statements:

1) `if`
2) `if-else`
3) `if-else-if`
4) `switch`

if Statement

General form:

```
if (expression) statement
```

If `expression` evaluates to `true`, execute `statement`, otherwise do nothing.

The expression must be of type `boolean`.

Simple/Compound Statement

The component statement may be:

1) simple

```
if (expression) statement;
```

2) compound

```
if (expression) {
      statement;
}
```

if-else Statement

Suppose you want to perform two different statements depending on the outcome of a `boolean` expression. `if-else` statement can be used.

General form:

```
if (expression) statement1
else statement2
```

Again, `statement1` and `statement2` may be simple or compound.

if-else-if Statement

General form:

```
if (expression1) statement1
else if (expression2) statement2
else if (expression3) statement3
…
else statement
```

Semantics:

1) statements are executed top-down
2) as soon as one expressions is true, its statement is executed
3) if none of the expressions is true, the last statement is executed

Example: if-else-if

```
class IfElse {
   public static void main(String args[]) {
      int month = 4;
      String season;
      if (month == 12 || month == 1 || month == 2)
         season = "Winter";
      else if(month == 3 || month == 4 || month == 5)
         season = "Spring";
      else if(month == 6 || month == 7 || month == 8)
         season = "Summer";
      else if(month == 9 || month == 10 || month == 11)
         season = "Autumn";
      else season = "Bogus Month";
      System.out.println("April is in the " + season + ".");
   }
}
```

switch Statement

`switch` provides a better alternative than `if-else-if` when the execution follows several branches depending on the value of an expression.

General form:

```
switch (expression) {
   case value1: statement1; break;
   case value2: statement2; break;
   case value3: statement3; break;
   …
   default: statement;
}
```

switch Assumptions/Semantics

Assumptions:

1) `expression` must be of type `byte`, `short`, `int` or `char`
2) each of the `case` values must be a literal of the compatible type
3) `case` values must be unique

Semantics:

1) `expression` is evaluated
2) its value is compared with each of the `case` values
3) if a match is found, the statement following the `case` is executed
4) if no match is found, the statement following `default` is executed

`break` makes sure that only the matching statement is executed.

Both `default` and `break` are optional.

Example: switch 1

```
class Switch {
   public static void main(String args[]) {
      int month = 4;
      String season;
      switch (month) {
         case 12:
         case 1:
         case 2: season = "Winter"; break;
         case 3:
         case 4:
         case 5: season = "Spring"; break;
         case 6:
         case 7:
         case 8: season = "Summer"; break;
```

Example: switch 2

```
            case 9:
            case 10:
            case 11: season = "Autumn"; break;
            default: season = "Bogus Month";
        }
        System.out.println("April is in " + season + ".");
    }
}
```

Nested switch Statement

A switch statement can be nested within another switch statement:

```
switch(count) {
   case 1:
      switch(target) {
         case 0:System.out.println("target is zero");
                  break;
         case 1:System.out.println("target is one");
                  break;
      }
      break;
   case 2: …
}
```

Since, every switch statement defines its own block, no conflict arises between the case constants in the inner and outer switch statements.

Comparing switch and if

Two main differences:

1) switch can only test for equality, while if can evaluate any kind of boolean expression
2) Java creates a “jump table” for switch expressions, so a switch statement is usually more efficient than a set of nested if statements

Iteration Statements

Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.

Java provides three iteration statements:

1) `while`
2) `do-while`
3) `for`

while Statement

General form:

```
while (expression) statement
```

where `expression` must be of type `boolean`.

Semantics:

1) repeat execution of `statement` until `expression` becomes false
2) `expression` is always evaluated before `statement`
3) if `expression` is false initially, `statement` will never get executed

Simple/Compound Statement

The component statement may be:

1) simple

```
while (expression) statement;
```

2) compound

```
while (expression) {
        statement;
}
```

Example: while

```
class MidPoint {
   public static void main(String args[]) {
      int i, j;
      i = 100;
      j = 200;
      while(++i < --j) {
         System.out.println("i is " + i);
         System.out.println("j is " + j);
      }
      System.out.println("The midpoint is " + i);
   }
}
```

do-while Statement

If a component statement has to be executed at least once, the `do-while` statement is more appropriate than the `while` statement.

General form:

```
do statement
while (expression);
```

where `expression` must be of type `boolean`.

Semantics:

1) repeat execution of `statement` until `expression` becomes false
2) `expression` is always evaluated after `statement`
3) even if `expression` is false initially, `statement` will be executed

Example: do-while

```
class DoWhile {
  public static void main(String args[]) {
     int i;
     i = 0;
     do
        i++;
     while ( 1/i < 0.001);
     System.out.println("i is " + i);
  }
}
```

for Statement

When iterating over a range of values, `for` statement is more suitable to use then `while` or `do-while`.

General form:

```
for (initialization; termination; increment)
   statement
```

where:

1) `initialization` statement is executed once before the first iteration
2) `termination` expression is evaluated before each iteration to determine when the loop should terminate
3) `increment` statement is executed after each iteration

for Statement Semantics

This is how the for statement is executed:

1) `initialization` is executed once
2) `termination` expression is evaluated:
    a) if false, the statement terminates
    b) otherwise, continue to (3)
3) `increment` statement is executed
4) component `statement` is executed
5) control flow continues from (2)

Loop Control Variable

The `for` statement may include declaration of a loop control variable:

```
for (int i = 0; i < 1000; i++) {
    …
}
```

The variable does not exist outside the `for` statement.

Example: for

```
class FindPrime {
  public static void main(String args[]) {
     int num = 14;
     boolean isPrime = true;
     for (int i=2; i < num/2; i++) {
           if ((num % i) == 0) {
                 isPrime = false;
              break;
           }
        }
        if (isPrime) System.out.println("Prime");
        else System.out.println("Not Prime");
  }
}
```

Many Initialization/Iteration Parts

The `for` statement may include several `initialization` and `iteration` parts.

Parts are separated by a comma:

```
int a, b;

for (a = 1, b = 4; a < b; a++, b--) {
   …
}
```

for Statement Variations

The `for` statement need not have all components:

```
class ForVar {
   public static void main(String args[]) {
      int i = 0;
      boolean done = false;
      for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
         }
   }
}
```

Empty for

In fact, all three components may be omitted:

```
public class EmptyFor {
   public static void main(String[] args) {
      int i = 0;
      for (; ;) {
         System.out.println("Infinite Loop " + i);
      }
   }
}
```

Jump Statements

Java jump statements enable transfer of control to other parts of program.

Java provides three jump statements:

1) `break`
2) `continue`
3) `return`

In addition, Java supports exception handling that can also alter the control flow of a program. Exception handling will be explained in its own section.

break Statement

The break statement has three uses:

1) to terminate a case inside the switch statement
2) to exit an iterative statement
3) to transfer control to another statement

(1) has been described.

We continue with (2) and (3).

Loop Exit with break

When `break` is used inside a loop, the loop terminates and control is transferred to the following instruction.

```
class BreakLoop {
   public static void main(String args[]) {
      for (int i=0; i<100; i++) {
         if (i == 10) break;
         System.out.println("i: " + i);
      }
      System.out.println("Loop complete.");
   }
}
```

break in Nested Loops

Used inside nested loops, `break` will only terminate the innermost loop:

```
class NestedLoopBreak {
   public static void main(String args[]) {
      for (int i=0; i<3; i++) {
         System.out.print("Pass " + i + ": ");
         for (int j=0; j<100; j++) {
            if (j == 10) break;
            System.out.print(j + " ");
         }
         System.out.println();
      }
      System.out.println("Loops complete.");
   }
}
```

Control Transfer with break

Java does not have an unrestricted “goto” statement, which tends to produce code that is hard to understand and maintain.

However, in some places, the use of gotos is well justified. In particular, when breaking out from the deeply nested blocks of code.

`break` occurs in two versions:

1) unlabelled
2) labeled

The labeled `break` statement is a “civilized” replacement for goto.

Labeled break

General form:

```
break label;
```

where `label` is the name of a label that identifies a block of code:

```
label: { … }
```

The effect of executing `break label;` is to transfer control immediately after the block of code identified by `label`.

Example: Labeled break

```
class Break {
  public static void main(String args[]) {
     boolean t = true;
     first: {
        second: {
           third: {
              System.out.println("Before the break.");
              if (t) break second;
              System.out.println("This won't execute");
           }
           System.out.println("This won't execute");
        }
        System.out.println("After second block.");
     }
} }
```

Example: Nested Loop break

```
class NestedLoopBreak {
  public static void main(String args[]) {
     outer: for (int i=0; i<3; i++) {
        System.out.print("Pass " + i + ": ");
        for (int j=0; j<100; j++) {
           if (j == 10) break outer; // exit both loops
           System.out.print(j + " ");
        }
        System.out.println("This will not print");
     }
  System.out.println("Loops complete.");
  }
}
```

break Without Label

It is not possible to break to any label which is not defined for an enclosing block. Trying to do so will result in a compiler error.

```
class BreakError {
   public static void main(String args[]) {
      one: for(int i=0; i<3; i++) {
         System.out.print("Pass " + i + ": ");
      }
      for (int j=0; j<100; j++) {
         if (j == 10) break one;
         System.out.print(j + " ");
      }
   }
}
```

continue Statement

The `break` statement terminates the block of code, in particular it terminates the execution of an iterative statement.

The `continue` statement forces the early termination of the current iteration to begin immediately the next iteration.

Like break, continue has two versions:

1) unlabelled – continue with the next iteration of the current loop
2) labeled – specifies which enclosing loop to continue

Example: Unlabeled continue

```
class Continue {
  public static void main(String args[]) {
     for (int i=0; i<10; i++) {
        System.out.print(i + " ");
        if (i%2 == 0) continue;
        System.out.println("");
     }
  }
}
```

Example: Labeled continue

```
class LabeledContinue {
  public static void main(String args[]) {
     outer: for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++) {
           if (j > i) {
              System.out.println();
              continue outer;
           }
           System.out.print(" " + (i * j));
        }
     }
     System.out.println();
  }
}
```

Return Statement

The return statement is used to return from the current method: it causes program control to transfer back to the caller of the method.

Two forms:

1) return without value

```
return;
```

2) return with value

```
return expression;
```

Example: Return

```
class Return {
  public static void main(String args[]) {
     boolean t = true;
     System.out.println("Before the return.");
     if (t) return; // return to caller
     System.out.println("This won't execute.");
  }
}
```

Exercise: Control Flow

1) Write a program that prints all the prime numbers between 1 and 49 to the console.

2) Write a program that prints the first 20 Fibonacci numbers.

The Fibonacci numbers are defined as follows:

The zeroth Fibonacci number is 1.
The first Fibonacci number is also 1.
The second Fibonacci number is 1 + 1 = 2.
The third Fibonacci number is 1 + 2 = 3.

In other words, except for the first two numbers each Fibonacci number is the sum of the two previous numbers.

Object-Orientation

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Data versus Operations

Data (nouns) versus operations (verbs):

```
data1
…
dataN
```

nouns

```
operation1 { … }
…
operationN { … }
```

verbs

Procedural Programming

Procedural programming is verb-oriented:

1) decomposition around o*perations*
2) operation are divided into smaller operations

Programming Languages:

1) C
2) Pascal
3) Fortran, etc.

Example: Procedural Program

```
program AddNums {
   Integer a;
   Integer b;

   a = 100;
   b = 200;
   midPoint(a, b);

   procedure midPoint(Integer a, Integer b) {
      while(a < b) {
         println("a is " + a); a = a+1;
         println("b is " + b); b = b-1;
      }
   }
}
```

Drawbacks

1) data is given a second-class status when compared with operations
2) difficult to relate to the real world – there are no functions in real world
3) difficult to create new data types – reduces extensibility
4) programs are difficult to debug – little restriction to data access
5) programs are hard to understand – many variables have global scope
6) programs are hard to reuse – data/functions are mutually dependent
7) little support for developing and comprehending really large programs
8) top-down development approach tends to produce monolithic programs

What is an Object?

Real world objects are things that have:

1) state
2) behavior

Example: your dog:

1) state – name, color, breed, sits?, barks?, wages tail?, runs?
2) behavior – sitting, barking, waging tail, running

A software object is a bundle of variables (state) and methods (operations).

What is a Class?

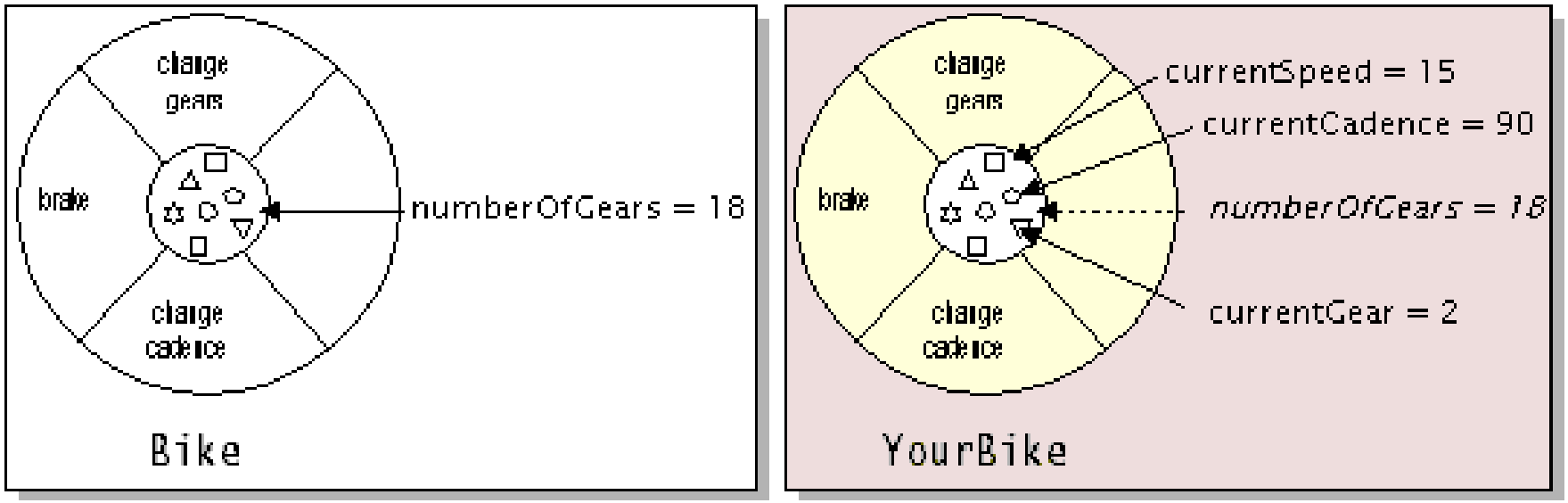
A class is a blueprint that defines the variables and methods common to all objects of a certain kind.

Example: ‘your dog’ is a object of the class Dog.

An object holds values for the variables defines in the class.

An object is called an instance of the Class

Objects versus Classes



Class **Instance of a Class**

1) operations: `changeGears`, `brake`, `changeCadence`
2) variables: `currentSpeed`, `currentCadence`, `currentGear`
3) static variable: `numberOfGears`

   It holds the same value for all objects of the class.

Object-Oriented Programming

Programming defined in terms:

1) objects (nouns) and
2) relationships between objects

Object-Oriented programming languages:

1) SmallTalk
2) C++
3) C#
4) Java

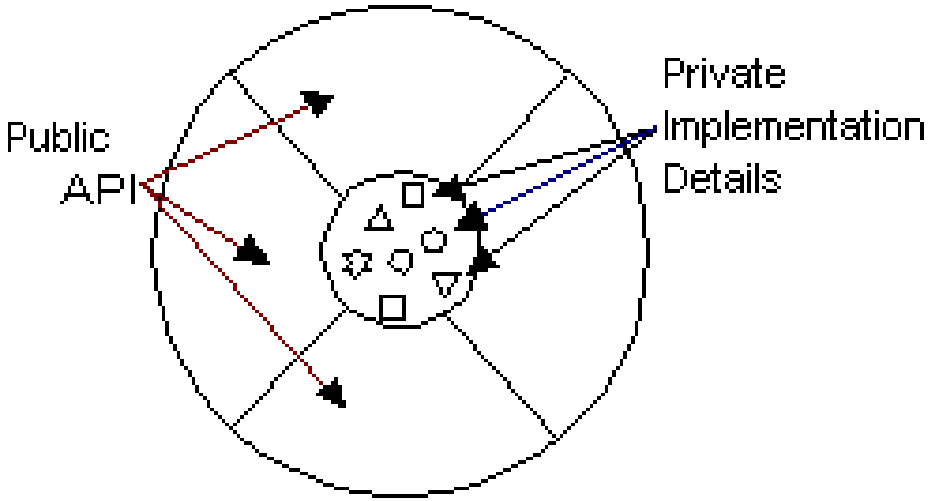
Object-Oriented Concepts

Three main concepts:

1) encapsulation
2) inheritance
3) polymorphism

Encapsulation

Illustration:

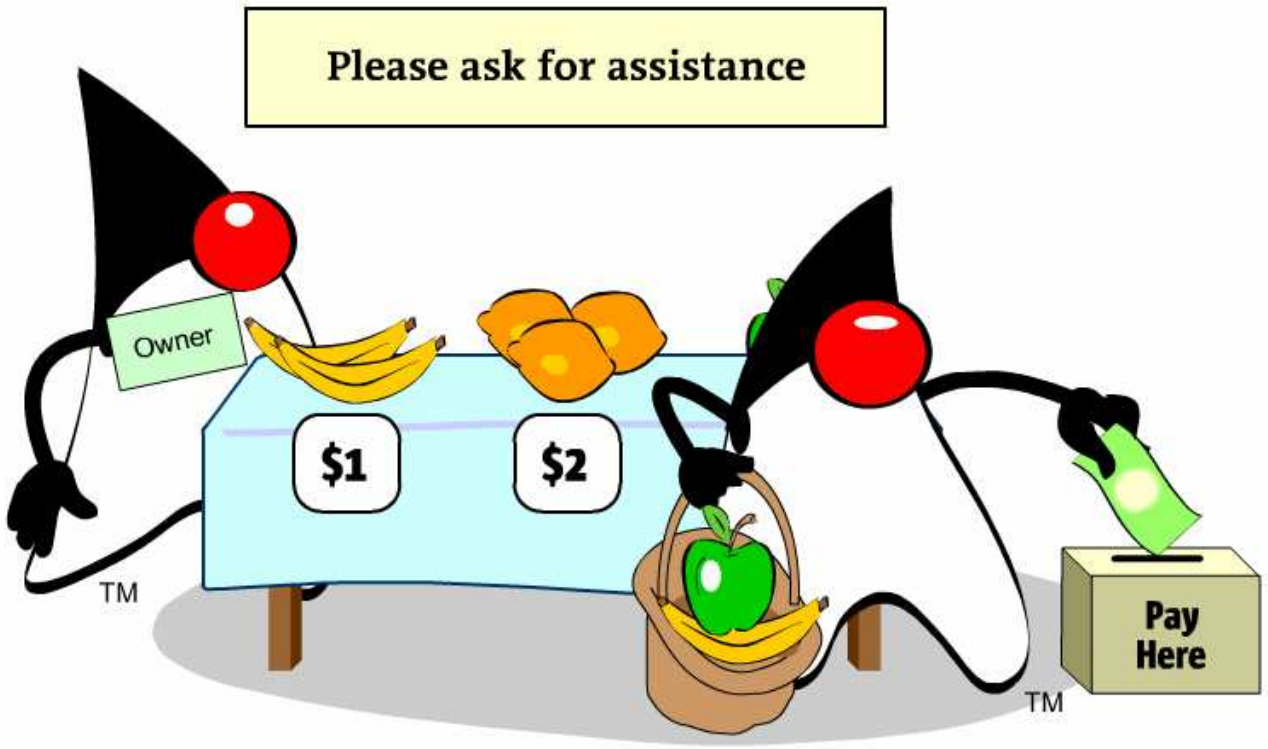


- Enclose data inside an object.
- Along with data, include operations on this data.
- Data cannot be accessed from outside except through the operations.
- Provides data security and facilitates code reuse.
- Operations provide the interface - internal state can change without affecting the user as long as the interface does not change.

Encapsulation 1

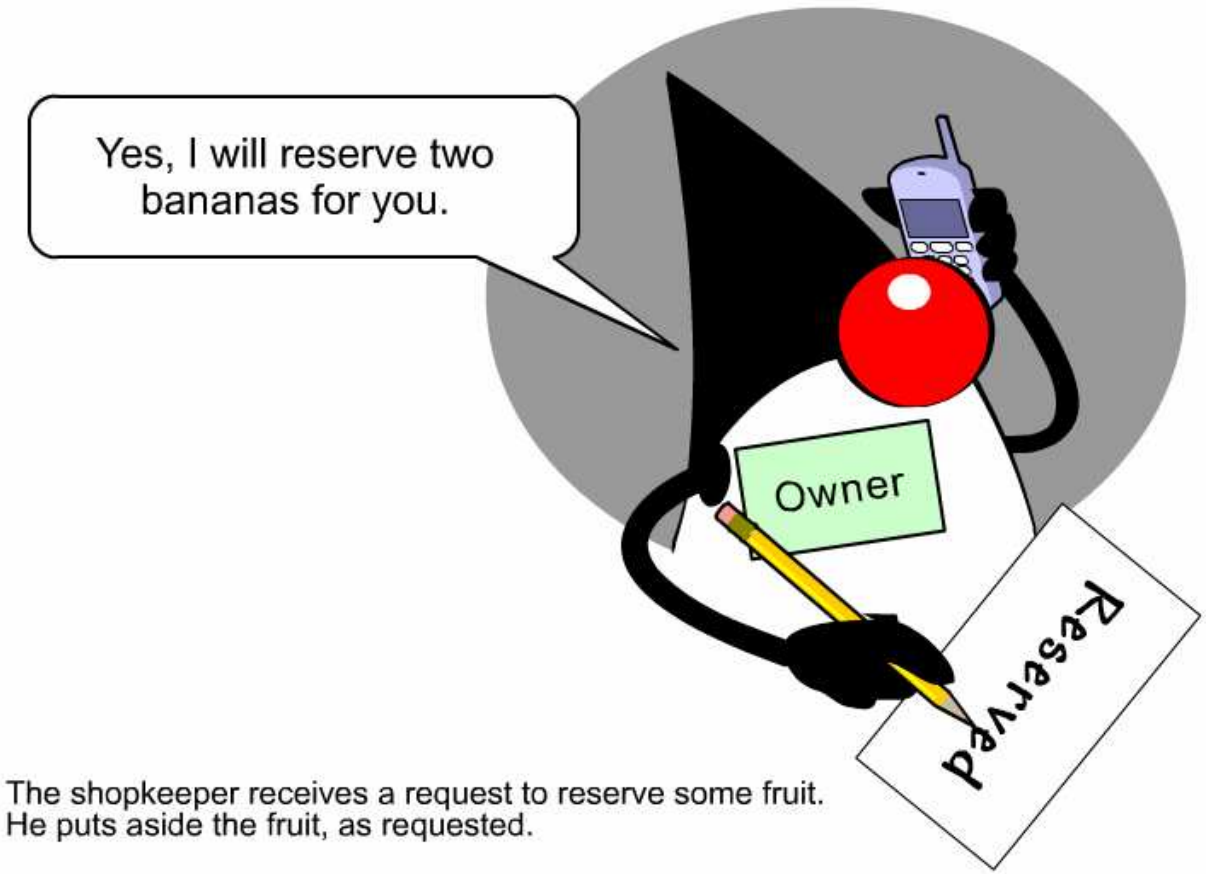
The shopkeeper's fruit stand is designed to serve all customers. However, the design of the fruit stand does not prevent self-service. The fruit is freely accessible to all customers.

Encapsulation 2



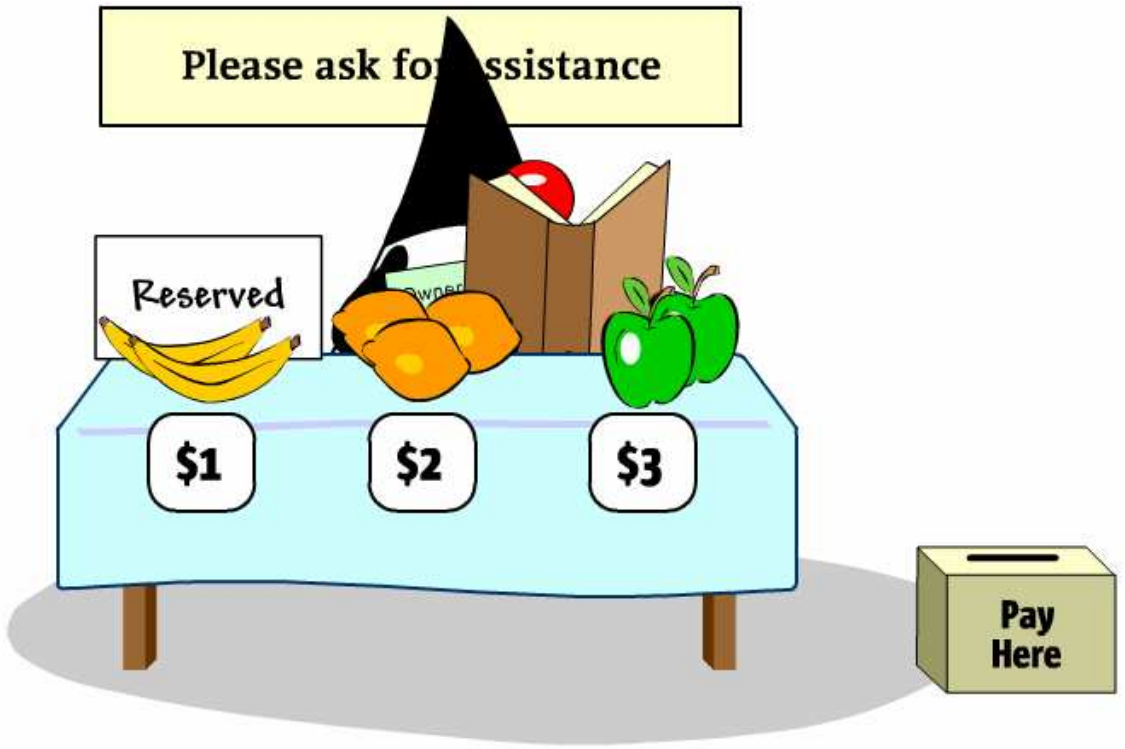
A customer can select some fruit and make payment, all without the shopkeeper's involvement.

Encapsulation 3



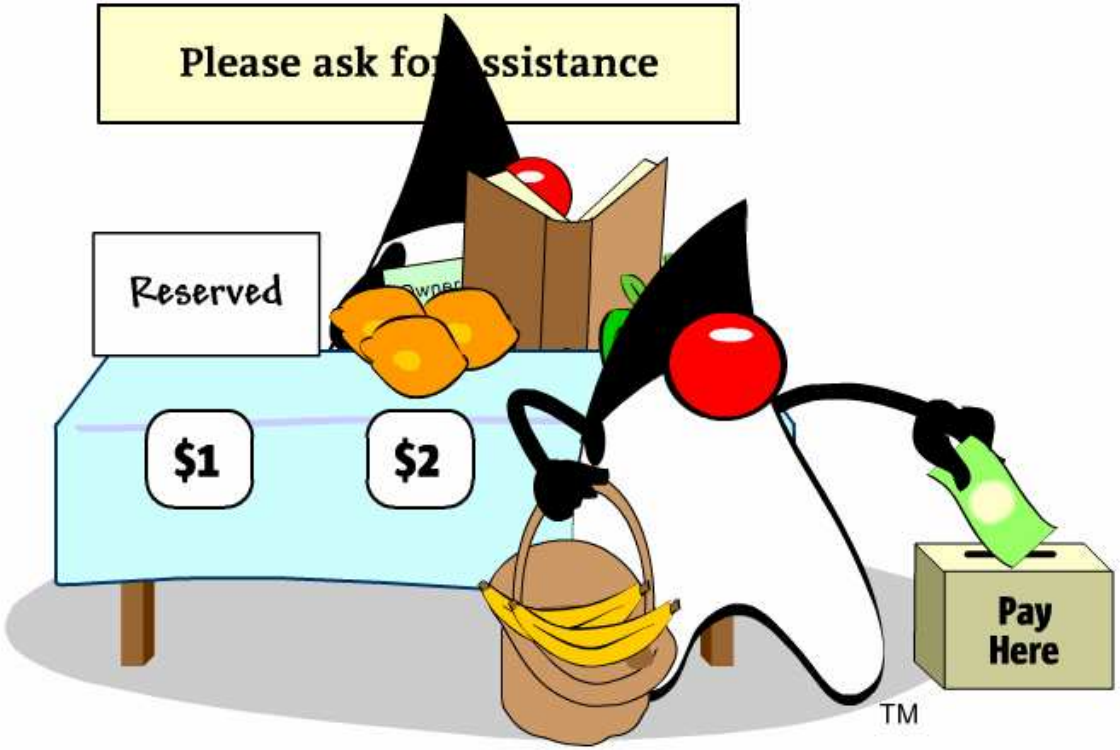
The shopkeeper receives a request to reserve some fruit.
He puts aside the fruit, as requested.

Encapsulation 4



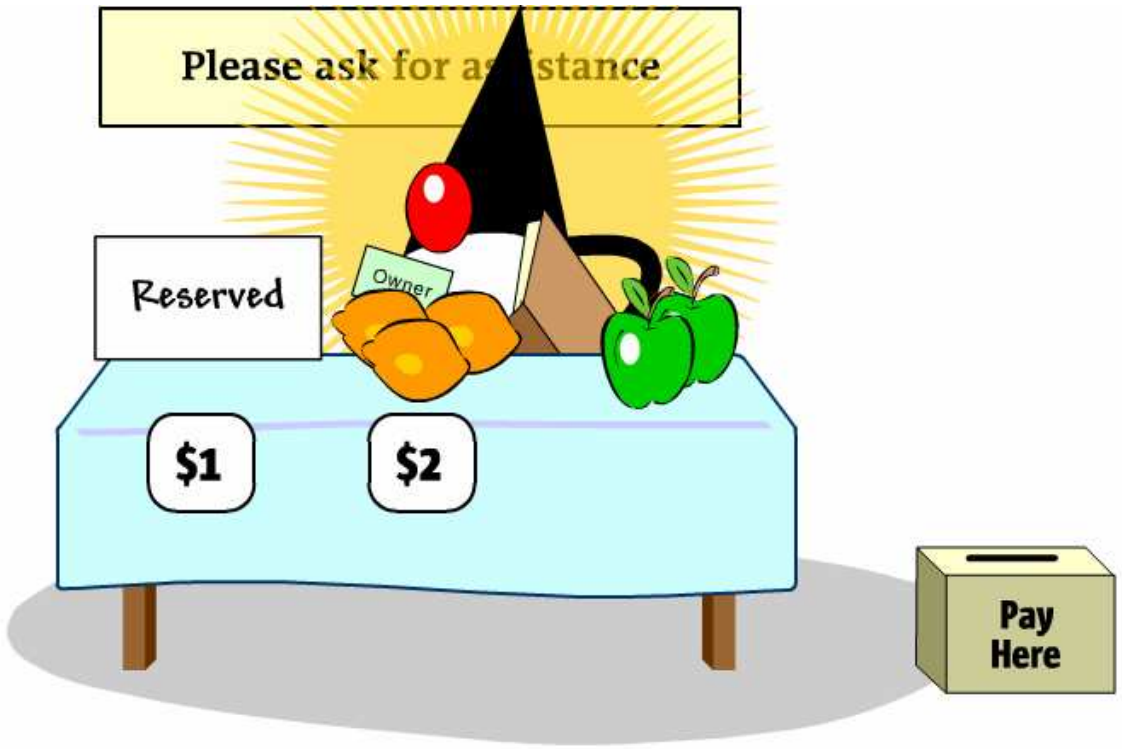
However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

Encapsulation 5



However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

Encapsulation 6

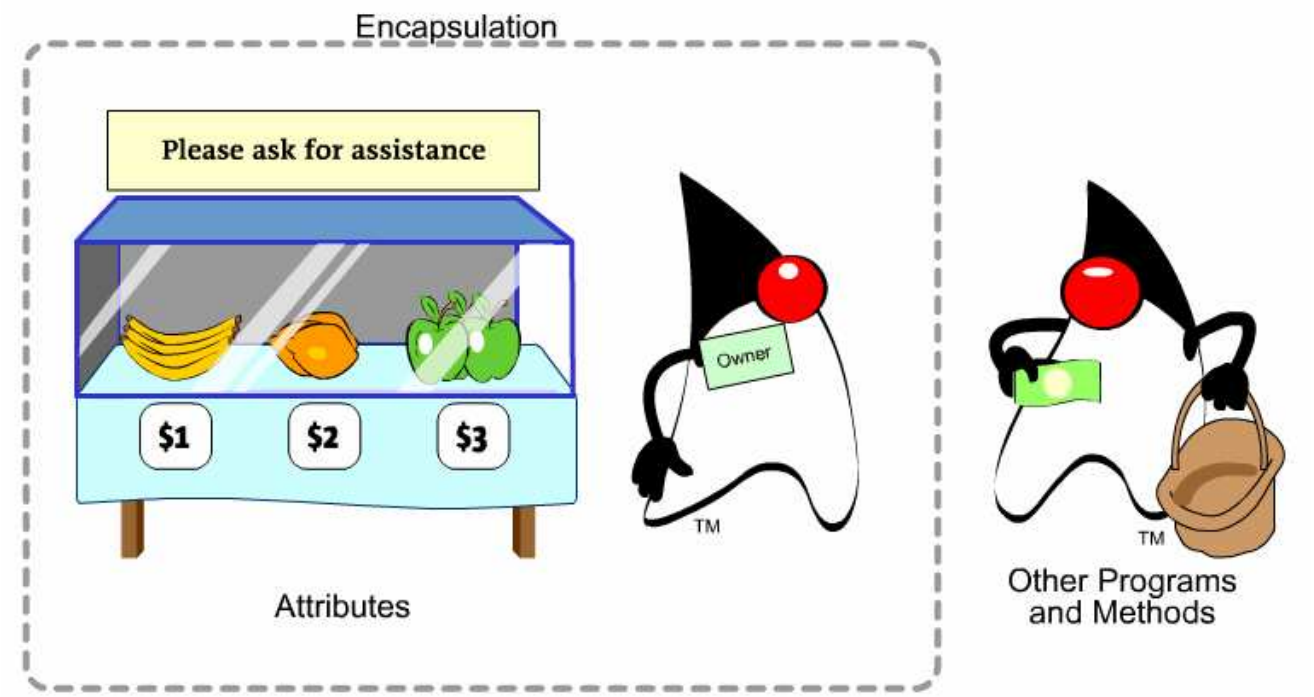


However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

Encapsulation 7

By enclosing the fruit behind glass, and controlling all access by customers, the shopkeeper has encapsulated the process of buying fruit.

Encapsulation 8

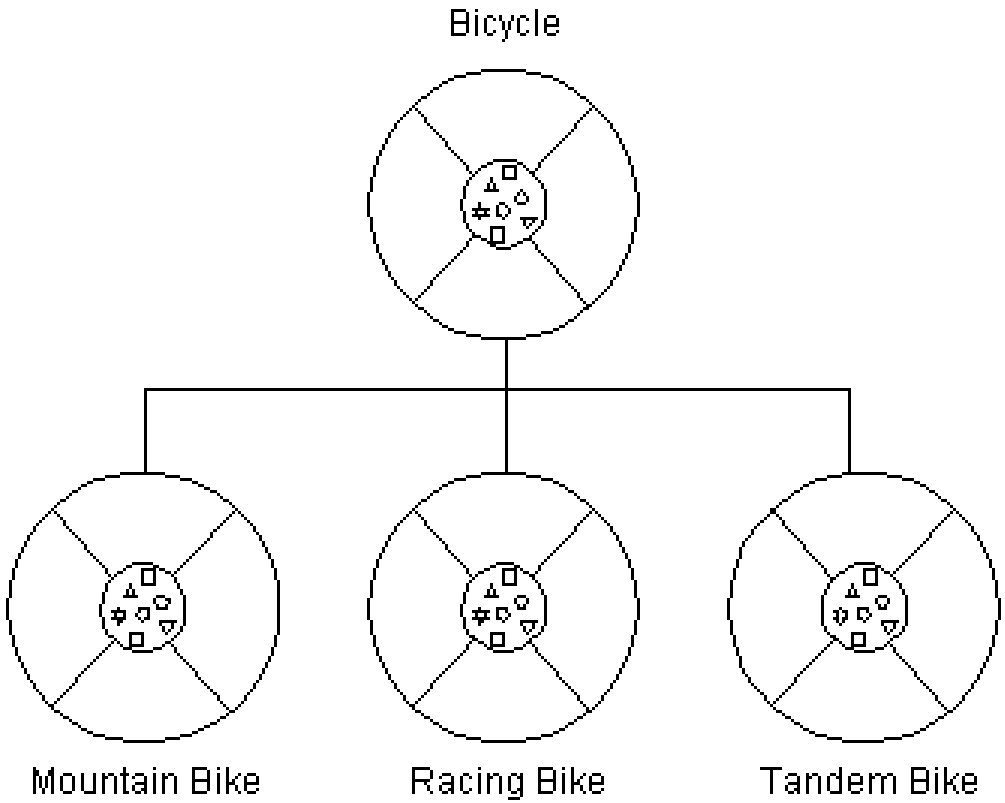


Encapsulation in the Java programming language is protection of the attributes from abritrary access by other programs and methods. You control how access to attributes is accomplished.

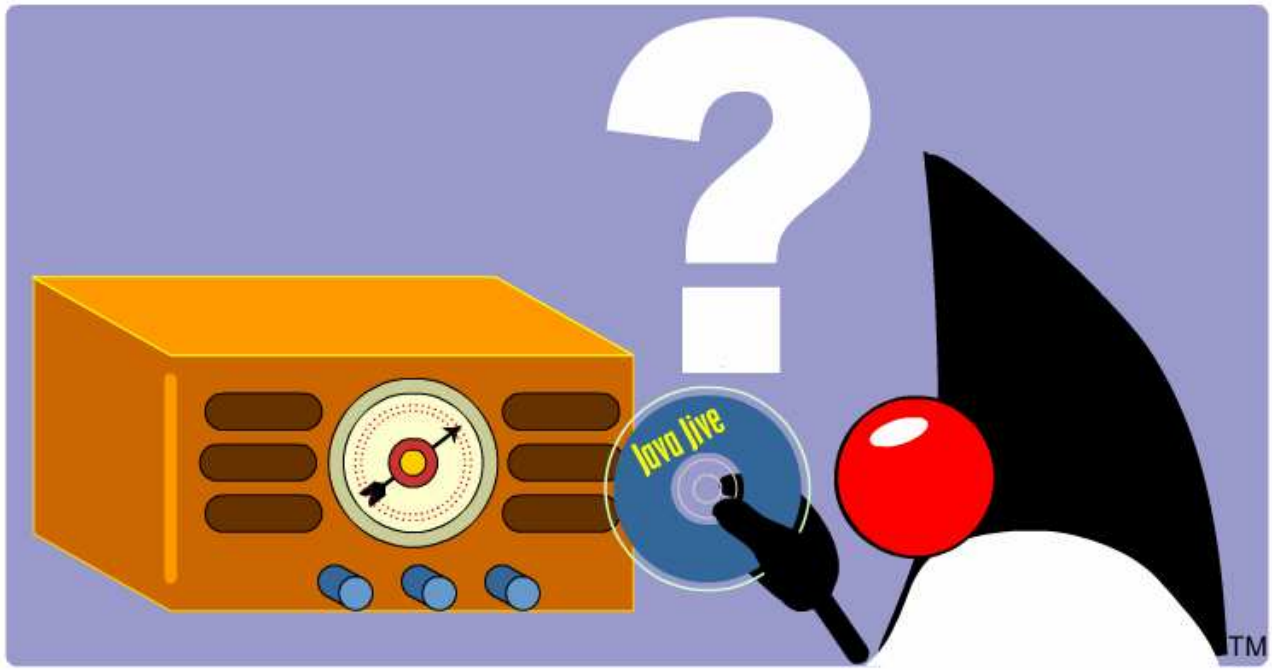
Inheritance

Features:

1) a class obtains variables and methods from another class
2) the former is called sub-class, the latter super-class
3) a sub-class provides a specialized behavior with respect to its super-class
4) inheritance faciliates code reuse and avoids duplication of data

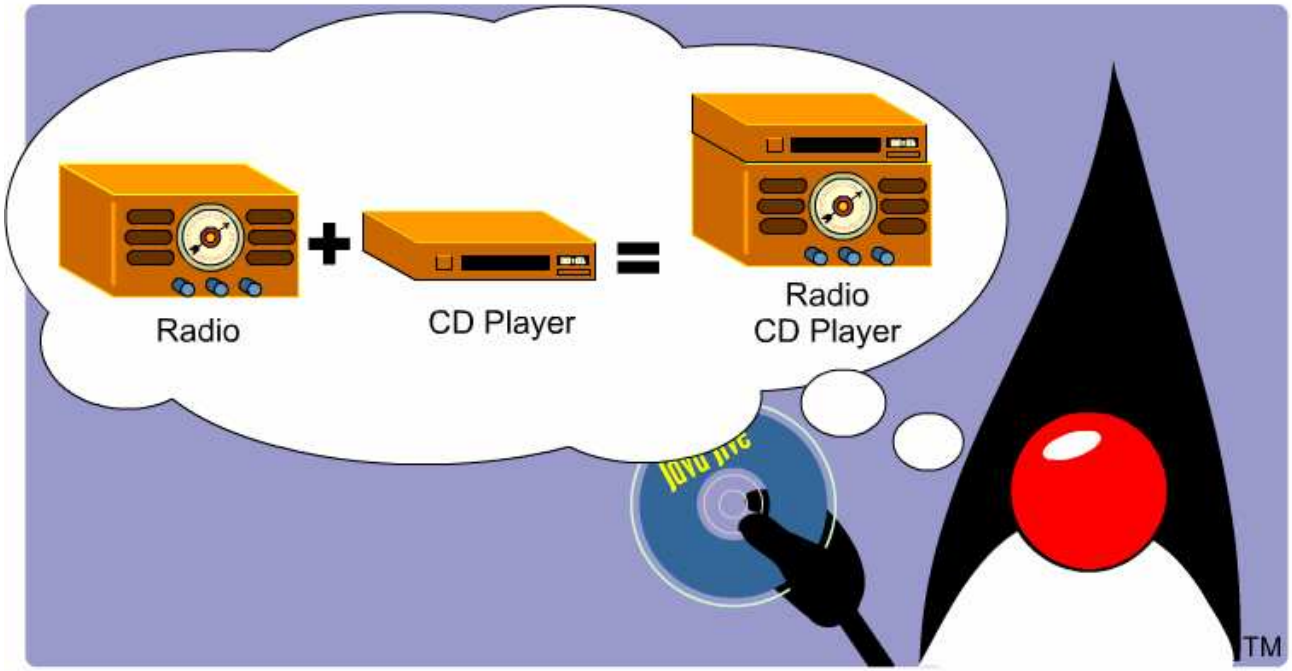


Inheritance 1



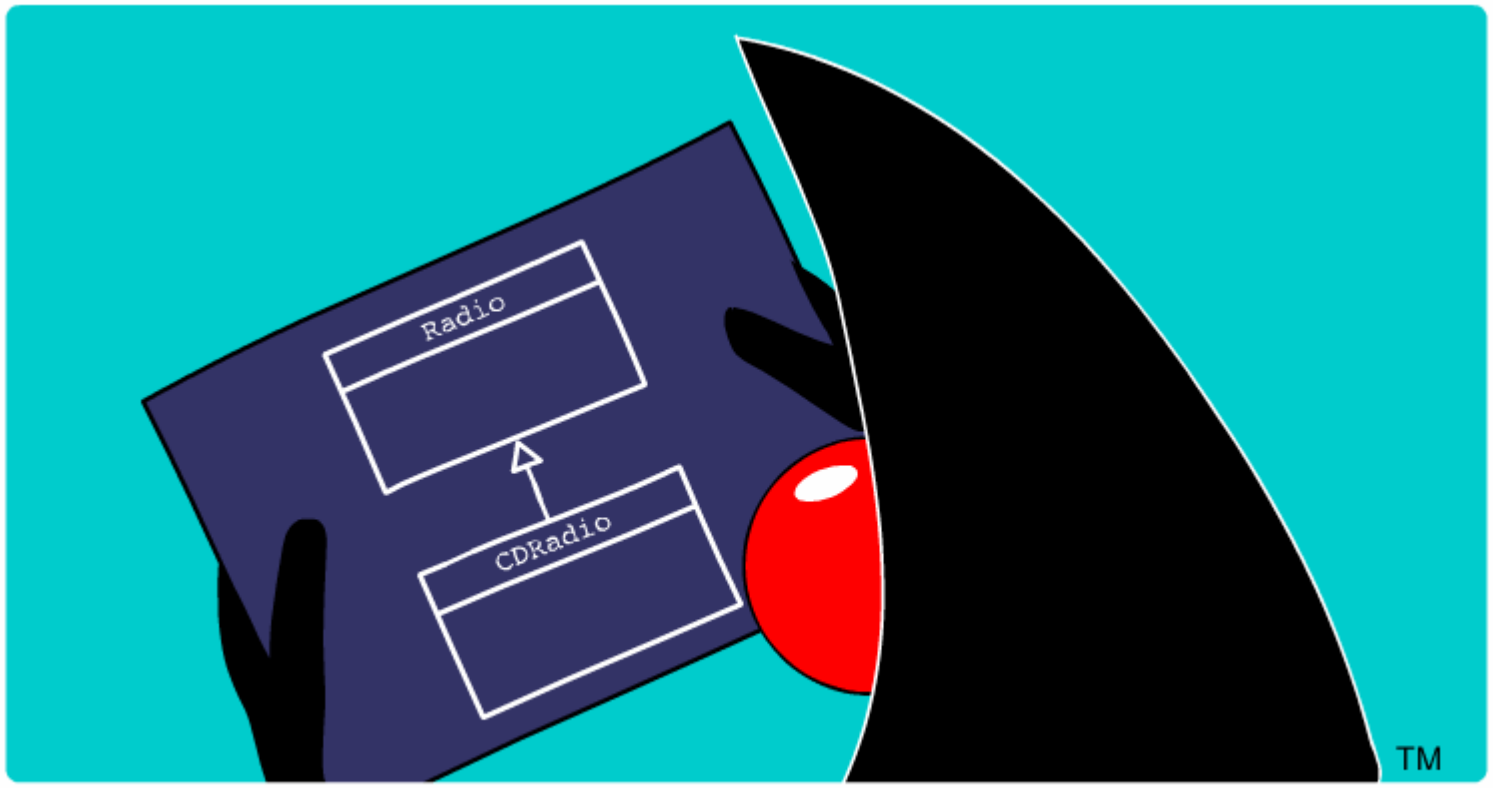
Duke would like to listen to his music CD, but his radio is not equipped with a CD player.

Inheritance 2



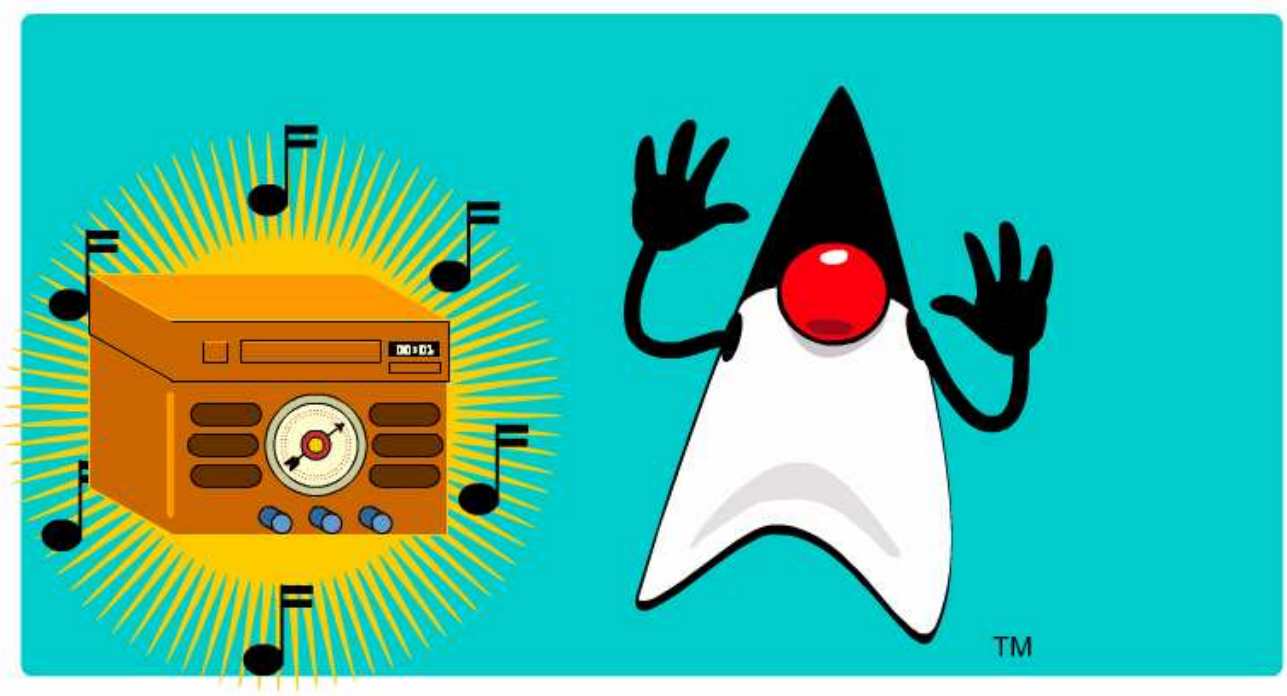
Duke decides he will create a specialized radio, one that plays CDs.

Inheritance 3



Duke creates a blueprint for a radio CD player.

Inheritance 4



Now Duke can enjoy listening to CDs on his radio CD player.

Polymorphism

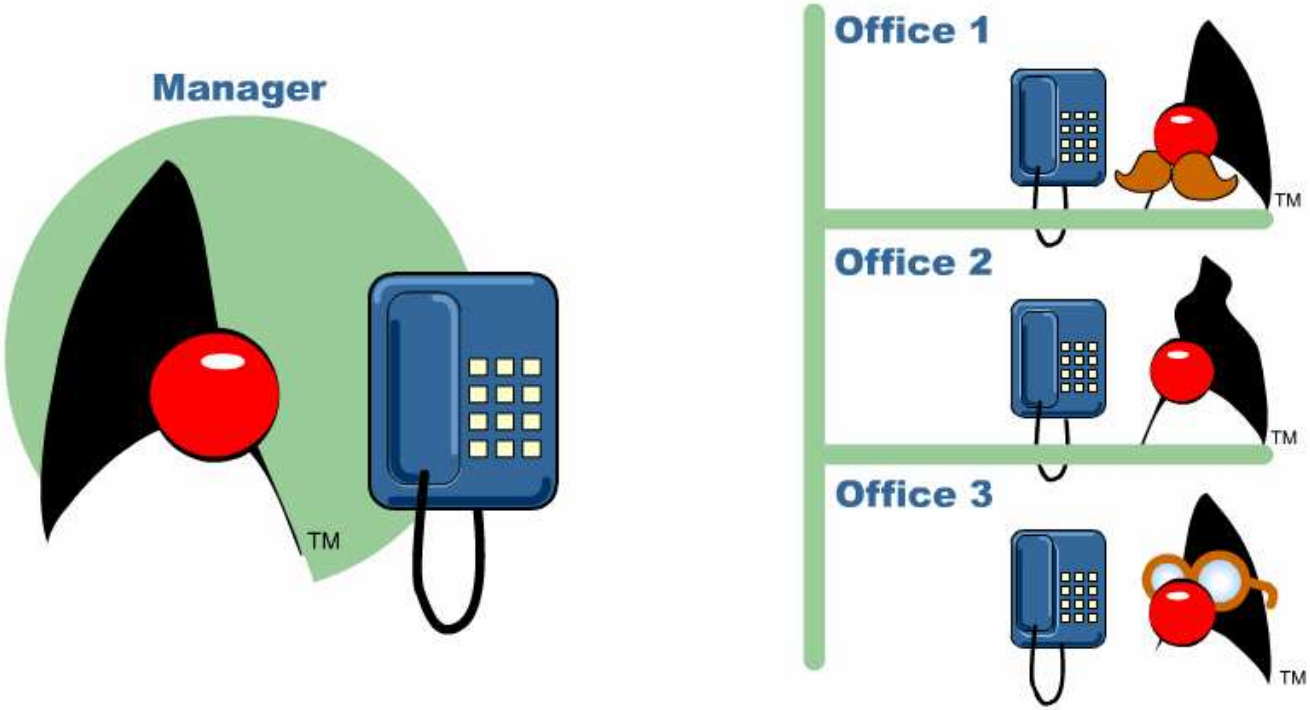
Polymorphism = many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked.

Polymorphism is enabled by inheritance:

1) a super-class defines an interface that all sub-classes must follow
2) it is up to the sub-classes how this interface is implemented; a sub-class may override methods of its super-class

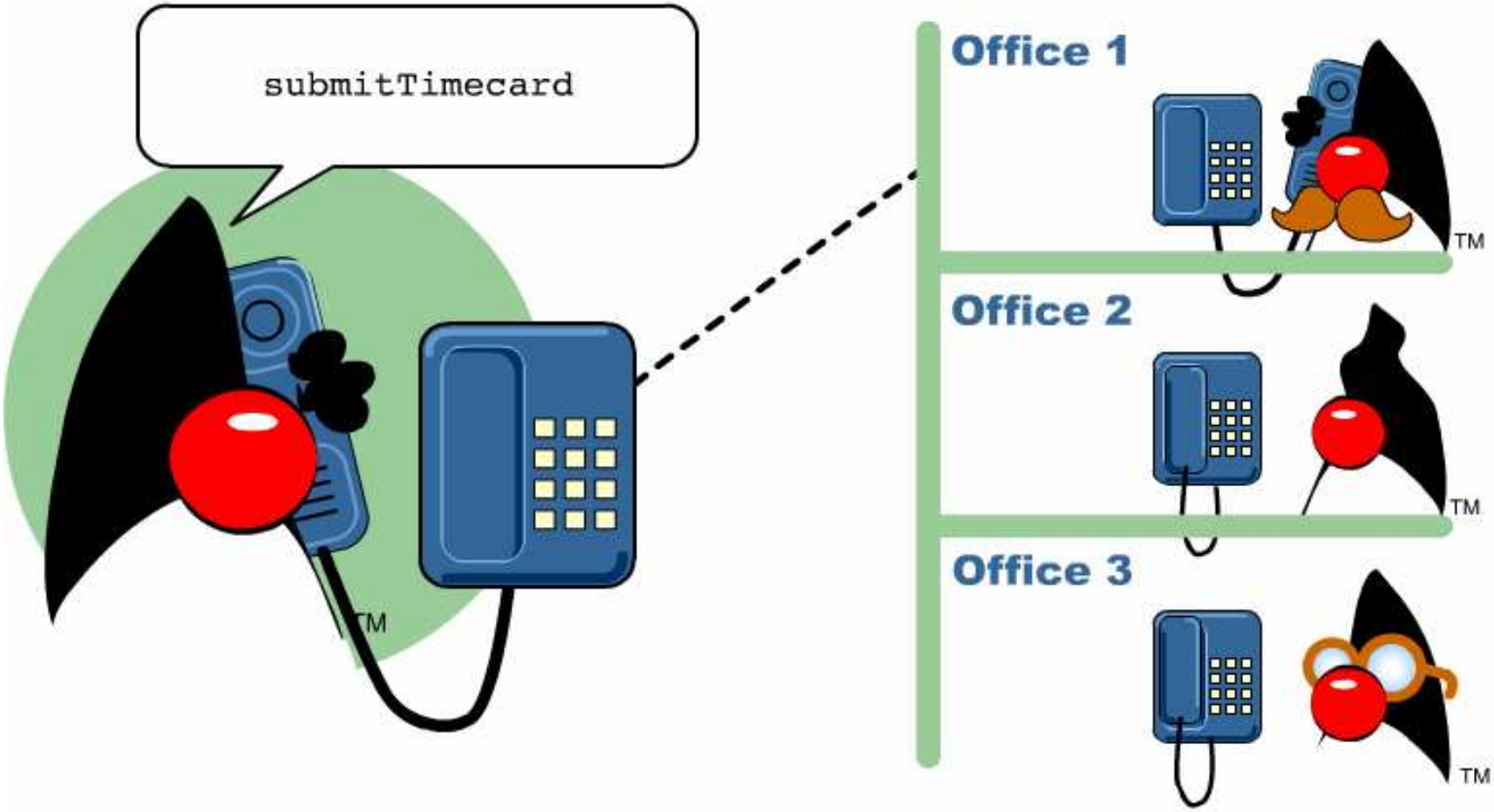
Therefore, objects from the classes related by inheritance may receive the same requests but respond to such requests in their own ways.

Polymorphism 1



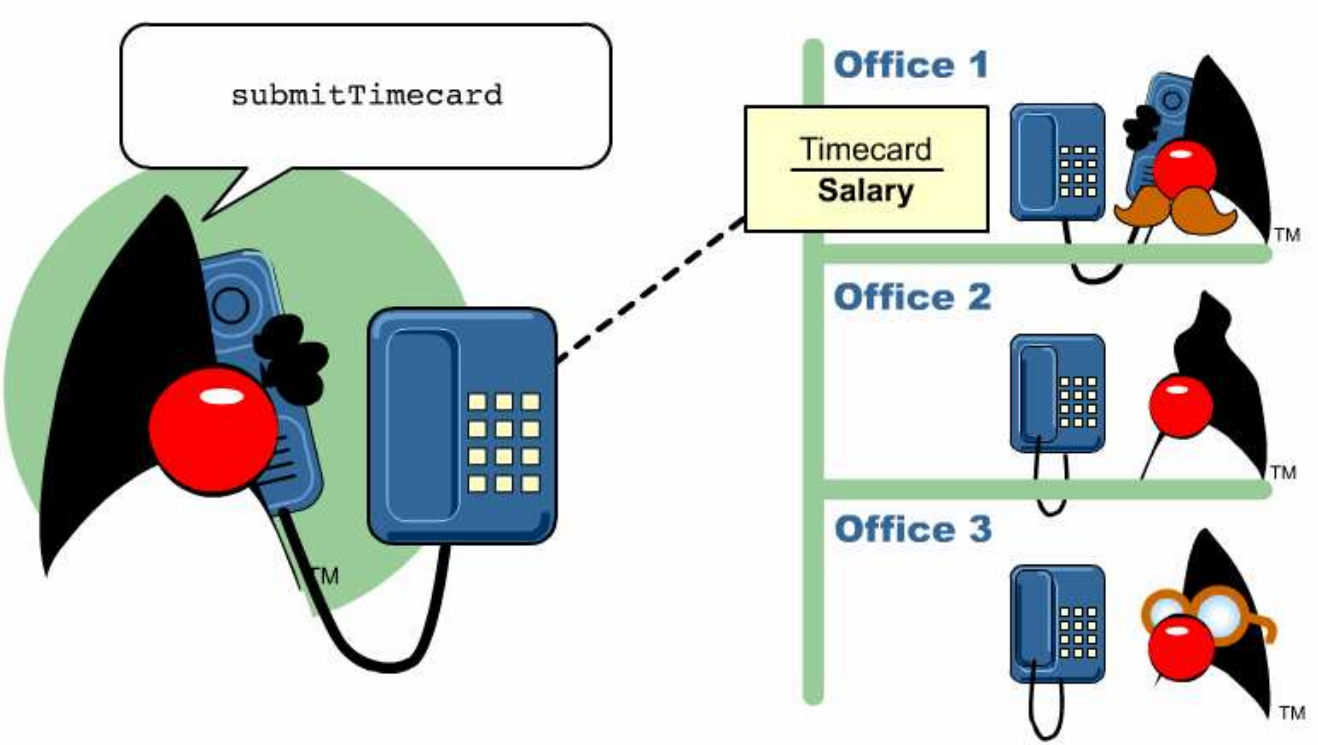
To illustrate polymorphism, imagine a manager with a phone connected to several employee offices.

Polymorphism 2



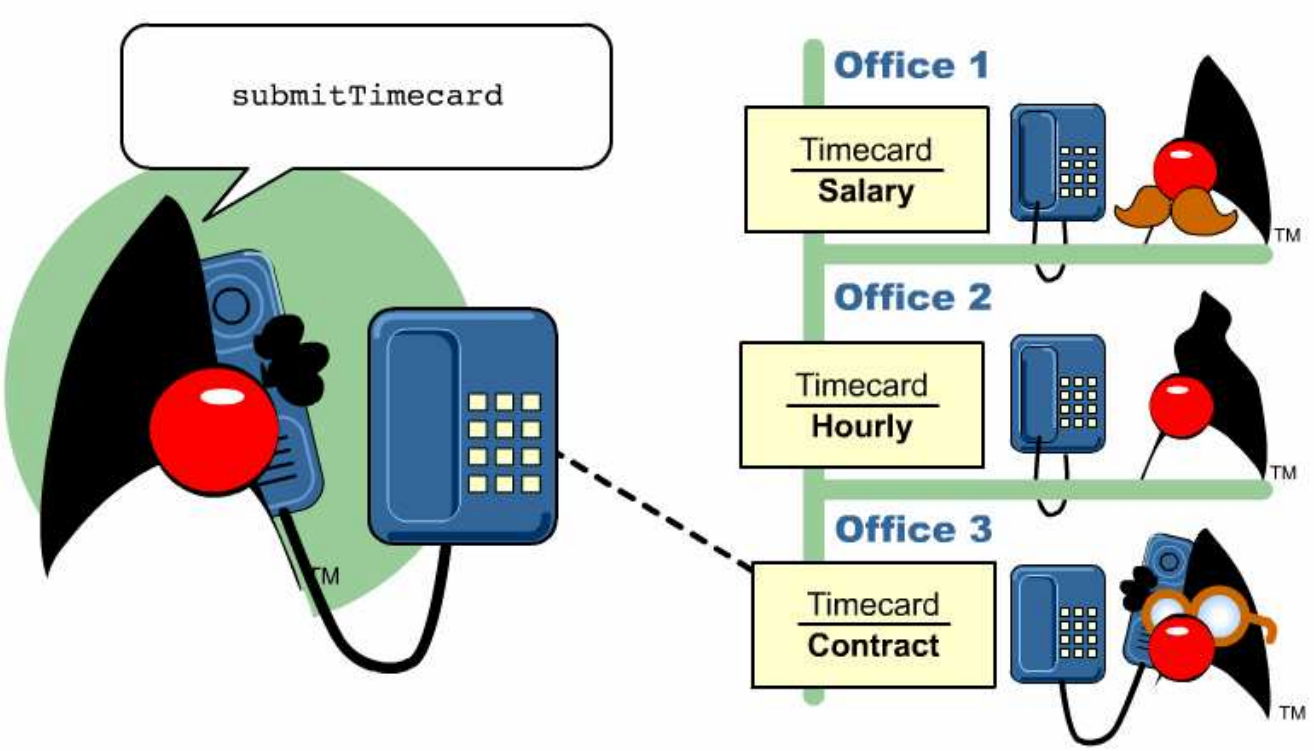
The manager places a call to the first office, and sends the `submitTimecard` method.

Polymorphism 3



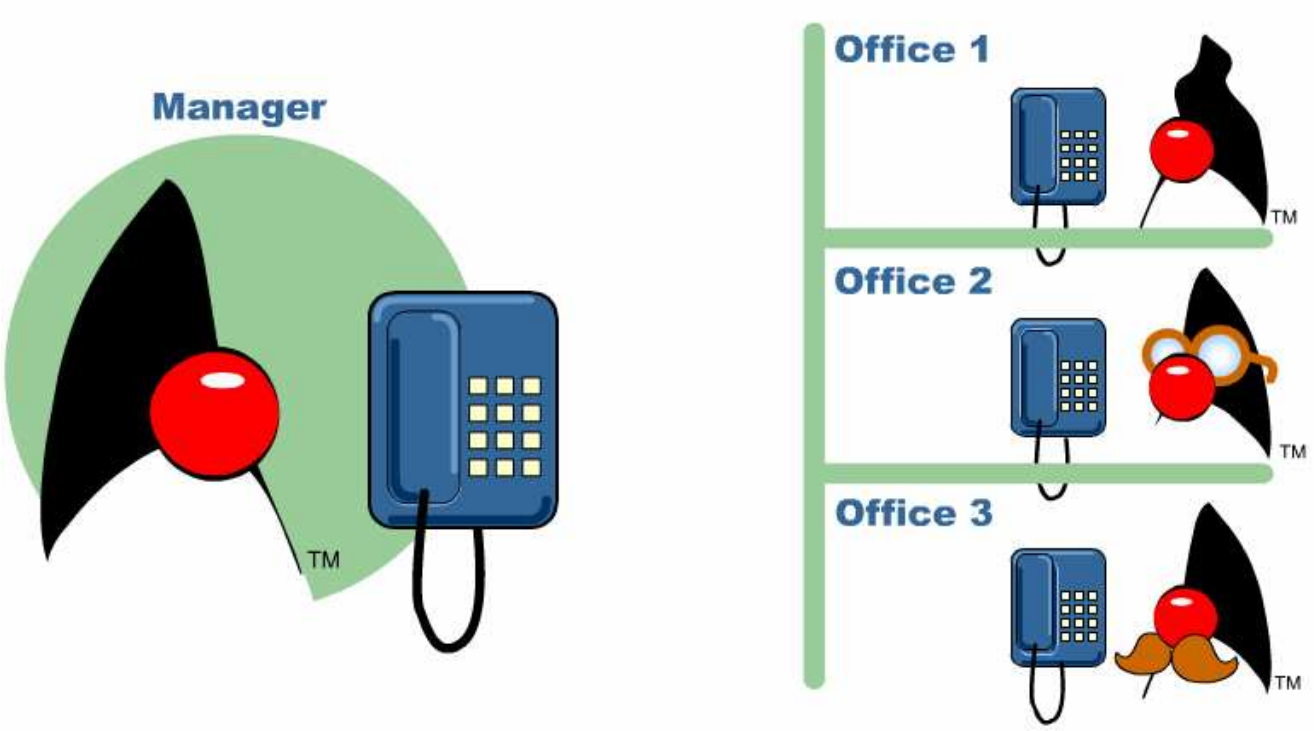
The employee in the first office is on salary. Salaried employees have their own method for submitting a timecard, which this employee follows.

Polymorphism 4



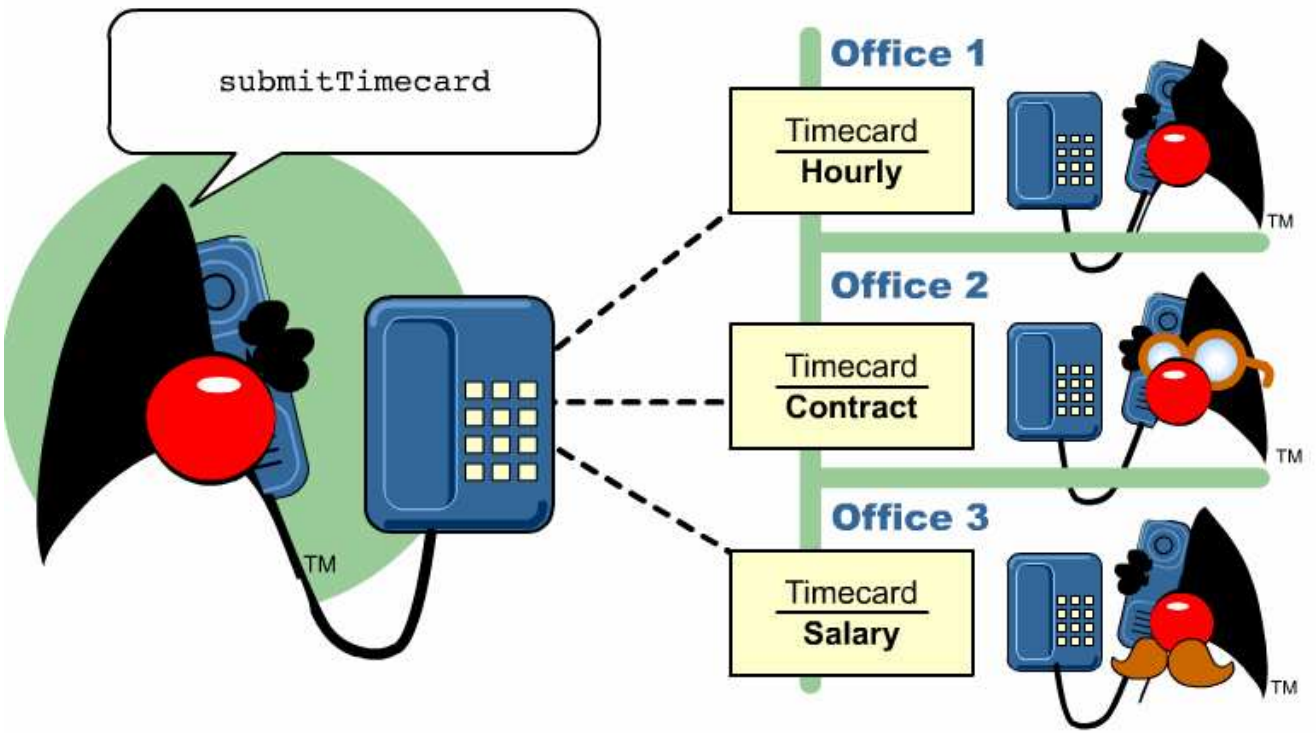
The manager places calls to each of the other offices. The employees in each office follow their own `submitTimecard` method.

Polymorphism 5



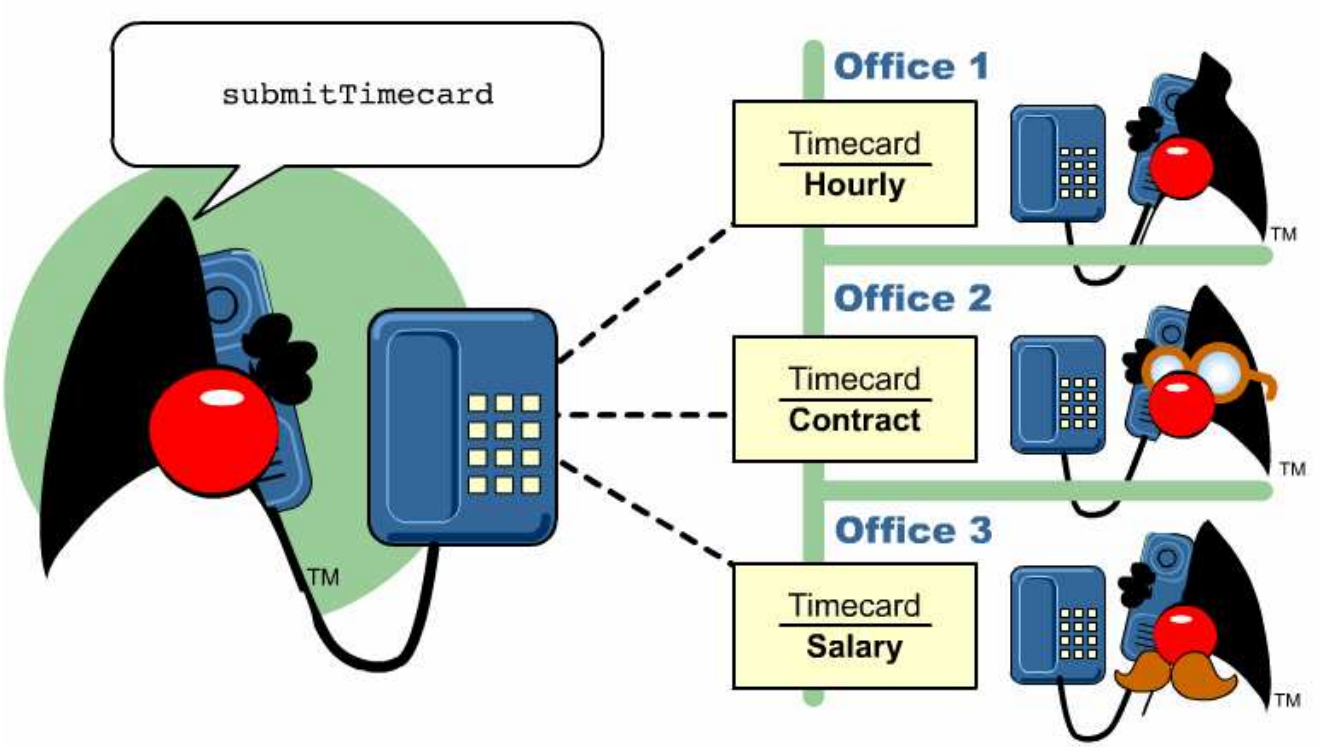
During the next pay period, the employees have switched offices.

Polymorphism 6



The manager can still place the same call to all the offices, knowing that the employees will follow their own `submitTimcard` method.

Polymorphism 7



Polymorphism is the ability for objects from separate, yet related classes to receive the same message, but act on it in their own way.

Exercise: OOP Concepts

1) Why Object-oriented programming?
2) Explain the three main concepts of Object-orientation.
3) How does Object-orientation enables data security?
4) Explain how to achieve Polymorphism through Inheritance.

Objects

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Objects

Everything in Java is an object.

Well ... almost.

Object lifecycle:

1) creation
2) usage
3) destruction

Object Creation

A variable s is declared to refer to the objects of type/class `String`:

```
String s;
```

The value of `s` is `null`; it does not yet refer to any object.

A new `String` object is created in memory with initial `"abc"` value:

```
String s = new String("abc");
```

Now `s` contains the address of this new object.

Object Usage

Objects are used mostly through variables.

Four usage scenarios:

1) one variable, one object
2) two variables, one object
3) two variables, two objects
4) one variable, two objects

One Variable, One Object

One variable, one object:

```
String s = new String("abc");
```

What can you do with the object addressed by `s`?

1) Check the length: `s.length() == 3`

2) Return the substring: `s.substring(2)`

3) etc.

Depending what is allowed by the definition of `String`.

Two Variables, One Object

Two variables, one object:

```
String s1 = new String("abc");
String s2;
```

Assignment copies the address, not value:

```
s2 = s1;
```

Now `s1` and `s2` both refer to one object. After

```
s1 = null;
```

`s2` still points to this object.

Two Variables, Two Objects

Two variables, two objects:

```
String s1 = new String("abc");
String s2 = new String("abc");
```

`s1` and `s2` objects have initially the same values:

```
s1.equals(s2) == true
```

But they are not the same objects:

```
(s1 == s2) == false
```

They can be changed independently of each other.

One Variable, Two Objects

One variable, two objects:

```
String s = new String("abc");
s = new String("cba");
```

The "abc" object is no longer accessible through any variable.

Object Destruction

A program accumulates memory through its execution.

Two mechanism to free memory that is no longer need by the program:

1) manual – done in C/C++
2) automatic – done in Java

In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.

Garbage collector is parts of the Java Run-Time Environment.

Exercise: Objects

1) Explain in detail the lifecycle of an object, making reference to it's
   a) creation
   b) usage and
   c) destruction
2) What is garbage collection?

Classes

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Class Outline

1) class definition
    a) attributes
    b) methods
    c) constructors
2) method overloading
3) method communication
    a) passing arguments
    b) returning results
4) recursive methods
5) arrays as object
6) static components
7) nested and internal classes

Class

A basis for the Java language.

Each concept we wish to describe in Java must be included inside a class.

A class defines a new data type, whose values are objects:

1) a class is a template for objects
2) an object is an instance of a class

Class Definition

A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.

General form of a class:

```
class classname {
   type instance-variable-1;
   …
   type instance-variable-n;

   type method-name-1(parameter-list) { … }
   type method-name-2(parameter-list) { … }
   …
   type method-name-m(parameter-list) { … }
}
```

Example: Class

A class with three variable members:

```
class Box {
   double width;
   double height;
   double depth;
}
```

A new `Box` object is created and a new value assigned to its width variable:

```
Box myBox = new Box();
myBox.width = 100;
```

Example: Class Usage

```
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    vol = mybox.width * mybox.height * mybox.depth;
    System.out.println("Volume is " + vol);
  }
}
```

Compilation and Execution

Place the `Box` class definitions in file `Box.java`:

```
class Box { … }
```

Place the `BoxDemo` class definitions in file `BoxDemo.java`:

```
class BoxDemo {
  public static void main(…) { … }
}
```

Compilation and execution:

```
> javac BoxDemo.java
> java BoxDemo
```

Variable Independence 1

Each object has its own copy of the instance variables: changing the variables of one object has no effect on the variables of another object.

Consider this example:

```
class BoxDemo2 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;

     mybox1.width = 10;
     mybox1.height = 20;
     mybox1.depth = 15;
```

Variable Independence 2

```
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

What are the printed volumes of both boxes?

Declaring Objects

Obtaining objects of a class is a two-stage process:

1) Declare a variable of the class type:

```
Box myBox;
```

   The value of `myBox` is a reference to an object, if one exists, or `null`. At this moment, the value of `myBox` is `null`.

2) Acquire an actual, physical copy of an object and assign its address to the variable. How to do this?

Operator new

Allocates memory for a `Box` object and returns its address:

```
Box myBox = new Box();
```

The address is then stored in the `myBox` reference variable.

`Box()` is a class constructor - a class may declare its own constructor or rely on the default constructor provided by the Java environment.

Memory Allocation

Memory is allocated for objects dynamically.

This has both advantages and disadvantages:

1) as many objects are created as needed
2) allocation is uncertain – memory may be insufficient

Variables of simple types do not require `new`:

```
int n = 1;
```

In the interest of efficiency, Java does not implement simple types as objects. Variables of simple types hold values, not references.

Assigning Reference Variables

Assignment copies address, not the actual value:

```
Box b1 = new Box();
Box b2 = b1;
```

Both variables point to the same object.

Variables are not in any way connected. After

```
b1 = null;
```

`b2` still refers to the original object.

Methods

General form of a method definition:

```
type name(parameter-list) {
   … return value; …
}
```

Components:

1) `type` - type of values returned by the method. If a method does not return any value, its return type must be `void`.
2) `name` is the name of the method
3) `parameter-list` is a sequence of type-identifier lists separated by commas
4) `return value` indicates what value is returned by the method.

Example: Method 1

Classes declare methods to hide their internal data structures, as well as for their own internal use:

Within a class, we can refer directly to its member variables:

```
class Box {
   double width, height, depth;
   void volume() {
      System.out.print("Volume is ");
      System.out.println(width * height * depth);
   }
}
```

Example: Method 2

When an instance variable is accessed by code that is not part of the class in which that variable is defined, access must be done through an object:

```
class BoxDemo3 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     mybox1.width = 10;    mybox2.width = 3;
     mybox1.height = 20;   mybox2.height = 6;
     mybox1.depth = 15;    mybox2.depth = 9;

     mybox1.volume();
     mybox2.volume();
  }
}
```

Value-Returning Method 1

The type of an expression returning value from a method must agree with the return type of this method:

```
class Box {
   double width;
   double height;
   double depth;

   double volume() {
      return width * height * depth;
   }
}
```

Value-Returning Method 2

```
class BoxDemo4 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;
     mybox1.width = 10;
     mybox2.width = 3;
     mybox1.height = 20;
     mybox2.height = 6;
     mybox1.depth = 15;
     mybox2.depth = 9;
```

Value-Returning Method 3

The type of a variable assigned the value returned by a method must agree with the return type of this method:

```
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Parameterized Method

Parameters increase generality and applicability of a method:

1) method without parameters

```
int square() { return 10*10; }
```

2) method with parameters

```
int square(int i) { return i*i; }
```

Parameter: a variable receiving value at the time the method is invoked.

Argument: a value passed to the method when it is invoked.

Example: Parameterized Method 1

```
class Box {
   double width;
   double height;
   double depth;

   double volume() {
      return width * height * depth;
   }

   void setDim(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
}
```

Example: Parameterized Method 2

```
class BoxDemo5 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;

     mybox1.setDim(10, 20, 15);
     mybox2.setDim(3, 6, 9);

     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

Constructor

A constructor initializes the instance variables of an object.

It is called immediately after the object is created but before the `new` operator completes.

1) it is syntactically similar to a method:
2) it has the same name as the name of its class
3) it is written without return type; the default return type of a class constructor is the same class

When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor 1

```
class Box {
   double width;
   double height;
   double depth;

   Box() {
      System.out.println("Constructing Box");
      width = 10; height = 10; depth = 10;
   }

   double volume() {
      return width * height * depth;
   }
}
```

Example: Constructor 2

```
class BoxDemo6 {
  public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

Parameterized Constructor 1

So far, all boxes have the same dimensions.

We need a constructor able to create boxes with different dimensions:

```
class Box {
   double width;
   double height;
   double depth;

   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }

   double volume() { return width * height * depth; }
}
```

Parameterized Constructor 2

```
class BoxDemo7 {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(3, 6, 9);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume is " + vol);
  }
}
```

finalize() Method

A constructor helps to initialize an object just after it has been created.

In contrast, the `finalize` method is invoked just before the object is destroyed:

1) implemented inside a class as:

```
protected void finalize() { … }
```

2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out

How is the `finalize` method invoked?

Garbage Collection

Garbage collection is a mechanism to remove objects from memory when they are no longer needed.

Garbage collection is carried out by the garbage collector:

1) The garbage collector keeps track of how many references an object has.
2) It removes an object from memory when it has no longer any references.
3) Thereafter, the memory occupied by the object can be allocated again.
4) The garbage collector invokes the `finalize` method.

Keyword this

Keyword `this` allows a method to refer to the object that invoked it.

It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

The above use of `this` is redundant but correct.

When is `this` really needed?

Instance Variable Hiding

Variables with the same names:

1) it is illegal to declare two local variables with the same name inside the same or enclosing scopes
2) it is legal to declare local variables or parameters with the same name as the instance variables of the class.

As the same-named local variables/parameters will hide the instance variables, using `this` is necessary to regain access to them:

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Example: Stack 1

A stack hold data in the first-in-last-out order.

Here is the implementation of a 10-element stack:

```
class Stack {
   int stck[] = new int[10];
   int tos;

   Stack() {
      tos = -1;
   }
```

Example: Stack 2

```
  void push(int item) {
     if (tos==9) System.out.println("Stack is full.");
     else stck[++tos] = item;
  }

  int pop() {
     if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else return stck[tos--];
  }
}
```

Example: Stack 3

```
class TestStack {
  public static void main(String args[]) {
     Stack mystack1 = new Stack();
     Stack mystack2 = new Stack();
     for (int i=0; i<10; i++) mystack1.push(i);
     for (int i=10; i<20; i++) mystack2.push(i);

     System.out.println("Stack in mystack1:");
     for(int i=0; i<10; i++)
        System.out.println(mystack1.pop());
     System.out.println("Stack in mystack2:");
     for(int i=0; i<10; i++)
        System.out.println(mystack2.pop());
  }
}
```

Method Overloading

It is legal for a class to have two or more methods with the same name.

However, Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.

Therefore the same-named methods must be distinguished:

1) by the number of arguments, or
2) by the types of arguments

Overloading and inheritance are two ways to implement polymorphism.

Example: Overloading 1

```
class OverloadDemo {
   void test() {
      System.out.println("No parameters");
   }
   void test(int a) {
      System.out.println("a: " + a);
   }
   void test(int a, int b) {
      System.out.println("a and b: " + a + " " + b);
   }
   double test(double a) {
      System.out.println("double a: " + a); return a*a;
   }
}
```

Example: Overloading 2

```
class Overload {
  public static void main(String args[]) {
     OverloadDemo ob = new OverloadDemo();
     double result;
     ob.test();
     ob.test(10);
     ob.test(10, 20);
     result = ob.test(123.2);
     System.out.println("ob.test(123.2): " + result);
  }
}
```

Different Result Types

Different result types are insufficient.

The following will not compile:

```
double test(double a) {
   System.out.println("double a: " + a);
   return a*a;
}

int test(double a) {
   System.out.println("double a: " + a);
   return (int) a*a;
}
```

Overloading and Conversion 1

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

When no exact match can be found, Java's automatic type conversion can aid overload resolution:

```
class OverloadDemo {
  void test() {
     System.out.println("No parameters");
  }
  void test(int a, int b) {
     System.out.println("a and b: " + a + " " + b);
  }
```

Overloading and Conversion 2

```
  void test(double a) {
     System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
     OverloadDemo ob = new OverloadDemo();
     int i = 88;
     ob.test();
     ob.test(10, 20);
     ob.test(i);
     ob.test(123.2);
  }
}
```

Overloading and Polymorphism

In the languages without overloading, methods must have a unique names:

```
int abs(int i)
long labs(int i)
float fabs(int i)
```

Java enables logically-related methods to occur under the same name:

```
static int abs(int i)
static long abs(long i)
static float abs(float i)
```

Constructor Overloading

Why overload constructors? Consider this:

```
class Box {
   double width, height, depth;

   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   double volume() {
      return width * height * depth;
   }
}
```

All `Box` objects can be created in one way: passing all three dimensions.

Example: Overloading 1

Three constructors: 3-parameter, 1-parameter, parameter-less.

```
class Box {
   double width, height, depth;
   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   Box() {
      width = -1;  height = -1;  depth = -1;
   }
   Box(double len) {
      width = height = depth = len;
   }
   double volume() { return width * height * depth; }
}
```

Example: Overloading 2

```
class OverloadCons {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box();
     Box mycube = new Box(7);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     vol = mycube.volume();
     System.out.println("Volume of mycube is " + vol);
  }
}
```

Object Argument 1

So far, all method received arguments of simple types.

They may also receive an object as an argument. Here is a method to check if a parameter object is equal to the invoking object:

```
class Test {
   int a, b;
   Test(int i, int j) {
      a = i; b = j;
   }
   boolean equals(Test o) {
      if (o.a == a && o.b == b) return true;
      else return false;
   }
}
```

Object Argument 2

```
class PassOb {
  public static void main(String args[]) {
     Test ob1 = new Test(100, 22);
     Test ob2 = new Test(100, 22);
     Test ob3 = new Test(-1, -1);
     System.out.println("ob1==ob2: " + ob1.equals(ob2));
     System.out.println("ob1==ob3: " + ob1.equals(ob3));
  }
}
```

Passing Object to Constructor 1

A special case of object-passing is passing an object to the constructor.

This is to initialize one object with another object:

```
class Box {
   double width, height, depth;

   Box(Box ob) {
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
```

Passing Object to Constructor 2

```
  Box(double w, double h, double d) {
     width = w;
     height = h;
     depth = d;
  }

  double volume() {
     return width * height * depth;
  }
}
```

Passing Object to Constructor 3

```
class OverloadCons2 {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(mybox1);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
  }
}
```

Argument-Passing

Two types of variables:

1) simple types
2) class types

Two corresponding ways of how the arguments are passed to methods:

1) by value – a method receives a cope of the original value; parameters of simple types
2) by reference – a method receives the memory address of the original value, not the value itself; parameters of class types

Simple Type Argument-Passing 1

Passing arguments of simple types takes place by value:

```
class Test {
   void meth(int i, int j) {
      i *= 2;
      j /= 2;
   }
}
```

Simple Type Argument-Passing 2

With by-value argument-passing what occurs to the parameter that receives the argument has no effect outside the method:

```
class CallByValue {
   public static void main(String args[]) {
      Test ob = new Test();
      int a = 15, b = 20;
      System.out.print("a and b before call: “);
      System.out.println(a + " " + b);
      ob.meth(a, b);
      System.out.print("a and b after call: ");
      System.out.println(a + " " + b);
   }
}
```

Class Type Argument-Passing 1

Objects are passed to the methods by reference: a parameter obtains the same address as the corresponding argument:

```
class Test {
   int a, b;

   Test(int i, int j) {
      a = i; b = j;
   }

   void meth(Test o) {
      o.a *=  2; o.b /= 2;
   }
}
```

Class Type Argument-Passing 2

As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:

```
class CallByRef {
   public static void main(String args[]) {
      Test ob = new Test(15, 20);
      System.out.print("ob.a and ob.b before call: “);
      System.out.println(ob.a + " " + ob.b);
      ob.meth(ob);
      System.out.print("ob.a and ob.b after call: ");
      System.out.println(ob.a + " " + ob.b);
   }
}
```

Returning Objects 1

So far, all methods returned no values or values of simple types.

Methods may also return objects:

```
class Test {
   int a;
   Test(int i) {
      a = i;
   }
   Test incrByTen() {
      Test temp = new Test(a+10);
      return temp;
   }
}
```

Returning Objects 2

Each time a method `incrByTen` is invoked a new object is created and a reference to it is returned:

```
class RetOb {
   public static void main(String args[]) {
      Test ob1 = new Test(2);
      Test ob2;
      ob2 = ob1.incrByTen();
      System.out.println("ob1.a: " + ob1.a);
      System.out.println("ob2.a: " + ob2.a);
      ob2 = ob2.incrByTen();
      System.out.print("ob2.a after second increase: “);
      System.out.println(ob2.a);
   }
}
```

Recursion

A recursive method is a method that calls itself:

What happens then?

1) all method parameters and local variables are allocated on the stack
2) arguments are prepared in the corresponding parameter positions
3) the method code is executed for the new arguments
4) upon return, all parameters and variables are removed from the stack
5) the execution continues immediately after the invocation point

Example: Recursion

```
class Factorial {
   int fact(int n) {
      if (n==1) return 1;
      return fact(n-1) * n;
   }
}

class Recursion {
   public static void main(String args[]) {
      Factorial f = new Factorial();
      System.out.print("Factorial of 5 is ");
      System.out.println(f.fact(5));
   }
}
```

Example: Recursion and Arrays 1

A method that recursively prints out a given number of elements in a table:

```
class RecTest {
   int values[];

   RecTest(int i) {
      values = new int[i];
   }
   void printArray(int i) {
      if (i==0) return;
      else printArray(i-1);
      System.out.print("[" + (i-1) + "] ");
      System.out.println(values[i-1]);
   }
}
```

Example: Recursion and Arrays 2

```
class Recursion2 {
  public static void main(String args[]) {
     RecTest ob = new RecTest(10);
     int i;
     for(i=0; i<10; i++)
        ob.values[i] = i;
     ob.printArray(10);
  }
}
```

Arrays Revisited

All arrays are implemented as objects.

In particular, all arrays have the `length` attribute to return the number of elements that an array may hold:

```
class Length {
  public static void main(String args[]) {
     int a1[] = new int[10];
     int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
     int a3[] = {4, 3, 2, 1};
     System.out.println("length of a1 is " + a1.length);
     System.out.println("length of a2 is " + a2.length);
     System.out.println("length of a3 is " + a3.length);
  }
}
```

Example: Arrays as Objects 1

An improved `Stack` example, able to handle stacks of any size:

```
class Stack {
  private int stck[];
  private int tos;

  Stack(int size) {
     stck = new int[size]; tos = -1;
  }

  void push(int item) {
     if (tos==stck.length-1)
        System.out.println("Stack is full.");
     else stck[++tos] = item;
  }
```

Example: Arrays as Objects 2

```
  int pop() {
     if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else
        return stck[tos--];
  }
}
```

Example: Arrays as Objects 3

```
class TestStack2 {
  public static void main(String args[]) {
     Stack mystack1 = new Stack(5);
     Stack mystack2 = new Stack(8);
     for (int i=0; i<5; i++) mystack1.push(i);
     for (int i=0; i<8; i++) mystack2.push(i);

     System.out.println("Stack in mystack1:");
     for (int i=0; i<5; i++)
        System.out.println(mystack1.pop());
     System.out.println("Stack in mystack2:");
     for (int i=0; i<8; i++)
        System.out.println(mystack2.pop());
  }
}
```

Static Class Members

Normally, the members of a class (its variables and methods) may be only used through the objects of this class.

Static members are independent of the objects:

1) variables
2) methods
3) initialization block

All declared with the `static` keyword.

Static Variables

Static variable:

```
static int a;
```

Essentially, it a global variable shared by all instances of the class.

It cannot be used within a non-static method.

Static Methods

Static method:

```
static void meth() { … }
```

Several restrictions apply:

- can only call static methods
- must only access static variables
- cannot refer to `this` or `super` in any way

Static Block

Static block:

```
static { … }
```

This is where the static variables are initialized.

The block is executed exactly once, when the class is first loaded.

Example: Static 1

```
class UseStatic {
   static int a = 3;
   static int b;
   static void meth(int x) {
      System.out.print("x = " + x + " a = " + a);
      System.out.println(" b = " + b);
   }
   static {
      System.out.println("Static block initialized.");
      b = a * 4;
   }
   public static void main(String args[]) {
      meth(42);
   }
}
```

Static Member Usage 1

How to use static members outside their class?

Consider this class:

```
class StaticDemo {
   static int a = 42;
   static int b = 99;
   static void callme() {
      System.out.println("a = " + a);
   }
}
```

Static Member Usage 2

Static variables/method are used through the class name:

```
StaticDemo.a
StaticDemo.callme()
```

Example:

```
class StaticByName {
   public static void main(String args[]) {
      StaticDemo.callme();
      System.out.println("b = " + StaticDemo.b);
   }
}
```

Nested Classes

It is possible to define a class within a class – nested class.

The scope of the nested class is its enclosing class: if class `B` is defined within class `A` then `B` is known to `A` but not outside.

Access rights:

1) a nested class has access to all members of its enclosing class, including its private members
2) the enclosing class does not have access to the members of the nested class

Types of Nested Classes

There are two types of nested classes:

1) static – cannot access the members of its enclosing class directly, but through an object; defined with the `static` keyword
2) non-static – has direct access to all members of the enclosing class in the same way as other non-static member of this class so

A static nested class is seldom used.

A non-static nested class is also called an inner class.

Example: Inner Classes 1

`Outer` has a variable `out_x`, an inner class `Inner` and a method `test` which creates an object of the `Inner` class and calls its `display` method:

```
class Outer {
   int outer_x = 100;
   void test() {
      Inner inner = new Inner();
      inner.display();
   }
   class Inner {
      void display() {
         System.out.println("outer_x = " + outer_x);
      }
   }
}
```

Example: Inner Classes 2

A demonstration class to create an object of the `Outer` class and invoke the `test` method on this object:

```
class InnerClassDemo {
   public static void main(String args[]) {
      Outer outer = new Outer();
      outer.test();
   }
}
```

The `Inner` class is only known within the `Outer` class. Any reference to `Inner` outside `Outer` will create a compile-time error.

Inner Members Visibility 1

Inner class has access to all member of the outer class.

The reverse is not true: members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

This is the `Outer` class with a variable, two methods and `Inner` class.

The first method refers to the `Inner` class correctly through an object:

```
class Outer {
   int outer_x = 100;
   void test() {
      Inner inner = new Inner();
      inner.display();
   }
```

Inner Members Visibility 2

`Inner` class declares variable `y` and refers to the `Outer` class variable:

```
  class Inner {
     int y = 10;
     void display() {
        System.out.println("outer_x = " + outer_x);
     }
  }
```

`showy` method refers incorrectly to the `Inner` class's `y` variable:

```
 void showy() {
    System.out.println(y);
  }
}
```

Inner Members Visibility 3

As a result, this program will not compile:

```
class InnerClassDemo {
   public static void main(String args[]) {
      Outer outer = new Outer();
      outer.test();
   }
}
```

Inner Class Declaration

So far, all inner classes were defined within the outer class scope.

In fact, an inner class may be defined within any block scope.

The following is an example of an inner class define within a for loop.

Example: Inner Class Declaration 1

```
class Outer {
   int outer_x = 100;
   void test() {
      for (int i=0; i<10; i++) {
         class Inner {
            void display() {
               System.out.println("outer_x= " + outer_x);
            }
         }
         Inner inner = new Inner();
         inner.display();
      }
   }
}
```

Example: Inner Class Declaration 2

A demonstration creates an `Outer` object and invokes a `test` method on it:

```
class InnerClassDemo {
  public static void main(String args[]) {
     Outer outer = new Outer();
     outer.test();
  }
}
```

Exercise: Classes 1

1) What's the difference between an object and a class?
2) Explain why constructors don't have return types.
3) Can *this* keyword be used within the context of a static method?. Why?.
4) What's Overloading?
5) Explain the mechanism of Argument passing in Java. Is it by value or by reference?
6) Explain the mechanism that Java uses to remove objects from memory when they are no longer needed.
7) An object has a handle to some non-Java resources such as file or window character font. How would you ensure that these resources are freed before the object is destroyed?

Exercise: Classes 2

1) Create a class with a default constructor (no arguments) that prints a message. Create an object of this class.
2) Add an overloaded constructor to the class in 1 which takes a String argument and prints it along with your message.
3) Make a two dimensional array of objects of the class you created in 2.
4) Create a class Counter with an adequate data member. Write four methods, one to get the value of the counter, one to increment the counter, one to decrement the counter, and one to reset the counter to zero. Make sure that the value of the counter never gets less than zero.
5) Create an object of your class Counter and assign it to two different variables. Change the state of the object, calling your method on one variable. See what happens if you call the get-value-method on the other variable.

Inheritance

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Inheritance

One of the pillars of object-orientation.

A new class is derived from an existing class:

1) existing class is called super-class
2) derived class is called sub-class

A sub-class is a specialized version of its super-class:

1) has all non-private members of its super-class
2) may provide its own implementation of super-class methods

Objects of a sub-class are a special kind of objects of a super-class.

Inheritance Syntax

Syntax:

```
class sub-class extends super-class {
   …
}
```

Each class has at most one super-class; no multi-inheritance in Java.

No class is a sub-class of itself.

Example: Super-Class

```
class A {
   int i;

   void showi() {
      System.out.println("i: " + i);
   }

}
```

Example: Sub-Class

```
class B extends A {

   int j;

   void showj() {
      System.out.println(“j: " + j);
   }

   void sum() {
      System.out.println("i+j: " + (i+j));
   }

}
```

Example: Testing Class

```
class SimpleInheritance {
  public static void main(String args[]) {
     A a = new A();
     B b = new B();
     a.i = 10;
     System.out.println("Contents of a: ");
     a.showi();
     b.i = 7; b.j = 8;
     System.out.println("Contents of b: ");
     subOb.showi(); subOb.showj();
     System.out.println("Sum of I and j in b:");
     b.sum();
  }
}
```

Inheritance and Private Members 1

A class may declare some of its members private.

A sub-class has no access to the private members of its super-class:

```
class A {
   int i;
   private int j;
   void setij(int x, int y) {
      i = x; j = y;
   }
}
```

Inheritance and Private Members 2

Class `B` has no access to the `A`'s private variable `j`.

This program will not compile:

```
class B extends A {
   int total;
   void sum() {
      total = i + j;
   }
}
```

Example: Box Class

The basic `Box` class with width, height and depth:

```
class Box {
   double width, height, depth;
   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   Box(Box b) {
      width = b.width;
      height = b.height; depth = b.depth;
   }
   double volume() {
      return width * height * depth;
   }
}
```

Example: BoxWeight Sub-Class

`BoxWeight` class extends `Box` with the new `weight` variable:

```
class BoxWeight extends Box {
   double weight;
   BoxWeight(double w, double h, double d, double m) {
      width = w; height = h; depth = d; weight = m;
   }
   BoxWeight(Box b, double w) {
      Box(b); weight = w;
   }
}
```

`Box` is a super-class, `BoxWeight` is a sub-class.

Example: BoxWeight Demo

```
class DemoBoxWeight {
  public static void main(String args[]) {
     BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
     BoxWeight mybox2 = new BoxWeight(mybox1);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     System.out.print("Weight of mybox1 is ");
     System.out.println(mybox1.weight);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     System.out.print("Weight of mybox2 is ");
     System.out.println(mybox2.weight);
  }
}
```

Another Sub-Class

Once a super-class exists that defines the attributes common to a set of objects, it can be used to create any number of more specific sub-classes.

The following sub-class of `Box` adds the `color` attribute instead of `weight`:

```
class ColorBox extends Box {
   int color;

   ColorBox(double w, double h, double d, int c) {
      width = w; height = h; depth = d;
      color = c;
   }
}
```

Referencing Sub-Class Objects

A variable of a super-class type may refer to any of its sub-class objects:

```
class SuperClass { … }
class SubClass extends SuperClass { … }

SuperClass o1;
SubClass o2 =  new SubClass();

o1 = o2;
```

However, the inverse is illegal:

```
o2 = o1
```

Example: Sub-Class Objects 1

```
class RefDemo {
  public static void main(String args[]) {
     BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
     Box plainbox = new Box(5, 5, 5);
     double vol;
     vol = weightbox.volume();
     System.out.print("Volume of weightbox is “);
     System.out.println(vol);
     System.out.print("Weight of weightbox is “);
     System.out.println(weightbox.weight);
     plainbox = weightbox;
     vol = plainbox.volume();
     System.out.println("Volume of plainbox is " + vol);
  }
}
```

Super-Class Variable Access

`plainbox` variable now refers to the `WeightBox` object.

Can we then access this object's `weight` variable through `plainbox`?

No. The type of a variable, not the object this variable refers to, determines which members we can access!

This is illegal:

```
System.out.print("Weight of plainbox is ");
System.out.println(plainbox.weight);
```

Super as a Constructor

Calling a constructor of a super-class from the constructor of a sub-class:

```
super(parameter-list);
```

Must occur as the very first instructor in the sub-class constructor:

```
class SuperClass { … }

class SubClass extends SuperClass {
   SubClass(…) {
      super(…);
      …
   }
   …
}
```

Example: Super Constructor 1

`BoxWeight` need not initialize the variable for the `Box` super-class, only the added `weight` variable:

```
class BoxWeight extends Box {
   double weight;

   BoxWeight(double w, double h, double d, double m) {
      super(w, h, d); weight = m;
   }

   BoxWeight(Box b, double w) {
      super(b); weight = w;
   }
}
```

Example: Super Constructor 2

```
class DemoSuper {
  public static void main(String args[]) {
     BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
     BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     System.out.print("Weight of mybox1 is ");
     System.out.println(mybox1.weight);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     System.out.print("Weight of mybox2 is ");
     System.out.println(mybox2.weight);
  }
}
```

Referencing Sub-Class Objects

Sending a sub-class object:

```
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
```

to the constructor expecting a super-class object:

```
BoxWeight(Box b, double w) {
    super(b); weight = w;
}
```

Uses of Super

Two uses of `super`:

1) to invoke the super-class constructor

```
super();
```

2) to access super-class members

```
super.variable;
super.method(…);
```

(1) was discussed, consider (2).

Super and Hiding

Why is `super` needed to access super-class members?

When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {
   int i = 1;
}

class B extends A {
   int i = 2;
   System.out.println("i is " + i);
}
```

The re-declared variables/methods hide those of the super-class.

Example: Super and Hiding 1

```
class A {
   int i;
}

class B extends A {
   int i;

   B(int a, int b) {
      super.i = a; i = b;
   }
   void show() {
      System.out.println("i in superclass: " + super.i);
      System.out.println("i in subclass: " + i);
   }
}
```

Example: Super and Hiding 2

Although the `i` variable in `B` hides the `i` variable in `A`, `super` allows access to the hidden variable of the super-class:

```
class UseSuper {
   public static void main(String args[]) {
      B subOb = new B(1, 2);
      subOb.show();
   }
}
```

Multi-Level Class Hierarchy 1

The basic `Box` class:

```
class Box {
   private double width, height, depth;
   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   Box(Box ob) {
      width = ob.width;
      height = ob.height; depth = ob.depth;
   }
   double volume() {
      return width * height * depth;
   }
}
```

Multi-Level Class Hierarchy 2

Adding the `weight` variable to the `Box` class:

```
class BoxWeight extends Box {
   double weight;

   BoxWeight(BoxWeight ob) {
      super(ob); weight = ob.weight;
   }

   BoxWeight(double w, double h, double d, double m) {
      super(w, h, d); weight = m;
   }

}
```

Multi-Level Class Hierarchy 3

Adding the `cost` variable to the `BoxWeight` class:

```
class Ship extends BoxWeight {
  double cost;

  Ship(Ship ob) {
    super(ob); cost = ob.cost;
  }

  Ship(double w, double h,
    double d, double m, double c) {
    super(w, h, d, m); cost = c;
  }
}
```

Multi-Level Class Hierarchy 4

```
class DemoShip {
  public static void main(String args[]) {
     Ship ship1 = new Ship(10, 20, 15, 10, 3.41);
     Ship ship2 = new Ship(2, 3, 4, 0.76, 1.28);
     double vol;

     vol = ship1.volume();
     System.out.println("Volume of ship1 is " + vol);
     System.out.print("Weight of ship1 is”);
     System.out.println(ship1.weight);
     System.out.print("Shipping cost: $");
     System.out.println(ship1.cost);
```

Multi-Level Class Hierarchy 5

```
        vol = ship2.volume();
        System.out.println("Volume of ship2 is " + vol);
        System.out.print("Weight of ship2 is ");
        System.out.println(ship2.weight);
        System.out.print("Shipping cost: $");
        System.out.println(ship2.cost);
    }
}
```

Constructor Call-Order

Constructor call-order:

1) first call super-class constructors
2) then call sub-class constructors

In the sub-class constructor, if `super(…)` is not used explicitly, Java calls the default, parameter-less super-class constructor.

Example: Constructor Call-Order 1

`A` is the super-class:

```
class A {
   A() {
      System.out.println("Inside A's constructor.");
   }
}
```

`B` and `C` are sub-classes of `A`:

```
class B extends A {
   B() {
      System.out.println("Inside B's constructor.");
   }
}
```

Example: Constructor Call-Order 2

```
class C extends B {
   C() {
      System.out.println("Inside C's constructor.");
   }
}
```

`CallingCons` creates a single object of the class `C`:

```
class CallingCons {
   public static void main(String args[]) {
      C c = new C();
   }
}
```

Exercise: Inheritance 1

1) Define a Class `Building` for building objects. Each building has a door as one of its components.
   a) In the class `Door`, model the fact that a door has a color and three states, "`open`" , "`closed`", "`locked`“ and “`unlocked`". To avoid illegal state changes, make the state private, write a method (`getState`) that inspects the state and four methods (open, close, lock and unlock) that change the state. Initialize the state to "closed" in the constructor. Look for an alternative place for this initialization.
   b) Write a method `enter` that visualizes the process of entering the building (unlock door, open door, enter, ...) by printing adequate messages, e.g. to show the state of the door.
   c) Write a corresponding method `quit` that visualizes the process of leaving the house. Don't forget to close and lock the door.
   d) Test your class by defining an object of type `Building` and visualizing the state changes when entering and leaving the building.

Exercise: Inheritance 2

3) Extend question 1 by introducing a subclass `HighBuilding` that contains an elevator and the height of the building in addition to the components of `Building`. Override the method `enter` to reflect the use of the elevator. Define a constructor that takes the height of the building as a parameter.
4) Define a subclass `Skyscraper` of `HighBuilding`, where the number of floors is stored with each skyscraper.

   What happens, if you don't define a constructor for class `Skyscraper` (Try it)?

   Write a constructor that takes the number of floors and the height as a parameter. Test the class by creating a skyscraper with 40 floors and using the inherited method `enter`.

Polymorphism

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Inheritance and Reuse

Reuse of code: every time a new sub-class is defined, programmers are reusing the code in a super-class.

All non-private members of a super-class are inherited by its sub-class:

1) an attribute in a super-class is inherited as-such in a sub-class

2) a method in a super-class is inherited in a sub-class:
   a) as-such, or
   b) is substituted with the method which has the same name and parameters (overriding) but a different implementation

Polymorphism: Definition

Polymorphism is one of three pillars of object-orientation.

Polymorphism: many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:

1) a super-class defines the common interface
2) sub-classes have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)

A sub-class provides a specialized behaviors relying on the common elements defined by its super-class.

Polymorphism: Behavior

Suppose we have a hierarchy of classes:

1) The top class in the hierarchy represents a common interface to all classes below. This class is the base class.
2) All classes below the base represent a number of forms of objects, all referred to by a variable of the base class type.

What happens when a method is invoked using the base class reference?

The object responds in accordance with its true type.

What if the user pointed to a different form of object using the same reference? The user would observe different behavior.

This is polymorphism.

Method Overriding

When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is overridden.

When an overridden method is called from within the sub-class:

1) it will always refer to the sub-class method
2) super-class method is hidden

Example: Hiding with Overriding 1

```
class A {
   int i, j;

   A(int a, int b) {
      i = a; j = b;
   }

   void show() {
      System.out.println("i and j: " + i + " " + j);
   }
}
```

Example: Hiding with Overriding 2

```
class B extends A {
   int k;

   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }

   void show() {
      System.out.println("k: " + k);
   }
}
```

Example: Hiding with Overriding 3

When `show()` is invoked on an object of type `B`, the version of `show()` defined in `B` is used:

```
class Override {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);
      subOb.show();
   }
}
```

The version of `show()` in `A` is hidden through overriding.

Super and Method Overriding

The hidden super-class method may be invoked using `super`:

```
class B extends A {
   int k;
   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }
   void show() {
      super.show();
      System.out.println("k: " + k);
   }
}
```

The super-class version of `show()` is called within the sub-class's version.

Overriding versus Overloading 1

Method overriding occurs only when the names and types of the two methods (super-class and sub-class methods) are identical.

If not identical, the two methods are simply overloaded:

```
class A {
   int i, j;

   A(int a, int b) {
      i = a; j = b;
   }
   void show() {
      System.out.println("i and j: " + i + " " + j);
   }
}
```

Overriding versus Overloading 2

The `show()` method in `B` takes a `String` parameter, while the `show()` method in `A` takes no parameters:

```
class B extends A {
   int k;

   B(int a, int b, int c) {
      super(a, b); k = c;
   }

   void show(String msg) {
      System.out.println(msg + k);
   }
}
```

Overriding versus Overloading 3

The two invocations of `show()` are resolved through the number of arguments (zero versus one):

```
class Override {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);

      subOb.show("This is k: ");
      subOb.show();
   }
}
```

Dynamic Method Invocation

Overriding is a lot more than the namespace convention.

Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.

Method overriding allows for dynamic method invocation:

1) an overridden method is called through the super-class variable
2) Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
3) when different types of objects are referred, different versions of the overridden method will be called.

Example: Dynamic Invocation 1

A super-class `A`:

```
class A {
   void callme() {
      System.out.println("Inside A's callme method");
   }
}
```

Example: Dynamic Invocation 2

Two sub-classes `B` and `C`:

```
class B extends A {
   void callme() {
      System.out.println("Inside B's callme method");
   }
}
class C extends A {
   void callme() {
      System.out.println("Inside C's callme method");
   }
}
```

`B` and `C` override the `A`'s `callme()` method.

Example: Dynamic Invocation 3

Overridden method is invoked through the variable of the super-class type. Each time, the version of the `callme()` method executed depends on the type of the object being referred to at the time of the call:

```
class Dispatch {
  public static void main(String args[]) {
     A a = new A();
     B b = new B();
     C c = new C();
     A r;
     r = a; r.callme();
     r = b; r.callme();
     r = c; r.callme();
  }
}
```

Polymorphism Again

One interface, many behaviors:

1) super-class defines common methods for sub-classes
2) sub-class provides specific implementations for some of the methods of the super-class

A combination of inheritance and overriding – sub-classes retain flexibility to define their own methods, yet they still have to follow a consistent interface.

Example: Polymorphism 1

A class that stores the dimensions of various 2-dimensional objects:

```
class Figure {
   double dim1;
   double dim2;
   Figure(double a, double b) {
      dim1 = a; dim2 = b;
   }
   double area() {
      System.out.println("Area is undefined.");
      return 0;
   }
}
```

Example: Polymorphism 2

`Rectangle` is a sub-class of `Figure`:

```
class Rectangle extends Figure {

   Rectangle(double a, double b) {
      super(a, b);
   }

   double area() {
      System.out.println("Inside Area for Rectangle.");
      return dim1 * dim2;
   }
}
```

Example: Polymorphism 3

`Triangle` is a sub-class of `Figure`:

```
class Triangle extends Figure {

   Triangle(double a, double b) {
      super(a, b);
   }

   double area() {
      System.out.println("Inside Area for Triangle.");
      return dim1 * dim2 / 2;
   }
}
```

Example: Polymorphism 4

Invoked through the `Figure` variable and overridden in their respective sub-classes, the `area()` method returns the area of the invoking object:

```
class FindAreas {
  public static void main(String args[]) {
     Figure f = new Figure(10, 10);
     Rectangle r = new Rectangle(9, 5);
     Triangle t = new Triangle(10, 8);
     Figure figref;
     figref = r; System.out.println(figref.area());
     figref = t; System.out.println(figref.area());
     figref = f; System.out.println(figref.area());
  }
}
```

Abstract Method

Inheritance allows a sub-class to override the methods of its super-class.

In fact, a super-class may altogether leave the implementation details of a method and declare such a method abstract:

```
abstract type name(parameter-list);
```

Two kinds of methods:

1) concrete – may be overridden by sub-classes
2) abstract – must be overridden by sub-classes

It is illegal to define abstract constructors or static methods.

Example: Abstract Method

The `area` method cannot compute the area of an arbitrary figure:

```
double area() {
    System.out.println("Area is undefined.");
    return 0;
}
```

Instead, `area` should be defined abstract in `Figure`:

```
abstract double area() ;
```

Abstract Class

A class that contains an abstract method must be itself declared abstract:

```
abstract class abstractClassName {
    abstract type methodName(parameter-list) {
        …
    }
    …
}
```

An abstract class has no instances - it is illegal to use the `new` operator:

```
abstractClassName a = new abstractClassName();
```

It is legal to define variables of the abstract class type.

Abstract Sub-Class

A sub-class of an abstract class:

1) implements all abstract methods of its super-class, or
2) is also declared as an abstract class

```
abstract class A {
   abstract void callMe();
}

abstract class B extends A {
   int checkMe;
}
```

Abstract and Concrete Classes 1

Abstract super-class, concrete sub-class:

```
abstract class A {
   abstract void callme();
   void callmetoo() {
      System.out.println("This is a concrete method.");
   }
}

class B extends A {
   void callme() {
      System.out.println("B's implementation.");
   }
}
```

Abstract and Concrete Classes 2

Calling concrete and overridden abstract methods:

```
class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Example: Abstract Class 1

`Figure` is an abstract class; it contains an abstract `area` method:

```
abstract class Figure {
   double dim1;
   double dim2;

   Figure(double a, double b) {
      dim1 = a; dim2 = b;
   }

   abstract double area();
}
```

Example: Abstract Class 2

`Rectangle` is concrete – it provides a concrete implementation for `area`:

```
class Rectangle extends Figure {
   Rectangle(double a, double b) {
      super(a, b);
   }

   double area() {
      System.out.println("Inside Area for Rectangle.");
      return dim1 * dim2;
   }
}
```

Example: Abstract Class 3

Triangle is concrete – it provides a concrete implementation for `area`:

```
class Triangle extends Figure {
   Triangle(double a, double b) {
      super(a, b);
   }

   double area() {
      System.out.println("Inside Area for Triangle.");
      return dim1 * dim2 / 2;
   }
}
```

Example: Abstract Class 4

Invoked through the `Figure` variable and overridden in their respective sub-classes, the `area()` method returns the area of the invoking object:

```
class AbstractAreas {
  public static void main(String args[]) {
     Rectangle r = new Rectangle(9, 5);
     Triangle t = new Triangle(10, 8);

     Figure figref;

     figref = r; System.out.println(figref.area());
     figref = t; System.out.println(figref.area());
  }
}
```

Abstract Class References

It is illegal to create objects of the abstract class:

```
Figure f = new Figure(10, 10);
```

It is legal to create a variable with the abstract class type:

```
Figure figref;
```

Later, `figref` may be used to assign references to any object of a concrete sub-class of `Figure` (e.g. `Rectangle`) and to invoke methods of this class:

```
Rectangle r = new Rectangle(9, 5);
figref = r; System.out.println(figref.area());
```

Uses of final

The final keyword has three uses:

1) declare a variable which value cannot change after initialization
2) declare a method which cannot be overridden in sub-classes
3) declare a class which cannot have any sub-classes

(1) has been discussed before.

Now is time for (2) and (3).

Preventing Overriding with final

A method declared `final` cannot be overridden in any sub-class:

```
class A {
   final void meth() {
      System.out.println("This is a final method.");
   }
}
```

This class declaration is illegal:

```
class B extends A {
   void meth() {
      System.out.println("Illegal!");
   }
}
```

final and Early Binding

Two types of method invocation:

1) early binding – method call is decided at compile-time
2) late binding – method call is decided at run-time

By default, method calls are resolved at run-time.

As a final method cannot be overridden, their invocations are resolved at compile-time. This is one way to improve performance of a method call.

Preventing Inheritance with final

A class declared `final` cannot be inherited – has no sub-classes.

```
final class A { … }
```

This class declaration is considered illegal:

```
class B extends A { … }
```

Declaring a class `final` implicitly declares all its methods `final`.

It is illegal to declare a class as both `abstract` and `final`.

Object Class

`Object` class is a super-class of all Java classes:

1) `Object` is the root of the Java inheritance hierarchy.
2) A variable of the `Object` type may refer to objects of any class.
3) As arrays are implemented as objects, it may also refer to any array.

Object Class Methods 1

Methods declared in the `Object` class:

1) `Object clone()` - creates an object which is an ideal copy of the invoking object.
2) `boolean equals(Object object)` - determines if the invoking object and the argument object are the same.
3) `void finalize()` – called before an unused object is recycled
4) `Class getClass()` – obtains the class description of an object at run-time
5) `int hashCode()` – returns the hash code for the invoking object

Object Class Methods 2

6) `void notify()` – resumes execution of a thread waiting on the invoking object
7) `void notifyAll()` – resumes execution of all threads waiting on the invoking object
8) `String toString()` – returns the string that describes the invoking object
9) three methods to wait on another thread of execution:
   a) `void wait()`
   b) `void wait(long milliseconds)`
   c) `void wait(long milliseconds, int nanoseconds)`

Overriding Object Class Methods

All methods except `getClass`, `notify`, `notifyAll` and `wait` can be overridden.

Two methods are frequently overridden:

1) `equals()`
2) `toString()`

This way, classes can tailor the equality and the textual description of objects to their own specific structure and needs.

Exercise: Polymorphism

1) Define a relationship among the following `Building`, `HighBuilding` and `Skyscraper` classes.

2) Define a class `Visits` that stores an array of 10 buildings (representing a street).

3) Define a method that enters all the buildings in the street using the method `enter`, one after another.

4) Fill the array with mixed objects from the classes `Building`, `HighBuilding` and `Skyscraper`.

   Make sure, that the output of your program visualizes the fact that different method implementations are used depending on the type of the actual object.

Access

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Name Space Management

Classes written so far all belong to a single name space: a unique name has to be chosen for each class to avoid name collision.

Some way to manage the name space is needed to:

1) ensure that the names are unique
2) provide a continuous supply of convenient, descriptive names
3) ensure that the names chosen by one programmer will not collide with those chosen by another programmers

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is a package.

Package

A package is both a naming and a visibility control mechanism:

1) divides the name space into disjoint subsets

   It is possible to define classes within a package that are not accessible by code outside the package.

2) controls the visibility of classes and their members

   It is possible to define class members that are only exposed to other members of the same package.

Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages.

Package Definition

A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

means that all classes in this file belong to the `myPackage` package. The package statement creates a name space where such classes are stored.

When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

Other files may include the same package instruction:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

```
package myPackage;
class MyClass3{ … }
```

A package may be distributed through several source files.

Packages and Directories

Java uses file system directories to store packages.

Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

The bytecode files `MyClass1.class` and `MyClass2.class` must be stored in a directory `myPackage`.

Case is significant! Directory names must match package names exactly.

Package Hierarchy

To create a package hierarchy, separate each package name with a dot:

```
package myPackage1.myPackage2.myPackage3;
```

A package hierarchy must be stored accordingly in the file system:

1) Unix `myPackage1/myPackage2/myPackage3`
2) Windows `myPackage1\myPackage2\myPackage3`
3) Macintosh `myPackage1:myPackage2:myPackage3`

You cannot rename a package without renaming its directory!

Finding Packages

As packages are stored in directories, how does the Java run-time system know where to look for packages?

Two ways:

1) The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.
2) Specify a directory path or paths by setting the `CLASSPATH` environment variable.

CLASSPATH Variable

`CLASSPATH` - environment variable that points to the root directory of the system's package hierarchy.

Several root directories may be specified in `CLASSPATH`, e.g. the current directory and the `C:\myJava` directory:

```
.;C:\myJava
```

Java will search for the required packages by looking up subsequent directories described in the `CLASSPATH` variable.

Finding Packages

Consider this package statement:

```
package myPackage;
```

In order for a program to find `myPackage`, one of the following must be true:

1) program is executed from the directory immediately above `myPackage` (the parent of `myPackage` directory)
2) `CLASSPATH` must be set to include the path to `myPackage`

Example: Package 1

```
package MyPack;

class Balance {
   String name;
   double bal;
   Balance(String n, double b) {
      name = n; bal = b;
   }
   void show() {
      if (bal<0) System.out.print("-->> ");
      System.out.println(name + ": $" + bal);
   }
}
```

Example: Package 2

```
class AccountBalance {
  public static void main(String args[]) {
     Balance current[] = new Balance[3];

     current[0] = new Balance("K. J. Fielding", 123.23);
     current[1] = new Balance("Will Tell", 157.02);
     current[2] = new Balance("Tom Jackson", -12.33);

     for (int i=0; i<3; i++) current[i].show();
  }
}
```

Example: Package 3

Save, compile and execute:

1) call the file `AccountBalance.java`
2) save the file in the directory `MyPack`
3) compile; `AccountBalance.class` should be also in `MyPack`
4) set access to `MyPack` in `CLASSPATH` variable, or make the parent of `MyPack` your current directory
5) run:

```
java MyPack.AccountBalance
```

Make sure to use the package-qualified class name.

Importing of Packages

Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.

The import statement allows to use classes or whole packages directly.

Importing of a concrete class:

```
import myPackage1.myPackage2.myClass;
```

Importing of all classes within a package:

```
import myPackage1.myPackage2.*;
```

Access Control

Classes and packages are both means of encapsulating and containing the name space and scope of classes, variables and methods:

1) packages act as a container for classes and other packages
2) classes act as a container for data and code

Access control is set separately for classes and class members.

Access Control: Classes

Two levels of access:

1) A class available in the whole program:

```
public class MyClass { … }
```

2) A class available within the same package only:

```
class MyClass { … }
```

Access Control: Members

Four levels of access:

1) a member is available in the whole program:

```
public int variable;
public int method(…) { … }
```

2) a member is only available within the same class:

```
private int variable;
private int method(…) { … }
```

Access Control: Members

3) a member is available within the same package (default access):

```
int variable;
int method(…) { … }
```

4) a member is available within the same package as the current class, or within its sub-classes:

```
protected int variable;
protected int method(…) { … }
```

The sub-class may be located inside or outside the current package.

Access Control Summary

Complicated?

Any member declared `public` can be accessed from anywhere.

Any member declared `private` cannot be seen outside its class.

When a member does not have any access specification (default access), it is visible to all classes within the same package.

To make a member visible outside the current package, but only to sub-classes of the current class, declare this member `protected`.

Table: Access Control

	`private`	default	`protected`	`public`
same class	yes	yes	yes	yes
same package subclass	no	yes	yes	yes
same package non-sub-class	no	yes	yes	yes
different package sub-class	no	no	yes	yes
different package non-sub-class	no	no	no	yes

Example: Access 1

Access example with two packages `p1` and `p2` and five classes.

A public `Protection` class is in the package `p1`.

It has four variables with four possible access rights:

```
package p1;

public class Protection {
   int n = 1;
   private int n_pri = 2;
   protected int n_pro = 3;
   public int n_pub = 4;
```

Example: Access 2

```
  public Protection() {
      System.out.println("base constructor");
      System.out.println("n = " + n);
      System.out.println("n_pri = " + n_pri);
      System.out.println("n_pro = " + n_pro);
      System.out.println("n_pub = " + n_pub);
  }
}
```

The rest of the example tests the access to those variables.

Example: Access 3

`Derived` class is in the same `p1` package and is the sub-class of `Protection`.

It has access to all variables of `Protection` except the private `n_pri`:

```
package p1;

class Derived extends Protection {
  Derived() {
     System.out.println("derived constructor");
     System.out.println("n = " + n);
     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
  }
}
```

Example: Access 4

`SamePackage` is in the `p1` package but is not a sub-class of `Protection`.

It has access to all variables of `Protection` except the private `n_pri`:

```
package p1;

class SamePackage {
   SamePackage() {
      Protection p = new Protection();
      System.out.println("same package constructor");
      System.out.println("n = " + p.n);
      System.out.println("n_pro = " + p.n_pro);
      System.out.println("n_pub = " + p.n_pub);
   }
}
```

Example: Access 5

`Protection2` is a sub-class of `p1.Protection`, but is located in a different package – package `p2`.

`Protection2` has access to the public and protected variables of `Protection`. It has no access to its private and default-access variables:

```
package p2;

class Protection2 extends p1.Protection {
   Protection2() {
      System.out.println("derived other package");
      System.out.println("n_pro = " + n_pro);
      System.out.println("n_pub = " + n_pub);
   }
}
```

Example: Access 6

`OtherPackage` is in the `p2` package and is not a sub-class of `p1.Protection`.

`OtherPackage` has access to the public variable of `Protection` only. It has no access to its private, protected or default-access variables:

```
class OtherPackage {
  OtherPackage() {
     p1.Protection p = new p1.Protection();
     System.out.println("other package constructor");
     System.out.println("n_pub = " + p.n_pub);
  }
}
```

Example: Access 7

A demonstration to use classes of the `p1` package:

```
package p1;

public class Demo {
   public static void main(String args[]) {
      Protection ob1 = new Protection();
      Derived ob2 = new Derived();
      SamePackage ob3 = new SamePackage();
   }
}
```

Example: Access 8

A demonstration to use classes of the `p2` package:

```
package p2;

public class Demo {
   public static void main(String args[]) {
      Protection2 ob1 = new Protection2();
      OtherPackage ob2 = new OtherPackage();
   }
}
```

Import Statement

The import statement occurs immediately after the package statement and before the class statement:

```
package myPackage;
import otherPackage1;otherPackage2.otherClass;
class myClass { … }
```

The Java system accepts this import statement by default:

```
import java.lang.*;
```

This package includes the basic language functions. Without such functions, Java is of no much use.

Name Conflict 1

Suppose a same-named class occurs in two different imported packages:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass { … otherClass … }
```

```
package otherPackage1;
class otherClass { … }
```

```
package otherPackage2;
class otherClass { … }
```

Name Conflict 2

Compiler will remain silent, unless we try to use `otherClass`.
Then it will display an error message.

In this situation we should use the full name:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass {
   …
   otherPackage1.otherClass
   …
   otherPackage2.otherClass
   …
}
```

Short versus Full References

Short reference:

```
import java.util.*;
class MyClass extends Date { … }
```

Full reference:

```
class MyClass extends java.util.Date { … }
```

Only the `public` components in imported package are accessible for non-sub-classes in the importing code!

Example: Packages 1

A package `MyPack` with one `public` class `Balance`. The class has two same-package variables: `public` constructor and a `public show` method.

```
package MyPack;
public class Balance {
   String name;
   double bal;
   public Balance(String n, double b) {
      name = n; bal = b;
   }
   public void show() {
      if (bal<0) System.out.print("-->> ");
      System.out.println(name + ": $" + bal);
   }
}
```

Example: Packages 2

The importing code has access to the `public` class `Balance` of the `MyPack` package and its two public members:

```
import MyPack.*;

class TestBalance {
  public static void main(String args[]) {
     Balance test = new Balance("J. J. Jaspers", 99.88);
     test.show();
  }

}
```

Java Source File

Finally, a Java source file consists of:

1) a single package instruction (optional)
2) several import statements (optional)
3) a single public class declaration (required)
4) several classes private to the package (optional)

At the minimum, a file contains a single public class declaration.

Exercise: Access

1) Create a package `emacao`. Don't forget to insert your package into a directory of the same name. Insert a class `AccessTest` into this packge. Define `public`, `default` and `private` data members and methods in your class `AccessTest`.
2) Define a second class `Accessor1` in your package that accesses the different kinds of data members of methods (`private`, `public`, `default`).

   See what compiler messages you get.

3) Define class `Accessor2` outside the package. Again try to access all methods and data members of the class `AccessTest`.

   See what compiler messages you get.

4) Where are the differences between `Accessor1` and `Accessor2` ?

Interfaces

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Interface

Using interface, we specify what a class must do, but not how it does this.

An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.

An interface is defined with an `interface` keyword.

Interface Format

General format:

```
access interface name {
   type method-name1(parameter-list);
   type method-name2(parameter-list);
   …
   type var-name1 = value1;
   type var-nameM = valueM;
   …
}
```

Interface Comments

Two types of access:

1) `public` – interface may be used anywhere in a program
2) default – interface may be used in the current package only

Interface methods have no bodies – they end with the semicolon after the parameter list. They are essentially abstract methods.

An interface may include variables, but they must be `final`, `static` and initialized with a constant value.

In a `public` interface, all members are implicitly `public`.

Interface Implementation

A class implements an interface if it provides a complete set of methods defined by this interface.

1) any number of classes may implement an interface
2) one class may implement any number of interfaces

Each class is free to determine the details of its implementation.

Implementation relation is written with the `implements` keyword.

Implementation Format

General format of a class that includes the `implements` clause:

```
access class name
   extends super-class
   implements interface1, interface2, …, interfaceN {
   …
}
```

Access is `public` or default.

Implementation Comments

If a class implements several interfaces, they are separated with a comma.

If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.

The methods that implement an interface must be declared `public`.

The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: Interface

Declaration of the `Callback` interface:

```
interface Callback {
   void callback(int param);
}
```

`Client` class implements the `Callback` interface:

```
class Client implements Callback {
   public void callback(int p) {
      System.out.println("callback called with " + p);
   }
}
```

More Methods in Implementation

An implementing class may also declare its own methods:

```
class Client implements Callback {
   public void callback(int p) {
      System.out.println("callback called with " + p);
   }

   void nonIfaceMeth() {
      System.out.println("Classes that implement “ +
         “interfaces may also define ” +
         “other members, too.");
   }
}
```

Interface as a Type

Variable may be declared with interface as its type:

```
interface MyInterface { … }
…
MyInterface mi;
```

The variable of an interface type may reference an object of any class that implements this interface.

```
class MyClass implements MyInterface { … }

MyInterface mi = new MyClass();
```

Call Through Interface Variable

Using the interface type variable, we can call any method in the interface:

```
interface MyInterface {
    void myMethod(…) ;
    …
}
class MyClass implements MyInterface { … }
…
MyInterface mi = new MyClass();
…
mi.myMethod();
```

The correct version of the method will be called based on the actual instance of the interface being referred to.

Example: Call Through Interface 1

Declaration of the `Callback` interface:

```
interface Callback {
   void callback(int param);
}
```

`Client` class implements the `Callback` interface:

```
class Client implements Callback {
   public void callback(int p) {
      System.out.println("callback called with " + p);
   }
}
```

Example: Call Through Interface 2

`TestIface` declares the `Callback` interface variable, initializes it with the new `Client` object, and calls the `callback` method through this variable:

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

Call Through Interface Variable 2

Call through an interface variable is one of the key features of interfaces:

1) the method to be executed is looked up dynamically at run-time
2) the calling code can dispatch through an interface without having to know anything about the callee

Allows classes to be created later than the code that calls methods on them.

Example: Interface Call 1

Another implementation of the `Callback` interface:

```
class AnotherClient implements Callback {
  public void callback(int p) {
     System.out.println("Another version of callback");
     System.out.println("p squared is " + (p*p));
  }
}
```

Example: Interface Call 2

`Callback` variable `c` is assigned `Client` and later `AnotherClient` objects and the corresponding `callback` is invoked depending on its value:

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
    AnotherClient ob = new AnotherClient();
    c = ob;
    c.callback(42);
  }
}
```

Compile-Time Method Binding

Normally, in order for a method to be called from one class to another, both classes must be present at compile time.

This implies:

1) a static, non-extensible classing environment
2) functionality gets pushed higher and higher in the class hierarchy to make them available to more sub-classes

Run-Time Method Binding

Interfaces support dynamic method binding.

Interface disconnects the method definition from the inheritance hierarchy:

1) interfaces are in a different hierarchy from classes
2) it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface

Interface and Abstract Class

A class that claims to implement an interface but does not implement all its methods must be declared abstract.

`Incomplete` class implements the `Callback` interface but not its `callback` method, so the class is declared `abstract`:

```
abstract class Incomplete implements Callback {
   int a, b;
   void show() {
      System.out.println(a + " " + b);
   }
}
```

Example: Stack Interface

Many ways to implement a stack but one interface:

```
interface IntStack {
   void push(int item);
   int pop();
}
```

Lets look at two implementations of this interface:

1) `FixedStack` – a fixed-length version of the integer stack
2) `DynStack` – a dynamic-length version of the integer stack

Example: FixedStack 1

A fixed-length stack implements the `IntStack` interface with two private variables, a constructor and two public methods:

```
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  FixedStack(int size) {
     stck = new int[size]; tos = -1;
  }
```

Example: FixedStack 2

```
  public void push(int item) {
     if (tos==stck.length-1)
        System.out.println("Stack is full.");
     else stck[++tos] = item;
  }

  public int pop() {
     if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else return stck[tos--];
  }
}
```

Example: FixedStack 3

A testing class creates two stacks:

```
class IFTest {
  public static void main(String args[]) {
     FixedStack mystack1 = new FixedStack(5);
     FixedStack mystack2 = new FixedStack(8);
```

Example: FixedStack 4

It pushes and them pops off some values from those stacks:

```
        for (int i=0; i<5; i++) mystack1.push(i);
        for (int i=0; i<8; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for (int i=0; i<5; i++)
           System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for (int i=0; i<8; i++)
           System.out.println(mystack2.pop());
    }
}
```

Example: DynStack 1

Another implementation of an integer stack.

A dynamic-length stack is first created with an initial length. The stack is doubled in size every time this initial length is exceeded.

```
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  DynStack(int size) {
     stck = new int[size];
     tos = -1;
  }
```

Example: DynStack 2

If stack if full, `push` creates a new stack with double the size of the old stack:

```
public void push(int item) {
   if (tos==stck.length-1) {
      int temp[] = new int[stck.length * 2];
      for (int i=0; i<stck.length; i++)
         temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
   }
   else stck[++tos] = item;
}
```

Example: DynStack 3

If the stack is empty, `pop` returns the zero value:

```
  public int pop() {
     if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
     }
     else return stck[tos--];
  }
}
```

Example: DynStack 4

The testing class creates two dynamic-length stacks:

```
class IFTest2 {
  public static void main(String args[]) {
     DynStack mystack1 = new DynStack(5);
     DynStack mystack2 = new DynStack(8);
```

Example: DynStack 5

It then pushes some numbers onto those stacks, dynamically increasing their size, then pops those numbers off:

```
        for (int i=0; i<12; i++) mystack1.push(i);
        for (int i=0; i<20; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for (int i=0; i<12; i++)
           System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for (int i=0; i<20; i++)
           System.out.println(mystack2.pop());
    }
}
```

Example: Two Stacks 1

Testing two stack implementations through an interface variable.

First, some numbers are pushed onto both stacks:

```
class IFTest3 {
  public static void main(String args[]) {
     IntStack mystack;
     DynStack ds = new DynStack(5);
     FixedStack fs = new FixedStack(8);

     mystack = ds;
     for (int i=0; i<12; i++) mystack.push(i);
     mystack = fs;
     for (int i=0; i<8; i++) mystack.push(i);
```

Example: Two Stacks 2

Then, those numbers are popped off:

```
        mystack = ds;
        System.out.println("Values in dynamic stack:");
        for (int i=0; i<12; i++)
           System.out.println(mystack.pop());
        mystack = fs;
        System.out.println("Values in fixed stack:");
        for (int i=0; i<8; i++)
           System.out.println(mystack.pop());
    }
}
```

Which stack implementation is the value of the `mystack` variable, therefore which version of `push` and `pop` are used, is determined at run-time.

Interface Variables

Variables declared in an interface must be constants.

A technique to import shared constants into multiple classes:

1) declare an interface with variables initialized to the desired values
2) include that interface in a class through implementation

As no methods are included in the interface, the class does not implement anything except importing the variables as constants.

Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;

interface SharedConstants {
   int NO = 0;
   int YES = 1;
   int MAYBE = 2;
   int LATER = 3;
   int SOON = 4;
   int NEVER = 5;
}
```

Example: Interface Variables 2

`Question` implements `SharedConstants`, including all its constants.

Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
     int prob = (int) (100 * rand.nextDouble());
     if (prob < 30)     return NO;
     else if (prob < 60) return YES;
     else if (prob < 75) return LATER;
     else if (prob < 98) return SOON;
     else return NEVER;
  }
}
```

Example: Interface Variables 3

`AskMe` includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {
  static void answer(int result) {
     switch(result) {
        case NO:    System.out.println("No"); break;
        case YES:   System.out.println("Yes"); break;
        case MAYBE: System.out.println("Maybe"); break;
        case LATER: System.out.println("Later"); break;
        case SOON:  System.out.println("Soon"); break;
        case NEVER: System.out.println("Never"); break;
     }
  }
```

Example: Interface Variables 4

The testing function relies on the fact that both `ask` and `answer` methods, defined in different classes, rely on the same constants:

```
   public static void main(String args[]) {
      Question q = new Question();
      answer(q.ask());
      answer(q.ask());
      answer(q.ask());
      answer(q.ask());
   }
}
```

Interface Inheritance

One interface may inherit another interface.

The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
    void myMethod1(…) ;
}
interface MyInterface2 extends MyInterface1 {
    void myMethod2(…) ;
}
```

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Inheritance and Implementation

When a class implements an interface that inherits another interface, it must provide implementations for all inherited methods:

```
class MyClass implements MyInterface2 {
    void myMethod1(…)   { … }
    void myMethod1(…)   { … }
    …
}
```

Example: Interface Inheritance 1

Consider interfaces `A` and `B`.

```
interface A {
   void meth1();
   void meth2();
}
```

`B` extends `A`:

```
interface B extends A {
   void meth3();
}
```

Example: Interface Inheritance 2

`MyClass` must implement all of `A` and `B` methods:

```
class MyClass implements B {
   public void meth1() {
      System.out.println("Implement meth1().");
   }
   public void meth2() {
      System.out.println("Implement meth2().");
   }
   public void meth3() {
      System.out.println("Implement meth3().");
   }
}
```

Example: Interface Inheritance 3

Create a new `MyClass` object, then invoke all interface methods on it:

```
class IFExtend {
  public static void main(String arg[]) {
     MyClass ob = new MyClass();
     ob.meth1();
     ob.meth2();
     ob.meth3();
  }
}
```

Exercise: Interface

1) Define two interfaces:
   a) an interface `CardUse` for card use with methods `read` to read the state and `reduceBy` with a parameter amount to change the state of the card. The user has to identify himself by a PIN for this operation;
   b) an interface `CardChange` for the administration of the card by authorized people, that need a method `reset` to reset the card state, a method `fill` to fill the card with an amount of money and a method `changePIN` to change the PIN for the card.

2) Define a third interface `CardAll` that includes all card operation (Use inheritance).

3) Change the interface `CardAll` into an abstract class that implements the balance of the card and a basic solution for the methods `fill` and `reduceBy` leaving the rest of the methods abstract. Choose the correct access specifier to make the balance accessible to subclasses but not to the public; Check this;

Exception-Handling

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Exceptions

Exception is an abnormal condition that arises when executing a program.

In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.

In contrast, Java:

1) provides syntactic mechanisms to signal, detect and handle errors
2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
3) brings run-time error management into object-oriented programming

Exception Handling

An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.

Exception handling involves the following:

1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
2) that method may choose to handle the exception itself or pass it on
3) either way, at some point, the exception is caught and processed

Exception Sources

Exceptions can be:

1) generated by the Java run-time system

   Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

2) manually generated by programmer's code

   Such exceptions are typically used to report some error conditions to the caller of a method.

Exception Constructs

Five constructs are used in exception handling:

1) `try` – a block surrounding program statements to monitor for exceptions
2) `catch` – together with try, catches specific kinds of exceptions and handles them in some way
3) `finally` – specifies any code that absolutely must be executed whether or not an exception occurs
4) `throw` – used to throw a specific exception from the program
5) `throws` – specifies which exceptions a given method can throw

Exception-Handling Block

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…
finally { … }
```

where:

1) `try { … }` is the block of code to monitor for exceptions
2) `catch(Exception ex) { … }` is exception handler for the exception `Exception`
3) `finally { … }` is the block of code to execute before the `try` block ends

Exception Hierarchy

All exceptions are sub-classes of the build-in class `Throwable`.

`Throwable` contains two immediate sub-classes:

1) `Exception` – exceptional conditions that programs should catch

   The class includes:

   a) `RuntimeException` – defined automatically for user programs to include: division by zero, invalid array indexing, etc.
   b) use-defined exception classes

2) `Error` – exceptions used by Java to indicate errors with the run-time environment; user programs are not supposed to catch them

Uncaught Exception

What happens when exceptions are not handled?

```
class Exc0 {
  public static void main(String args[]) {
     int d = 0;
     int a = 42 / d;
  }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and throws this object.

This will cause the execution of `Exc0` to stop – once an exception has been thrown it must be caught by an exception handler and dealt with.

Default Exception Handler

As we have not provided any exception handler, the exception is caught by the default handler provided by the Java run-time system.

This default handler:

1) displays a string describing the exception,
2) prints the stack trace from the point where the exception occurred
3) terminates the program

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Any exception not caught by the user program is ultimately processed by the default handler.

Stack Trace Display

The stack trace displayed by the default error handler shows the sequence of method invocations that led up to the error.

Here the exception is raised in `subroutine()` which is called by `main()`:

```
class Exc1 {
   static void subroutine() {
      int d = 0;
      int a = 10 / d;
   }
   public static void main(String args[]) {
      Exc1.subroutine();
   }
}
```

Own Exception Handling

Default exception handling is basically useful for debugging.

Normally, we want to handle exceptions ourselves because:

1) if we detected the error, we can try to fix it
2) we prevent the program from automatically terminating

Exception handling is done through the try and catch block.

Try and Catch 1

Try and catch:

1) `try` surrounds any code we want to monitor for exceptions
2) `catch` specifies which exception we want to handle and how.

When an exception is thrown in the try block:

```
try {
    d = 0;
    a = 42 / d;
    System.out.println("This will not be printed.");
}
```

Try and Catch 2

control moves immediately to the catch block:

```
catch (ArithmeticException e) {
   System.out.println("Division by zero.");
}
```

The exception is handled and the execution resumes.

The scope of catch is restricted to the immediately preceding try statement - it cannot catch exceptions thrown by another try statements.

Try and Catch 3

Resumption occurs with the next statement after the try/catch block:

```
try { … }
catch (ArithmeticException e) { … }
System.out.println("After catch statement.");
```

Not with the next statement after `a = 42/d;` which caused the exception!

```
a = 42 / d;
System.out.println("This will not be printed.");
```

Catch and Continue 1

The purpose of catch should be to resolve the exception and then continue as if the error had never happened.

Try/catch block inside a loop:

```
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
     int a=0, b=0, c=0;
     Random r = new Random();
```

Catch and Continue 2

After exception-handling, the program continues with the next iteration:

```
        for (int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```

Exception Display

All exception classes inherit from the `Throwable` class.

`Throwable` overrides `toString()` to describe the exception textually:

```
try { … }
catch (ArithmeticException e) {
   System.out.println("Exception: " + e);
}
```

The following text will be displayed:

```
Exception: java.lang.ArithmeticException: / by zero
```

Multiple Catch Clauses

When more than one exception can be raised by a single piece of code, several `catch` clauses can be used with one `try` block:

1) each `catch` catches a different kind of exception
2) when an exception is thrown, the first one whose type matches that of the exception is executed
3) after one `catch` executes, the other are bypassed and the execution continues after the try/catch block

Example: Multiple Catch 1

Two different exception types are possible in the following code: division by zero and array index out of bound:

```
class MultiCatch {
  public static void main(String args[]) {
     try {
        int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[] = { 1 };
        c[42] = 99;
     }
```

Example: Multiple Catch 2

Both exceptions can be caught by the following catch clauses:

```
        catch(ArithmeticException e) {
         System.out.println("Divide by 0: " + e);
      } catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Array index oob: " + e);
      }
      System.out.println("After try/catch blocks.");
   }
}
```

Order of Multiple Catch Clauses

Order is important:

1) catch clauses are inspected top-down
2) a clause using a super-class will catch all sub-class exceptions

Therefore, specific exceptions should appear before more general ones.

In particular, exception sub-classes must appear before super-classes.

Example: Multiple Catch Order 1

A try block with two catch clauses:

```
class SuperSubCatch {
  public static void main(String args[]) {
     try {
        int a = 0;
        int b = 42 / a;
```

This exception is more general but occurs first:

```
     } catch(Exception e) {
        System.out.println("Generic Exception catch.");
     }
```

Example: Multiple Catch Order 2

This exception is more specific but occurs last:

```
        catch(ArithmeticException e) {
            System.out.println("This is never reached.");
        }
    }
}
```

The second clause will never get executed. A compile-time error (unreachable code) will be raised.

Nested try Statements

The try statements can be nested:

1) if an inner try does not catch a particular exception
2) the exception is inspected by the outer try block
3) this continues until:
    a) one of the catch statements succeeds or
    b) all the nested try statements are exhausted
4) in the latter case, the Java run-time system will handle the exception

Example: Nested try 1

An example with two nested try statements:

```
class NestTry {
  public static void main(String args[]) {
```

Outer try statement:

```
    try {
       int a = args.length;
```

Division by zero when no command-line argument is present:

```
       int b = 42 / a;
       System.out.println("a = " + a);
```

Example: Nested try 2

Inner try statement:

```
try {
```

Division by zero when one command-line argument is present:

```
if (a==1) a = a/(a-a);
```

Array index out of bound when two command-line arguments are present:

```
if (a==2) {
   int c[] = { 1 };
   c[42] = 99;
}
```

Example: Nested try 3

Catch statement for the inner try statement, catches the array index out of bound exception:

```
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(e);
        }
```

Catch statement for the outer try statement, catches both division-by-zero exceptions for the inner and outer try statements:

```
    } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
    }
  }
}
```

Method Calls and Nested try 1

Nesting of try statements with method calls:

1) method call is enclosed within one try statement
2) the method includes another try statement

Still, the try blocks are considered nested within each other.

```
class MethNestTry {
```

Method Calls and Nested try 2

Method `nesttry` contains the try statement:

```
static void nesttry(int a) {
   try {
      if (a==1) a = a/(a-a);
      if (a==2) {
         int c[] = { 1 }; c[42] = 99;
      }
   } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println(e);
   }
}
```

Method Calls and Nested try 3

Method `main` contains another try block which includes the call to `nesttry`:

```
  public static void main(String args[]) {
     try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        nesttry(a);
     }
     } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
     }
  }
}
```

Throwing Exceptions

So far, we were only catching the exceptions thrown by the Java system.

In fact, a user program may throw an exception explicitly:

```
throw ThrowableInstance;
```

`ThrowableInstance` must be an object of type `Throwable` or its subclass.

throw Follow-up

Once an exception is thrown by:

```
throw ThrowableInstance;
```

1) the flow of control stops immediately
2) the nearest enclosing `try` statement is inspected if it has a `catch` statement that matches the type of exception:
   1) if one exists, control is transferred to that statement
   2) otherwise, the next enclosing `try` statement is examined
3) if no enclosing `try` statement has a corresponding `catch` clause, the default exception handler halts the program and prints the stack

Creating Exceptions

Two ways to obtain a `Throwable` instance:

1) creating one with the `new` operator

   All Java built-in exceptions have at least two constructors: one without parameters and another with one String parameter:

```
throw new NullPointerException("demo");
```

2) using a parameter of the `catch` clause

```
try { … } catch(Throwable e) { … e … }
```

Example: throw 1

```
class ThrowDemo {
```

The method `demoproc` throws a `NullPointerException` exception which is immediately caught in the try block and re-thrown:

```
  static void demoproc() {
     try {
        throw new NullPointerException("demo");
     } catch(NullPointerException e) {
        System.out.println("Caught inside demoproc.");
        throw e;
     }
  }
```

Example: throw 2

The main method calls `demoproc` within the try block which catches and handles the `NullPointerException` exception:

```
   public static void main(String args[]) {
      try {
         demoproc();
      } catch(NullPointerException e) {
         System.out.println("Recaught: " + e);
      }
   }
}
```

throws Declaration

If a method is capable of causing an exception that it does not handle, it must specify this behavior by the `throws` clause in its declaration:

```
type name(parameter-list) throws exception-list {
   …
}
```

where `exception-list` is a comma-separated list of all types of exceptions that a method might throw.

All exceptions must be listed except `Error` and `RuntimeException` or any of their subclasses, otherwise a compile-time error occurs.

Example: throws 1

The `throwOne` method throws an exception that it does not catch, nor declares it within the `throws` clause.

```
class ThrowsDemo {
   static void throwOne() {
      System.out.println("Inside throwOne.");
      throw new IllegalAccessException("demo");
   }
   public static void main(String args[]) {
      throwOne();
   }
}
```

Therefore this program does not compile.

Example: throws 2

Corrected program: `throwOne` lists exception, `main` catches it:

```
class ThrowsDemo {
   static void throwOne() throws IllegalAccessException {
      System.out.println("Inside throwOne.");
      throw new IllegalAccessException("demo");
   }
   public static void main(String args[]) {
      try {
         throwOne();
      } catch (IllegalAccessException e) {
         System.out.println("Caught " + e);
      }
   }
}
```

Motivating finally

When an exception is thrown:

1) the execution of a method is changed
2) the method may even return prematurely.

This may be a problem is many situations.

For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.

The `finally` block is used to address this problem.

finally Clause

The `try/catch` statement requires at least one `catch` or `finally` clause, although both are optional:

```
try { … }
catch(Exception1 ex1) { … } …
finally { … }
```

Executed after try/catch whether of not the exception is thrown.

Any time a method is to return to a caller from inside the try/catch block via:

1) uncaught exception or
2) explicit return

the finally clause is executed just before the method returns.

Example: finally 1

Three methods to exit in various ways.

```
class FinallyDemo {
```

`procA` prematurely breaks out of the `try` by throwing an exception, the `finally` clause is executed on the way out:

```
  static void procA() {
     try {
        System.out.println("inside procA");
        throw new RuntimeException("demo");
     } finally {
        System.out.println("procA's finally");
     }
  }
```

Example: finally 2

`procB`'s `try` statement is exited via a `return` statement, the `finally` clause is executed before `procB` returns:

```
static void procB() {
   try {
      System.out.println("inside procB");
      return;
   } finally {
      System.out.println("procB's finally");
   }
}
```

Example: finally 3

In `procC`, the `try` statement executes normally without error, however the `finally` clause is still executed:

```
static void procC() {
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
}
```

Example: finally 4

Demonstration of the three methods:

```
  public static void main(String args[]) {
     try {
        procA();
     } catch (Exception e) {
        System.out.println("Exception caught");
     }
     procB();
     procC();
  }
}
```

Java Built-In Exceptions

The default `java.lang` package provides several exception classes, all sub-classing the `RuntimeException` class.

Two sets of build-in exception classes:

1) unchecked exceptions – the compiler does not check if a method handles or throws there exceptions
2) checked exceptions – must be included in the method's `throws` clause if the method generates but does not handle them

Unchecked Built-In Exceptions 1

Methods that generate but do not handle those exceptions need not declare them in the `throws` clause:

`ArithmeticException`	arithmetic error such as divide-by-zero
`ArrayIndexOutOfBoundsException`	array index out of bounds
`ArrayStoreException`	assignment to an array element of the wrong type
`ClassCastException`	invalid cast
`IllegalArgumentException`	illegal argument used to invoke a method
`IllegalMonitorStateException`	illegal monitor behavior, e.g. waiting on an unlocked thread
`IllegalStateException`	environment of application is in incorrect state

Unchecked Built-In Exceptions 2

`IllegalThreadStateException`	requested operation not compatible with current thread state
`IndexOutOfBoundsException`	some type of index is out-of-bounds
`NegativeArraySizeException`	array created with a negative size
`NullPointerException`	invalid use of null reference
`NumberFormatException`	invalid conversion of a string to a numeric format
`SecurityException`	attempt to violate security
`StringIndexOutOfBounds`	attempt to index outside the the bounds of a string
`UnsupportedOperationException`	an unsupported operation was encountered

Checked Built-In Exceptions

Methods that generate but do not handle those exceptions must declare them in the `throws` clause:

`ClassNotFoundException`	class not found
`CloneNotSupportedException`	attempt to clone an object that does not implement the Cloneable interface
`IllegalAccessException`	access to a class is denied
`InstantiationException`	attempt to create an object of an abstract class or interface
`InterruptedException`	one thread has been interrupted by another thread
`NoSuchFieldException`	a requested field does not exist
`NoSuchMethodException`	a requested method does not exist

Creating Own Exception Classes

Build-in exception classes handle some generic errors.

For application-specific errors define your own exception classes.

How? Define a subclass of `Exception`:

```
class MyException extends Exception { … }
```

`MyException` need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

Throwable Class 1

`Exception` itself is a sub-class of `Throwable`.

All user exceptions have the methods defined by the `Throwable` class:

1) `Throwable fillInStackTrace()` – returns a `Throwable` object that contains a completed stack trace; the object can be rethrown.
2) `Throwable getCause()` – returns the exception that underlies the current exception. If no underlying exception exists, null is returned.
3) `String getLocalizedMessage()` – returns a localized description of the exception.
4) `String getMessage()` – returns a description of the exception
5) `StackTraceElement[] getStackTrace()` – returns an array that contains the stack trace; the method at the top is the last one called before exception.

Throwable Class 2

More methods defined by the `Throwable` class:

6) `Throwable initCause(Throwable causeExc)` – associates `causeExc` with the invoking exception as its cause, returns the exception reference
7) `void printStackTrace()` – displays the stack trace
8) `void printStackTrace(PringStream stream)` – sends the stack trace to the specified stream
9) `void setStackTrace(StackTraceElement elements[])` – sets the stack trace to the elements passed in elements; for specialized applications only
10) `String toString()` – returns a `String` object containing a description of the exception; called by `print()` when displaying a `Throwable` object.

Example: Own Exceptions 1

A new exception class is defined, with a private `detail` variable, a one-parameter constructor and an overridden `toString` method:

```
class MyException extends Exception {
   private int detail;

   MyException(int a) {
      detail = a;
   }

   public String toString() {
      return "MyException[" + detail + "]";
   }
}
```

Example: Own Exceptions 2

```
class ExceptionDemo {
```

The static `compute` method throws the `MyException` exception whenever its `a` argument is greater than `10`:

```
  static void compute(int a) throws MyException {
     System.out.println("Called compute(" + a + ")");
     if (a > 10) throw new MyException(a);
     System.out.println("Normal exit");
  }
```

Example: Own Exceptions 3

The `main` method calls `compute` with two arguments within a `try` block that catches the `MyException` exception:

```
  public static void main(String args[]) {
     try {
        compute(1);
        compute(20);
     } catch (MyException e) {
        System.out.println("Caught " + e);
     }
  }
}
```

Chained Exceptions 1

The chained exception allows to associate with a given exception another exception that describes its cause.

`Throwable` class includes two constructors to handle chained exceptions:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

They both create an exception with `causeExc` being its underlying reason and optional `msg` providing the textual description.

Chained Exceptions 2

`Throwable` class also includes two methods to handle chained exceptions:

1) `Throwable getCause()` – returns an exception that is the cause of the current exception, or null if there is no underlying exception.
2) `Throwable initCause(Throwable causeExc)` – associates `causeExc` with the invoking exception and returns a reference to the exception:

   a) `initCause` allows to associate a cause with an existing exception
   b) the cause exception can be set only once
   c) if the cause exception was set by a constructor, it is not possible to set it again with `initCause`

Example: Chained Exceptions 1

```
class ChainExcDemo {
```

The `demoproc` method creates a new `NullPointerException` exception `e`, associates `ArithmethicException` as its cause, then throws `e`:

```
  static void demoproc() {
     NullPointerException e =
        new NullPointerException("top layer");
     e.initCause(new ArithmeticException("cause"));
     throw e;
  }
```

Example: Chained Exceptions 2

The `main` method calls `demoproc` within the `try` block that catches `NullPointerException`, then displays the exception and its cause:

```
   public static void main(String args[]) {
      try {
         demoproc();
      } catch(NullPointerException e) {
         System.out.println("Caught: " + e);
         System.out.println("Cause: " + e.getCause());
      }
   }
}
```

A cause exception may itself have a cause. In fact, the cause-chain of exceptions may be arbitrarily long.

Exceptions Usage

New Java programmers should break the habit of returning error codes to signal abnormal exit from a method.

Java provides a clean and powerful way to handle errors and unusual boundary conditions through its:

1) `try`
2) `catch`
3) `finally`
4) `throw` and
5) `throws` statements.

However, Java's exception-handling should not be considered a general mechanism for non-local branching!

Exercise: Exception-Handling

1) Create exception classes called `Even` and `Odd`
2) Generate numbers within an endless loop. Print the generated numbers.
3) If the number is even, throw the `Even` exception with the message "`The number thrown an even number`" along with the number.
4) If the number is odd, throw the `Odd` exception with the message "`The number thrown an odd number`" along with the number.
5) Catch the `Even` exception within the endless loop and print the message.
6) Catch the `Odd` exception outside of the loop and print the message.

Multi-Threading

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Multi-Tasking

Two kinds of multi-tasking:

1) process-based multi-tasking
2) thread-based multi-tasking

Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.

Processes are heavyweight tasks:

1) that require their own address space
2) inter-process communication is expensive and limited
3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

Thread-based multi-tasking is about a single program executing concurrently several tasks e.g. a text editor printing and spell-checking text.

Threads are lightweight tasks:

1) they share the same address space
2) they cooperatively share the same process
3) inter-thread communication is inexpensive
4) context-switching from one thread to another is low-cost

Java multi-tasking is thread-based.

Reasons for Multi-Threading

Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.

There is plenty of idle time for interactive, networked applications:

1) the transmission rate of data over a network is much slower than the rate at which the computer can process it
2) local file system resources can be read and written at a much slower rate than can be processed by the CPU
3) of course, user input is much slower than the computer

Single-Threading

In a single-threaded environment, the program has to wait for each of these tasks to finish before it can proceed to the next.

Single-threaded systems use event loop with pooling:

1) a single thread of control runs in an infinite loop
2) the loop pools a single event queue to decide what to do next
3) the pooling mechanism returns an event
4) control is dispatched to the appropriate event handler
5) until this event handler returns, nothing else can happen

Threads: Model

Thread exist in several states:

1) ready to run
2) running
3) a running thread can be suspended
4) a suspended thread can be resumed
5) a thread can be blocked when waiting for a resource
6) a thread can be terminated

Once terminated, a thread cannot be resumed.

Threads: Priorities

Every thread is assigned priority – an integer number to decide when to switch from one running thread to the next (context-switching).

Rules for context switching:

1) a thread can voluntarily relinquish control (sleeping, blocking on I/O, etc.), then the highest-priority ready to run thread is given the CPU.
2) a thread can be preempted by a higher-priority thread – a lower-priority thread is suspended

When two equal-priority threads are competing for CPU time, which one is chosen depends on the operating system.

Threads: Synchronization

Multi-threading introduces asynchronous behavior to a program. How to ensure synchronous behavior when we need it?

For instance, how to prevent two threads from simultaneously writing and reading the same object?

Java implementation of monitors:

1) classes can define so-called synchronized methods
2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Class

To create a new thread a program will:

1) extend the `Thread` class, or
2) implement the `Runnable` interface

`Thread` class encapsulates a thread of execution.

The whole Java multithreading environment is based on the `Thread` class.

Thread Methods

`getName`	obtain a thread's name
`getPriority`	obtain a thread's priority
`isAlive`	determine if a thread is still running
`join`	wait for a thread to terminate
`run`	entry-point for a thread
`sleep`	suspend a thread for a period of time
`start`	start a thread by calling its run method

The Main Thread

The main thread is a thread that begins as soon as a program starts.

The main thread:

1) is invoked automatically
2) is the first to start and the last to finish
3) is the thread from which other “child” threads will be spawned

It can be obtained through the

```
public static Thread currentThread()
```

method of `Thread`.

Example: Main Thread 1

```
class CurrentThreadDemo {
  public static void main(String args[]) {
```

The main thread is obtained, displayed, its name changed and re-displayed:

```
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    t.setName("My Thread");
    System.out.println("After name change: " + t);
```

Example: Main Thread 2

A loop performs five iterations pausing for a second between the iterations.

It is performed within the `try/catch` block – the `sleep` method may throw `InterruptedException` if some other thread wanted to interrupt:

```
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

Example: Thread Methods

`Thread` methods used by the example:

1) `static void sleep(long milliseconds)`
   `throws InterruptedException`

   Causes the thread from which it is executed to suspend execution for the specified number of milliseconds.

2) `final String getName()`

   Allows to obtain the name of the current thread.

3) `final void setName(String threadName)`

   Sets the name of the current thread.

Creating a Thread

Two methods to create a new thread:

1) by implementing the `Runnable` interface
2) by extending the `Thread` class

We look at each method in order.

New Thread: Runnable

To create a new thread by implementing the `Runnable` interface:

1) create a class that implements the `run` method (inside this method, we define the code that constitutes the new thread):

```
public void run()
```

2) instantiate a `Thread` object within that class, a possible constructor is:

```
Thread(Runnable threadOb, String threadName)
```

3) call the `start` method on this object (`start` calls `run`):

```
void start()
```

Example: New Thread 1

A class `NewThread` that implements `Runnable`:

```
class NewThread implements Runnable {
   Thread t;
```

Creating and starting a new thread. Passing `this` to the `Thread` constructor – the new thread will call this object's `run` method:

```
   NewThread() {
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
      t.start();
   }
```

Example: New Thread 2

This is the entry point for the newly created thread – a five-iterations loop with a half-second pause between the iterations all within try/catch:

```
   public void run() {
      try {
         for (int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
         }
      } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
   }
}
```

Example: New Thread 3

```
class ThreadDemo {
  public static void main(String args[]) {
```

A new thread is created as an object of `NewThread`:

```
    new NewThread();
```

After calling the `NewThread` `start` method, control returns here.

Example: New Thread 4

Both threads (new and main) continue concurrently.

Here is the loop for the main thread:

```
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

New Thread: Extend Thread

The second way to create a new thread:

1) create a new class that extends `Thread`
2) create an instance of that class

Thread provides both `run` and `start` methods:

1) the extending class must override `run`
2) it must also call the `start` method

Example: New Thread 1

The new thread class extends `Thread`:

```
class NewThread extends Thread {
```

Create a new thread by calling the `Thread`'s constructor and `start` method:

```
  NewThread() {
     super("Demo Thread");
     System.out.println("Child thread: " + this);
     start();
  }
```

Example: New Thread 2

`NewThread` overrides the `Thread`'s `run` method:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println("Child Thread: " + i);
           Thread.sleep(500);
        }
     } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
     }
     System.out.println("Exiting child thread.");
  }
}
```

Example: New Thread 3

```
class ExtendThread {
  public static void main(String args[]) {
```

After a new thread is created:

```
    new NewThread();
```

the new and main threads continue concurrently…

Example: New Thread 4

This is the loop of the main thread:

```
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

New Thread: Which Approach?

The `Thread` class defines several methods that can be overriden.

Of these methods, only `run` must be overriden.

Creating a new thread:

1) implement `Runnable` if only `run` is overriden
2) extend `Thread` if other methods are also overriden

Example: Multiple Threads 1

So far, we were using only two threads - main and new, but in fact a program may spawn as many threads as it needs.

`NewThread` class implements the `Runnable` interface:

```
class NewThread implements Runnable {
   String name;
   Thread t;

   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start();
   }
```

Example: Multiple Threads 2

Here is the implementation of the `run` method:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println(name + ": " + i);
           Thread.sleep(1000);
        }
     } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
     }
     System.out.println(name + " exiting.");
  }
}
```

Example: Multiple Threads 3

The demonstration class creates three threads then waits until they all finish:

```
class MultiThreadDemo {
  public static void main(String args[]) {
     new NewThread("One");
     new NewThread("Two");
     new NewThread("Three");
     try {
        Thread.sleep(10000);
     } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
     }
     System.out.println("Main thread exiting.");
  }
}
```

Using isAlive and join Methods

How can one thread know when another thread has ended?

Two methods are useful:

1) `final boolean isAlive()` - returns `true` if the thread upon which it is called is still running and `false` otherwise
2) `final void join() throws InterruptedException` – waits until the thread on which it is called terminates

Example: isAlive and join 1

Previous example improved to use `isAlive` and `join` methods.

New thread implements the `Runnable` interface:

```
class NewThread implements Runnable {
   String name;
   Thread t;

   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start();
   }
```

Example: isAlive and join 2

Here is the new thread's run method:

```
  public void run() {
     try {
        for (int i = 5; i > 0; i--) {
           System.out.println(name + ": " + i);
           Thread.sleep(1000);
        }
     } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
     }
     System.out.println(name + " exiting.");
  }
}
```

Example: isAlive and join 3

```
class DemoJoin {
  public static void main(String args[]) {
```

Creating three new threads:

```
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");
```

Checking if those threads are still alive:

```
    System.out.println(ob1.t.isAlive());
    System.out.println(ob2.t.isAlive());
    System.out.println(ob3.t.isAlive());
```

Example: isAlive and join 4

Waiting until all three threads have finished:

```
try {
   System.out.println("Waiting to finish.");
   ob1.t.join();
   ob2.t.join();
   ob3.t.join();
} catch (InterruptedException e) {
   System.out.println("Main thread Interrupted");
}
```

Example: isAlive and join 5

Testing again if the new threads are still alive:

```
        System.out.println(ob1.t.isAlive());
        System.out.println(ob2.t.isAlive());
        System.out.println(ob3.t.isAlive());

        System.out.println("Main thread exiting.");
    }
}
```

Thread Priorities

Priority is used by the scheduler to decide when each thread should run.

In theory, higher-priority thread gets more CPU than lower-priority thread and threads of equal priority should get equal access to the CPU.

In practice, the amount of CPU time that a thread gets depends on several factors besides its priority.

Setting and Checking Priorities

Setting thread's priority:

```
final void setPriority(int level)
```

where level specifies the new priority setting between:

1) `MIN_PRIORITY` (1)
2) `MAX_PRIORITY` (10)
3) `NORM_PRIORITY` (5)

Obtain the current priority setting:

```
final int getPriority()
```

Example: Priorities 1

A new thread class with `click` and `running` variables:

```
class Clicker implements Runnable {
   int click = 0;
   Thread t;
   private volatile boolean running = true;
```

A new thread is created, its priority initialised:

```
   public Clicker(int p) {
      t = new Thread(this);
      t.setPriority(p);
   }
```

Example: Priorities 2

When running, `click` is incremented. When stopped, `running` is `false`:

```
  public void run() {
     while (running) {
        click++;
     }
  }
  public void stop() {
     running = false;
  }

  public void start() {
     t.start();
  }
}
```

Example: Priorities 3

```
class HiLoPri {
  public static void main(String args[]) {
```

The main thread is set at the highest priority, the new threads at two above and two below the normal priority:

```
    Thread.currentThread().
       setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
```

Example: Priorities 4

The threads are started and allowed to run for 10 seconds:

```
lo.start();
hi.start();
try {
   Thread.sleep(10000);
} catch (InterruptedException e) {
   System.out.println("Main thread interrupted.");
}
```

Example: Priorities 5

After 10 seconds, both threads are stopped and click variables printed:

```
        lo.stop();
        hi.stop();
        try {
           hi.t.join();
           lo.t.join();
        } catch (InterruptedException e) {
           System.out.println("InterruptedException");
        }

        System.out.println("Low-priority: " + lo.click);
        System.out.println("High-priority: " + hi.click);
    }
}
```

Volatile Variable

The `volatile` keyword is used to declare the `running` variable:

```
private volatile boolean running = true;
```

This is to ensure that the value of `running` is examined at each iteration of:

```
while (running) {
    click++;
}
```

Otherwise, Java is free to optimize the loop in such a way that a local copy of `running` is created. The use of `volatile` prevents this optimization.

Synchronization

When several threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

This way is called synchronization.

Synchronization uses the concept of monitors:

1) only one thread can enter a monitor at any one time
2) other threads have to wait until the threat exits the monitor

Java implements synchronization in two ways: through the synchronized methods and through the synchronized statement.

Synchronized Method

All objects have their own implicit monitor associated with them.

To enter an object's monitor, call this object's `synchronized` method.

While a thread is inside a monitor, all threads that try to call this or any other `synchronized` method on this object have to wait.

To exit the monitor, it is enough to return from the `synchronized` method.

Consider first an example without synchronization…

Example: No Synchronization 1

The `call` method tries to print the message string inside brackets, pausing the current thread for one second in the middle:

```
class Callme {
   void call(String msg) {
      System.out.print("[" + msg);
      try {
         Thread.sleep(1000);
      } catch(InterruptedException e) {
         System.out.println("Interrupted");
      }
      System.out.println("]");
   }
}
```

Example: No Synchronization 2

`Caller` constructor obtains references to the `Callme` object and `String`, stores them in the `target` and `msg` variables, then creates a new thread:

```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
```

Example: No Synchronization 3

The Caller's `run` method calls the `call` method on the `target` instance of `Calllme`, passing in the `msg` string:

```
   public void run() {
      target.call(msg);
   }
}
```

Example: No Synchronization 4

`Synch` class creates a single instance of `Callme` and three of `Caller`, each with a message. The `Callme` instance is passed to each `Caller`:

```
class Synch {
  public static void main(String args[]) {
     Callme target = new Callme();
     Caller ob1 = new Caller(target, "Hello");
     Caller ob2 = new Caller(target, "Synchronized");
     Caller ob3 = new Caller(target, "World");
```

Example: No Synchronization 5

Waiting for all three threads to finish:

```
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}
```

No Synchronization

Output from the earlier program:

```
[Hello[Synchronized[World]
]
]
```

By pausing for one second, the call method allows execution to switch to another thread. A mix-up of the outputs from of the three message strings.

In this program, nothing exists to stop all three threads from calling the same method on the same object at the same time.

Synchronized Method

To fix the earlier program, we must serialize the access to `call`:

```
class Callme {
   synchronized void call(String msg) {
      …
   }
}
```

This prevents other threads from entering call while another thread is using it. The output result of the program is now as follows:

```
[Hello]
[Synchronized]
[World]
```

Synchronized Statement

How to synchronize access to instances of a class that was not designed for multithreading and we have no access to its source code?

Put calls to the methods of this class inside the `synchronized` block:

```
synchronized(object) {
   …
}
```

This ensures that a call to a method that is a member of the `object` occurs only after the current thread has successfully entered the `object`'s monitor.

Example: Synchronized 1

Now the `call` method is not modified by `synchronized`:

```
class Callme {
   void call(String msg) {
      System.out.print("[" + msg);
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         System.out.println("Interrupted");
      }
      System.out.println("]");
   }
}
```

Example: Synchronized 2

```
class Caller implements Runnable {
   String msg;
   Callme target;
   Thread t;

   public Caller(Callme targ, String s) {
      target = targ;
      msg = s;
      t = new Thread(this);
      t.start();
   }
```

Example: Synchronized 3

The `Caller`'s `run` method uses the `synchronized` statement to include the call the `target`'s `call` method:

```
   public void run() {
      synchronized(target) {
         target.call(msg);
      }
   }
}
```

Example: Synchronized 4

```
class Synch1 {
  public static void main(String args[]) {
     Callme target = new Callme();
     Caller ob1 = new Caller(target, "Hello");
     Caller ob2 = new Caller(target, "Synchronized");
     Caller ob3 = new Caller(target, "World");
     try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
     } catch(InterruptedException e) {
        System.out.println("Interrupted");
     }
  }
}
```

Inter-Thread Communication

Inter-thread communication relies on three methods in the `Object` class:

1) `final void wait() throws InterruptedException`

   tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.

2) `final void notify()`

   wakes up the first thread that called `wait()` on the same object

3) `final void notifyAll()`

   wakes up all the threads that called `wait()` on the same object; the highest-priority thread will run first.

All three must be called from within a `synchronized` context.

Queuing Problem

Consider the classic queuing problem where one thread (producer) is producing some data and another (consumer) is consuming this data:

1) producer should not overrun the consumer with data
2) consumer should not consume the same data many times

We consider two solutions:

1) incorrect with `synchronized` only
2) correct with synchronized and `wait`/`notify`

Example: Incorrect Queue 1

The one-place queue class `Q`, with the variable `n` and methods `get` and `put`. Both methods are `synchronized`:

```
class Q {
   int n;

   synchronized int get() {
      System.out.println("Got: " + n);
      return n;
   }
   synchronized void put(int n) {
      this.n = n;
      System.out.println("Put: " + n);
   }
}
```

Example: Incorrect Queue 2

`Producer` creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
   Q q;
   Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
   }
   public void run() {
      int i = 0;
      while(true) {
         q.put(i++);
      }
   }
}
```

Example: Incorrect Queue 3

`Consumer` creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
   Q q;
   Consumer(Q q) {
      this.q = q;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      while(true) {
         q.get();
      }
   }
}
```

Example: Incorrect Queue 4

The `PC` class first creates a single `Queue` instance `q`, then creates a `Producer` and `Consumer` that share this `q`:

```
class PC {
  public static void main(String args[]) {
     Q q = new Q();
     new Producer(q);
     new Consumer(q);
     System.out.println("Press Control-C to stop.");
  }
}
```

Why Incorrect?

Here is the output:

```
Put: 1
Got: 1
Got: 1
Put: 2
Put: 3
Get: 3
…
```

Nothing stops the producer from overrunning the consumer, nor the consumer from consuming the same data twice.

Example: Corrected Queue 1

The correct producer-consumer system uses `wait` and `notify` to synchronize the behavior of the producer and consumer.

The queue class introduces the additional `boolean` variable `valueSet` used by the `get` and `put` methods:

```
class Q {
   int n;
   boolean valueSet = false;
```

Example: Corrected Queue 2

Inside `get`, `wait` is called to suspend the execution of `Consumer` until `Producer` notifies that some data is ready:

```
synchronized int get() {
   if (!valueSet)
      try {
         wait();
      }
      catch(InterruptedException e) {
         System.out.println("InterruptedException");
      }
```

Example: Corrected Queue 3

After the data has been obtained, `get` calls `notify` to tell `Producer` that it can put more data on the queue:

```
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
```

Example: Corrected Queue 4

Inside `put`, `wait` is called to suspend the execution of `Producer` until `Consumer` has removed the item from the queue:

```
synchronized void put(int n) {
   if (valueSet)
      try {
         wait();
      } catch(InterruptedException e) {
         System.out.println("InterruptedException");
      }
```

Example: Corrected Queue 5

After the next item of data is put in the queue, `put` calls `notify` to tell `Consumer` that it can remove this item:

```
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}
```

Example: Corrected Queue 6

`Producer` creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
   Q q;
   Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
   }
   public void run() {
      int i = 0;
      while(true) {
         q.put(i++);
      }
   }
}
```

Example: Corrected Queue 7

`Consumer` creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
   Q q;
   Consumer(Q q) {
      this.q = q;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      while(true) {
         q.get();
      }
   }
}
```

Example: Corrected Queue 8

The `PCFixed` class first creates a single `Queue` instance `q`, then creates a `Producer` and `Consumer` that share this `q`:

```
class PCFixed {
  public static void main(String args[]) {
     Q q = new Q();
     new Producer(q);
     new Consumer(q);
     System.out.println("Press Control-C to stop.");
  }
}
```

Deadlock

Multi-threading and synchronization create the danger of deadlock.

Deadlock: a circular dependency on a pair of synchronized objects.

Suppose that:

1) one thread enters the monitor on object `X`
2) another thread enters the monitor on object `Y`
3) the first thread tries to call a synchronized method on object `Y`
4) the second thread tries to call a synchronized method on object `X`

The result: the threads wait forever – deadlock.

Example: Deadlock 1

Class `A` contains the `foo` method which takes an instance `b` of class `B` as a parameter. It pauses briefly before trying to call the `b`'s `last` method:

```
class A {
  synchronized void foo(B b) {
     String name = Thread.currentThread().getName();
     System.out.println(name + " entered A.foo");
     try {
        Thread.sleep(1000);
     } catch(Exception e) {
        System.out.println("A Interrupted");
     }
     System.out.println(name + " trying B.last()");
     b.last();
  }
```

Example: Deadlock 2

Class `A` also contains the `synchronized` method `last`:

```
   synchronized void last() {
      System.out.println("Inside A.last");
   }
}
```

Example: Deadlock 3

Class `B` contains the `bar` method which takes an instance `a` of class `A` as a parameter. It pauses briefly before trying to call the `a`'s `last` method:

```
class B {
   synchronized void bar(A a) {
      String name = Thread.currentThread().getName();
      System.out.println(name + " entered B.bar");
      try {
         Thread.sleep(1000);
      } catch(Exception e) {
         System.out.println("B Interrupted");
      }
      System.out.println(name + " trying A.last()");
      a.last();
   }
```

Example: Deadlock 4

Class `B` also contains the `synchronized` method `last`:

```
   synchronized void last() {
      System.out.println("Inside A.last");
   }
}
```

Example: Deadlock 5

The main `Deadlock` class creates the instances `a` of `A` and `b` of `B`:

```
class Deadlock implements Runnable {

   A a = new A();
   B b = new B();
```

Example: Deadlock 6

The constructor creates and starts a new thread, and creates a lock on the `a` object in the `main` thread (running `foo` on `a`) with `b` passed as a parameter:

```
Deadlock() {
    Thread.currentThread().setName("MainThread");
    Thread t = new Thread(this, "RacingThread");
    t.start();
    a.foo(b);
    System.out.println("Back in main thread");
}
```

Example: Deadlock 7

The run method creates a lock on the `b` object in the new thread (running `bar` on `b`) with `a` passed as a parameter:

```
  public void run() {
     b.bar(a);
     System.out.println("Back in other thread");
  }
```

Create a new `Deadlock` instance:

```
  public static void main(String args[]) {
     new Deadlock();
  }
}
```

Deadlock Reached

Program output:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

`RacingThread` owns the monitor on `b` while waiting for the monitor on `a`.
`MainThread` owns the monitor on `a` while it is waiting for the monitor on `b`.

The program deadlocks!

Suspending/Resuming Threads

Thread management should use the `run` method to check periodically whether the thread should suspend, resume or stop its own execution.

This is usually accomplished through a flag variable that indicates the execution state of a thread, e.g.

1) `running` – the thread should continue executing
2) `suspend` – the thread must pause
3) `stop` – the thread must terminate

Example: Suspending/Resuming 1

`NewThread` class contains the `boolean` variable `suspendFlag` to control the execution of a thread, initialized to `false`:

```
class NewThread implements Runnable {
   String name;
   Thread t;
   boolean suspendFlag;

   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      suspendFlag = false;
      t.start();
   }
```

Example: Suspending/Resuming 2

The `run` method contains the `synchronized` statement that checks `suspendFlag`. If `true`, the `wait` method is called.

```
public void run() {
    try {
        for (int i = 15; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(200);
            synchronized(this) {
                while(suspendFlag)    wait();
            }
        }
    }
```

Example: Suspending/Resuming 3

```
      catch (InterruptedException e) {
       System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
}
```

Example: Suspending/Resuming 4

The `mysuspend` method sets `suspendFlag` to true:

```
    void mysuspend() {
        suspendFlag = true;
    }
```

The `myresume` method sets `suspendFlag` to `false` and invokes `notify` to wake up the thread:

```
    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

Example: Suspending/Resuming 5

`SuspendResume` class creates two instances `ob1` and `ob2` of `NewThread`, therefore two new threads, through its `main` method:

```
class SuspendResume {

  public static void main(String args[]) {
     NewThread ob1 = new NewThread("One");
     NewThread ob2 = new NewThread("Two");
```

The two threads are kept running, then suspended, then resumed from the main thread:

Example: Suspending/Resuming 6

```
try {
    Thread.sleep(1000);
    ob1.mysuspend();
    System.out.println("Suspending thread One");
    Thread.sleep(1000);
    ob1.myresume();
    System.out.println("Resuming thread One");
    ob2.mysuspend();
    System.out.println("Suspending thread Two");
    Thread.sleep(1000);
    ob2.myresume();
    System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
```

Example: Suspending/Resuming 7

The main thread waits for the two child threads to finish, then finishes itself:

```
        try {
            System.out.println("Waiting to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

The Last Word on Multi-Threading

Multi-threading is a powerful tool to writing efficient programs.

When you have two subsystems within a program that can execute concurrently, make them individual threads.

However, creating too many threads can actually degrade the performance of your program because of the cost of context switching.

Exercise: Multi-Threading 1

1) Create a new main class called `MultiThread`
2) Create a new class called `MemoryThread`. This class should implement the interface `Runnable` so that it can be run as a thread. This Thread will monitor the memory usage on the system.
   a) Add `void start()`, `stop()` and `run()` methods.
   b) The `run()` method should print to the screen every 5 seconds the amount of memory presently being used. This can be done using these two lines of code:

```
Runtime r = Runtime.getRuntime();
long memoryUsed = r.totalMemory()- r.freeMemory();
```

3) Create a new class called `ClockThread`. This class should implement the interface `Runnable` so that it can be run as a thread. This Thread will monitor what the time is.
   a) The constructor for `ClockThread` should take an argument that sets the time (in seconds) that this thread will run for.
   b) Add `void start()`, `stop()` and `run()` methods.

Exercise: Multi-Threading 2

c) The `run()` method should print the current time to the screen every 5 seconds. This can be done using the built in `Date` class.

```
import java.util.Date;
Date timeNow = new Date();
System.out.println(timeNow.toString());
```

The `run()` method should stop when the elapsed time set in the constructor has been reached.

4) In the `main()` method of `MultiThread` create the two thread objects and start them; using the *`start()`* method.

5) Add a `while` loop, that will print the number of active Threads every one second. A one second pause can be implemented as follows:

```
while (true) {
      Thread.sleep(1000);
  }
```

6) The number of threads can be monitored using the `ThreadGroup` class as follows:

```
int numThreads =
Thread.currentThread().getThreadGroup().activeCount();
```

Exercise: Multi-Threading 3

7) This while loop should terminate after one minute and the two threads stopped by invoking their `stop()` methods.
8) When printing to the screen - the Threads names should be appended so that it is clear from which process the data is originating.
9) The `MemoryThread` thread should be assigned a maximum priority and the `ClockThread` thread a minimum priority.

Horizontal Libraries

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Horizontal Libraries

Horizontal libraries are APIs that are used across the language.

Java provides a rich set of horizontal libraries:

a) String handling – for handling sequence of characters
b) event handling – helps to handle how programs respond to actions generated by the user.
c) Object collection – handling a group of objects

String Handling

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

String Object

String is a sequence of characters.

Unlike many other programming languages that implements string as character arrays, Java implements strings as object of type `String`.

This provides a full compliment of features that make string handling convenient. For example, Java String has methods to:

1) compare two strings
2) Search for a substring
3) Concatenate two strings and
4) Change the case of letters within a string
5) Can be constructed a number of ways making it easy to obtain a string when needed

String is Immutable

Once a String Object has been created, you cannot change the characters that comprise that string.

This is not a restriction. It means each time you need an altered version of an existing string, a new string object is created that contains the modification.

It is more efficient to implement immutable strings than changeable ones.

To solve this, Java provides a companion class to `String` called `StringBuffer.`

`StringBuffer` objects can be modified after they are created.

String Constructors 1

String supports several constrictors:

1) to create an empty String

```
String s = new String();
```

2) to create a string that have initial values

```
String(char chars[])
```

Example:

```
char chars[] = {'a','b','c'};
String s = new String(chars);
```

String Constructors 2

3) to create a string as a subrange of a character array

```
String(char chars[], int startindex, int numchars)
```

Here, `startindex` specifies the index at which the subrange begins, and `numChars` specifies the number of characters to use.

Example:

```
char chars[] = {'a','b','c','d','e','f'};
String s = new String(chars,2,3);
```

This initializes `s` with the characters `cde`.

String Constructors 3

4) to construct a String object that contains the same character sequence as another String object

```
String(String obj)
```

Example

```
class MakeString {
  public static void main(String args[]) {
     char c[] = {'J', 'a', 'v', 'a'};
     String s1 = new String(c);
     String s2 = new String(s1);
     System.out.println(s1);
     System.out.println(s2);
  }
}
```

String Length

The length of a string is the number of characters that it contains.

To obtain this value call the `length()` method:

```
int length()
```

The following fragment prints "`3`", since there are three characters in the string `s`.

```
char chars[] = {'a','b','c'};
String s = new String(chars);
System.out.println(s.lenght());
```

String Operations

Strings are a common and important part of programming.

Java provides several string operations within the syntax of the language.

These operations include:

1) automatic creation of new `String` instances from literals
2) concatenation of multiple `String` objects using the `+` operator
3) conversion of other data types to a string representation

There are explicit methods to perform all these functions, but Java does them automatically for the convenience of the programmer and to add clarity.

String Literals

Using String literals is an easier way of creating Strings Objects.

For each String literal, Java automatically constructs a `String` object.

You can use String literal to initialize a `String` object.

Example:

```
char chars[] = {'a','b','c'};
String s1 = new String(chars);
```

Using String literals

```
String s2 = "abc";
```

String Concatenation

Java does not allow operations to be applied to a `String` object.

The one exception to this rule is the `+` operator, which concatenates two strings producing a string object as a result.

With this you ca chain together a series of `+` operations.

Example:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

Concatenation Usage

One practical use is found when you are creating very long strings.

Instead of letting long strings wrap around your source code, you can break them into smaller pieces, using the + to concatenate them.

Example:

```
class ConCat {
      public static void main(String args[]) {
            String longStr = "This could have been " +
            "a very long line that would have " +
            "wrapped around.  But string concatenation "
            + "prevents this.";
            System.out.println(longStr);
  }
}
```

Concatenation & Other Data Type

You can concatenate Strings with other data types.

Example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

The compiler will convert an operand to its string equivalent whenever the other operand of the + ia an instance of `String`.

Be careful:

```
String s = "Four:" + 2 + 2;
System.out.println(s);
```

Prints `Four:22` rather than `Four: 4`.

To achieve the desired result, bracket has to be used.

```
String s = "Four:" + (2 + 2);
```

Conversion and toString() Method

When Java converts data into its string representation during concatenation, it does so by calling one of its overloaded `valueOf()` method defined by `String`.

`valueOf()` is overloaded for

1) simple types – which returns a string that contains the human – readable equivalent of the value with which it is called.
2) object types – which calls the toString() method of the object.

Example: toString() Method 1

Override toString() for Box class

```
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

Example: toString() Method 2

```
    public String toString() {
        return "Dimensions are " + width + " by " +
                    depth + " by " + height + ".";
  }
}
```

Example: toString() Method 3

```
class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

`Box's toString()` method is automatically invoked when a `Box` object is used in a concatenation expression or in a Call to `println()`.

Character Extraction

String class provides a number of ways in which characters can be extracted from a String object.

String index begin at zero.

These extraction methods are:

1) charAt()
2) getChars()
3) getBytes()
4) toCharArray()

Each will considered.

charAt()

To extract a single character from a String.

General form:

```
char charAt(int where)
```

`where` is the index of the character you want to obtain. The value of `where` must be nonnegative and specify alocation within the string.

Example:

```
char ch;
ch = "abc".charAt(1);
```

Assigns a value of "`b`" to `ch`.

getChars()

Used to extract more than one character at a time.

General form:

```
void getChars(int sourceStart, int sourceEnd, char[]
                                target, int targetStart)
```

`sourceStart` – specifies the index of the beginning of the substring

`sourceEnd` – specifies an index that is one past the end of the desired subString

`target` – is the array that will receive the characters

`targetStart` – is the index within target at which the subString will be copied is passed in this parameter

getChars()

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "This is a demo of the getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];
    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

getBytes()

Alternative to getChars() that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform.

General form:

```
byte[] getBytes()
```

Usage:

Most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

For example, most internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray()

To convert all the characters in a String object into character array.

It returns an array of characters for the entire string.

General form:

```
char[] toCharArray()
```

It is provided as a convenience, since it is possible to use `getChars()` to achieve the same result.

String Comparison

The String class includes several methods that compare strings or substrings within strings.

They are:

1) `equals() and equalsIgnoreCase()`
2) `regionMatches()`
3) `startWith() and endsWith()`
4) `equals() Versus ==`
5) `comapreTo()`

Each will be considered.

equals()

To compare two Strings for equality, use equals()

General form:

```
boolean equals(Object str)
```

`str` is the `String` object being compared with the invoking String object.

It returns true if the string contain the same character in the same order, and false otherwise.

The comparison is case-sensitive.

equalsIgnoreCase()

To perform operations that ignores case differences.

When it compares two strings, it considers `A-Z` as the same as `a-z`.

General form:

```
boolean equalsIgnoreCase(Object str)
```

`str` is the `String` object being compared with the invoking String object.

It returns `true` if the string contain the same character in the same order, and `false` otherwise.

The comparison is case-sensitive.

equals and equalsIgnoreCase() 1

Example:

```
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                                         s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                                         s1.equals(s3));
```

equals and equalsIgnoreCase() 2

```
    System.out.println(s1 + " equals " + s4 + " -> " +
                                        s1.equals(s4));

    System.out.println(s1 + " equalsIgnoreCase " + s4 +
                      " -> " + s1.equalsIgnoreCase(s4));

  }

}
```

regionMatches() 1

Compares a specific region inside a string with another specific region in another string.

There is an overloaded form that allows you to ignore case in such comparison.

General form:

```
boolean regionMatches(int startindex, String str2,
                      int str2StartIndex, int numChars)


boolean regionMatches(boolean ignoreCase, int
              startindex, String str2, int str2StartIndex,
                                              int numChars)
```

regionMatches() 2

In both versions, `startIndex` specifies the index at which the region begins within the invoking String object.

The string object being compared is specified as `str`.

The index at which the comparison will start within `str2` is specified by `str2StartIndex`.

The length of the substring being comapred is passed in `numChars`.

In the second version, if the `ignoreCase` is `true`, the case of the characters is ignored. Otherwise case is significant.

startsWith() and endsWith() 1

`String` defines two routines that are more or less the specialised forms of regionMatches().

The `startsWith()` method determines whether a given string begins with a specified string.

Conversely, `endsWith()` method determines whether the string in question ends with a specified string.

General form:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

`str` is the `String` being tested. If the string matches, true is returned, otherwise false is returned.

startsWith() and endsWith() 2

Example:

```
"Foobar".endsWith("bar");
```

and

```
"Foobar".startsWith("Foo");
```

are both `true`.

startsWith() and endsWith() 3

A second form of `startsWith()`, let you specify a starting point:

General form:

```
boolean startWith(String str, int startIndex)
```

Where `startIndex` specifies the index into the invoking string at which point the search will begin.

Example:

```
"Foobar".startsWith("bar", 3);
```

returns `true.`

equals() Versus ==

It is important to understand that the two metyhod performs different functions.

1) `equals()` method compares the characters inside a `String` object.
2) `==` operator compares two object references to see whether they refer to the same instance.

Example: equals() Versus ==

```
class EqualsNotEqualTo {
   public static void main(String args[]) {
       String s1 = "Hello";
       String s2 = new String(s1);
       System.out.print(s1 + " equals " + s2 + " -> ");
       System.out.println(s1.equals(s2));
       System.out.print(s1 + " == " + s2 + " -> ")
       System.out.println((s1 == s2));
   }
}
```

compareTo() 1

It is not enough to know that two Strings are identical. You need to know which is less than, equal to, or greater than the next.

A string is less than the another if it comes before the other in the dictionary order.

A string is greater than the another if it comes after the other in the dictionary order.

The `String` method `compareTo()` serves this purpose.

compareTo() 2

General form:

```
int compareTo(String str)
```

`str` is the string that is being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

Less than zero	The invoking string is less than `str`
Greater than zero	The invoking string is greater than `str`
Zero	The two strings are equal

Example: compareTo()

```
class SortString {
      static String arr[] =
      {"Now", "is", "the", "time", "for", "all", "good,”
“men","to", "come", "to", "the", "aid", "of","their",
"country"};
      public static void main(String args[]) {
            for(int j = 0; j < arr.length; j++) {
                  for(int i = j + 1; i < arr.length;
i++) {
                        if(arr[i].compareTo(arr[j]) < 0)
{
                              String t = arr[j];
                              arr[j] = arr[i];
                              arr[i] = t;
```

Searching String 1

String class provides two methods that allows you search a string for a specified character or substring:

1) `indexOf()` – Searches for the first occurrence of a character or substring.
2) `lastIndexOf()` – Searches for the last occurrence of a charater or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or -1 on failure.

Searching String 2

To seach for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

To search for the first and the last occurence of a substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

Here `str` specifies the substring.

Searching String 3

You can specify a starting point for the serach using these forms:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

`startIndex` – specifies the index at which point the search begins.

For `indexOf()`, the search runs from startIndex to the end of the string.

For `lastIndexOf()`, the search runs from startIndex to zero.

Example: Searching String 1

```
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";
    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
```

Example: Searching String 2

```
    System.out.println("indexOf(t, 10) = " +
                                        s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                                      s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                                     s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                                     s.lastIndexOf("the", 60));
  }
}
```

Modifying a String

String object are immutable.

Whenever you want to modify a String, you must either copy it into a StringBuffer or use the following String methods,, which will construct a new copy of the string with your modification complete.

They are:

1) `subString()`
2) `concat()`
3) `replace()`
4) `trim()`

Each will be discussed.

subString() 1

You can extract a substring using subString().

It has two forms:

```
String substring(int startIndex)
```

`startIndex` specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

subString() 2

The second form allows you to specify both the beginning and ending index of the substring.

```
String substring(int startIndex, int ensIndex)
```

`startIndex` specifies the index beginning index, and

`endIndex` specifies the stopping point.

The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

Example: subString()

```
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;
    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
```

Example: subString()

```
                result = result + sub;
                result = result + org.substring(i +
                                  search.length());
        org = result;
      }
    } while(i != -1);
  }
}
```

concat()

You can concatenate two string using concat()

General form:

```
String concat(String str)
```

This method creates a new object that contains the invoking string with the contents of `str` appended to the end.

`concat()` performs the same function as +.

Example:

```
String s1 =“one”;
String s2 = s1.concat(“two”);
```

Or

```
String s2 = s1 + “two”;
```

replace()

Replaces all occurences of one character in the invoking string with another character.

General form:

```
String replace(char original, char replacement)
```

`original` – specifies the character to be replaced by the charscter specified by `replacement`. The resulting string is returned.

Example:

```
String s = "Hello".replace('l','w');
```

Puts the string "`Hewwo`" into `s`.

trim()

Returns a copy of the involving string from which any leading and trailing whitespace has been removed.

General form:

```
String trim();
```

Example:

```
String s = "     Hello world    ".trim();
```

This puts the string `"Hello world"` into `s`.

It is quite useful when you process user commands.

Example: trim() 1

```
import java.io.*;
class UseTrim {
  public static void main(String args[]) throws
                                              IOException{
    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));
    String str;
    System.out.println("Enter 'stop' to quit.");
    System.out.println("Enter State: ");
    do {
      str = br.readLine();
```

Example: trim() 2

```
        str = str.trim(); // remove whitespace
        if(str.equals("Illinois"))
             System.out.println("Capital is pringfield.");
        else if(str.equals("Missouri"))
          System.out.println("Capital is Jefferson
                                                City.");
        else if(str.equals("California"))
          System.out.println("Capital is Sacramento.");
        else if(str.equals("Washington"))
          System.out.println("Capital is Olympia.");
      } while(!str.equals("stop"));
    }
}
```

Data Conversion Using valueOf()

Converts data from its internal format into human-readable form.

It is a static method that is overloaded within `String` for all of Java's built-in types, so that each of the type can be converted properly into a String.

`valueOf()` can be overloaded for type `Object` so an object of any class type you create can also be used as an argument.

General forms:

```
static String valueOf(double num)
static String valueOf(long  num)
static String valueOf(Object obj)
static String valueOf(char chars[])
```

Case of Characters

The method `toLowerCase()` converts all the characters in a string from uppercase to lowercase.

The `toUpperCase()` method converts all the characters in a string from lowercase to uppercase.

Non-alphabetical characters, such as digits are unaffected.

General form:

```
String toLowercase()
String toUppercase()
```

Example: Case of Characters

```
class ChangeCase {
  public static void main(String args[]) {
    String s = "This is a test.";
    System.out.println("Original: " + s);
    String upper = s.toUpperCase();
    String lower = s.toLowerCase();
    System.out.println("Uppercase: " + upper);
    System.out.println("Lowercase: " + lower);
  }
}
```

StringBuffer

`StringBuffer` is a peer class of string that provides much of the functionality of Strings.

`String` is immutable. `StringBuffer` represents growable and writable character sequence.

`StringBuffer` may have characters and substring inserted in the middle or appended to the end.

`StringBuffer` will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

Defines three constructors:

1) `StringBuffer()` – default and reserves room for 16 characters without reallocation

2) `StringBuffer(int size)` – accepts an integer argument that explicitly sets the size of the buffer

3) `StringBuffer(String str)` – accepts a `String` argument that initially sets the content of the `StringBuffer` Object and reserves room for more16 characters without reallocation.

Length() and capacity()

Current length of a `StringBuffer` can be found via the `length()` method, while the total allocated capacity can be found throught the `capacity()` method.

General form:

```
int length()
Int capacity()
```

Example: Length() and capacity()

```
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

ensureCapacity()

Use `ensureCapacity()` to set the size of the buffer in order to preallocate room for a certain number of characters after a `StringBuffer` has been constructed.

General form:

void `ensureCapacity(int capacity)`

Here, `capacity` specifies the size of the buffer.

Usage:

Useful if you know in advance that you ill be appending a large number of small strings to a `StringBuffer.`

setLength()

To set the length of the buffer within a `StringBuffer` object.

General form:

void `setlengthint len)`

Here, `len` specifies the lenght of the buffer.

Usage:

When you increase the length of the buffer, null characters are added to the end of the existing buffer. If you call setLength() with a value less than the current value returned by length(), then the characters stored beyond the new length will be lost.

charAt() and setCharAt()

To obtain the value of a single character, use CharAt().

To set the value of a character within `StringBuffer`, use setCharAt().

General form:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

For `charAt()`, where specifies the index of the characters being obtained.

For `setCharAt()`, where specifies the index of the characters being set, and ch specifies the new value of that character.

`where` must be non negative and must not specify a location beyond the end of the buffer.

Example:charAt() and setCharAt()

```
class setCharAtDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer before = " + sb);
    System.out.println("charAt(1) before = " +
                                          sb.charAt(1));
      sb.setCharAt(1, 'i');
      sb.setLength(2);
      System.out.println("buffer after = " + sb);
      System.out.println("charAt(1) after = " +
sb.charAt(1));
  }
}
```

getChars()

To copy a substring of a StringBuffer into an array.

General form:

```
void getChars(int srcBegin, int srcEnd, char[] dst,
                                          int dstBegin)
```

Where:

`srcBegin` - start copying at this offset.

`srcEnd` - stop copying at this offset.

`dst` - the array to copy the data into.

`dstBegin` - offset into dst.

append()

Concatenates the string representation of any other type of data to the end of the invoking `StringBuffer` object.

General form:

```
StringBuffer append(Object obj)
StringBuffer append(String str)
StringBuffer append(int num)
```

`String.valueOf()` is called for each parameter to obtain its string representation. The result is appended to the current `StringBuffer` object.

Example: append()

```
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);
    s = sb.append("a =

      ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

insert()

Inserts one string into another. It is overloaded to accept values of all the simple types, plus String and Objects.

General form:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, `index` specifies the index at which point the String will be inserted into the invoking StringBuffer object.

Example: insert()

```
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");

    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

reverse()

To reverse the character within a StringBuffer object.

General form:

```
StringBuffer reverse()
```

This method returns the reversed on which it was called.

For example:

```
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");
    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

replace()

Replaces one set of characters with another set inside a StringBuffer object.

General form:

```
StringBuffer replace(int startIndex, String endIndex,
                                          String str)
```

The substring being replaced is specified by the indexes `startIndex` and `endIndex`. Thus, the substring at `startIndex` through `endIndex-1` is replaced. The replacement string is passed in `str`. The resulting `StringBuffer` object is returned.

Example: replace()

```
class replaceDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a


      test.");
    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

substring()

Returns a portion of a StringBuffer.

General form:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

The first form returns the substring that starts at `startIndex` and runs to the end of the invoking `StringBuffer` object.

The second form returns the substring that starts at `startIndex` and runs through `endIndex-1.`

These methods work just llike those defined for `String` that were described earlier.

Exercise: String Handling

1.) Write a program that computes your initials from your full name and displays them.

2.) Write a program to test if a word is a palindrome.

3.) Write a program to read English text to end-of-data, and print a count of word lengths, i.e. the total number of words of length 1 which occurred, the number of length 2, and so on.

Type in question 3 as input to test your program.

4.) An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one string is an anagram of another string. The program should ignore white space and punctuation.

Event Handling

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Event Handling

For the user to interact with a GUI, the underlying operating system must support event handling.

1) operating systems constantly monitor events such as keystrokes, mouse clicks, ink input, voice command, etc.
2) operating systems sort out these events and report them to the appropriate application programs
3) each application program then decides what to do in response to these events

Complexity vs. Power

Many programming languages have their ways of implementing events:

1) Visual Basic
    a) Each component responds to a fixed set of events
2) C
    a) A giant loop with a massive switch statement
3) Java
    a) Event delegation model – events are transmitted from event sources to event listeners
    b) You can designate any object to be an event listener

Event Handling Components 1

Event handling involves three components:

1) listener object is an instance of a class that implements a special interface called a listener interface.
2) event source is an object that can register listener objects and send them event objects
3) event source sends out event objects to all registered listeners when that event occurs.

The listener objects reacts based on the information in the event.

Event Handling Process 1

Register listener object with the event source object

```
eventSourceObject.addEventListener(eventListenerObject);
```

For example

```
ActionListener listener = …;
JButton button = new JButton("OK");
button.addActionListener(listener);
```

The listener object is notified whenever an action event occurs in the button (i.e., a button click)

Event Handling Process 2

The class to which the listener object belongs should implement the appropriate interface (in this case, the `ActionListener` interface)

Listener class must have a method `actionPerformed` that receives an `ActionEvent` object as a parameter

For Example:

```
   class MyListener implements ActionListener {
          ...
          public void actionPerformed(ActionEvent event)
{
               // reaction to button click goes here
               ...
          }
    }
```

Event Handling Process 3

Whenever the user clicks the button,

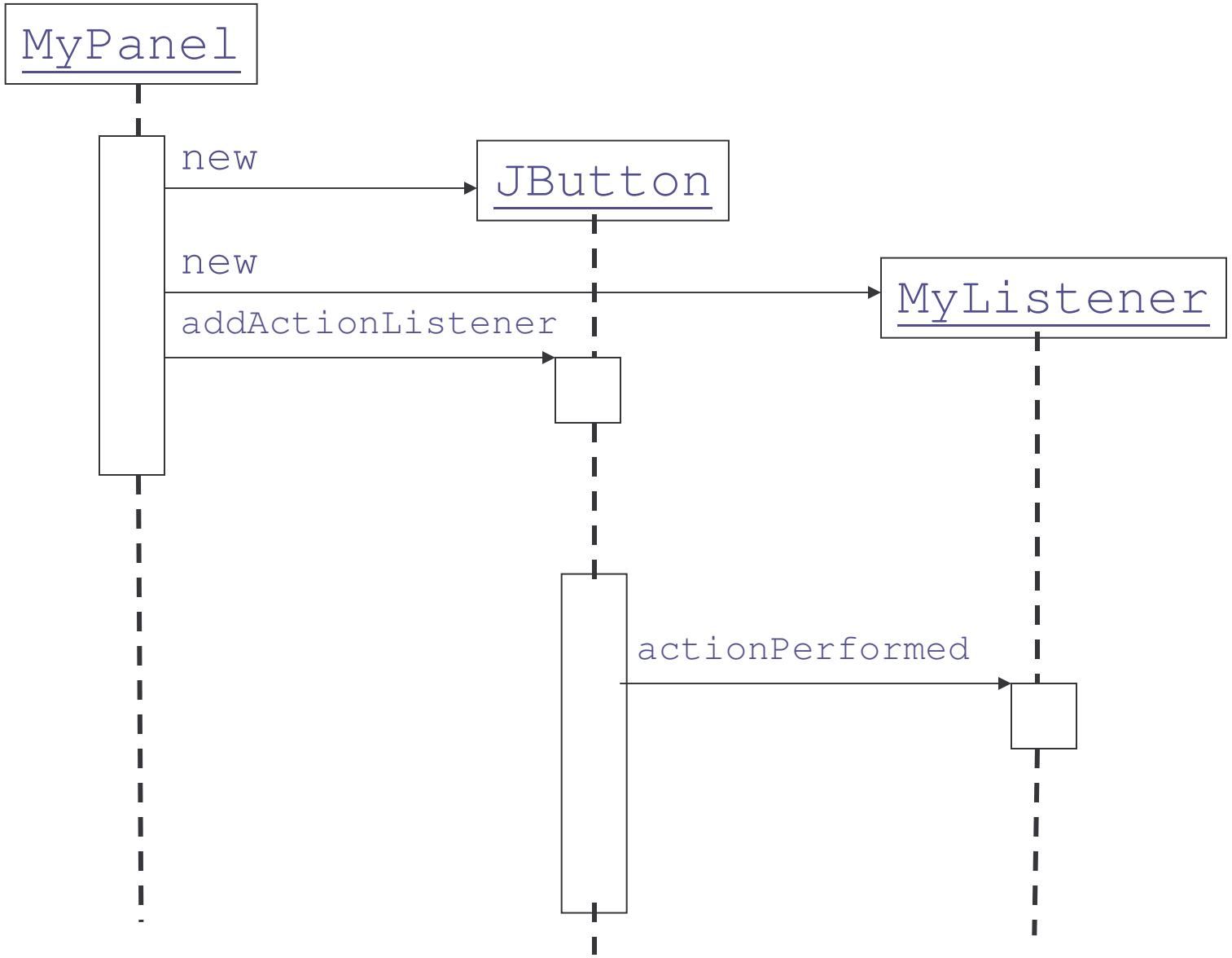
the `JButton` object creates an `ActionEvent` object and

calls `listener.actionPerformed(event)`, passing that event object

It is possible for multiple objects to be added as listeners to an event source by using the `addActionListener(listener)`.

The `actionPerformed` methods of all listeners will be called.

ActionEvent Handling Example



Event and Listener Objects

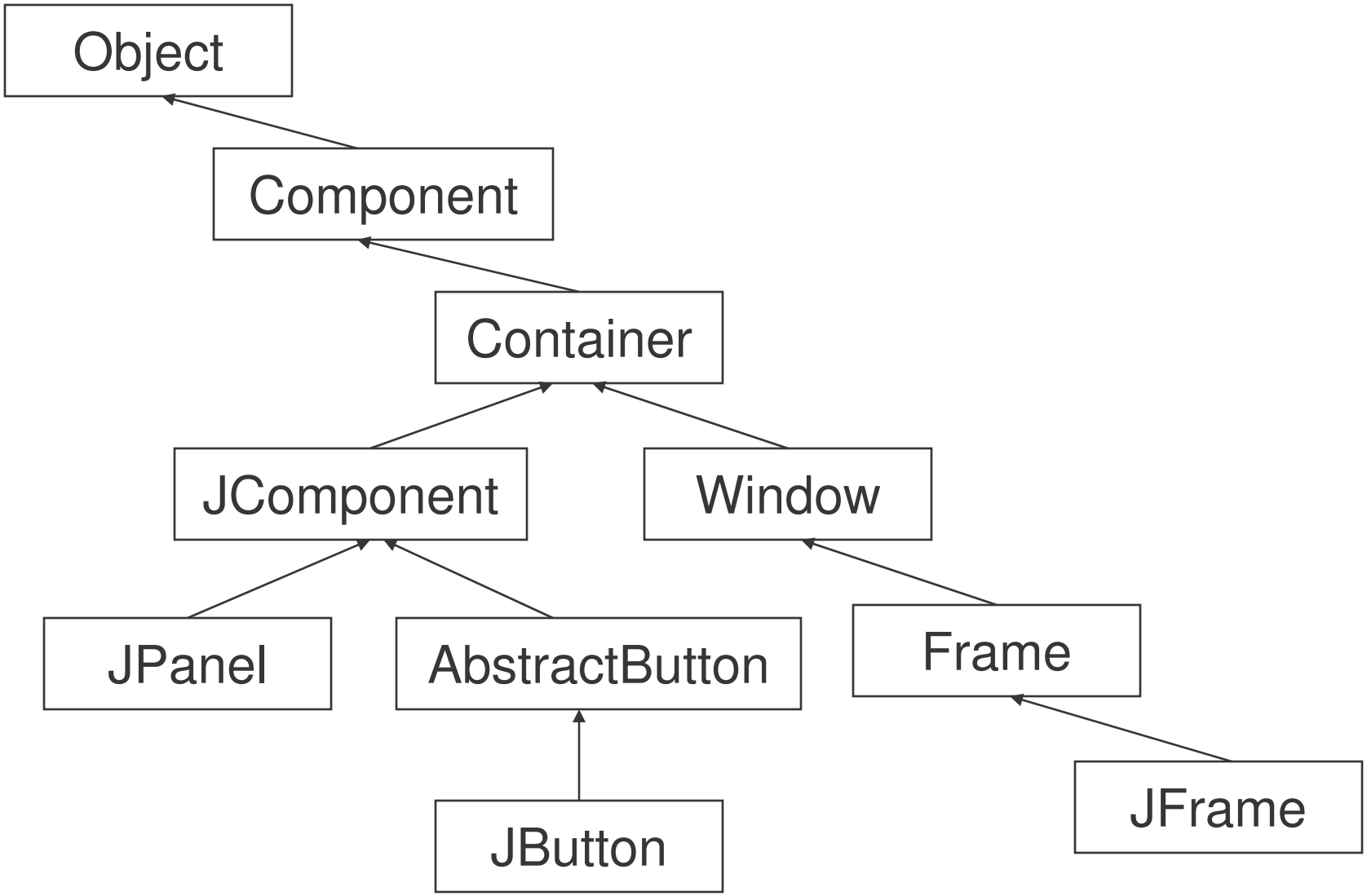
Different event sources can produce different kinds of events

a) `ActionEvent` – sent by `JButton` click and other components
b) `WindowEvent` – sent by `JFrame`

Different listener objects implement different required methods from interfaces

1) `ActionListener` – action performed
2) `WindowListener` – activated, closed, closing, deactivated, deiconified, iconified, opened

Example: JButton



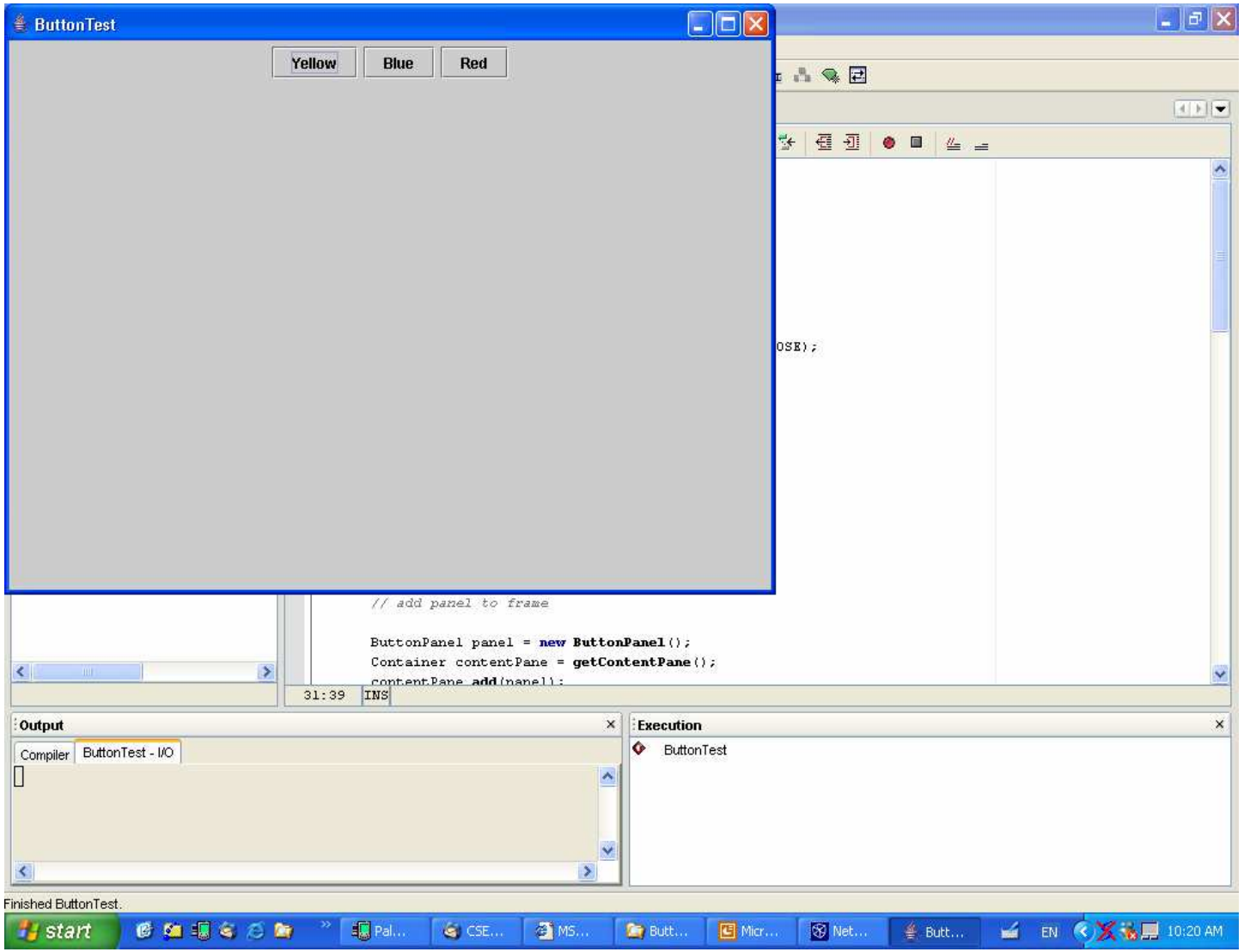
Example: JButton Event 1

```
class ButtonPanel extends JPanel {
   public ButtonPanel( ) {
      JButton yellowButton = new JButton("Yellow");
      JButton blueButton = new JButton("Blue");
      JButton redButton = new JButton("Red");
      add(yellowButton);
      add(blueButton);
      add(redButton);
   }
}
```

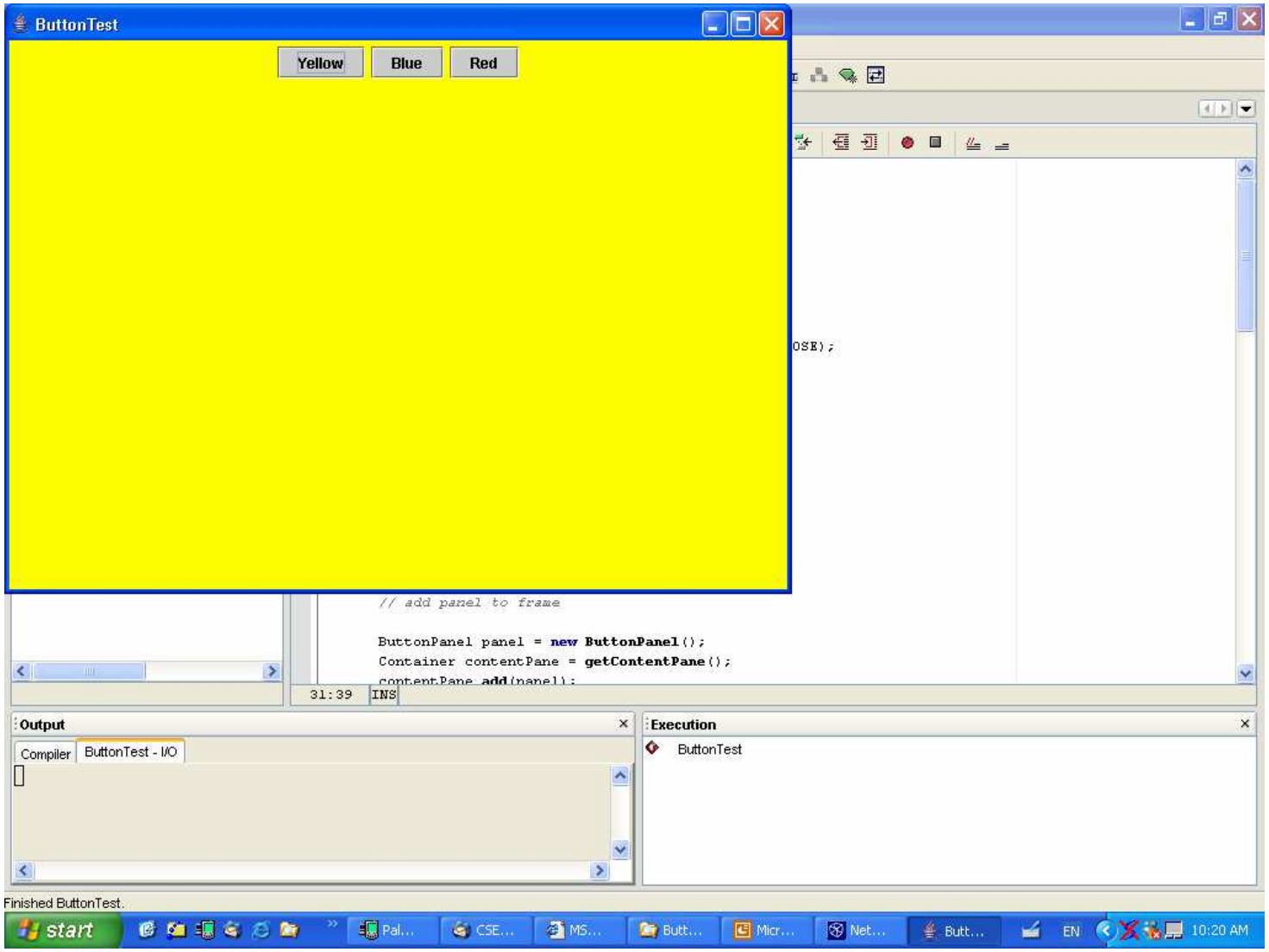
Constructors – button with a label string, an icon, or both a label string and an icon

```
JButton blueButton = new JButton("Blue");
JButton blueButton = new JButton(new ImageIcon("blue-
                                        ball.gif"));
```

Example: JButton Event 2



Example: JButton Event 3



Event Hierarchy 1

All events in Java descend from the `EventObject` class in the `java.util` package

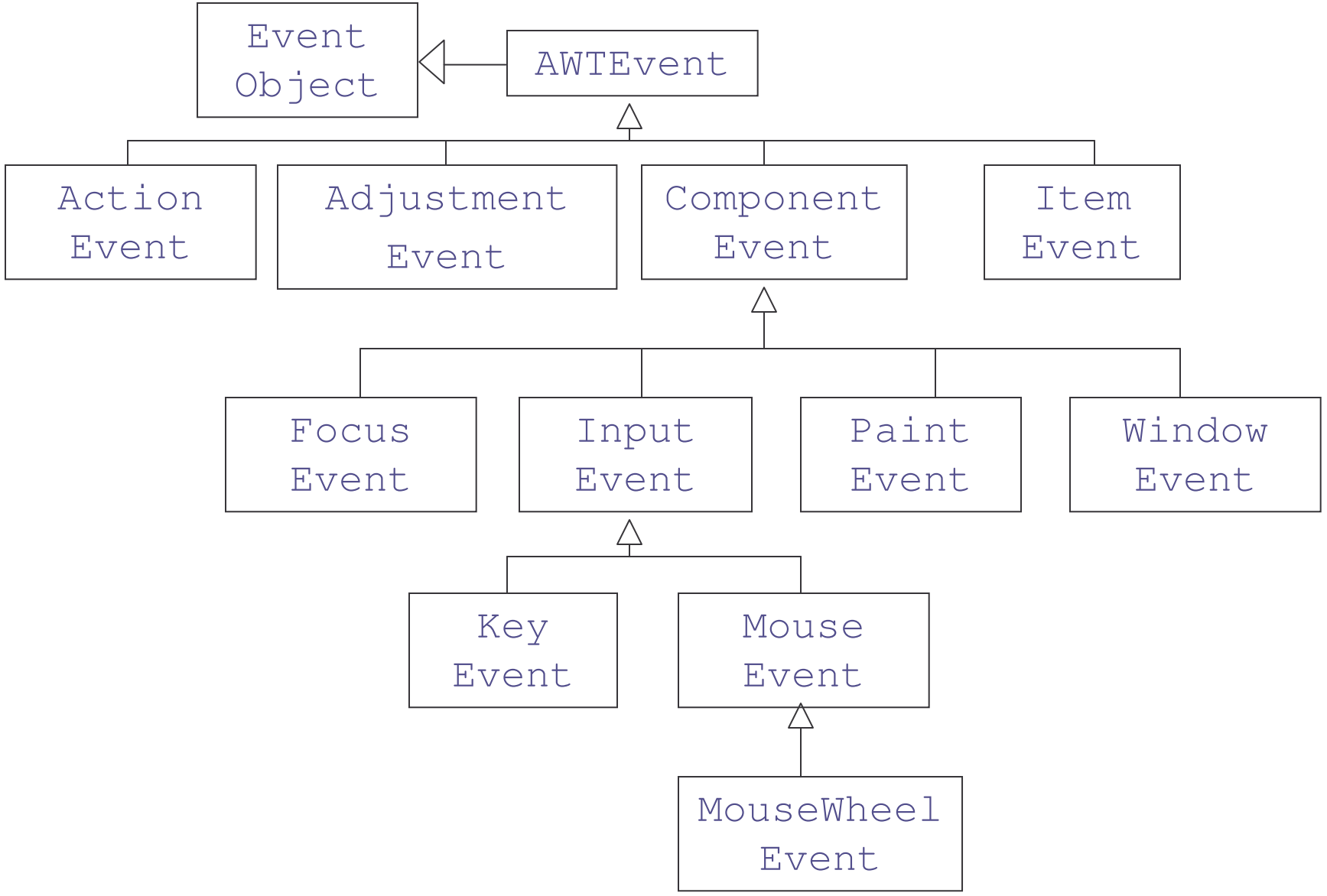
a) The `EventObject` class has a subclass `AWTEvent`, which is the parent of all AWT event classes

b) Some of the Swing components generate event objects that directly extend `EventObject`, not `AWTEvent`

c) You can add your own custom events by subclassing `EventObject` or any of the subclasses

The event objects encapsulate information about the event that the event source communicates to its listeners.

Event Hierarchy 2



Example: AWT Event Objects

Commonly used AWT event types

1) `ActionEvent`
2) `AdjustmentEvent`
3) `FocusEvent`
4) `ItemEvent`
5) `KeyEvent`
6) `MouseEvent`
7) `MouseWheelEvent`
8) `WindowEvent`

Example: AWT Listener Interfaces

The following interfaces listen to these events

1) `ActionListener`
2) `AdjustmentListener`
3) `FocusListener`
4) `ItemListener`
5) `KeyListener`
6) `MouseListener`
7) `MouseMotionListener`
8) `MouseWheelListener`
9) `WindowListener`
10) `WindowFocusListener`
11) `WindowStateListener`

Adapter Classes

Adapter Class exists as convenience for creating a listener object.

Extend this class to create a listener for a particular listener interface and override the methods for the events of interest.

It defines `null` methods for all of the methods in the listener interface, so you can only have to define methods for events you care about.

Commonly used adapter classes:

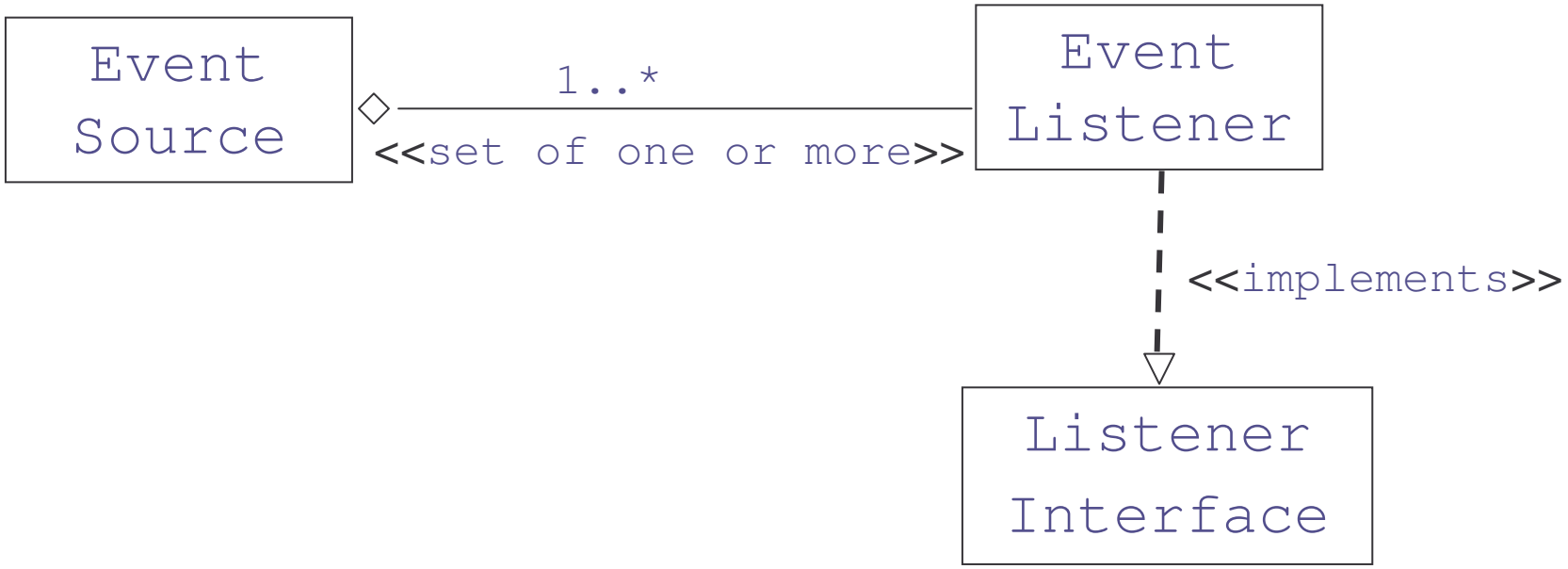
1) `FocusAdapter`
2) `KeyAdapter`
3) `MouseAdapter`
4) `MouseMotionAdapter`
5) `WindowAdapter`

Event Types

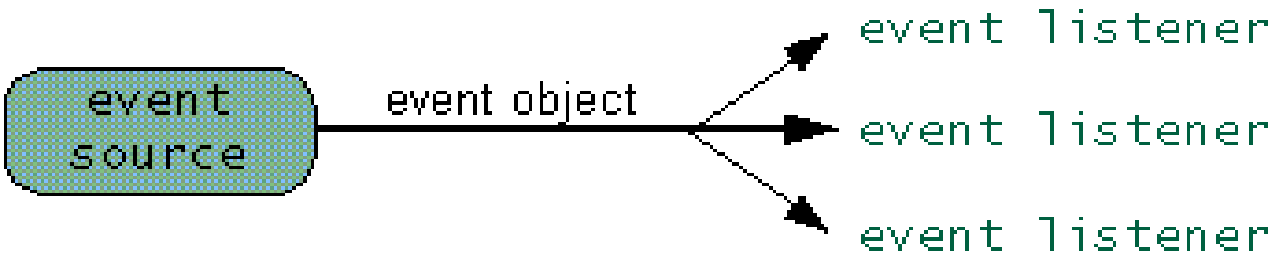
There are two types of events:

1) Semantic event – expresses what the user is doing
   a) `ActionEvent` – button click, menu selection, list item selection, press ENTER in a text field
   b) `AdjustmentEvent` – scrollbar adjustment
   c) `ItemEvent` – selection from a set of checkbox or list items
2) Low-level event –makes user interactions possible
   a) `FocusEvent` – a component got focus or lost focus
   b) `KeyEvent` – key was pressed or released
   c) `MouseEvent` – mouse button pressed, released, moved, dragged
   d) `MouseWheelEvent` – mouse wheel rotated
   e) `WindowEvent` – window state changed

Event Handling Summary



Implementation:



Exercise: Event Handling

1) What listener would you implement to be notified when a particular component has appeared on screen? What method tells you this information?
2) What listener would you implement to be notified when the user has finished editing a text field by pressing Enter?
3) What listener would you implement to be notified as each character is typed into a text field? Note that you should not implement a general-purpose key listener, but a listener specific to text.
4) What listener would you implement to be notified when a spinner's value has changed? How would you get the spinner's new value?
5) The default behavior for the focus subsystem is to consume the focus traversal keys, such as Tab and Shift Tab. Say you want to prevent this from happening in one of your application's components. How would you accomplish this?

Object Collections

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Overview 1

1) Introduction - tells you what collections are, and how they will make your job easier and your programs better.

2) Interfaces - describes the core collection interfaces, which are the heart and soul of the Java Collections Framework. You will learn:
   a) general guidelines for effective use of these interfaces, including when to use which interface
   b) idioms for each interface that will help you get the most out of the interfaces.

Overview 2

3) Implementations - describes the JDK's general-purpose collection implementations and tells you when to use which implementation.

4) Algorithms - describes the polymorphic algorithms provided by the JDK to operate on collections. With any luck you will never have to write your own sort routine again!

What Is a Collection?

Collection (sometimes called a container) is simply an object that groups multiple elements into a single unit.

Usage:

1) to store and retrieve data

2) to manipulate data

3) to transmit data from one method to another

Collections typically represent data items that form a natural group like:

1) a poker hand (a collection of cards)

2) a mail folder (a collection of letters)

3) a telephone directory (a collection of name-to-phone-number mappings)

Collection Framework 1

A collections framework is a unified architecture for representing and manipulating collections.

All collections frameworks contain three things:

1) Interfaces
    a) abstract data types representing collections.
    b) Interfaces allow collections to be manipulated independently of the details of their representation.

Collection Framework 2

2) Implementations

   a. concrete implementations of the collection interfaces.

   b. In essence, these are reusable data structures.

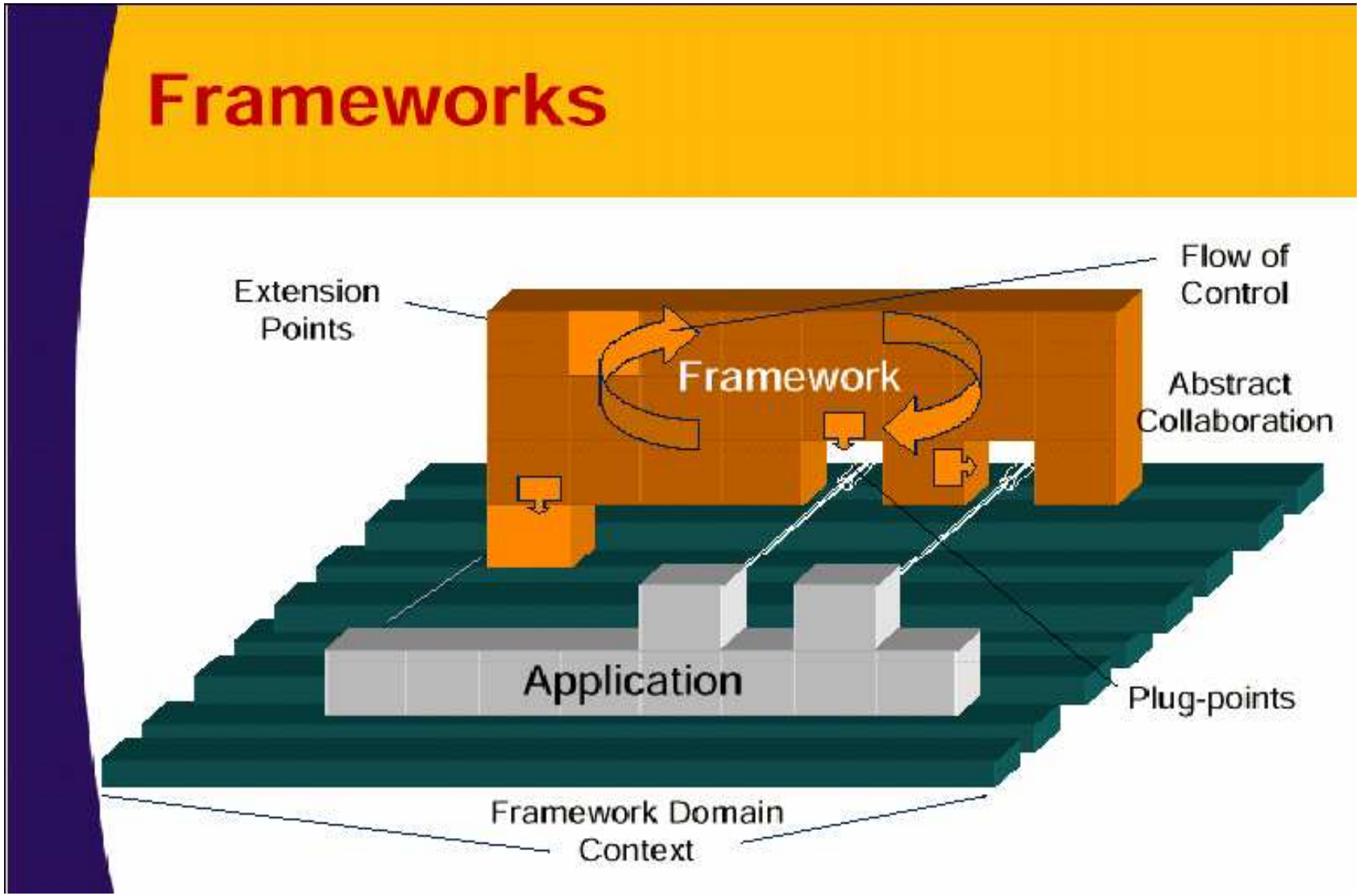
3) Algorithms

   a) methods that perform useful computations like searching and sorting, on objects that implement collection interfaces.

   b) they are polymorphic because the same method can be used on many different implementations of the appropriate collections interface.

   c) In essence, they are reusable functionality.

Collection Framework 3



Benefits 1

Collection Framework offers the following benefits:

1) It reduces programming effort
    a) Powerful data structures and algorithms

2) It increases program speed and quality
    a) High quality implementations
    b) Fine tuning by switching implementations

3) It allows interoperability among unrelated APIs
    a) Passing objects around from one API to another

Benefits 2

4) It reduces the effort to learn and use new APIs
    a) Uniformity of the framework
    b) APIs of applications

5) It reduces effort to design new APIs

6) It fosters software reuse
    a) New data structures and algorithms

Core Collection Interfaces 1

These are interfaces used to manipulate collections, and pass them from one method to another.

Purpose:

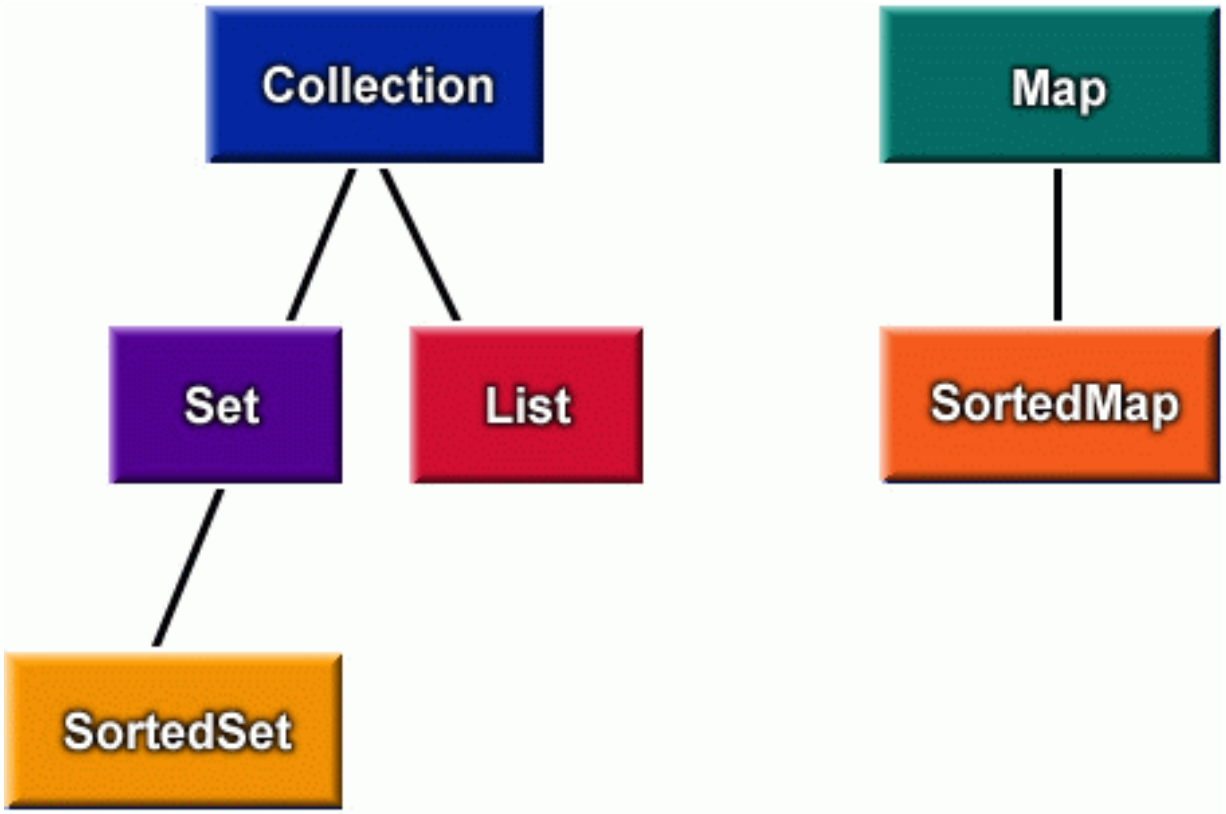
1)To allow collections to be manipulated independently of the details of their representation.

Note:

1) They are the heart and soul of the collections framework.
2) When you understand how to use these interfaces, you know most of what there is to know about the framework.

Core Collection Interfaces 2

The core collections interfaces are shown below:



Core Collection Interfaces 3

There are four basic core collection interfaces:

1) `Collection`
2) `Set`
3) `List`
4) `Map`

Collection

The `Collection` interface is the root of the collection hierarchy.

Usage:

To pass around collections of objects where maximum generality is desired.

Behaviors:

1) Basic Operations
2) Bulk Operations
3) Array Operations

Collection Methods 1

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional
```

Collection Methods 2

```
    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

It has methods to tell you:

1) how many elements are in the collection (size, isEmpty)
2) to check if a given object is in the collection (contains)
3) to add and remove an element from the collection (add, remove),
4) and to provide an iterator over the collection (iterator)

Set 1

A `Set` is a `Collection` that cannot contain duplicate elements.

`Set` models the mathematical set abstraction.

Example:

1) Set of Cars - {BMW, Ford, Jeep, Chevrolet, Nissan, Toyota, VW}

2) Nationalities in the class - {Chinese, American, Canadian, Indian}

It extends Collection and contains no methods other than those inherited from Collection.

Set 2

`Set` extends Collection to add the following functionality:

a) stronger contract on the behavior of the equals and hashCode operations,

b) allowing Set objects with different implementation types to be compared meaningfully.

Two `Set` objects are equal if they contain the same elements.

Set Methods 1

The `Set` interface is shown below:

```
public interface Set {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
```

Set Methods 2

```
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Provides two general purpose implementations:

1) `HashSet` – which stores its elements in a hash table, is the best-performing
2) `TreeSet` – which stores its elements in a red-black tree, guarantees the order of iteration.

Example: Set

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.print("Duplicate”)
                System.out.println(“detected: "+args[i]);
                System.out.println(s.size()+" distinct”)
                System.out.println(“words detected: "+s);
    }
}
```

List 1

An ordered collection (sometimes called a sequence)

Lists may contain duplicate elements

Collection operations:

1) remove operation removes the first occurrence of the specified element
2) add and addAll operations always appends new elements to the end of the list.
3) Two List objects are equal if they contain the same elements in the same order.

List 2

1) New List operations
    a) Positional access
    b) Search
    c) Iteration (ordered, backward)
    d) Range-view operations

2) General Purpose Implementation
    a) ArrayList
    b) ListIterator

Example: List

```
private static void swap(List a, int i, int j)
{
  Object tmp = a.get(i);
  a.set(i, a.get(j));
  a.set(j, tmp);
}

for (ListIterator i=l.listIterator(l.size());
   i.hasPrevious(); ) {
  Foo f = (Foo)i.previous();
  ...
}
```

Exampe: List

```
public static void replace(List l, Object val, List
   newVals){
  for (ListIterator i = l.listIterator(); i.hasNext() ;
   ) {
    if (val==null ? i.next()==null :
   val.equals(i.next())) {
      i.remove();
      for (Iterator j = newVals.iterator(); j.hasNext();
   )
        i.add(j.next());
    }
  }
}
```

Iterator

A mechanism for iterating over a collection's elements.

Represented by the Iterator interface.

Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics

The only safe way to modify a collection during iteration

Location: `java.util.Iterator`

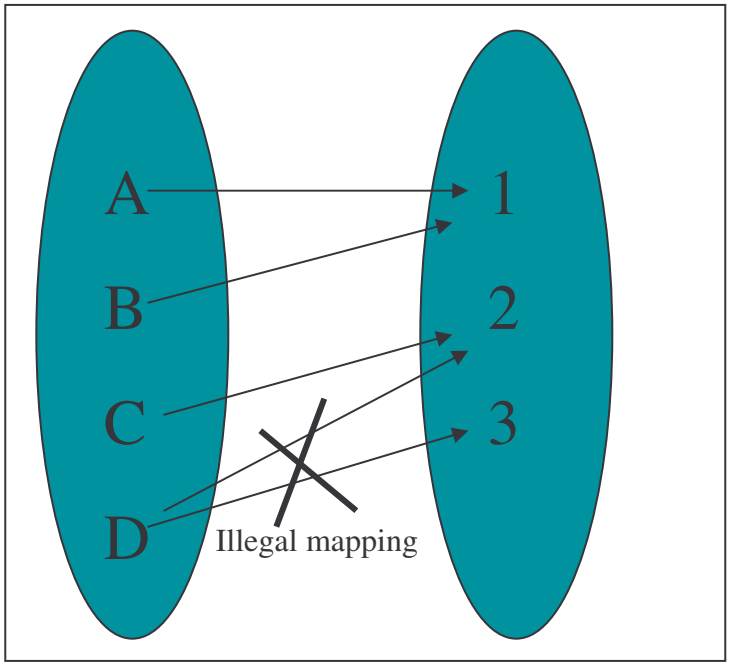
Example: Iterator

```
static void filter(Collection c)
{
  for (Iterator it = c.iterator() ; it.hasNext(); )
    if (!cond(it.next()))
      it.remove();
}

static void filter(Collection c)
{
  Iterator it = c.iterator();
  while (it.hasNext())
    if (!cond(it.next()))
      it.remove();
}
```

Map 1

A Map is an object that maps keys to values. Maps cannot contain duplicate keys.

1) Each key can map to at most one value
2) A map cannot contain duplicate keys.
3) Two Map objects are equal if they represent the same key-value mappings



Map

Examples:

a) Think of a dictionary:
   word <-> description
b) address book
   name <-> phone number

Map 2

1) Collection-view methods allow a Map to be viewed as a Collection
   a) keySet – The Set of keys contained in the Map
   b) values – The Collection of values contained in the Map
   c) entrySet – The Set of key-value pairs contained in the Map

2) With all three Collection-views, calling an Iterator's remove operation removes the associated entry from the backing Map.
   a) This is the only safe way to modify a Map during iteration.

3) Every object can be used as a hash key

4) Two Map objects are equal if they represent the same key-value mappings

Example: Map 1

```
import java.util.*;
public class Freq {
  private static final Integer ONE = new Integer(1);
  public static void main(String args[]){
    Map m = new HashMap();
    for (int i=0; i<args.length; i++) {
      Integer freq = (Integer)m.get(args[i]);
      m.put(args[i], (freq==null ?
          ONE : new Integer(freq.intValue() + 1)));
    }
    System.out.println(m.size() + " distinct words
   detected:");
    System.out.println(m);
  } }
```

Example: Map 2

```
for (Iterator i = m.keySet().iterator() ;
   i.hasNext() ; )
  System.out.println(i.next());

for (Iterator i = m.values().iterator();
   i.hasNext() ; )
  System.out.println(i.next());

for (Iterator i = m.entrySet().iterator();
   i.hasNext() ; ) {
  Map.Entry e = (Map.Entry)i.next();
  System.out.println(e.getKey() + ": " +
   e.getValue());
}
```

SortedSet Interface

1) A Set that maintains its elements in ascending order
   a) according to elements' natural order
   b) according to a Comparator provided at SortedSet creation time
2) Set operations
   a) Iterator traverses the sorted set in order
3) Additional operations
   a) Range view – range operations
   b) Endpoints – return the first or last element
   c) Comparator access
4) Location: java.util.SortedSet

SortedMap Interface

1) A Map that maintains its entries in ascending order
   a) According to keys' natural order
   b) According to a Comparator provided at creation time.
2) Map operations
   a) Iterator traverses elements in any of the sorted map's collection-views in key-order.
3) Additional Operations
   a) Range view
   b) End points
   c) Comparator access
4) Location: java.util.SortedMap

Implementations

1) Implementations are the actual data objects used to store elements
    a) Implement the core collection interfaces
2) There are three kinds of implementations
    a) General purpose implementations
    b) Wrapper implementations
    c) Convenience implementations

General Purpose Implementations

		Implementations			
		Hash Table	**Resizeable Array**	**Balanced Tree**	**Linked List**
Interfaces	**Set**	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Older Implementations

1) The collections framework was introduced in JDK1.2.
2) Earlier JDK versions included collection implementations that were not part of any framework
   a) java.util.Vector
   b) java.util.Hashtable
3) These implementations were extended to implement the core interfaces but still have all their legacy operations
   a) Be careful to always manipulate them only through the core interfaces.

Algorithms 1

Algorithms are pieces of reusable functionality provided by the JDK.

All of them come from the `Collections` class.

All take the form of static methods whose first argument is the collection on which the operation is to be performed.

The great majority of the algorithms provided by the Java platform operate on `List` objects,

A couple of them (min and max) operate on arbitrary `Collection` objects.

Algorithms 2

The algorithms are described below:

1)Sorting

a) reorders a List so that its elements are ascending order according to some ordering relation.

b) The important things to know about this algorithm are that it is:

- Fast: This algorithm is guaranteed to run in n log(n) time, and runs substantially faster on nearly sorted lists.
- Stable: That is to say, it doesn't reorder equal elements.

2) Shuffling

a) does the opposite of what sort does - it destroys any trace of order that may have been present in a List.

Algorithms 3

3) Routine Data Manipulation

a) The Collections class provides three algorithms for doing routine data manipulation on List objects: `Reverse`, `Fill` and `Copy`

4) Searching

a) The binarySearch algorithm searches for a specified element in a sorted List using the binary search algorithm.

5) Finding Extreme Values

a) The min and max algorithms return, respectively, the minimum and maximum element contained in a specified Collection.

Exercise: Object Collection

1) Write a class that stores an object of `Building`, `HighBuilding`, and `Skyscraper` into a Map (either a `HashMap` or `TreeMap`)
2) Store each of these elements into an `ArrayList`.
3) Create another class that takes a variable `ArrayList` as a parameter.
4) Call a method on the class to Iterate through the `ArrayList` and call `enter` method of each object. Direct your output to the console.
5) Write a program to count the number of different words occurring in a text. Whenever we inspect the next word in the text we need to know if it has already occurred or not, so we need to store the words that have already occurred in *some* data structure in our program.

   For this program we need a data structure that stores the words we have already encountered in the text in order to look following words up and decide whether to count one more different word or not.

   Test your program with the text of exercise 5. Print out the content of the data structure.

Vertical Libraries

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Graphical Interface

Course Outline

1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

Overview

1) A Quick Start Guide
2) Swing Features and Concepts
3) Summary

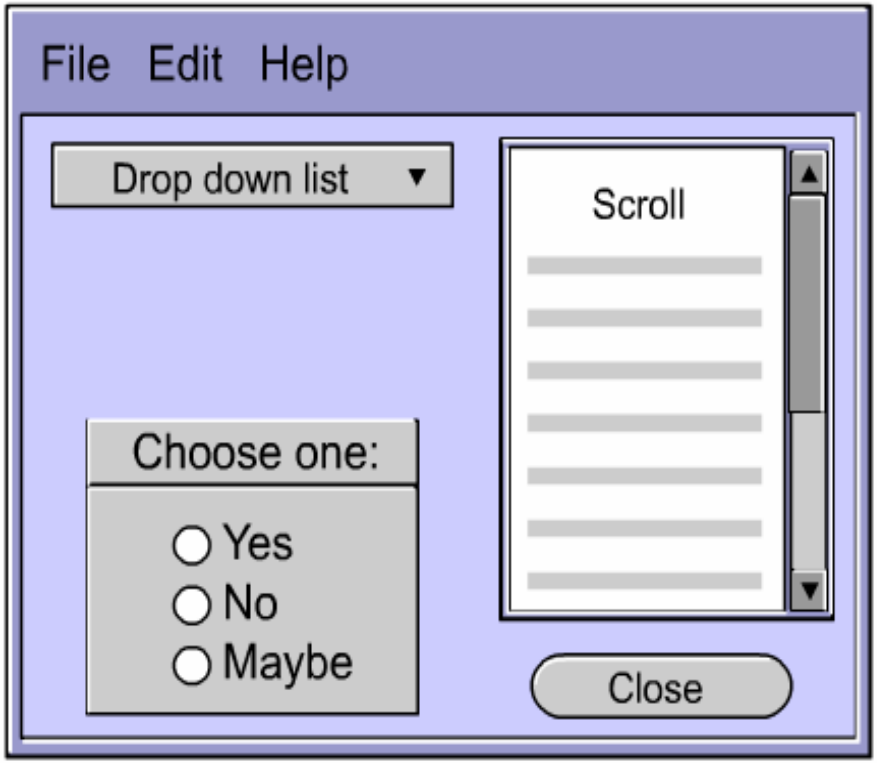
Overview

1) A Quick Start Guide
2) Swing Features and Concepts
3) Summary

A Quick Start Guide

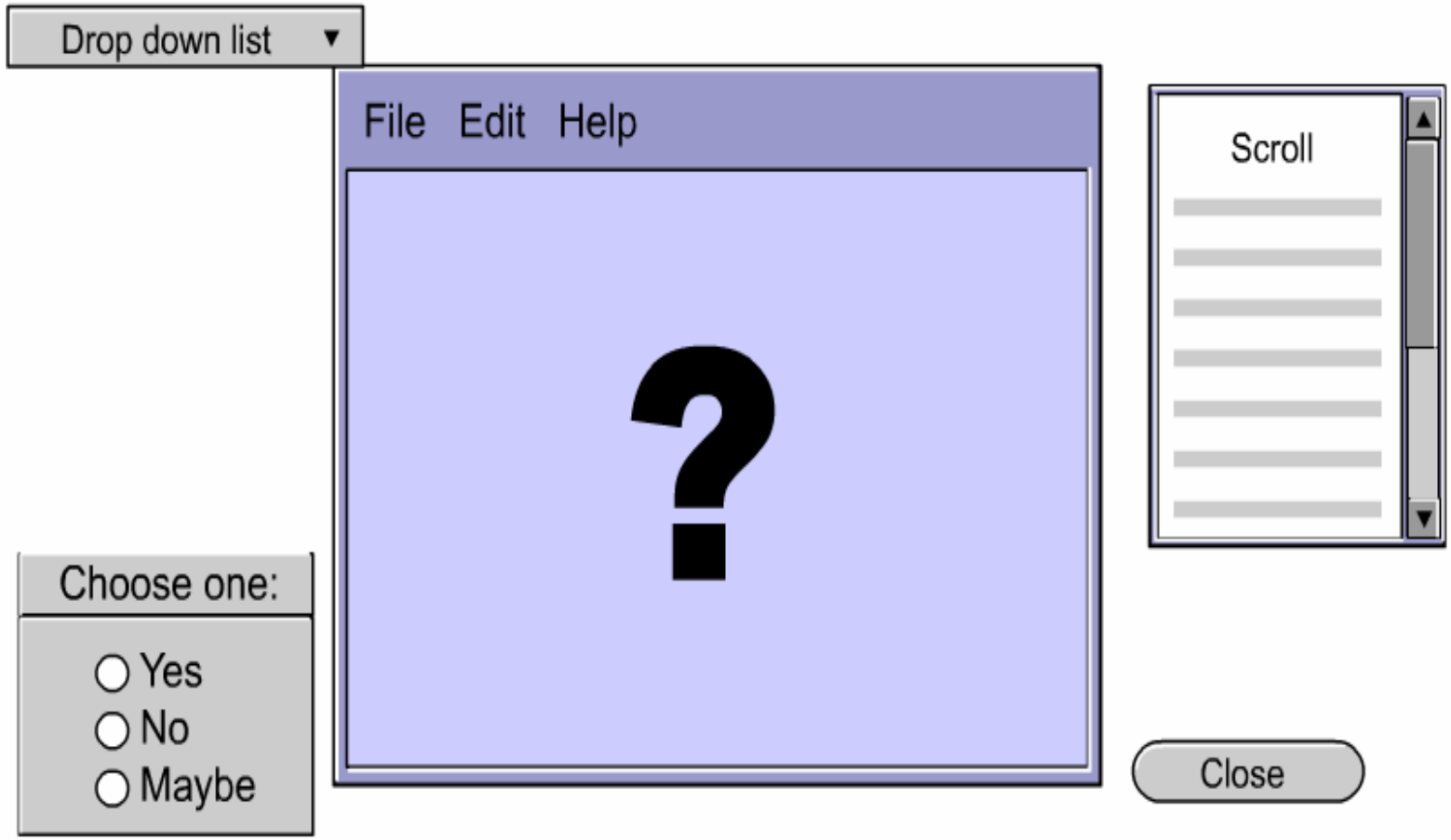
1) Overview of Swing API
2) First Swing Application
3) Swing Application (SA)
    a) Look and Feel
    b) Setting up Button, Label
    c) Handling Events
    d) Border and Component
4) CelsiusConverter
    a) Adding HTML
    b) Adding an Icon
5) LunarPhases
    a) Compound Border
    b) Combo Boxes
    c) Multiple Images
6. Layout Management

Overview of Swing API 1



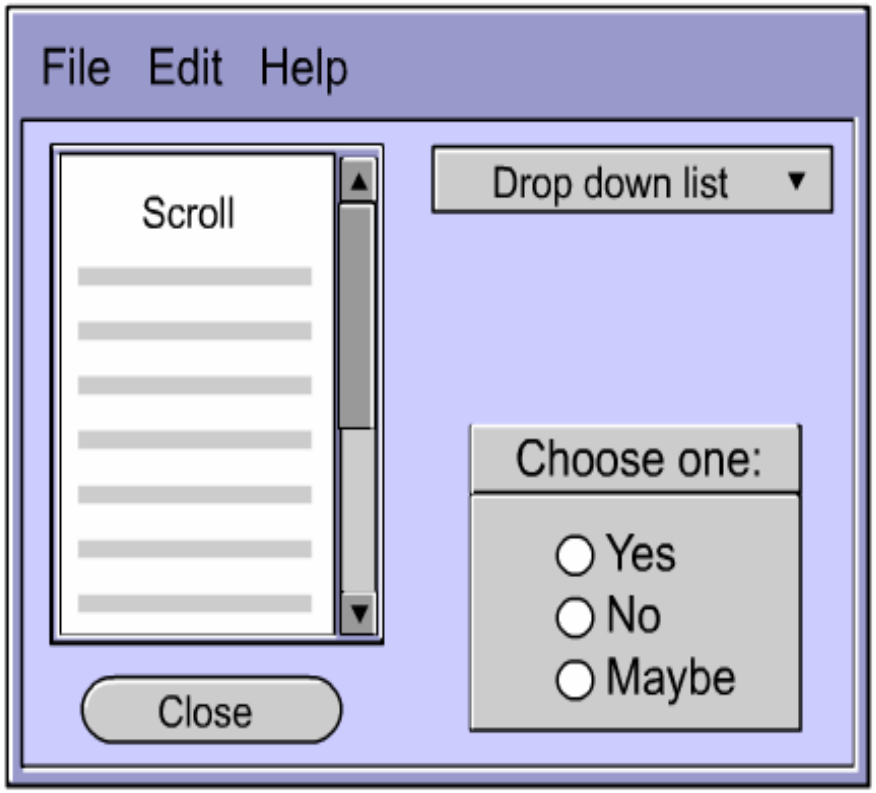
Modern computer applications usually use a graphical user interface, or GUI, to interact with users.

Overview of Swing API 2



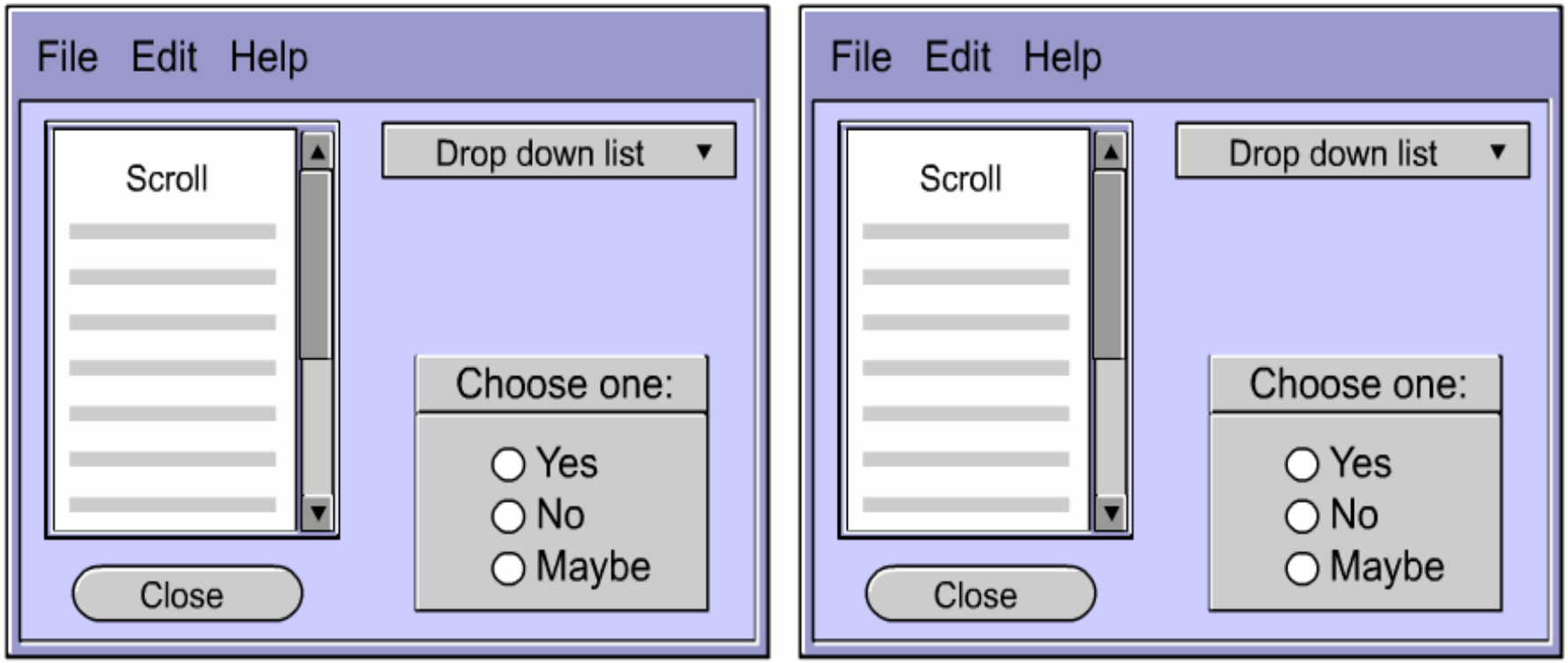
One item to consider is how to postion each component on the GUI.

Overview of Swing API 3



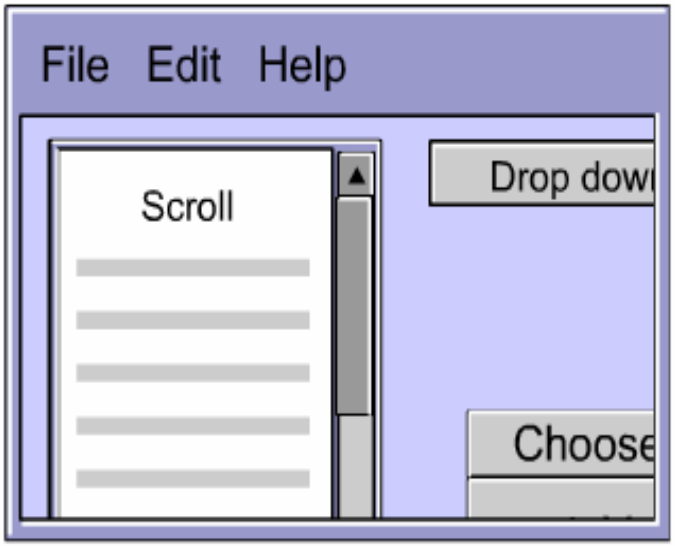
One item to consider is how to postion each component on the GUI.

Overview of Swing API 4



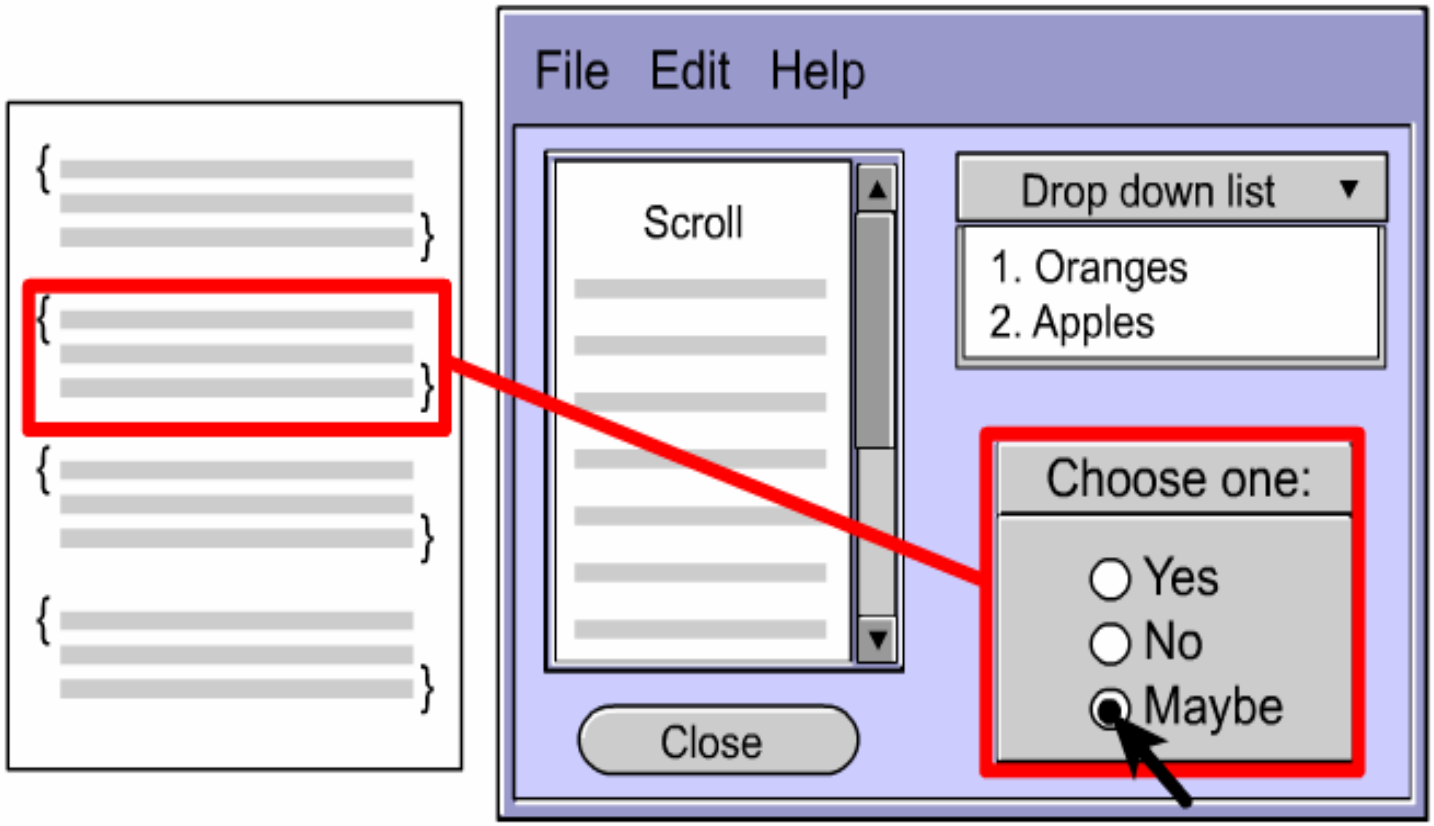
Another thing to consider is what happens when the window is resized.

Overview of Swing API 5



Another thing to consider is what happens when the window is resized.

Overview of Swing API 6



Finally, the GUI needs to be functional. Specific behaviors need to be associated with each component.

Overview of Swing API 7

What are the JFC and Swing?

Definition

JFC is short for Java Foundation Classes, which encompass a group of features to help people build graphical user interfaces (GUIs).

First announced at the 1997 JavaOne developer conference and is defined as containing the following features:

1) Swing Components - from buttons to split panes to tables
2) Pluggable Look and Feel Support - looks and feels
3) Accessibility API - enables assistive technologies
4) Java 2D API - high-quality 2D graphics, text, images
5) Drag and Drop Support - ability to drag and drop between a Java application and a native application

Overview of Swing API 7

The first three JFC features were implemented without any native code, relying only on the API defined in JDK 1.1

This extension was released as JFC 1.1, which is sometimes called "the Swing release."

The API in JFC 1.1 is often called "the Swing API."

Note: "Swing" was the codename of the project that developed the new components.

Overview of Swing API 8

Prior to the introduction of the Swing, AWT (Abstract Window Toolkit) provided all the UI components in the JDK 1.0 and 1.1 platforms.

Java 2 Platform still supports the AWT components

AWT is in java.awt while Swing is in javax.swing

Swing Components starts with “J”

AWT button – Button

Swing Button - JButton

First Swing Application 1

This section examines the code for a simple program, HelloWorldSwing. This introduces some basic features of Swing.

```
import javax.swing.*;
public class HelloWorldSwing {
      public static void main(String[] args) {
      JFrame frame = new JFrame("HelloWorldSwing");
      final JLabel label = new JLabel("Hello World");
      frame.getContentPane().add(label);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_C
                                          LOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

First Swing Application 2

The code demonstrates the basic code in every Swing program:

1) Import the pertinent packages
2) Set up a top-level container

The first line imports the main Swing package:

```
import javax.swing.*;
```

However, most Swing programs also need to import two AWT packages because Swing components use the AWT infrastructure, including the AWT event model.

```
import java.awt.*;

import java.awt.event.*;
```

First Swing Application 3

Every program with a Swing GUI must contain at least one top-level Swing container.

Definition

A top-level Swing container provides the support that Swing components need to perform their painting and event handling.

There are three top-level containers:

1) `JFrame`- implements a single main window
2) `JDialog`- implements a secondary window (a window that's dependent on another window)
3) `JApplet`- implements an applet's display area within a browser window

First Swing Application 4

Here is the code that sets up and shows the frame:

```
JFrame frame = new JFrame("HelloWorldSwing");
...
frame.pack();
frame.setVisible(true);
```

These two lines of code construct a label and then add the component to the frame.

```
final JLabel label = new JLabel("Hello World");
frame.getContentPane().add(label);
```

This code closes the window when is clicked

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

First Swing Application 5

JFrame provides the `setDefaultCloseOperation` method to configure the default action for when the user clicks the close button.

The `EXIT_ON_CLOSE` constant lets you specify this, as of version 1.3 of the Java 2 Platform.

To implement this in earlier version, you have to write a window event listener.

```
frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }
    }
);
```

Basic Features

We use the following applications to illustrate the basic features of swing:

1) `SwingApplication.java`
2) `CelsiusConverter.java`
3) `LunarPhases.java`

Example: SwingApplication 1

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SwingApplication {
    private static String labelPrefix = "Number of
                                        button clicks: ";
    private int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix +
                                        "0    ");
        JButton button = new JButton("I'm a Swing
                                        button!");
        button.setMnemonic(KeyEvent.VK_I);
```

Example: SwingApplication 2

```
button.addActionListener(new ActionListener()
                                                {
public void actionPerformed(ActionEvent e) {
    numClicks++;
    label.setText(labelPrefix + numClicks);
    }
});
label.setLabelFor(button);
JPanel pane = new JPanel();
```

Example: SwingApplication 3

```
pane.setBorder(BorderFactory.createEmptyBorder(
                                              30, //top
                                              30, //left
                                              10, //bottom
                                              30) //right
                                              );
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);

        return pane;
    }
```

Example: SwingApplication 4

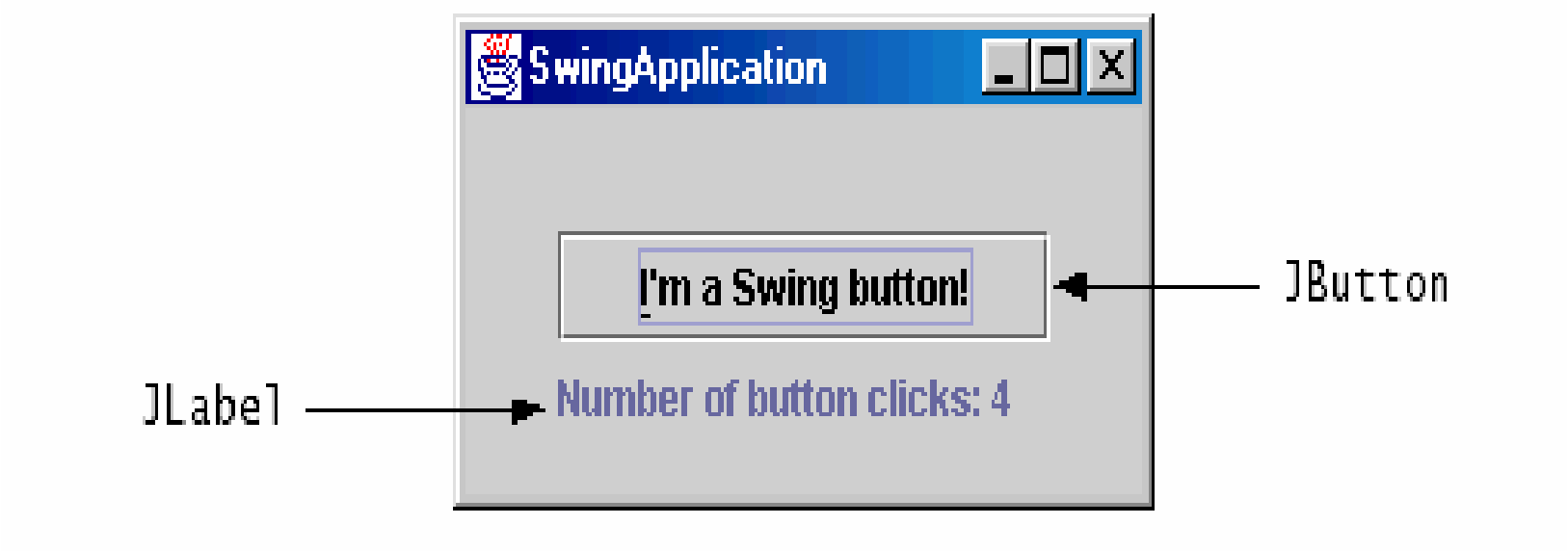
```
public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName(
                                                ));
        } catch (Exception e) {}
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents,
                                   BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_C
                                              LOSE);
        frame.pack(); frame.setVisible(true);}}
```

Example: SwingApplication 5

This program illustrates the following features of swing:

1) Look and Feel
2) Setting Up Buttons and Labels
3) Handling Events
4) Adding Borders Around Components

Example: SwingApplication 5



Look and Feel 1

Swing allows you to specify which look and feel your program uses

1) Java look and feel

2) CDE/Motif look and feel

3) Windows look and feel, and so on

Example: Look and Feel

The code snippet shows you how SwingApplication specifies its look and feel:

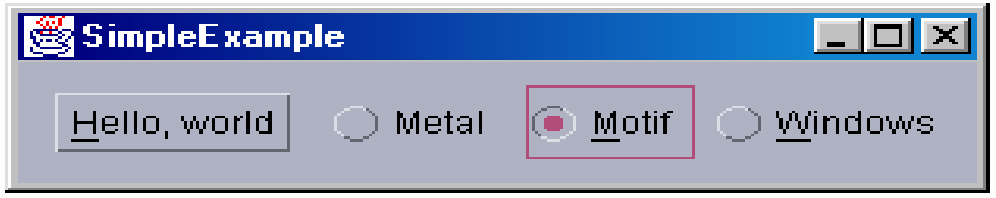
```
public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getCrossPlat
                          formLookAndFeelClassName());
    } catch (Exception e) { }
    ...// Create and show the GUI...
}
```

Use cross-platform look and feel.

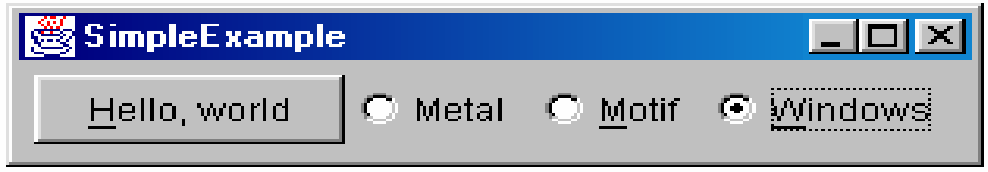
Example: Look and Feel

The following figure shows three views of a GUI that uses Swing components.

Java look and feel



CDE/Motif look and feel



Windows look and feel

Buttons and Labels

Here's the code that initializes the button:

```
JButton button = new JButton("I'm a Swing button!");
button.setMnemonic('i');
button.addActionListener(...an actionlistener...);


private static String labelPrefix = "Number of button
clicks: ";
private int numClicks = 0;
...// in GUI initialization code:
final JLabel label = new JLabel(labelPrefix + "0
"); …
label.setLabelFor(button);
...// in the event handler for button clicks:
label.setText(labelPrefix + numClicks);
```

Example: CelsiusConverter 1

This program illustrates two features of swing:

1) Adding HTML to a Lablel
2) Adding Icon to a swing Button

Example: CelsiusConverter 1

This program illustrates two features of swing:

1) Adding HTML to a Lablel
2) Adding Icon to a swing Button

Example: CelsiusConverter 3

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CelsiusConverter implements
  ActionListener {
    JFrame converterFrame;
    JPanel converterPanel;
    JTextField tempCelsius;
    JLabel celsiusLabel, fahrenheitLabel;
    JButton convertTemp;
    public CelsiusConverter() {
  converterFrame = new JFrame("Convert Celsius to
                                    Fahrenheit");
```

Example: CelsiusConverter 4

```
converterPanel = new JPanel();
converterPanel.setLayout(new GridLayout(2, 2));
addWidgets();
converterFrame.getContentPane().add(
    converterPanel, BorderLayout.CENTER);
converterFrame.setDefaultCloseOperation(JFrame.
                                    EXIT_ON_CLOSE);
converterFrame.pack();
converterFrame.setVisible(true);
  }
```

Example: CelsiusConverter 5

```
  private void addWidgets() {
tempCelsius = new JTextField(2);
celsiusLabel = new JLabel("Celsius",
                          SwingConstants.LEFT);
convertTemp = new JButton("Convert...");
fahrenheitLabel = new JLabel("Fahrenheit",
                          SwingConstants.LEFT);
convertTemp.addActionListener(this);
converterPanel.add(tempCelsius);
converterPanel.add(celsiusLabel);
converterPanel.add(convertTemp);
converterPanel.add(fahrenheitLabel);
```

Example: CelsiusConverter 6

```
  celsiusLabel.setBorder(BorderFactory.
createEmptyBorder(5,5,5,5));
  fahrenheitLabel.setBorder(BorderFactory.
                  createEmptyBorder(5,5,5,5));
    }


    public void actionPerformed(ActionEvent event) {
        int tempFahr =
  (int)((Double.parseDouble(tempCelsius.getText()))
                           * 1.8 + 32);
  fahrenheitLabel.setText(tempFahr + " Fahrenheit");
    }
```

Example: CelsiusConverter 7

```
    public static void main(String[] args) {
  try {
      UIManager.setLookAndFeel(
  UIManager.getCrossPlatformLookAndFeelClassName(
                                                  ));
  } catch(Exception e) {}
  CelsiusConverter converter = new
      CelsiusConverter();
    }
}
```

Example: CelsiusConverter 2

Adding HTML

You can use HTML to specify the text on some Swing components, such as buttons and labels.

To do this we modify the actionPerformed event as follows

```
if (tempFahr <= 32) {

    fahrenheitLabel.setText("<html><font color=blue>" +
    tempFahr + "&#176 Fahrenheit </font></html>");

} else if (tempFahr <= 80) {
    fahrenheitLabel.setText("<html><font color=green>"
    + tempFahr + "&#176 Fahrenheit </font></html>");

} else {

    fahrenheitLabel.setText("<html><font color=red>" +
    tempFahr + "&#176 Fahrenheit </font></html>");

 }
```

Adding Icon

Some Swing components can be decorated with an icon--a fixed size

A Swing icon is an object that adheres to the Icon interface.

Swing provides a particularly useful implementation of the Icon interface: ImageIcon.

`ImageIcon` paints an icon from a GIF or a JPEG image.

Here's the code that adds the arrow graphic to the convertTemp button:

```
ImageIcon icon = new
ImageIcon("images/convert.gif", "Convert
                                  temperature");

      ...

 convertTemp = new JButton(icon);
```

Example: LunarPhrases 1

This program illustrates two features of swing

1) Compound Borders
2) Combo Boxes
3) Loading Multiple Images

Example: LunarPhrases

Compound Border

Both subpanels, selectPanel and displayPanel, have a compound border.

This compound border consists of a titled border (an outlined border with a title) and an empty border (to add extra space),

The code for the selectPanel border follows.

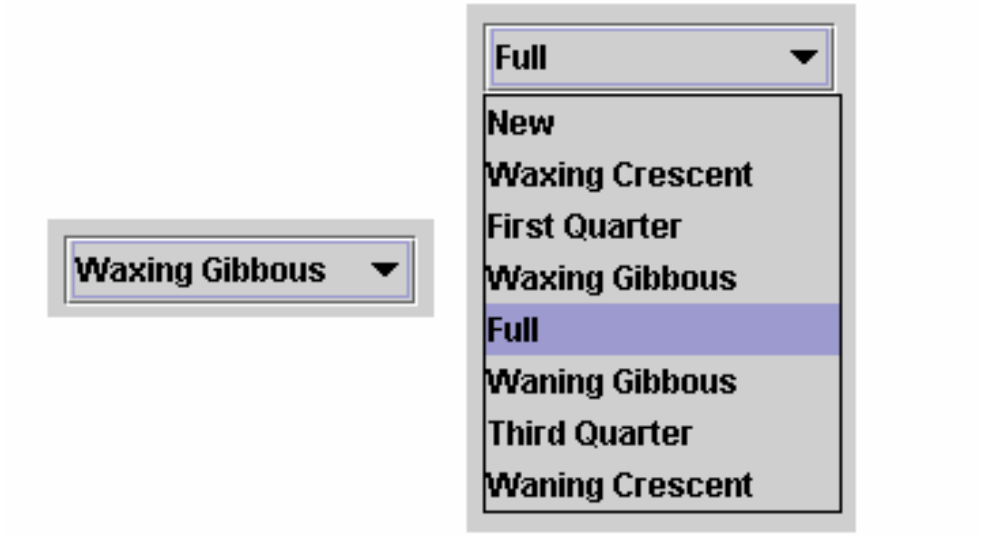
```
selectPanel.setBorder(BorderFactory.createCompoundB
order(BorderFactory.createTitledBorder("Select
Phase"),BorderFactory.createEmptyBorder(5,5,5,5)));
```

Combo Boxes 1

1) A combo box enables user choice.

2) A combo box can be either editable or uneditable, but by default it is uneditable

3) An uneditable combo box looks like a button until the user interacts with it. When the user clicks it, the combo box displays a menu of items.



Combo Boxes 2

1) A combo box enables user choice.
2) A combo box can be either editable or uneditable, but by default it is uneditable
3) An uneditable combo box looks like a button until the user interacts with it. When the user clicks it, the combo box displays a menu of items.

4) When to use uneditable combo box
   a) when space is limited,
   b) when the number of choices is large, or when the menu items are computed at runtime.

Combo Boxes 2

To populate a combo box:

1. initialise with an array of Strings or Vector
2. add the array of Strings or Vector to the constructor

Example:

```
JComboBox phaseChoices = null;
...
// Create combo box with lunar phase choices
String[] phases = { "New", "Waxing Crescent", "First
Quarter","Waxing Gibbous", "Full", "Waning Gibbous",
                "Third Quarter", "Waning Crescent" };
phaseChoices = new JComboBox(phases);
phaseChoices.setSelectedIndex(START_INDEX);
```

Combo Box Event

The combo box fires an action event when the user selects an item from the combo box's menu.

```
phaseChoices.addActionListener(this);
...
public void actionPerformed(ActionEvent event) {
    if
   ("comboBoxChanged".equals(event.getActionCommand
   ())) {
         // update the icon to display the new
  phase
  phaseIconLabel.setIcon(images[phaseChoices.getSe
  lectedIndex()]);
    }
}
```

Multiple Images 1

1) In previous program, we saw how to add a single ImageIcon to a button.

2) LP eight images all loaded at a time.

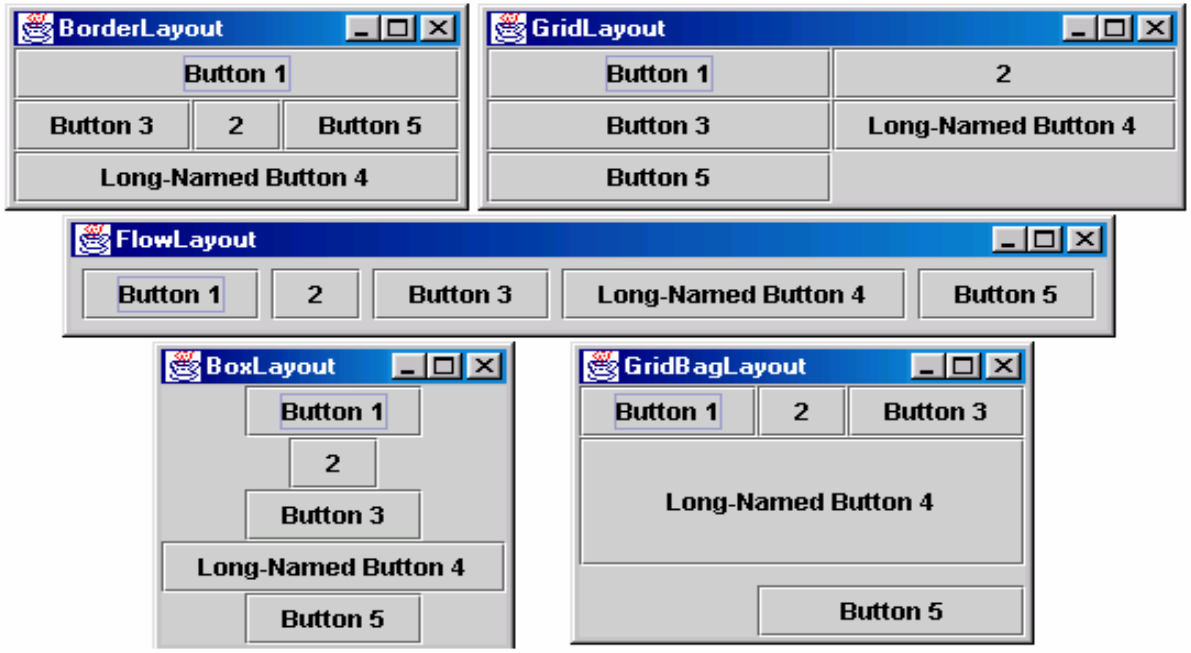
```
final static int NUM_IMAGES = 8;
final static int START_INDEX = 3;

ImageIcon[] images = new ImageIcon[NUM_IMAGES];
...
private void addWidgets() {
    // Get the images and put them into an array of
  ImageIcon.
    for (int i = 0; i < NUM_IMAGES; i++) {
```

Multiple Images 2

```
String imageName = "images/image" + i + ".jpg";
        URL iconURL =
  ClassLoader.getSystemResource(imageName);

        ImageIcon icon = new ImageIcon(iconURL);
        images[i] = icon;
    }
}
```

Note: the use of getSystemResource, a method in ClassLoader that searches the classpath to find the image file names so that we don't have to specify the fully qualified path name.

Layout Management



1) The figure above shows the GUI of five different programs.

2) The buttons are identical and the code are almost identical.
   So why do the GUIs look so different?

   a) Because they use different layout managers to control the size and the position of the buttons.

Using Layout Managers 1

The Java platform supplies five commonly used layout managers:

1) BorderLayout
2) BoxLayout
3) FlowLayout
4) GridBagLayout, and
5) GridLayout

Using Layout Managers 2

1) By default, every container has a layout manager
2) JPanel objects use a FlowLayout by default
3) main containers in JApplet, JDialog, and JFrame objects use BorderLayout by default
4) As a rule, the only time you have to think about layout managers is when you create a JPanel or add components to a content pane.

Using Layout Managers 3

If you don't like the default layout manager that a panel or content pane uses, you can change it to a different one by setting the setLayout method of the content pane.

```
JPanel pane = new JPanel();

pane.setLayout(new BorderLayout());
```

BorderLayout 1

BorderLayout

1. the default layout manager for every content pane.
2. the main container in all frames, applets, and dialogs
3. has five areas available to hold components

   north, south, east, west, and center

Example: BorderLayout 2

How to use BorderLayout

```
...
//Container pane = aFrame.getContentPane()
...
JButton button = new JButton("Button 1
   (PAGE_START)");
pane.add(button, BorderLayout.PAGE_START);

//Make the center component big, since that's the
//typical usage of BorderLayout.
button = new JButton("Button 2 (CENTER)");
button.setPreferredSize(new Dimension(200, 100));
pane.add(button, BorderLayout.CENTER);
```

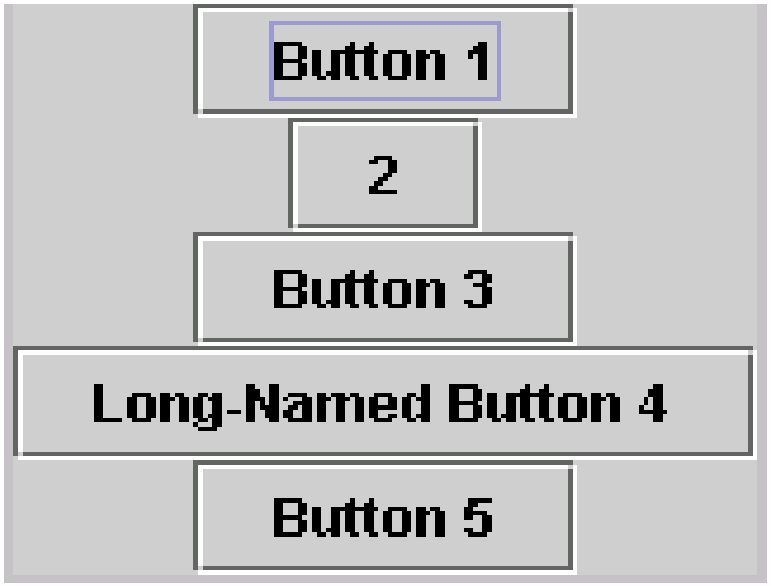
BorderLayout 3

```
button = new JButton("Button 3 (LINE_START)");
pane.add(button, BorderLayout.LINE_START);

button = new JButton("Long-Named Button 4PAGE_END)");
pane.add(button, BorderLayout.PAGE_END);

button = new JButton("5 (LINE_END)");
pane.add(button, BorderLayout.LINE_END);
```

BoxLayout 1

BoxLayout

1) puts components in a single row or column
2) respects the components' requested maximum sizes
3) also lets you align components



Example: BoxLayout 2

```
public static void addComponentsToPane(Container
  pane) {
        pane.setLayout(new BoxLayout(pane,
                                BoxLayout.Y_AXIS));
        addAButton("Button 1", pane);
        addAButton("Button 2", pane);
        addAButton("Button 3", pane);
        addAButton("Long-Named Button 4", pane);
        addAButton("5", pane);
    }
```

FlowLayout

FlowLayout

1) default layout manager for every JPanel
2) lays out components from left to right, starting new rows, if necessary
3) respects the components' requested maximum sizes
4) also lets you align components

Example: FlowLayout

```
contentPane.setLayout(new FlowLayout());
 contentPane.add(new JButton("Button 1"));
 contentPane.add(new JButton("Button 2"));
 contentPane.add(new JButton("Button 3"));
 contentPane.add(new JButton("Long-Named Button 4"));
 contentPane.add(new JButton("5"));
```

FlowLayout

FlowLayout API

The FlowLayout class has three constructors:

```
public FlowLayout()

public FlowLayout(int alignment)

public FlowLayout(int alignment, int
                  horizontalGap, int verticalGap)
```

The alignment argument can be

```
FlowLayout.LEADING,

FlowLayout.CENTER, or

FlowLayout.TRAILING
```

GridLayout

Makes a bunch of components equal in size and displays them in the requested number of rows and columns

Example: GridLayout

```
pane.setLayout(new GridLayout(0,2));

pane.add(new JButton("Button 1"));
pane.add(new JButton("Button 2"));
pane.add(new JButton("Button 3"));
pane.add(new JButton("Long-Named Button 4"));
pane.add(new JButton("5"));
```

GridLayout

The GridLayout class has two constructors:

```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns,
                  int horizontalGap, int verticalGap)
```

1) At least one of the rows and columns arguments must be nonzero
2) the rows argument has precedence over the columns argument
3) The horizontalGap and verticalGap arguments to the second constructor allow you to specify the number of pixels between cells. If you don't specify gaps, their values default to zero.

GrigBagLayout

1) the most sophisticated and flexible layout manager
2) aligns components by placing them within a grid of cells, allowing some components to span more than one cell
3) The rows in the grid aren't necessarily all the same height; similarly, grid columns can have different widths.

Example: GrigBagLayout 1

```
JButton button;
pane.setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
c.fill = GridBagConstraints.HORIZONTAL;

button = new JButton("Button 1");
c.weightx = 0.5;
c.gridx = 0;
c.gridy = 0;
pane.add(button, c);
```

Exanmple: GrigBagLayout 2

```
button = new JButton("Button 2");
c.gridx = 1;
c.gridy = 0;
pane.add(button, c);
button = new JButton("Button 3");
c.gridx = 2;
c.gridy = 0;
pane.add(button, c);
button = new JButton("Long-Named Button 4");
c.ipady = 40;      //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);
```

Exanmple: GrigBagLayout 3

```
button = new JButton("5");
c.ipady = 0;        //reset to default
c.weighty = 1.0;    //request any extra vertical space
c.anchor = GridBagConstraints.PAGE_END; //bottom of
                                        space
c.insets = new Insets(10,0,0,0);  //top padding
c.gridx = 1;        //aligned with button 2
c.gridwidth = 2;    //2 columns wide
c.gridy = 2;        //third row
pane.add(button, c);
```

Overview

1) A Quick Start Guide
2) Swing Features and Concepts
3) Summary

Swing Features and Concepts

1) Components and the Containment Hierarchy
2) Event Handling

Components 1

1) SwingApplication.java creates four commonly used Swing components

   a) a frame, or main window (JFrame)
   b) a panel, sometimes called a pane (JPanel)
   c) a button (JButton)
   d) a label (JLabel)

2) The frame is a top-level container

   a) exists mainly to provide a place for other Swing components to paint themselves
   b) The other commonly used top-level containers are dialogs (JDialog) and applets (JApplet).

Components 2

1) The panel is an intermediate container
   a) Its only purpose is to simplify the positioning of the button and label
   b) Other intermediate Swing containers, such as scroll panes (JScrollPane) and tabbed panes (JTabbedPane), typically play a more visible, interactive role in a program's GUI.

2) The button and label are atomic components
   a) self-sufficient entities that present bits of information to the user
   b) atomic components also get input from the user
      1) Combo boxes (JComboBox)
      2) Text fields (JTextField)
      3) Tables (JTable)

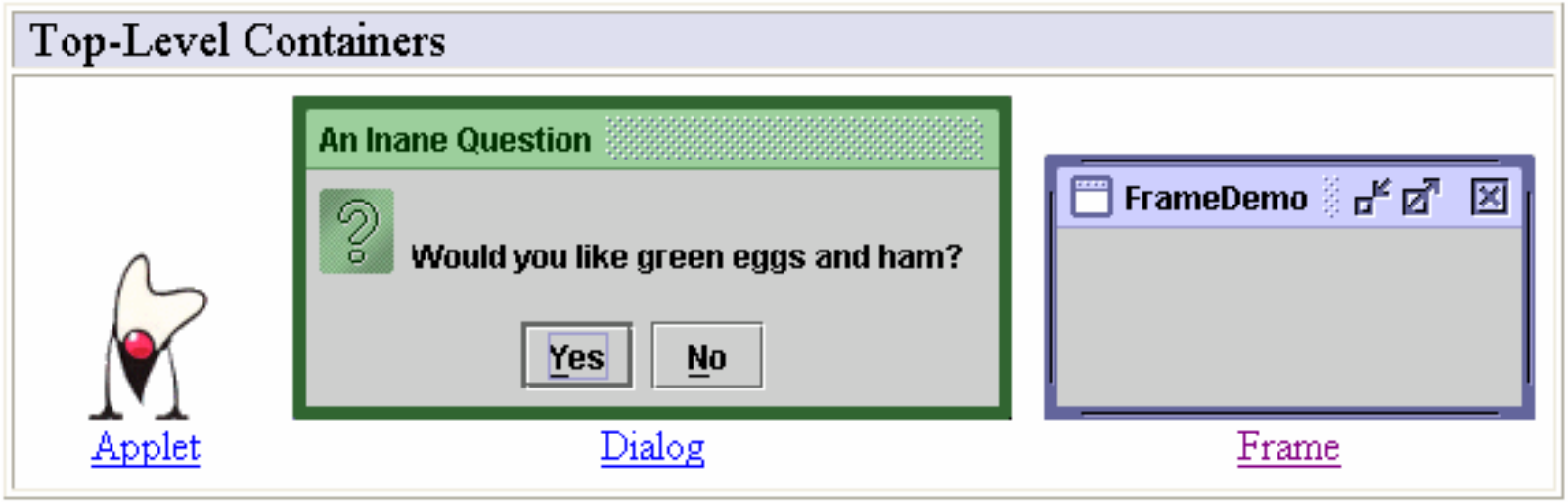
Components 3

Swing Components are divided into the following categories

1) Top-level Containers
2) General Purpose Containers
3) Special Purpose Containers
4) Basic Controls
5) Uneditable Information Display
6) Interactive Display of Highly Formatted Information (IDHFI)

Top-Level Containers

The components at the top of any Swing containment hierarchy

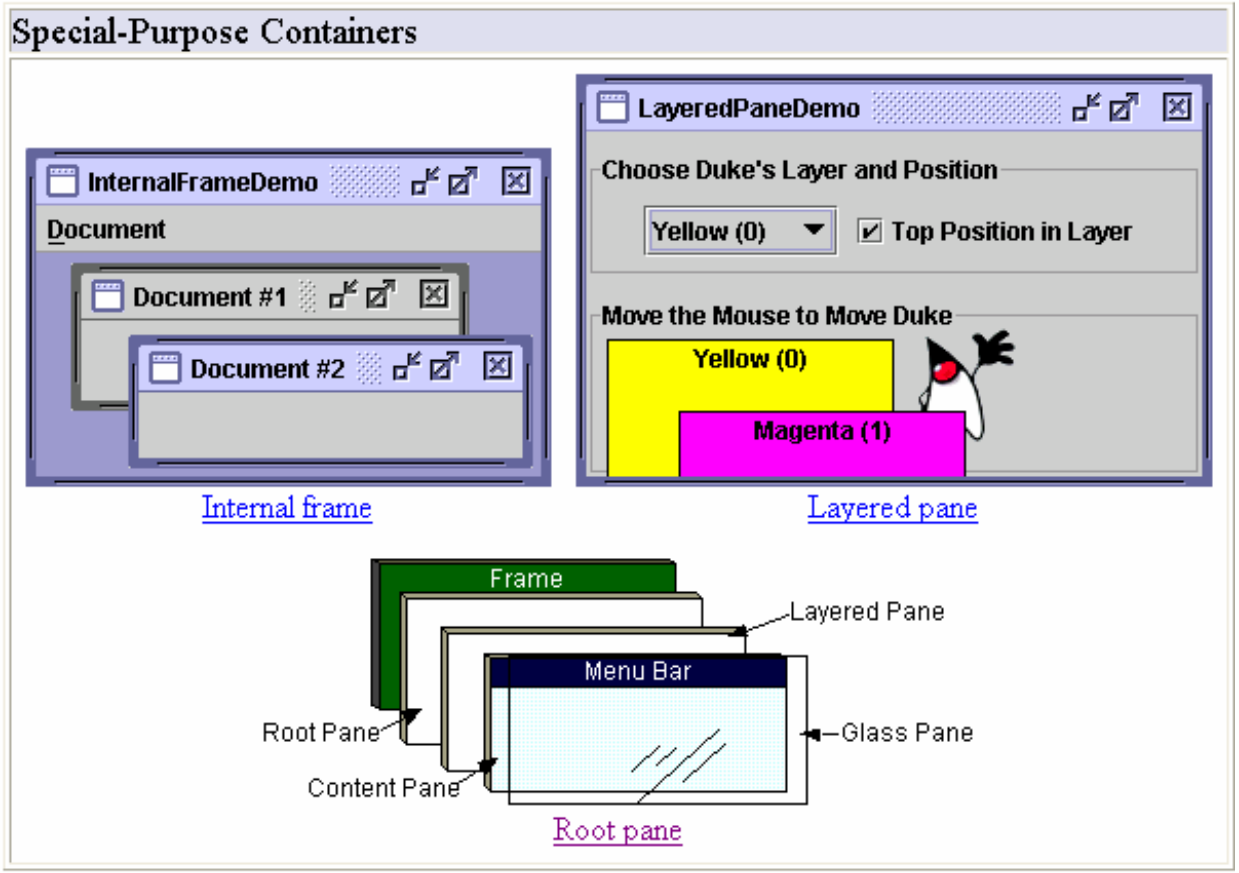


General Purpose Containers

Intermediate containers that can be used under many different circumstances.

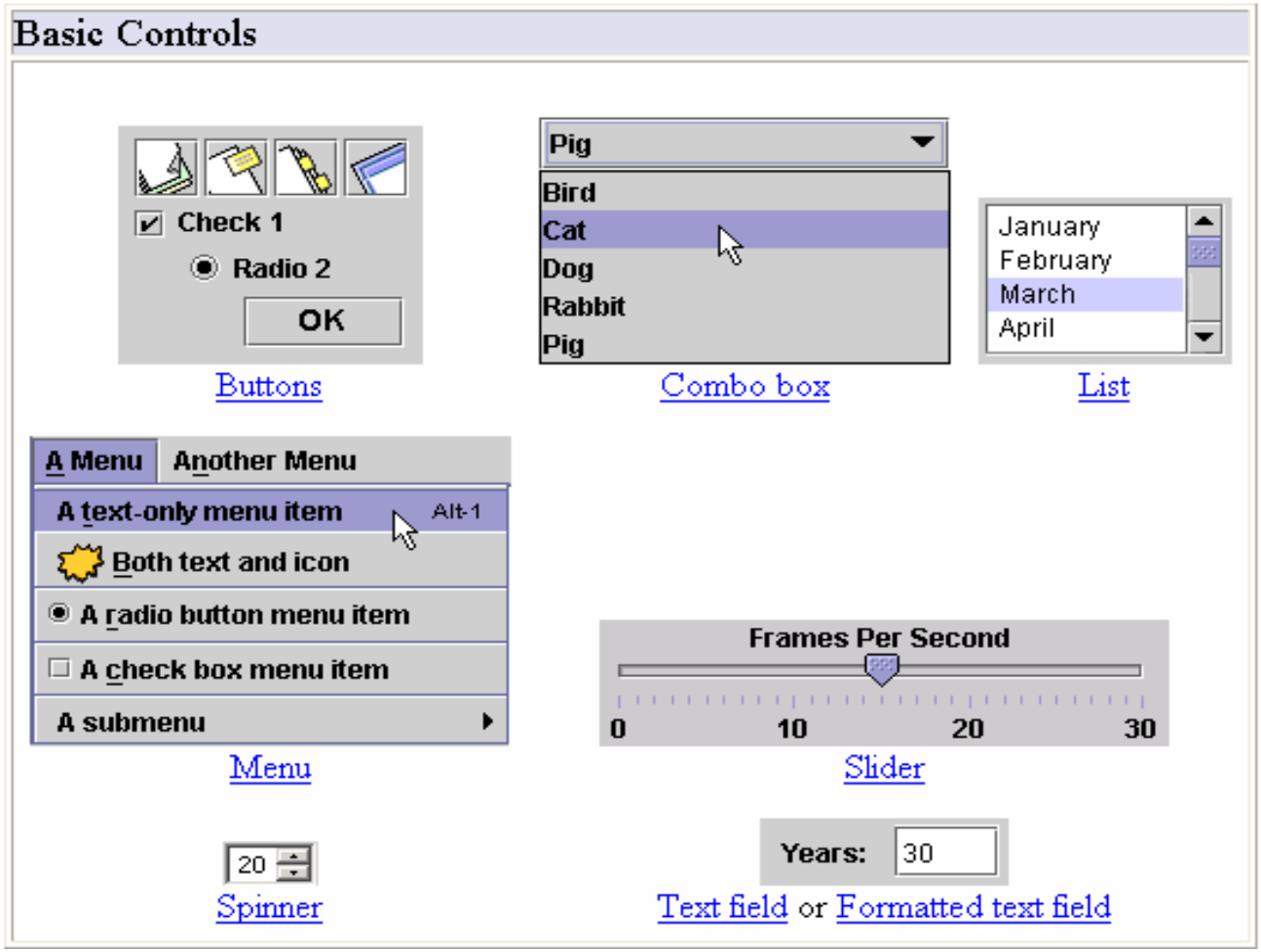
Special Purpose Container

Intermediate containers that play specific roles in the UI.



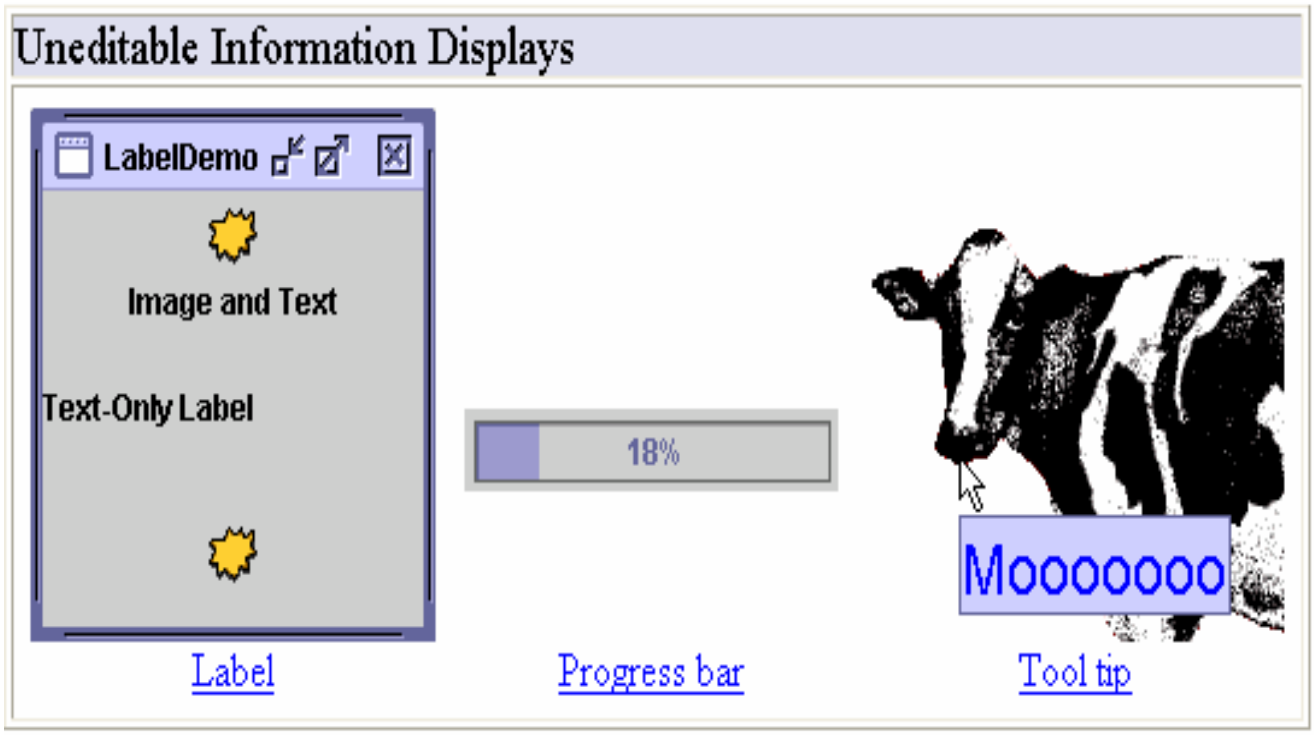
Basic Controls

Atomic components that exist primarily to get input from the user; they generally also show simple state.



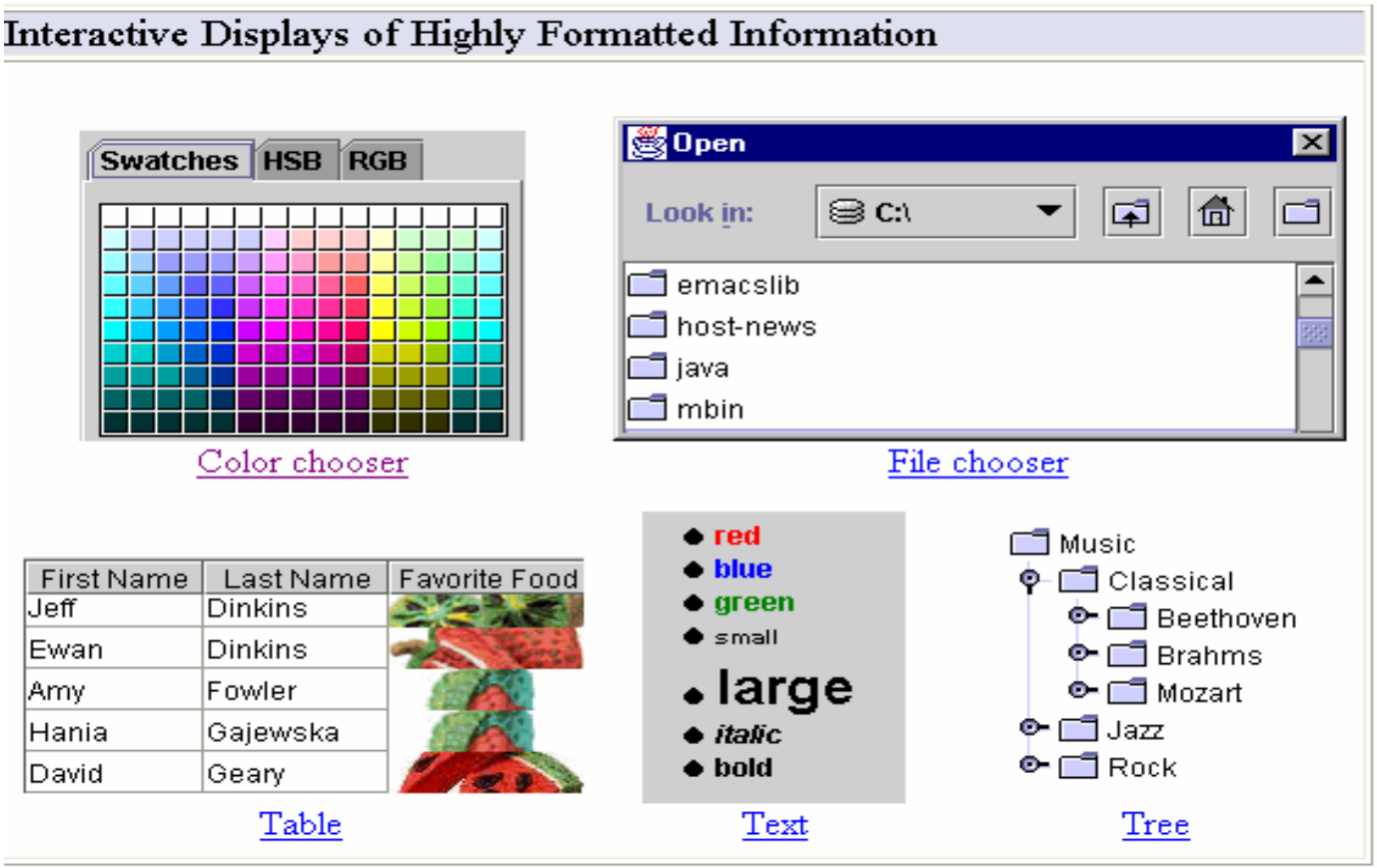
Uneditable Information Display

Atomic components that exist solely to give the user information.



Interactive Displays

Atomic components that display highly formatted information that (if you choose) can be modified by the user.

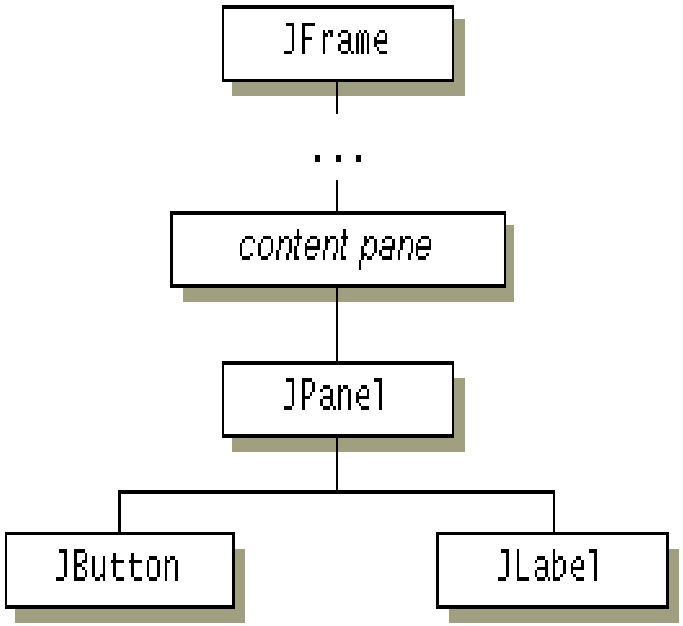


Containment Hierarchy 1

1) The diagram shows the containment hierarchy for the window shown by SwingApplication.

2) The root of the containment hierarchy is always a top-level container.

3) The top-level container provides a place for its descendent Swing components to paint themselves.



Containment Hierarchy 2

1) Every top-level container indirectly contains an intermediate container known as a content pane.

2) the content pane contains, directly or indirectly, all of the visible components in the window's GUI.

3) The big exception to the rule is that if the top-level container has a menu bar, then by convention the menu bar goes in a special place outside of the content pane.

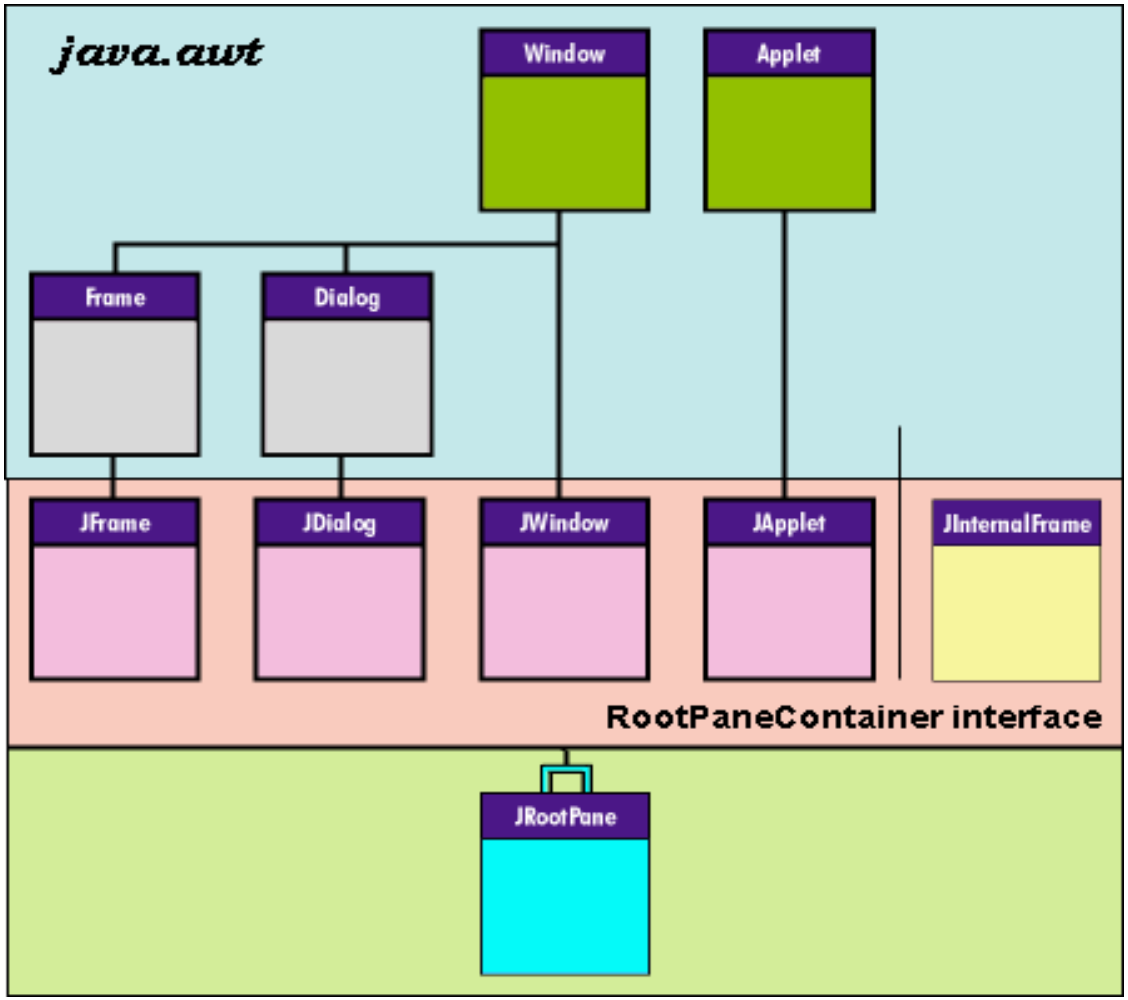
Containment Hierarchy 3

Here is the code that adds the button and label to the panel, and the panel to the content pane

```
frame = new JFrame(...);
button = new JButton(...);
label = new JLabel(...);
pane = new JPanel();
pane.add(button);
pane.add(label);
frame.getContentPane().add(pane, BorderLayout.CENTER);
```

Containment Hierarchy 4

Basic Swing Container Architecture



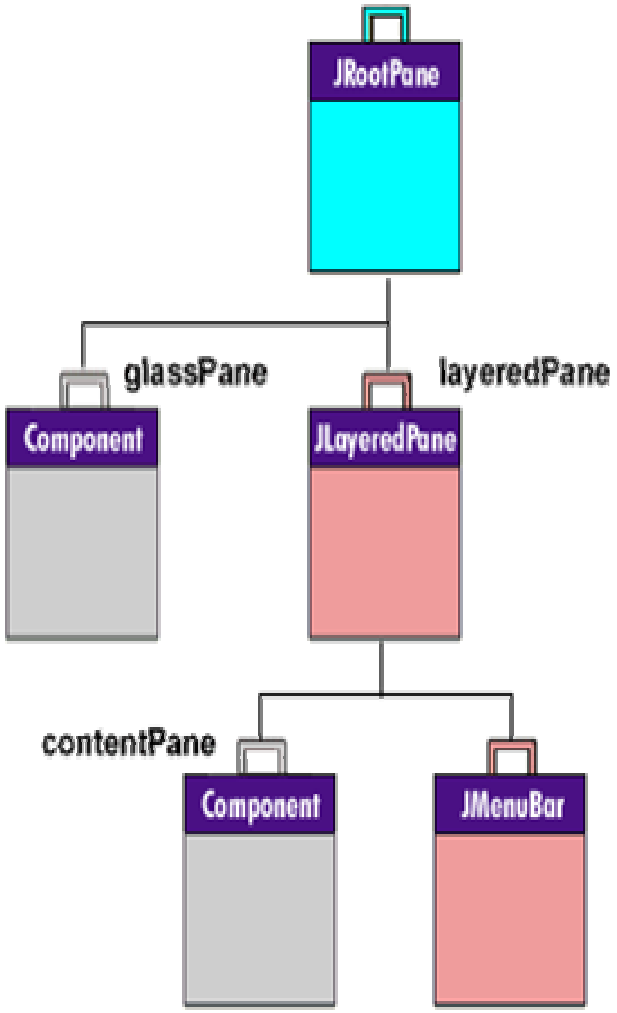
Containment Hierarchy 5

Architectural Underpinnings

1) A JRootPane object is made up of
   a) a glassPane,
   b) a contentPane, and
   c) an optional menu bar

2) the layeredPane, contentPane, and glassPane shown in this diagram always exist

3) What happens when you invoke getContentPane() method of a top-level container?



Containment Hierarchy 6

1) Take for example a JFrame.

2) JFrame delegates the operation to it's container. But how?

3) Consider the inheritance and implementation hierarchy

Containment Hierarchy 7

public interface RootPaneContainer

Method Summary	
Container	**getContentPane**() Returns the contentPane.
Component	**getGlassPane**() Returns the glassPane.
JLayeredPane	**getLayeredPane**() Returns the layeredPane.
JRootPane	**getRootPane**() Return this component's single JRootPane child.
void	**setContentPane**(Container contentPane) The "contentPane" is the primary container for application specific components.
void	**setGlassPane**(Component glassPane) The glassPane is always the first child of the rootPane and the rootPanes layout manager ensures always as big as the rootPane.
void	**setLayeredPane**(JLayeredPane layeredPane) A Container that manages the contentPane and in some cases a menu bar.

Containment Hierarchy 8

Implementation of getContentPane method

```
public Container getContentPane(){
   return getRootPane().getContentPane();
 }
```

JRootPane objects, in turn, delegates its getContentPane() methods to their JLayeredPane instances.

Applet

Course Outline

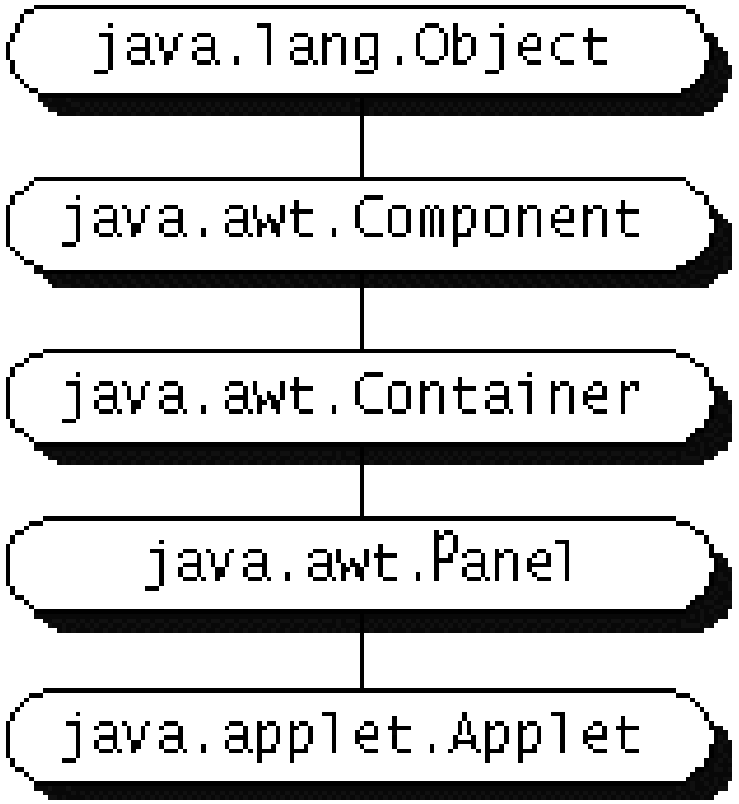
1) introduction
2) language
   a) syntax
   b) types
   c) variables
   d) arrays
   e) operators
   f) control flow
3) object-orientation
   a) objects
   b) classes
   c) inheritance
   d) polymorphism
   e) access
   f) interfaces
   g) exception handling
   h) multi-threading
4) horizontal libraries
   a) string handling
   b) event handling
   c) object collections
5) vertical libraries
   a) graphical interface
   b) applets
   c) input/output
   d) networking
6) summary

What is an Applet?

1) Program embedded in another application, generally a Web browser that provides a JVM.
   a) An applet's host program provides an *applet context* in which the applet executes.
   b) Launched from an HTML document with an `APPLET` tag that specifies the URL for the applet
2) A class that extends Applet or a descendant of the Applet class
   a) javax.swing.JApplet extends Applet

The Hierarchy

1) An applet is an object whose class descends ultimately from `Applet` or `JApplet`.
2) An applet is an embeddable `Panel`, which is a simple `Container` window.
3) An applet's class **must be public**.
4) An applet typically overrides the inherited `init`, `start`, `paint`, `stop`, and `destroy` methods.



What does an applet do?

1) An applet can react to major events in the following ways:
   a) It can *initialize* itself
      - init() method called when an applet is first loaded
   b) It can *start* running
      - start() method called when browser receives focus
   c) It can *stop* running.
      - stop() method called when the browser minimized or user leaves page
   d) It can perform a *final cleanup*, in preparation for being unloaded.
      - destroy() method called when browser closed

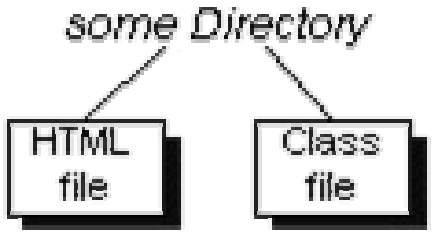
What to override?

1) *If you don't override one of the following methods, your applet will not do anything!*
   a) init()
      - Called once. Place "constructor code" here.
   b) start()
      - Usually performs applet's work. Called many times.
   c) paint(Graphics g)
      - Can override to simply draw oneself
   d) stop()
      - Should stop execution of applet (ie: pause a timer thread)
   e) destroy()
      - Performs cleanup of resources (ie: release DB connection)

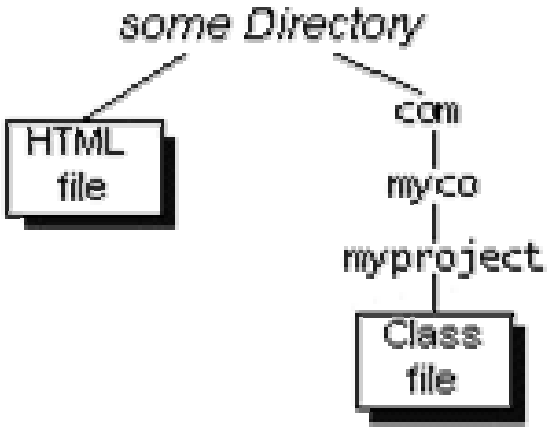
Where to put stuff

1) Where the applet class file must be, relative to the HTML document that contains the `<APPLET>` tag.

No Package Statements in Applet Code



`package com.myco.myproject;`



Applet Lifecycle 1

1) When a Java-enabled browser encounters an `<APPLET>` tag…
   a) it reserves a display area of the specified width and height for the applet
   b) loads the bytecode for the specified Applet subclass,
   c) creates an instance of the subclass
   d) invokes the `init` method to enable once-only initialization (e.g., setting colors, fonts, and the like)

Applet Lifecycle 2

a) invokes the `start` method.

- If the applet is multithreaded, other threads can be constructed and started in this method.
- This results in the paint method being called If the page that launched the applet is exited, the browser typically invokes the `stop` method, which then can perform appropriate cleanup operations.

b) If the launching page is entered again, the browser again invokes `start`.

c) `destroy` method

- Ends the lifecycle when the browser window is closed

JApplet Methods

Table 13-3 Some Members of the class JApplet (package javax.swing)

```
public void init()
  //Called by the browser or applet viewer to inform this applet
  //that it has been loaded into the system.
```

```
public void start()
  //Called by the browser or applet viewer to inform this applet
  //that it should start its execution. It is called after the init
  //method and each time the applet is revisited in a Web page.
```

```
public void stop()
  //Called by the browser or applet viewer to inform this applet
  //that it should stop its execution. It is called before the
  //method destroy.
```

```
public void destroy()
  //Called by the browser or applet viewer. Informs this applet that
  //it is being reclaimed and that it should destroy any resources
  //that it has allocated. The method stop is called before destroy.
```

```
public void showStatus(String msg)
  //Displays the string msg in the status bar.
```

```
public Container getContentPane()
  //Returns the ContentPane object for this applet.
```

JApplet Methods

Table 13-3 Some Members of the `class` `JApplet` (`package` `javax.swing`) (continued)

```
public JMenuBar getJMenuBar()
  //Returns the JMenuBar object for this applet.
```

```
public URL getDocumentBase()
  //Returns the URL of the document that contains this applet.
```

```
public URL getCodeBase()
  //Returns the URL of this applet.
```

```
public void update(Graphics g)
  //Calls the paint() method.
```

```
protected String paramString()
  //Returns a string representation of this JApplet; mainly used
  //for debugging.
```

Applet Methods to Override

1) No main method
2) Methods init, start, and paint guaranteed to be invoked in sequence
3) To develop an applet
   a) Override any/all of the methods above
   b) Override stop and destroy to stop threads or free resources (eg: close database connections)

Init and Paint Methods

1) init Method
   a) Initializes variables
   b) Gets data from user
   c) Places various GUI components
2) paint Method
   a) Performs output

Skeleton of a Java Applet

```
import java.awt.Graphics;
import javax.swing.JApplet;

public class WelcomeApplet extends JApplet
{

}
```

Displaying Welcome Message

```
//Welcome Applet

import java.awt.Graphics;
import javax.swing.JApplet;

public class WelcomeApplet extends JApplet {

  public void paint(Graphics g) {
    super.paint(g); // <-- make sure you always call this!
    g.drawString("Welcome to Java Programming",30, 30);
  }
}
```

HTML to Run Applet

```
<html>
   <head>
       <title>This title shows at the top of the browser
   window</title>
   </head>
   <body>
       <applet code="WelcomeApplet" height="100" width="400"/>
   </body>
</html>
```

class Font

1) Shows text in different fonts
2) Contained in package java.awt
3) Available fonts
   a) Serif/SanSerif
   b) Monospaced
   c) Dialog/DialogInput
4) Arguments for constructor
   a) String specifying the Font face name
   b) int value specifying Font style
   c) int value specifying Font size
      - Expressed in points (72 points = 1 inch)

Methods of the class Font

Table 13-4 Some Constructors and Methods of the `class` Font

Method
`public Font(String name, int style, int size)` `  //Constructor` `  //Creates a new Font from the specified name, style, and point` `  //size.`
`public String getFamily()` `  //Returns the family name of this Font.`
`public String getFontName()` `  //Returns the font face name of this Font.`

class Color

1) Shows text in different colors
2) Changes background color of component
3) Contained in package java.awt

Constructors of the class Color

Table 13-5 Some Constructors and Methods of the `class` Color

```
Color(int r, int g, int b)
  //Constructor
  //Creates an instance of Color with red value r, green value g,
  //and blue value b. In this case, r, g, and b can be
  //between 0 and 255.
  //Example: new  Color(0,255,0)
  //      creates a color with no red or blue component.
```

```
Color(int rgb)
  //Constructor
  //Creates an instance of Color with red value r, green
  //value g, and blue value b, choose rgb as
  //r * 65536 + g * 256 + b. In this case, r, g, and b
  //can be between 0 and 255.
  //Example: new  Color(255)
  //      creates a color with no red or green component.
```

Methods of the class Color

```
Color(float r, float g, float b)
  //Constructor
  //Creates an instance of Color with red value r, green value g,
  //and blue value b. In this case, r, g, and b can be between 0
  //and 1.0.
  //Example: new  Color(1.0,0,0)
  //      creates a color with no green or blue component.
```

```
public Color brighter()
  //Returns a Color that is brighter.
```

```
public Color darker()
  //Returns a Color that is darker.
```

```
public boolean equals(Object o)
  //Returns true if the color of this object is the same as the
  //color of the object o; false otherwise.
```

Methods of the class Color

```
public int getBlue()
  //Returns the value of the blue component.
```

```
public int getGreen()
  //Returns the value of the green component.
```

```
public int getRGB()
  //Returns the RGB value. RGB value is  r * 65536 + g * 256 + b.
```

```
public int getRed()
  //Returns the value of the red component.
```

```
public String toString()
  //Returns a String with information about the color.
```

Color Constants

Table 13-6 Constants Defined in the `class` `Color`

`Color.black: (0,0,0)`	`Color.magenta: (255,0,255)`
`Color.blue: (0,0,255)`	`Color.orange: (255,200,0)`
`Color.cyan: (0,255,255)`	`Color.pink: (255,175,175)`
`Color.darkGray: (64,64,64)`	`Color.red: (255,0,0)`
`Color.gray: (128,128,128)`	`Color.white: (255,255,255)`
`Color.green: (0,255,0)`	`Color.yellow: (255,255,0)`
`Color.lightGray: (192,192,192)`	

class Graphics

1) Provides methods for drawing items such as lines, ovals, and rectangles on the screen
2) Contains methods to set the properties of graphic elements including clipping area, fonts, and colors
3) Contained in the package java.awt

Graphics class

```
protected Graphics()
  //Constructs a Graphics object that defines a context in which the
  //user can draw. This constructor cannot be called directly.
```

```
public void draw3Rect(int x, int y, int w, int h, boolean t)
  //Draws a 3D rectangle at (x, y) with width w, height h. If t is
  //true, rectangle will appear raised.
```

```
public abstract void drawArc(int x, int y, int w, int h,
                             int sangle, int aangle)
  //Draws an arc in the rectangle at position (x, y) of width w and
  //height h. The arc starts at angle sangle with an arc angle aangle.
  //Both angles are measured in degrees.
```

```
public abstract boolean drawImage(Image img, int xs1, int ys1,
                      int xs2, int ys2, int xd1, int yd1,
                      int xd2, int yd2, Color c, ImageObserver ob)
  //Draws the image specified by img from the area defined by
  //bounding rectangle, (xs1, ys1) to (xs2, ys2), in the area defined
  //by the rectangle (xd1, yd1) to (xd2, yd2). Any transparent color
  //pixels are drawn in color c. The ob monitors the progress of
  //the image.
```

Graphics class

```
public abstract void drawLine(int xs, int ys, int xd, int yd)
  //Draws a line from (xs, ys) to (xd, yd).

public abstract void drawOval(int x, int y, int w, int h)
  //Draws an oval at position (x, y) of width w and height h.

public abstract void drawPolygon(int[] x, int[] y, int num)
  //Draws a polygon with points (x[0], y[0]), ...,
  //(x[num-1], y[num-1]). Here num is the number of points in
  //the polygon.

public abstract void drawPolygon(Polygon poly)
  //Draws a polygon as defined by the object poly.

public abstract void drawRect(int x, int y, int w, int h)
  //Draws a rectangle at position (x, y) having a width w and
  //height h.

public abstract void drawRoundRect(int x, int y, int w, int h,
                                   int arcw, int arch)
  //Draws a round-cornered rectangle at position (x, y) having a
  //width w and height h. The shape of the rounded corners is
  //determined by arc with width arcw and height arch.
```

Graphics class

```
public abstract void drawString(String s, int x, int y)
  //Draws the string s at (x, y).

public void fill3Rect(int x, int y, int w, int h, boolean t)
  //Draws a 3D filled rectangle at (x, y) with width w, height h.
  //If t is true, rectangle will appear raised. The rectangle is
  //filled with current color.

public abstract void fillArc(int x, int y, int w, int h,
                             int sangle, int aangle)
  //Draws a filled arc in the rectangle at position (x, y) of width
  //w and height h starting at angle sangle with an arc
  //angle aangle. Both angles are measured in degrees. The arc is
  //filled with current color.

public abstract void fillOval(int x, int y, int w, int h)
  //Draws a filled oval at position (x, y) having a width w and
  //height h. The oval is filled with current color.

public abstract void fillPolygon(int[] x, int[] y, int num)
  //Draws a filled polygon with points (x[0], y[0]), ...,
  //(x[num-1], y[num-1]). Here num is the number of points in
  //the polygon. The polygon is filled with current color.

public abstract void fillPolygon(Polygon poly)
  //Draws a filled polygon as defined by the object poly. The polygon
  //is filled with current color.

public abstract void fillRect(int x, int y, int w, int h)
  //Draws a filled rectangle at position (x, y) having a width w
  //and height h. The rectangle is filled with current color.
```

Graphics class

```
public abstract void fillRoundRect(int x, int y, int w, int h,
                                   int arcw, int arch)
  //Draws a filled, round-cornered rectangle at position (x, y)
  //having a width w and height h. The shape of the rounded corners
  //is determined by the arc with width arcw and height arch. The
  //rectangle is filled with current color.

public abstract Color getColor()
  //Returns the current color for this graphics context.

public abstract void setColor(Color c)
  //Sets the current color for this graphics context to c.

public abstract Font getFont()
  //Returns the current font for this graphics context.

public abstract void setFont(Font f)
  //Sets the current font for this graphics context to f.

public FontMetrics getFontMetrics()
  //Returns the font metrics associated with this graphics context.

public FontMetrics getFontMetrics(Font f)
  //Returns the font metrics associated with Font f.

public void String toString()
  //Returns a string representation of this graphics context.
```

GUI Components and Applets

1) Applets inherit from Panel
2) Can add UI components
3) Call setLayout(LayoutManager lm) to change to a different layout
    a) Default is BorderLayout
4) Place all adding of UI components in init() method
    a) No longer need to override paint() as UI components will automatically update themselves
    b) Call validate() when done adding components

GUI Application vs. Applet

1) Applications extend JFrame
2) GUI components setup in constructor
3) Things start in main method
4) Typically call `setTitle, setSize, pack, setVisible, setDefaultCloseOperation`

1) Applets extend JApplet
2) GUI components setup in init() method
3) No main method, browser creates instance automatically
4) Not called in applet, as it doesn't have it's own window

System.out.print and Applets

1) When you use System.out to print text, it does not show up in an applet
2) Treated as debugging information
3) Most browsers have a way to view the output
    a) IE has a Java console under “Tools” → “Sun Java Console”

Applets and Security

1) Tight security called *sandbox security*
2) Every browser implements security policies to keep applets from compromising system security.
3) The implementation of the security policies differs from browser to browser.
4) Security policies are subject to change.
    a) For example, if a browser is developed for use only in trusted environments, then its security policies will likely be much more lax than those described here.

Sandbox Security?

1) In general, applets loaded over the net are prevented from:
   a) Reading and writing files on the client file system
   b) Making network connections
      - Except to the originating host
   c) Starting other programs on the client
   d) Loading libraries
   e) Define native method calls
      - If an applet could define native method calls, that would give the applet direct access to the underlying computer.

The SecurityManager

1) Each browser has a `SecurityManager` object that implements its security policies.
2) When a `SecurityManager` detects a violation, it throws a `SecurityException`.
3) Your applet can catch this `SecurityException` and react appropriately.

What can applets do?

1) Make network connections to the host they came from.
2) Cause HTML documents to be displayed.
3) Invoke public methods of other applets on the same page.
4) (if loaded from the local file system, i.e. from a directory in the user's CLASSPATH), and Applet does not have the restrictions that applets loaded over the network do
5) Play audio clips

Distributed Programming Assessment

Set 1

1	What will be displayed if you try to run the following code and there is no file called Hello.txt in the current directory? `import java.io.*;` `public class Mine {` `  public static void main(String argv[]){` `  Mine m=new Mine();` `  System.out.println(m.amethod());` `  }` `  public int amethod() {` `  try {` `  FileInputStream dis=new FileInputStream("Hello.txt");` `  }catch (FileNotFoundException fne) {` `  System.out.println("No such file found");` `  return -1;` `  }catch(IOException ioe) {` `  }` `  return 0;` `  }` `}`		tick
	a	No such file found	
	b	No such file found ,-1	X
	c	No such file found, Doing finally, -1	
	d	0	

2	Which of the following statements can be used to change the current working directory to a new directory called “DirName” using an instance of the File class called “FileName”?		tick
	a	FileName.chdir("DirName")	
	b	FileName.cd("DirName")	
	c	FileName.cwd("DirName")	
	d	The File class does not support directly changing the current directory.	X

3	Which of the following operations cannot be performed using the File class?		tick
	a	Change the current directory	X
	b	Return the name of the parent directory.	
	c	Delete a file	
	d	Find if a file contains text or binary information.	X

4	Which of the following is not a benefit of distributed programming?		tick
	a	Separation of concern	
	b	Ease of maintenance	
	C	Balance resource loading	
	d	Object serialization	X

5	Which of the following classes is not part of the java.io package?		tick
	a	File	
	b	PushInpuStream	X
	c	InputStream	
	d	RandomAccessFile	

6	Which of the following would read in characters encoded in BIG-5 from a file?		tick
	a	`BufferedReader br = new BufferedReader(new FileReader("Fileaname"));`	
	b	`BufferedInputStream bis = new BufferedInputStream(new FileReader(new InputStreamReader("filename")));`	
	c	`BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream("filename")));`	x
	d	`FileReader fr = new FileReader("fielname");`	

7	What will happen when you try to compile and run the following code?		tick
	`` import java.io.*; public class URLConnectionReader { public static void main(String[] args) throws Exception { URL yahoo = new URL("http://www.yahoo.com/"); URLConnection yc = yahoo.openConnection(); BufferedReader in = new BufferedReader(new InputStreamReader (yc.getInputStream())); String inputLine while ((inputLine = in.readLine()) != null) System.out.println(inputLine); in.close(); } } ```		
	a	Retrieves URL data from the specified location	
	b	Read and does not display the content of the stream	
	c	Compile time error occurs	x
	d	Prints www.yahoo.com	

8	Which of the following statements are required for database connection?		tick
	a	`DriverManager.registerDriver(Driver driver);`	x
	b	`class.forname(Driver driver).newIntsance();`	x
	c	`DriverManager.getConnection(String url, String ui, String pwd);`	X
	d	`Statement stm = con.createStatement();`	

9	Creating an RMI system involves the following actions except		tick
	a	Implementing java.rmi.Remote interface	
	b	Hosting the RMI service in a server process	
	c	Extending java.rmi.server.UnicastRemoteObject	
	d	Generating skeletons for the client using rmic	x

10	Which of the following is a function of an Object Request Broker?		tick
	a	Implements IDL module on the server	
	b	Mediates between two CORBA objects	x
	c	Prevents function calls across the network to the target object	
	d	Acts as a proxy for the client CORBA object	

11	What will happen when you try to compile and run the following code?		tick

```
import java.io.*;
public class Extension implements FilenameFilter {
  String ext;
  public OnlyExt(String ext){
    this.ext=ext;
  }
  public Boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}

import java.io.*;
class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new Extension("class");
    String s[] = f1.list(only);
    for (int i=0; i < s.length; i++){
        System.out.println(s[i]);
    }
  }
}
```

	a	Prints out the list of classes with .class extension	
	b	Compile time error occurs	x
	c	Prints out the list of classes	
	d	Exception is thrown: `IllegalArgumentException`	

12	The following methods are defined by the `URL` class except		tick
	a	`boolean sameFile(URL other)`	
	B	`Int getPort()`	
	c	`String Userinfo()`	x
	d	`String getFile()`	

13	Which of the following APIs need to be implemented by a developer?		tick
	a	`java.sql.Driver`	
	b	`java.sql.Connection`	
	c	`com.oracle.jdbc.OracleResultSet`	
	d	none	x

14	Which of these message types are not supported by JMS API?		tick
	a	`TextMessage`	
	b	`XMLMessage`	x
	C	`HashMapMessage`	x
	d	`ByteMessage`	

15	Which of the following statements are appropriate for deleting a message from an INBOX folder?		tick
	a	`message.setFlag(Flags.Flag.DELETED, true);` `folder.open();`	
	b	`message.setFlag(Flags.Flag.DELETED, true);` `folder.open(Folder.READ_WRITE);`	x
	c	`message.setFlag(Flags.Flag.DELETED, true);` `store.open(Folder.READ_WRITE);`	
	d	`message.setFlag(Flags.Flag.DELETED, true);` `folder.open(Folder.DELETED);`	

Set 2

1	What will happen when you try to compile and run the following code?		tick
	`` import java.io.*; public class URLConnectionReader { public static void main(String[] args) throws Exception { URL yahoo = new URL("http://www.yahoo.com/"); URLConnection yc = yahoo.openConnection(); BufferedReader in = new BufferedReader(new InputStreamReader(yc.getInputStream())); String inputLine; while ((inputLine = in.readLine()) != null) System.out.println(inputLine); in.close(); } } ```		
	a	It retrieves URL data from the specified location	x
	b	It reads but does not display the content of the stream	
	c	Compile time error occurs	
	d	It prints www.yahoo.com	

2	Which of the following operations can be performed using the File class?		tick
	a	Changing the current directory	x
	b	Returning the name of the parent directory.	
	c	Deleting a file	
	d	Checking if a file contains text or binary information.	

3	Which of the following statements are not required for database connection?		tick
	a	`DriverManager.registerDriver(Driver driver);`	
	b	`Class.forname(Driver driver).newIntsance();`	
	c	`DriverManager.getConnection(String url, String ui, String pwd);`	
	d	`Statement stm = con.createStatement();`	x

4	Creating an RMI system involves the following actions except		tick
	a	Implementing java.rmi.Remote interface	x
	b	Hosting the RMI service in a server process	x
	c	Extending java.rmi.server.UnicastRemoteObject	x
	d	Generating skeletons for the client using rmic	

5	Which of the following is a function of an Object Request Broker?		tick
	a	Implements IDL module on the server	
	b	Mediates between two CORBA objects	x
	c	Prevents function calls across the network to the target object	
	d	Acts as a proxy for the client CORBA object	

6	Which of the following APIs need to be implemented by a developer?		tick
	a	`java.sql.Driver`	
	b	`java.sql.Connection`	
	c	`com.oracle.jdbc.OracleResultSet`	x
	d	none	

7	What will be displayed if you try to run the following code and there is no file called Hello.txt in the current directory?		tick

```
import java.io.*;
public class Mine {
    public static void main(String argv[]){
        Mine m=new Mine();
        System.out.println(m.amethod());
    }
    public int amethod() {
        try {
           FileInputStream dis=new
                            FileInputStream("Hello.txt");
        }catch (FileNotFoundException fne) {
            System.out.println("No such file found");
            return -1;
        }catch(IOException ioe) {
        } finally{
            System.out.println("Doing finally");
        }
        return 0;
    }
}
```

	a	“No such file found” is displayed	
	b	“No such file found ,-1” is displayed	x
	c	“No such file found, Doing finally, -1” is displayed	
	d	0 is displayed	

8	Which of the following classes encodes chars for output?		tick
	a	java.io.OutputStream	
	B	java.io.OutputStreamWriter	x
	c	Java.io.BufferedOutputStream	
	d	Java.io.EncodeWriter	

9	Which of these message types are not supported by JMS API?		tick
	a	TextMessage	
	b	XMLMessage	x
	c	HashMapMessage	x
	d	ByteMessage	

10	Which of the following classes are not part of the java.io package?		tick
	a	File	
	b	PushbackInpuStream	x
	c	Inputstream	
	d	RandomAccessFile	

11	What will happen when you try to compile and run the following code?		tick

```
import java.io.*;
public class Extension implements FilenameFilter {
  String ext;
  public OnlyExt(String ext){
    this.ext=ext;
  }
  public Boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}

import java.io.*;
class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new Extension("class");
    String s[] = f1.list(only);
    for (int i=0; i < s.length; i++){
        System.out.println(s[i]);
    }
  }
}
```

	a	The list of classes with .class extension is displayed	
	b	Compile time error occurs	x
	c	The list of classes is displayed	
	d	Exception is thrown: IllegalArgumentException	

12	Which of the following statements can be used to change the current working directory to a new directory called “DirName” using an instance of the File class called “FileName”?		tick
	a	`FileName.chdir("DirName")`	
	b	`FileName.cd("DirName")`	
	c	`FileName.cwd("DirName")`	
	d	The File class does not support directly changing the current directory.	x

13	Which of the following is not a benefit of distributed programming?		tick
	a	Separation of concerns	
	b	Persistence	x
	c	Resource Load balancing	
	d	Ease of Maintenance	

14	The following methods are defined by the `URL` class except		tick
	a	`boolean sameFile(URL other)`	x
	b	`Int getPort()`	
	c	`String Userinfo()`	
	d	`String getFile()`	x

15	Which of the following statements must be executed before you can read a message from an INBOX?		tick
	a	`Transport.send(message);`	
	b	`Session session = Session.getInstance(props, null);` `MimeMessage message = new MimeMessage(session);`	
	c	`Store store = session.getStore("imap");`	
	d	`folder.open(Folder.READ_ONLY);`	x

Distributed Programming with Java

Training Course

Gabriel Oteniya
Chau Keng Fong

e-Macao Report 21

Version 1.0, November 2005

Table of Contents

1. Overview

The history of computing began with the development of programs that ran within a single computer. Users time-shared the central processing unit (CPU) resource of a mainframe system and accessed their "address space" from a terminal. As computers became more powerful, smaller, and capable of executing a greater number of programs, the computer moved from the central location to the desktop. People soon realized the advantage of linking distributed computers to a network and sharing common resources (applications) on the network using servers. With this approach, a client computer downloads or accesses an application from a server computer and runs the application entirely in its address space. Overall, this scenario greatly improves the performance of the application, as the client machine is the determining factor in the application's speed.

This document presents a course on how to create distributed applications using Java Technology. It begins by an introduction to distributed programming environments. Afterwards, it explains the classes in the `java.io` package used for processing streams of data. Networking classes located in the `java.net` package, database programming and various distributed architectures (distributed objects, messaging, etc.) are discussed next.

The rest of this document explains the objectives, prerequisites and methodology for teaching the course in Sections 2, 3 and 4 respectively. The content of the course is introduced in Section 5, followed by assessment and organization in Sections 6 and 7 respectively. Following references, Appendix A includes the complete set of slides and Appendix B contains two sets of assessment questions with answers.

2. Objectives

The course has three main objectives:

1) To teach the fundamental concepts of distributed programming, with particular emphasis on enterprise application development.
2) To introduce various distributed programming architectures and how to apply them.
3) To lean the importance of distributed computing and outline the factors to address when designing a distributed system.

3. Prerequisites

The course assumes that:

1) The students have some knowledge of Object-Oriented Analysis and Design.
2) The students have the ability to develop desktop applications using Java.
3) The students have the basic understanding of the networking concepts and TCP/IP.

4. Methodology

The course has been designed relying on some general didactic principles:

1) Concepts are first defined and illustrated with some concrete examples, before they are used or applied in any context.
2) Visual illustrations are used as much as possible.
3) Assessment is carried out at the end of every section.
4) Most exercises require hands-on participation.
5) The overall assessment is administered at the end of the course to test the general understanding of the material presented.

Generally, an instructor is expected to administer the course in a tutorial style.

5. Content

The course is divided into five main sections: Introduction, Streams, Networking, Database Connectivity and Architectures. They are described as follows.

5.1. Introduction

This section, comprising the slides 14 through 54 provides an overview of distributed programming. It starts by reviewing the basic terms and concepts in the networking area. Next, it presents the issues to address when building distributed applications. The section also describes the seven layers of the ISO networking model. It concludes by presenting the components and middleware comprising distributed applications.

5.2. Streams

This section consists of the slides 43 through 151. It describes the process of receiving information into a program and sending it out again through the use of the stream classes located in the `java.io` package. The section also discusses two byte streams classes - `ObjectInputStream` and `ObjectOutputStream` used to read and write objects (serialization). In addition to discussing object streams, it also presents the classes and interfaces in `java.io` that help classes perform serialization for their instances.

5.3. Networking

This section comprises the slides 152 through 241. It first provides a brief overview of the networking terms and concepts. It then discusses how a Java application can use URL addresses to access information on the Internet. The section describes what a URL address is, shows how to create and parse a URL, explains how to open a connection to a URL address, and how to read from and write to that connection. The section also explains how to use sockets so that programs can communicate with one another over a network. The section concludes by presenting a complete client/server example, which shows how to implement both client and server sides of a distributed application.

5.4. Database Connectivity

This section comprises the slides 242 to 342. It starts by introducing enterprise information systems, describes different types of such systems, and explains the relevance of enterprise integration and the Java solution to this problem. Next, the section reviews basic database concepts and provides a brief introduction to the Structured Query Language (SQL). Afterwards, the section presents the Java Database Connectivity API and how it can help create tables, insert values into them, query tables, retrieve results of queries, and update tables. It also explains the usage of simple, prepared and callable statements. Finally, the section shows how to perform transactions and to move a cursor in a scrollable result set.

5.5. Architectures

This section, comprising the slides 343 to 622, presents typical architectures of distributed applications. It is divided into two subsections: Distributed Objects and Message Orientation. Under Distributed Objects, two technologies for calling remote objects in Java are explained: Remote Method Invocation (RMI) and Java Interface Definition Language (Java IDL). RMI technology allows the transfer and execution of Java code across a network, while Java IDL technology allows building and testing client and server objects conforming to the CORBA specification. Message Orientation section concludes by explaining how email functionality can be integrated into applications through the JavaMail API. Enterprise System Integration using Java Message Service (JMS) API is also discussed.

6. Assessment

Assessment questions are provided at the end of the course to test the students' understanding of the various topics taught. The assessment is more like a quiz which tests the general understanding of the course. The assessment does not replace the exercises provided across the various sections, nor the tasks students are asked to perform. The tasks, in particular, are more rigorous and aim to test the depth of understanding in specific areas. The assessment consists of multiple choice questions.

7. Organization

The course is designed to be taught during 42 hours. In the context of the e-Macao Core Team Training Programme, the course was taught over seven days. For the e-Macao Extended Team Training, the same course was taught over four days.

References

1. Distributed Programming with Java, Qusay H. Mahmoud, Manning Publisher 2000.
2. Java in Distributed Systems: Concurrency, Distribution and Persistence, Marko Boger, 2001.
3. Developing Distributed and E-commerce Applications, 2nd edition, Darrel Ince, 2nd edition, Pearson Addison Wesly, 2004.
4. Java Message Service (O'Reilly Java Series), Richard Monson-Haefel, David Chappell.
5. Sun SL 301 Distributed Programming with Java.
6. Java Tutorial, http://java.sun.com/docs/books/tutorial/index.html.

Appendix

A. Slides

A.1. Introduction

Distributed Programming

Gabriel Oteniya and Milton Chau Keng Fong

UNU-IIST

e-Macao-16-3-2

The Course

1) objectives - what do we intend to achieve?
2) outline - what content will be taught?
3) resources - what teaching resources will be available?
4) organization - duration, major activities, daily schedule

e-Macao-16-3-3

Course Objectives

1) learn the fundamental concepts of distributed programming for enterprise application development
2) learn the various distributed programming architectures and how to apply them
3) lean the importance of distributed computing and outline the factors to consider when designing a distributed system
4) presents different Distributed Architecture

e-Macao-16-3-4

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-5

Outline: Introduction

Presents an overview of the distributed programming.

Main points:

1) what are distributed systems?.
2) why distributed programming?.
3) nature and design considerations.
4) types of networks.
5) distributed architectures.

e-Macao-16-3-6

Outline: Stream

Presents the `java.io` package

Main points:

1) what is a stream?
2) types of Streams
3) characteristics of Streams
4) working with streams
5) bridging Streams
6) stream chaining

e-Macao-16-3-7

Outline: Networking

Presents the network programming in Java language.

Main points:

1) review the basic network concepts and Java Implementation
2) discuss the usage of java.net package.
3) introduce the Secure Socket.
4) introduce the New I/O API.
5) introduce the Java implementation for UDP protocol.

e-Macao-16-3-8

Outline: Database Connectivity

Presents JDBC API from the basics of SQL to the more esoteric features of advanced JDBC.

Main points:

1) introduction to Database and Structured Query Language
2) JDBC architecture
3) JDBC core interfaces
4) query processing
5) transaction Management

e-Macao-16-3-9

Outline: Message Orientation

Presents how to build a loosely coupled application using messaging mechanism.

Main points:

1) JavaMail – to provide asynchronous communication between application components and human users.
2) Java Message Service – to provide asynchronous/synchronous communication software components

e-Macao-16-3-10

Outline: Distributed Objects

This section basically addresses:

1) Remote Method Invocation (RMI)
2) Common Object Request Broker Architecture (CORBA)
3) Interface Definition Language (IDL)
4) IDL to Java mapping (JavaIDL)

e-Macao-16-3-11

Outline: Summary

Revision of the material introduced during the course.

How this course provides a foundation for the remaining courses:

1) Java XML processing
2) Java Web Services
3) J2EE web components
4) J2EE business components

e-Macao-16-3-12

Course Resources

1) books
   a) Distributed Programming with Java, Qusay H. Mahmoud, Manning Publisher 2000
   b) Java in Distributed Systems: Concurrency, Distribution and Persistence, Marko Boger, 2001
   c) Developing Distributed and E-commerce Applications, 2nd edition, Darrel Ince, 2nd edition, Pearson Addison Westly, 2004.
   d) Java Message Service (O'Reilly Java Series), Richard Monson-Haefel, David Chappell
2) tools
   a) mySQL Database engine
   b) JBoss 4.0.1
   c) JBossMQ
   d) Hermes 1.8 (JBossMQ Browser)

e-Macao-16-3-13

Course Logistics

1) duration - 36 hours
2) activities - lecture (hands-on), development
3) sessions/day - morning 09:00–13:00 and afternoon 14:30–16:30
4) number of sessions - 6 morning and 6 afternoon
5) style – interactive, Lab work and tutorial

e-Macao-16-3-14

Course Prerequisite

1) some experience in object-oriented programming:
   a) C++
   b) Delphi
   c) Java Programming Language
   d) any other object-oriented language
2) basic understanding of TCP/IP networking concepts

Introduction

e-Macao-16-3-16

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-17

Distributed System

Distributed system can be defined as a combination of several computers with separate memory, linked over a network, and on which it is possible to run a distributed applications.

Characteristics:

1) capable of communicating over a network
2) the network is usually stable
3) fail-safe
4) each device has a permanent identification within the network

Hence, it is a collection of independent computers, interconnected via a network, capable of collaborating on a task.

e-Macao-16-3-18

Distributed Application

A distributed application consist of several parts of a program communicating with each other, which cooperate to carry out a common task.

For example, client server application.

Typically, but not necessarily, the parts of the application are distributed across several computers.

The distribution can also be simulated on one computer.

In this case, however, information is not transmitted via a common memory or address space, but with the aid of techniques of remote communication.

e-Macao-16-3-19

Distributed Programming

Distributed programming is a model in which processing occurs in many different places (or nodes) around a network.

Characteristics:

1) processing can occur whenever it makes the most sense
2) carried out on a distributed system
3) making calls to other address spaces possibly on different machines
4) tasks are handled in parallel

e-Macao-16-3-20

Why Distributed Programming?

1) balance resource loading

2) lower cost of development since clients can access remote codes for services

3) separation of concerns

4) Platform independence

e-Macao-16-3-21

Design Considerations

In general, three aspects need to be put into consideration:

1) Concurrency – actual or apparent parallelism of control flows

   issues: how to manage both heavy and light weight processes

2) Distribution – is the logical and spatial distance of objects from each other

   Issue: how these object can locate, access and communicate with each other

3) Persistence – is the long-term storage of data or objects on non-volatile media

   issues: how to persist data and objects. Persistence achives the distribution of data or objects in time.

e-Macao-16-3-22

Protocol Layers

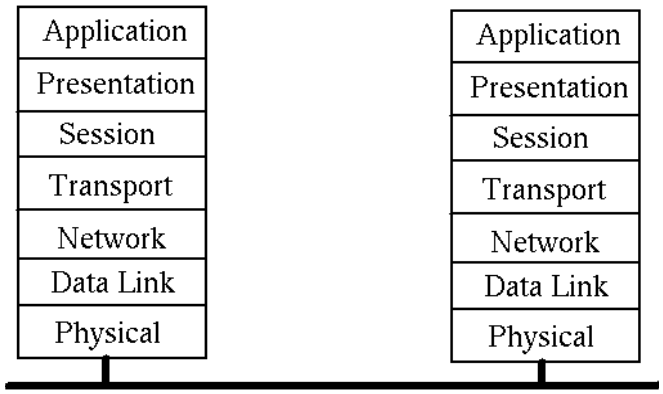
1) Communications betweens processes takes place using agreed conventions - protocols

2) Network communications requires protocols to cover high-level application communication all the way down to wire communication

3) Complexity handled by encapsulation in protocol layers

e-Macao-16-3-23

ISO OSI Protocol



e-Macao-16-3-24

OSI layers

1) Network layer provides switching and routing technologies

2) Transport layer provides transparent transfer of data between end systems and is responsible for end-to-end error recovery and flow control

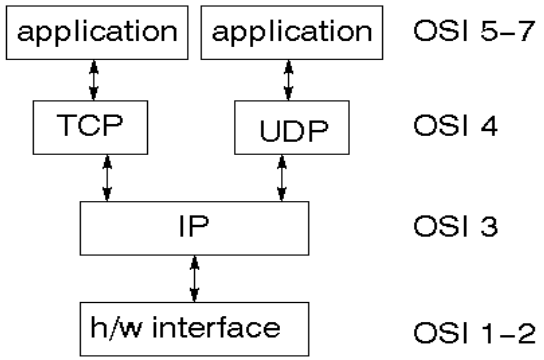
3) Session layer establishes, manages and terminates connections between applications.

4) Presentation layer provides independence from differences in data representation (e.g. encryption)

5) Application layer supports application and end-user processes

e-Macao-16-3-25

TCP/IP Protocol



e-Macao-16-3-26

Port and Socket

1) port
   a) conduit into a computer through which information flows and assigned a unique number
   b) usually port numbers 0 to 1023 are reserved for special purposes (e.g. HTTP – 80, FTP – 21, SMTP – 25
   c) TCP/IP-based computer is identified by a pair of IP address and Port number

2) socket
   a) a socket is one end of a process that an application is using to communicate
   b) defined by two addresses: the IP address of the host computer; and the port address of the application or process running on the host

e-Macao-16-3-27

Connection Models

There are two types of connection models:

1) Connection oriented
2) Connectionless

Connection oriented transports may be established on top of connectionless ones –TCP over IP

Connectionless transports my be established on top of connection oriented ones – HTTP over TCP

e-Macao-16-3-28

Connection oriented

1) A single connection is established for the session
2) Two-way communications flow along the connection
3) When the session is over, the connection is broken
4) The analogy is to a phone conversation
5) An example is TCP

e-Macao-16-3-29

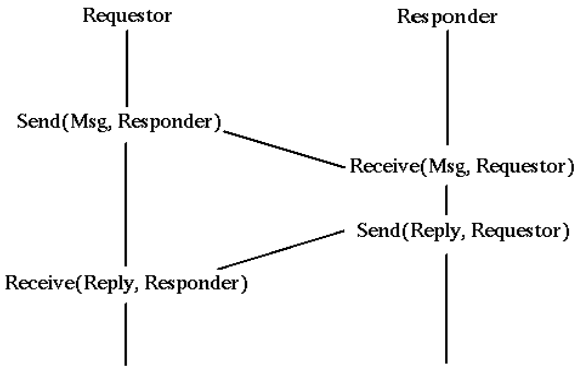
Connectionless

1) In a connectionless system, messages are sent independent of each other
2) Ordinary mail is the analogy
3) Connectionless messages may arrive out of order
4) An example is the IP protocol

e-Macao-16-3-30

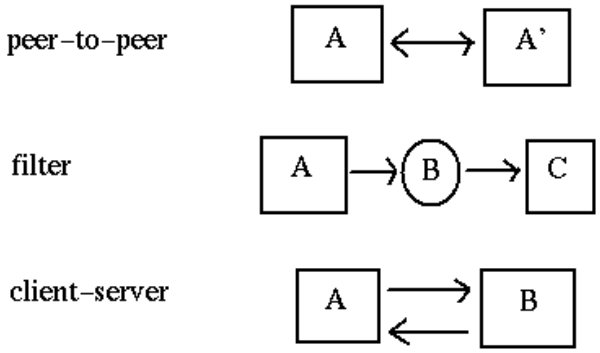
Communications Model

Message passing



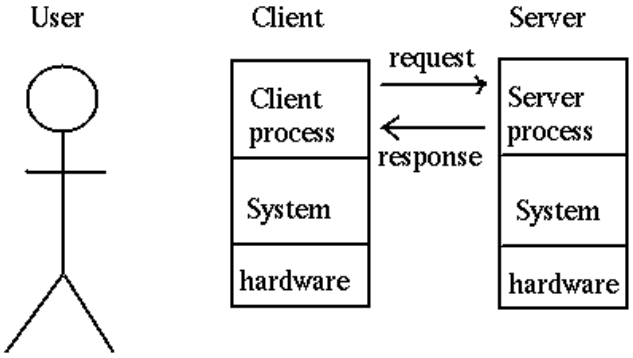
e-Macao-16-3-31

Distributed Computing Models



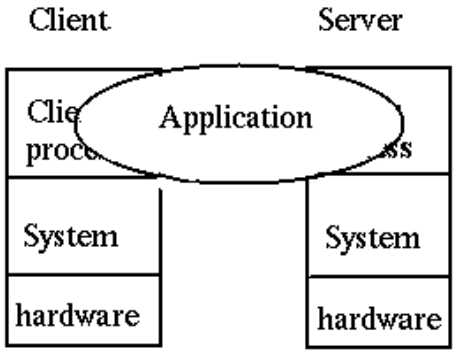
e-Macao-16-3-32

Client/Server System



e-Macao-16-3-33

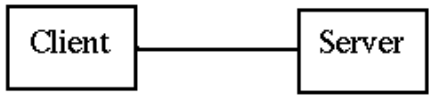
Client/Server Application



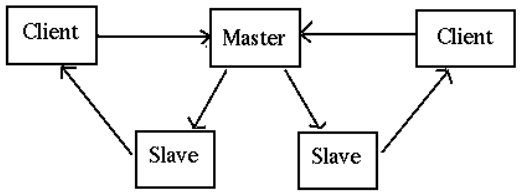
e-Macao-16-3-34

Server Distribution 1

Single client, single server



multiple clients, single server



e-Macao-16-3-35

Server Distribution 2

single client, multiple servers

multiple clients, multiple servers

e-Macao-16-3-36

Component Distribution

Every distribution is made up of three components:

1) Presentation component
2) Application logic
3) Data access

e-Macao-16-3-37

Middleware 1

1) intermediate layers between client and server

2) what exactly is it?
   a) a vague term that covers all the distributed software needed to support interactions between client and server

3) where does the middleware start and where does it end?
   a) It starts with the API set on the client side that is used to invoke a service, and it covers the transmission of the request over the network and the resulting response"

e-Macao-16-3-38

Middleware 2

1) The network services include things like TCP/IP

2) The middleware layer is application-independent s/w using the network services

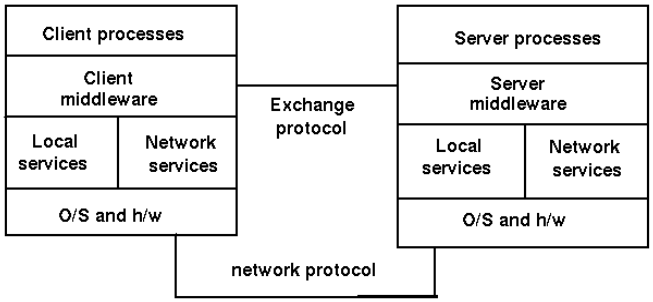
3) Examples of middleware are: DCE, RPC, Corba

4) Middleware may only perform one function (such as RPC) or many (such as DCE)

e-Macao-16-3-39

Middleware Model

The middleware model is



e-Macao-16-3-40

Example: Middleware

1) Primitive services such as terminal emulators, file transfer, email

- Basic services such as RPC

1) Integrated services such as DCE, Network O/S

- Distributed object services such as CORBA, OLE/ActiveX
- Mobile object services such as RMI, Jini
- World Wide Web

e-Macao-16-3-41

Middleware Functions

1) Initiation of processes at different computers

- Session management

1) Directory services to allow clients to locate servers

- remote data access
- Concurrency control to allow servers to handle multiple clients
- Security and integrity
- Monitoring
- Termination of processes both local and remote

e-Macao-16-3-42

Project Exercise 1

1) Describe a typical distributed application in use in your agency
2) Which of the following distributed architecture models best represents the distributed system described in question 1?
3) List the different components of the systems listed in 1
4) Provide a model of the system described in question 1 using a UML deployment diagram showing the various components listed in question two as well as the nodes hosting these components.
5) Identify the possible points of failures in the distributed system using the model presented in question 4.

A.2. Streams

Streams

e-Macao-16-3-44

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-45

Overview

1) what is a stream?
2) types of Streams
3) characteristics of Streams
4) working with streams
5) bridging Streams
6) stream chaining

e-Macao-16-3-46

Introduction

Most programs use data in one form or another, whether as input, output, or both.

The sources of input and output can vary between a local file, a socket on the network, a database, variables in memory, or another program.

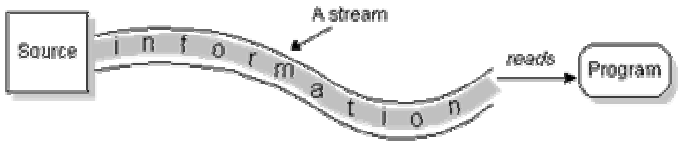
Even the type of data can vary between objects, characters, multimedia, and others.

e-Macao-16-3-47

Reading Data

To bring data into a program, a Java program:

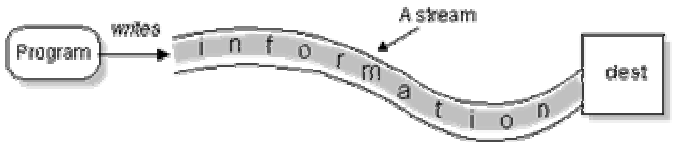
1) opens a stream to a data source, such as a file or remote socket
2) and reads the information serially



e-Macao-16-3-48

Writing Data

On the flip side, a program can open a stream to a data source and write to it in a serial fashion.



e-Macao-16-3-49

Reading and Writing Data

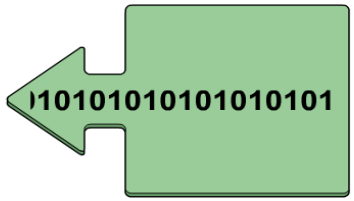
The concept of serially reading from, and writing to different data sources is the same.

For that very reason, once you understand the top level classes the remaining classes are straightforward to work with.

These classes are stored in the `java.io` package.

e-Macao-16-3-50

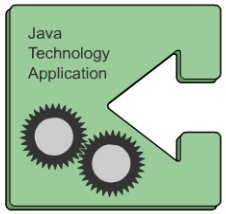
Streams and Data Sources 1



Applications often need to read data to and write data from other sources. Streams provide a means for reading and writing sequences of bytes.

e-Macao-16-3-51

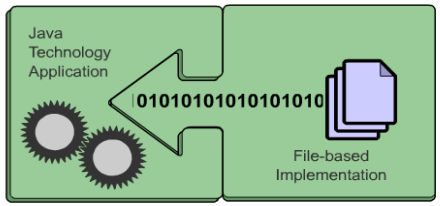
Streams and Data Sources 2



Streams are often used for character values, although this is not the only possibility. The stream concept gives consistent access to any source of data. The source of the data might be a file.

e-Macao-16-3-52

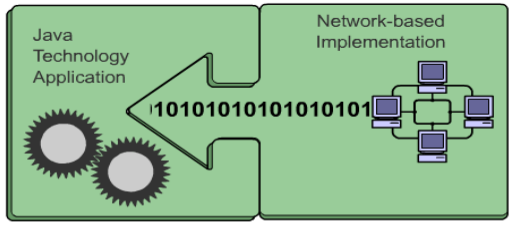
Streams and Data Sources 3



Streams are often used for character values, although this is not the only possibility. The stream concept gives consistent access to any source of data. The source of the data might be a file.

e-Macao-16-3-53

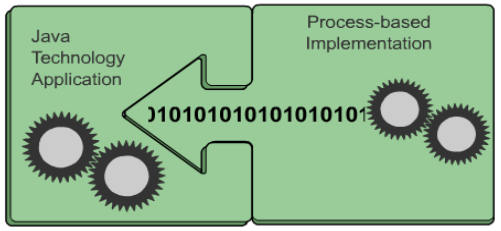
Streams and Data Sources 4



The source of this data might be a network.

e-Macao-16-3-54

Streams and Data Sources 5



The source of this data might even be another process.

e-Macao-16-3-55

Reading and Writing Algorithms

No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same:

Reading:

```
open a stream
while more information
    read information
close the stream
```

Writing:

```
open a stream
while more information
    write information
close the stream
```

e-Macao-16-3-56

Example: Reading Text from File

```
try {
      BufferedReader in = new BufferedReader(new
                                FileReader("file"));
      String str;
      while ((str = in.readLine()) != null) {
            process(str);
      }
      in.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

e-Macao-16-3-57

Lab Work: Reading a File

1) Based on the code snippet write a program to read a file and displays the content to the console.

e-Macao-16-3-58

Stream Types

There are two categories of streams:

1) 8-bit `byte` streams
2) 16-bit Unicode `character` streams

Prior to JDK 1.1, the input and output classes (mostly found in the `java.io` package) only supported 8-bit `byte` streams.

The concept of 16-bit Unicode `character` streams was introduced in JDK 1.1.

e-Macao-16-3-59

Stream Support

Support for `byte` streams are provided by:
1) `java.io.InputStream` abstract class
2) `java.io.OutputStream` abstract class
3) and their subclasses.

While the support for `character` streams are provided by:
1) `java.io.Reader` abstract class
2) `java.io.Writer` abstract class
3) and their subclasses.

e-Macao-16-3-60

Character versus Byte 1

Most of the functionality available for `byte` streams is also provided for `character` streams.

Methods for character streams generally accept parameters of data type `char` while methods for byte streams accept `byte`.

The names of the methods in both sets of classes are almost identical except for the suffix

1) character-stream classes end with the suffix Reader or Writer

2) byte-stream classes end with the suffix InputStream and OutputStream

e-Macao-16-3-61

Character versus Byte 2

For example:

1) to read files using `character` streams you would use `java.io.FileReader` class.

2) to read files using `byte` streams you would use `java.io.FileInputStream`.

Unless you are working with binary data, such as image and sound files, you should use readers and writers (`character` streams) to read and write the data.

Why?

e-Macao-16-3-62

Character versus Byte 3

`Character` streams are always preferred to `byte` streams when reading and writing information because:

1) They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).

2) They are easier to internationalize because they are not dependent upon a specific character encoding.

3) They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

e-Macao-16-3-63

I/O Streams Organization

`java.io` is a large collection of classes, consisting of over 50 classes.

For the purpose of understanding the relationships that exist among these classes, they are categorized using the following criteria:

1) Data flow
2) Function and
3) Type of Data they process

e-Macao-16-3-64

Data Flow

Stream classes that channel data into a program are called input streams.
Example:

1) `FileInputStream`
2) `FileReader`
3) `ObjectInputStream`
4) `PipedInputStream`
5) `etc`.

Stream classes that channel data out of a program are called output streams.
Example:

1) `FileOutStream`
2) `FileWriter`
3) `ObjectOutputStream`
4) `PipedOutputStream`
5) `etc`.

e-Macao-16-3-65

Function

Streams can also be grouped by the function they perform.

There are two categories:

1) Node or Data sink Streams - nature of the resource at the other end of the stream, For example,
   a) `FileInputStream` reads byte data from a file,
   b) `PipedWriter` writes `character` data to a pipe (Thread)

2) Process or Filter Streams - type of processing performed on the contents of the stream,
   For example,
   a) `BufferedReader` to buffer reading to reduce disk/network access
   b) `ObjectOutputStream` for object serialization

e-Macao-16-3-66

Data Sink Streams

`CharArrayReader, CharArrayWriter`	For reading from or writing to character buffers in memory
`FileReader, FileWriter`	For reading from or writing to files
`PipedReader, PipedWriter`	Used to forward the output of one thread as the input to another thread
`StringReader StringWriter`	For reading from or writing to strings in memory
`ByteArrayInputStream ByteArrayOutputStream`	For reading from or writing to byte buffers in memory
`FileInpuStream, FileOutputStream`	For reading from or writing bytes to files
`PipedInputStream, PipedOutputStream`	Used to forward the output of one thread as the input to another thread

e-Macao-16-3-67

Example: Data Sink Streams

Displays contents of a file.

```
import java.io.*;
public class Type{
   public static void main(String args[]) throws
                                          Exception{
      FileReader fr = new FileReader(args[0]);
      PrintWriter pw = new PrintWriter(System.out,
                                              true);
      char c[] = new char[4096];
      int read = 0;
      while ((read = fr.read(c)) != -1)
            pw.write(c, 0, read);
      fr.close(); pw.close();
   }}
```

e-Macao-16-3-68

Filter Streams

`BufferedReader , BufferedWriter`	For buffered reading/writing to reduce disk/network access for more efficiency
`InputStreamReader , OutputStreamWriter`	Provide a bridge between byte and character streams
`SequenceInputStream`	Concatenates multiple input streams.
`ObjectInputStream , ObjectOutputStream`	Use for object serialization.
`DataInputStream , DataOutputStream`	For reading/writing raw bytes to Java native data types.
`PushbackReader`	Allows to "peek" ahead in a stream by one character.
`LineNumberReader`	For reading while keep tracking of the line number.

e-Macao-16-3-69

Example: Filter Streams

Displays contents of many files

```
import java.io.*;
class cat  {
  public static void main (String args[]) {
      String thisLine;
      for (int i=0; i < args.length; i++) {
      try {
       BufferedReader br = new BufferedReader(new
                           FileReader(args[i]));
       while ((thisLine = br.readLine()) != null) {
         System.out.println(thisLine);
       }
     }catch (IOException e) {
       System.err.println("Error: " + e);
     }}}}
```

e-Macao-16-3-70

Data Type

At the Simplest level, the `java.io` package can be decomposed into classes that process either of two types of data:

1) Byte Stream
2) Character Stream

Fundamental Stream Classes

	Byte Stream	Character Stream
Read Data	`InputStream`	`Reader`
Write data	`OutputStream`	`Writer`

e-Macao-16-3-71

Byte Stream Classes

They process raw bytes.

They come in two basic forms:
1) InputStreams – channel `byte` data into the program
   Example:
   `java.io.FileInputStream`

2) OutputStreams – channel `byte` data from the program
   Example:
   `java.io.FileOutputStream`

Byte-stream classes end with the suffix InputStream and OutputStream

e-Macao-16-3-72

Byte-Stream Parent Classes

`InputStream` and `OutputStream` are the abstract parent classes for byte-stream based classes in the `java.io` package.

Usage:
1) `InputStream` classes are used to read 8-bit byte streams and

2) `OutputStream` classes are used to write to 8-bit byte streams.

Methods for reading and writing to streams:

```
int read()
int read(byte[] b)
int read(byte[] b, int offset, int length)
int write(int b)
int write(byte[] b)
int write(byte[] b, int offset, int length)
```

e-Macao-16-3-73

InputStream

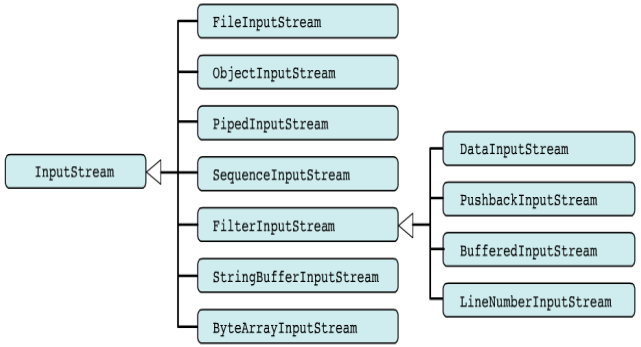
This abstract class provides the core methods used to read bytes from an input node.

The methods are:

```
int read()
int read(byte[] b)
int read(byte[] b, int offset, int length)
void close()
int available()
long skip( long l)
boolean markSupported()
void mark( int i)
void reset()
```

e-Macao-16-3-74

InputStream Hierarchy



e-Macao-16-3-75

OutputStream

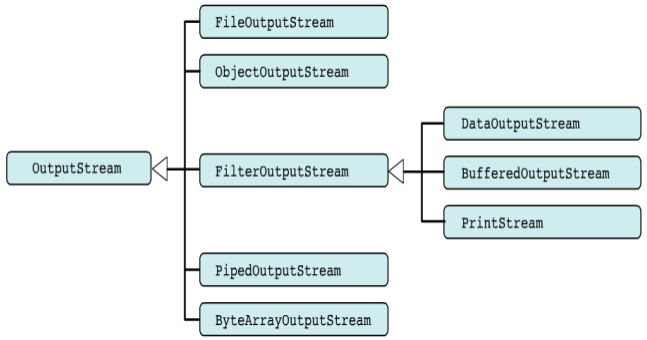
This abstract class provides the core methods used to write bytes to an output node.

The methods are:

```
int write(int b)
int write(byte[] b)
int write(byte[] b, int offset, int length)
void close()
void flush()
```

e-Macao-16-3-76

OutputStream Hierarchy



e-Macao-16-3-77

Data Sink Byte-Stream

Classes that take byte input from different types of nodes (file, pipe, byte array, etc)

Example:

`FileInputStream`

`PipedInputStrean`

Classes that send byte output to different types of node (file, pipe, byte array, etc)

Example:

`FileOutputStream`

`PipedOutputStream`

e-Macao-16-3-78

Example: Data Sink Byte-Stream

1) `FileInputStream`/`FileOutputStream`

Usage: is meant for reading/writing streams of raw bytes such as image data to/from files

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new
BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

2) `ByteArrayInputStream`/`ByteArrayOutputStream`

Usage: are useful for holding data when the underlying data type is irrelevant to the purpose of the application

```
byte[] c
...
ByteArrayInputSteam r = new ByteArrayInputSteam(c);
```

e-Macao-16-3-79

Example: Data Sink Byte-Stream

3) `PipedInputStream`/`PipedOutputStrea`

Usage: read from or write to pipes. Often used to exchange data between threads.

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

Or

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

Or

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
pi.connect(po);
```

e-Macao-16-3-80

Example: FileInputStream 1

```
import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) throws
  Exception {
    int size;
    InputStream f =
      new FileInputStream("FileInputStreamDemo.java");

    System.out.print("Total Available Bytes: " )
     System.out.println((size = f.available()));
    int n = size/40;
    System.out.println("First " + n +
          " bytes of the file one read() at a time");
    for (int i=0; i < n; i++) {
      System.out.print((char) f.read());
    }
```

e-Macao-16-3-81

Example: FileInputStream 2

```
System.out.println("\nStill Available: " +
                                f.available());
System.out.println("Reading the next " + n +
                   " with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
  System.err.println("couldn't read " + n + "
                                      bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size =
                                f.available()));
System.out.println("Skipping half of remaining
                           bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " +
                                f.available());
```

e-Macao-16-3-82

Example: FileInputStream 3

```
    System.out.println("Reading " + n/2 + " into the
                                      end of array");
    if (f.read(b, n/2, n/2) != n/2) {
      System.err.println("couldn't read " + n/2 + "
                                        bytes.");
    }
    System.out.println(new String(b, 0, b.length));
    System.out.println("\nStill Available: " +
                                      f.available());
    f.close();
  }
}
```

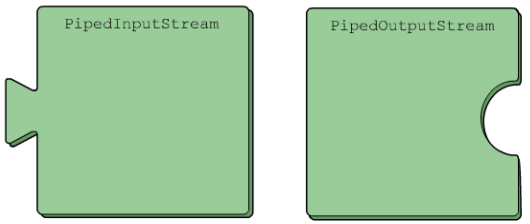
e-Macao-16-3-83

Lab Work: Reading a File

1) Write a program that reads an MP3 file.

e-Macao-16-3-84

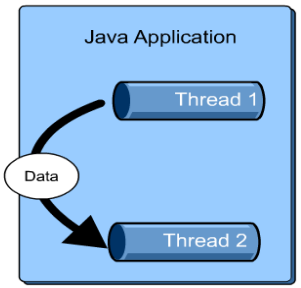
PipedInput/Output Stream 1



The `PipedInputStream` and `PipedOutputStream` classes are designed to be used as a pair. They allow a mechanism for communication among threads, although they have other uses, too.

e-Macao-16-3-85

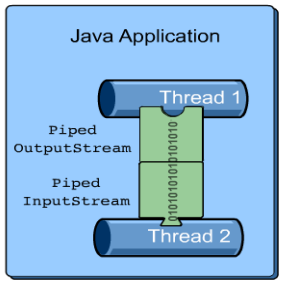
PipedInput/Output Stream 2



To illustrate how the piped streams can be used, suppose you have an application which must transfer data between two separate threads.

e-Macao-16-3-86

PipedInput/Output Stream 3



The `PipedInputStream` /`PipedOutputStream` are created as a pair, so that what is written into the output stream can be read from the input stream. The effect is much like a pipeline that carries data from one point in an application to another.

e-Macao-16-3-87

Example: Piped Stream 1

Shows how to exchange data between two threads

```
import java.io.*;

class ReadThread extends Thread implements Runnable {
      InputStream pi = null;
      OutputStream po = null;
      String process = null;
      ReadThread( String process, InputStream pi,
                                  OutputStream po) {
            this.pi = pi;
            this.po = po;
            this.process = process;
      }
```

e-Macao-16-3-88

Example: Piped Stream 2

```
public void run() {
      int ch;
      byte[] buffer = new byte[512];
      int bytes_read;
      try {
            for(;;) {
                  bytes_read = pi.read(buffer);
                  if (bytes_read == -1) { return; }
                  po.write(buffer, 0, bytes_read);
            }
      } catch (Exception e) {
                  e.printStackTrace();
      } finally {  }
  }

}
```

e-Macao-16-3-89

Example: Piped Stream 3

```
class SystemStream {
 public static void main( String [] args) {
    try {
      int ch;
      while (true) {
        PipedInputStream  in = new PipedInputStream();
        PipedOutputStream out = new PipedOutputStream(
                                                 in );
        FileOutputStream writeOut = new
                            FileOutputStream("out");

        ReadThread rt = new ReadThread("reader",
                                    System.in, out );
        ReadThread wt = new ReadThread("writer",  in,
                                       System.out );
        rt.start();
        wt.start();
      }
```

e-Macao-16-3-90

Example: Piped Stream 4

```
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

e-Macao-16-3-91

Filter Byte-Stream

They convert bytes to primitive data

They write primitive data

Example:

`BufferedInputStream/BufferedOutputStream`
`DataInputStream/DataoutputStream`

e-Macao-16-3-92

Example: Filter Byte-Stream 1

1) `BufferedInputStream/BufferedOutputStream`
   Usage:
   These classes buffer data emanating from an InputStream object or in route to an OutputStream object.

Benefits:

a) Improved performance - Buffered streams cache data to reduce the need to access slower transmission media.
b) Simplicity - Buffered streams manage the data cache themselves, so you do not have to.

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

e-Macao-16-3-93

Example: Filter Byte-Stream 2

2) `DataInputStream/DataoutputStream`

Usage:

These classes transform bytes emanating from an InputStream type into primitives (such as int, long, or double) or primitives in route to an OutputStream type into bytes.

Attach a `DataInputStream` filter to an InputStream object when you need to read primitives from a stream.

Attach a `DataOutputStream` filter to an OutputStream object when you need to write primitives to a stream.

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

e-Macao-16-3-94

Example: BufferedInputStream 1

```
import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) throws
  IOException {
    String s = "This is a &copy; copyright symbol " +

      "but this is &copy not.\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new


      ByteArrayInputStream(buf);
    BufferedInputStream f = new BufferedInputStream(in);
    int c;
    boolean marked = false;
```

e-Macao-16-3-95

Example: BufferedInputStream 2

```
    while ((c = f.read()) != -1) {
      switch(c) {
      case '&':
        if (!marked) {
          f.mark(32);
          marked = true;
        } else {
          marked = false;
        }
        break;
      case ';':
        if (marked) {
          marked = false;
          System.out.print("(c)");
        } else
```

e-Macao-16-3-96

Example: BufferedInputStream 3

```
          System.out.print((char) c);
          break;
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
              System.out.print((char) c);
          break;
        }}}}
```

e-Macao-16-3-97

Serialization

Serialization is a process of writing an object to a byte stream.

Writing an Object

```
FileOutputStream out = new FileOutputStream("tmp");
ObjectOutput objOut = new ObjectOutputStream(out);
objOut.writeObject(Color.red);
```

Reading an Object

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream objIn = new ObjectInputStream(in);
Color c = (Color)objIn.readObject();
```

e-Macao-16-3-98

Object Serialization 1

Provides a way for objects to be written as a stream of bytes and then later recreated from that stream of bytes.

The job of an `ObjectInputStream` class is to convert collections of bytes into objects.

Sending an object over a stream was a cumbersome process. How?

Essentially, you had to decompose the object into its constituent parts, sending each to the stream individually, and then reconstruct the object manually at the other end of the stream.

Process is cumbersome. Solution?

e-Macao-16-3-99

Object Serialization 2

The introduction of new interface to the `java.io` package, the `Serializable` interface

The `Serializable` interface eliminates the drawbacks of sending objects across streams.

Each object to be sent has to implement this interface

```
import java.io.* ;
class Date implements Serializable {
      int m, d, y ;
      public Date( int m, int d, int y ) {
                  this.m = m ; this.d = d ; this.y = y ;
      }
}
```

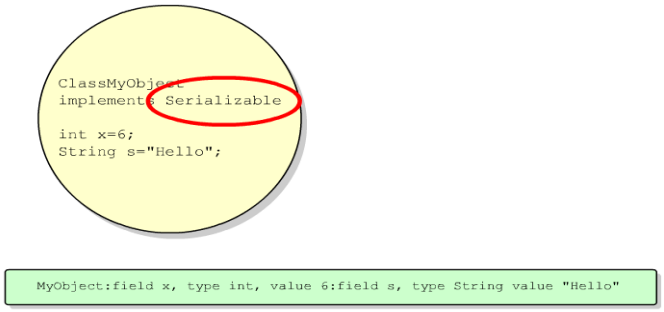
e-Macao-16-3-100

Object Serialization 3

Serialization takes the state (that is the instance variables) of an object and represents them as a sequence of bytes

e-Macao-16-3-101

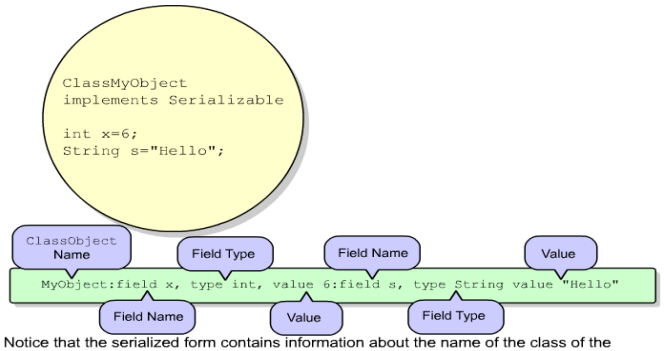
Object Serialization 4



It is important that the class that defines the object declares that the object is serializable. Otherwise, this process does not work.

e-Macao-16-3-102

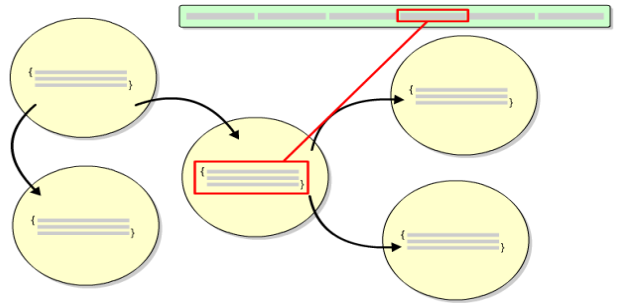
Object Serialization 5



Notice that the serialized form contains information about the name of the class of the object, the names and types of the fields, as well as the values themselves.

e-Macao-16-3-103

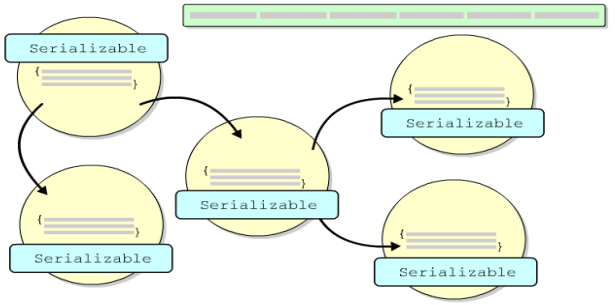
Object Serialization 6



Most objects have references to other objects within them, and sometimes those reference objects also have other references to other objects. All the objects that are reachable through these references form the object graph.

e-Macao-16-3-104

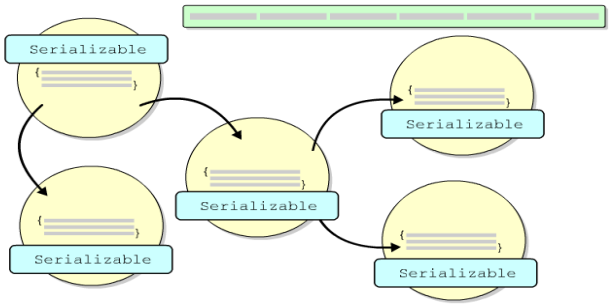
Object Serialization 7



Normally, when an object is serialized, all the objects in the object graph must be serialized. This means that all the objects in the object graph must implement `Serializable` as well. Otherwise, the serialization will fail.

e-Macao-16-3-105

Object Serialization 8



Normally, when an object is serialized, all the objects in the object graph must be serialized. This means that all the objects in the object graph must implement `Serializable` as well. Otherwise, the serialization will fail.

e-Macao-16-3-106

Character Stream Classes

They process 16 bits Unicode characters.

They come in two basic forms:

1) Reader – channel `character` data into the program
   Example:
   `java.io.FileReader`

2) Writer – channel `character` data from the program
   Example:
   `java.io.FileWriter`

Character-stream classes end with the suffix Reader and Writer

e-Macao-16-3-107

Character Parent Classes

`Reader` and `Reader` are the abstract parent classes for character-stream based classes in the `java.io` package.

Usage:

1) `Reader` classes are used to read 16-bit character streams and

2) `Writer` classes are used to write to 16-bit character streams.

Methods for reading and writing to streams:

```
int read()
int read(char[] c)
int read(char[] c, int offset, int length)
int write(int c)
int write(char[] c)
int write(char[] c, int offset, int length)
```

e-Macao-16-3-108

Reader

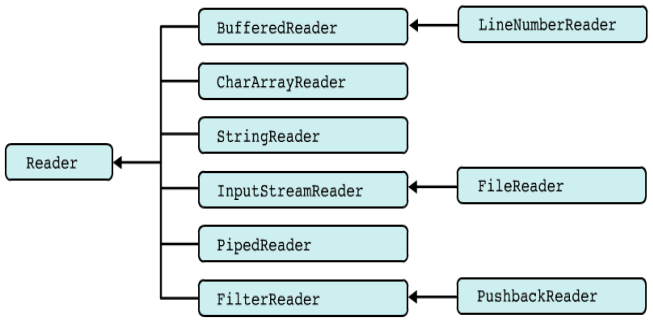
This abstract class provides the core methods used to read characters from an input node.

The methods are:

```
int read()
int read(char[] c)
int read(char[] c, int offset, int length)
void close()
long skip( long l)
boolean markSupported()
void mark( int i)
void reset()
```

e-Macao-16-3-109

Reader Hierarchy



e-Macao-16-3-110

Writer

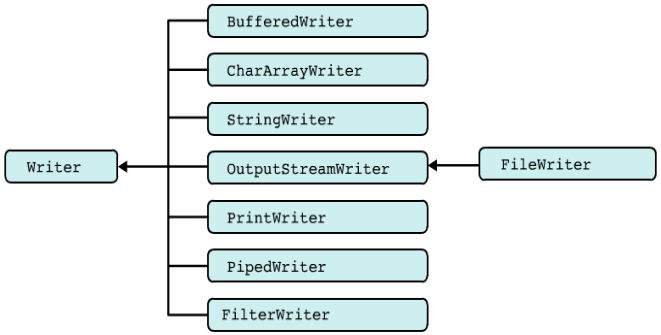
This abstract class provides the core methods used to write characters to an output node.

The methods are:

```
int write(int b)
int write(char[] c)
int write(char[] c, int offset, int length)
void close()
void flush()
```

e-Macao-16-3-111

Writer Hierarchy



e-Macao-16-3-112

Data Sink Character-Stream

Classes that take input from different types of nodes (file, pipe, char array, etc)

Example:

```
FileReader
PipedReader
```

Classes that send output to different types of node (file, pipe, char array, etc)

Example:

```
FileWriter
PipedWriter
```

e-Macao-16-3-113

Example: Node Character-Stream

1) `FileReader`/`FileWriter`

Usage: is meant for reading/writing streams of character data to/from files

```
File f = new File("mydata.txt");
FileReader fis = new FileReader(f);
```

2) `CharArrayReader`/`CharArrayWriter`

Usage: are useful for holding data when the underlying data type is irrelevant to the purpose of the application

```
char[] c
...
CharArrayReader r = new CharArrayReader(c);
```

e-Macao-16-3-114

Bridging Streams 1

To bridge the gap between the `byte` and `character` stream classes, JDK 1.1 and JDK 1.2 provide the `java.io.InputStreamReader` and `java.io.OutputStreamWriter` classes.

Usage:

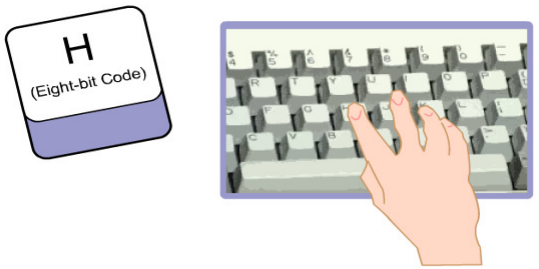
The only purpose of these classes is to convert byte data into character-based data according to a specified (or the platform default) encoding.

For example, the static data member "`in`" in the "`System`" class is essentially a handle to the Standard Input (stdin) device. If you want to wrap this inside the `java.io.BufferedReader` class that works with character-streams, you use `InputStreamReader` class as follows:

```
BufferedReader in = new BufferedReader(new
                        InputStreamReader(System.in));
```

e-Macao-16-3-115

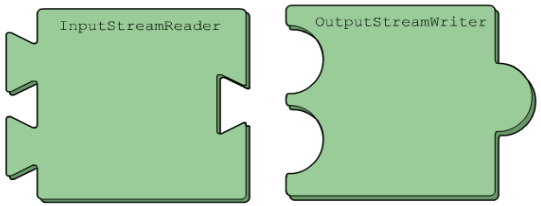
Bridging Streams 2



When a key is pressed, most platforms produce an 8bit code to represent a character. There are many different character sets that a platform may use to encode characters. Internally, the JVM uses 16bit Unicode characters.

e-Macao-16-3-116

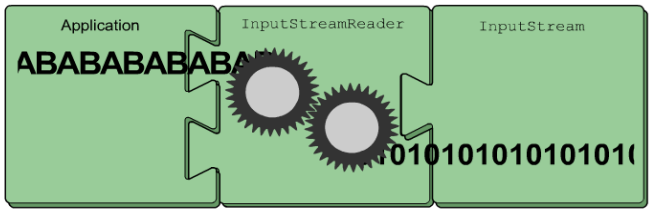
Bridging Streams 3



When characters enter or leave the program, they must be converted between these distinct encoding formats. This is the job of the `InputStreamReader` and `OutputStreamWriter` classes.

e-Macao-16-3-117

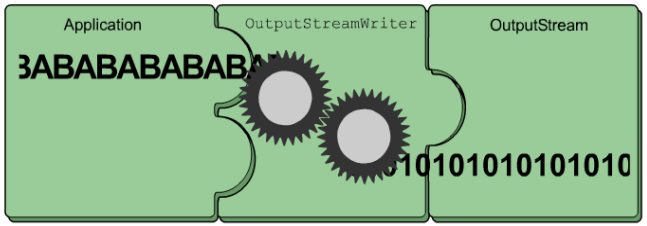
Bridging Streams 4



This is the `InputStreamReader` object. This class uses a `Reader` type object on the left hand side, but can plug into an `InputStream` type object on the right hand side. This allows it to read data that are encoded in a platorm-specific way from an `InputStream` object and convert those data into 16 bit unicode characters that can be read from the `Reader` interface by the client program.

e-Macao-16-3-118

Bridging Streams 5



When an application wants to write data out to the local platform, the `OutputStreamWriter` performs the inverse conversion, taking Unicode characters and mapping them to bytes in the local encoding format.

e-Macao-16-3-119

Summary: Filter Streams

Type	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Converting between bytes and character	InputStreamReader OuptutStreamWriter	
Object serialization		ObjectInputStream ObjectOutputStream
Data conversion		DataInputStream DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

e-Macao-16-3-120

Reading Text from Standard Input

```
try {
        BufferedReader in = new BufferedReader(new
                            InputStreamReader(System.in));
        String str = "";
        while (str != null) {
            System.out.print("> prompt ");
            str = in.readLine();
            process(str);
        }
    } catch (IOException e) {
    }
```

e-Macao-16-3-121

Reading Text from a File

```
try {
        BufferedReader in = new BufferedReader(new
                        FileReader("infilename"));
        String str;
        while ((str = in.readLine()) != null) {
            process(str);
        }
        in.close();
    } catch (IOException e) {
    }
```

e-Macao-16-3-122

Writing to a File

```
try {
        BufferedWriter out = new BufferedWriter(new
                        FileWriter("outfilename"));
        out.write("aString");
        out.close();
    } catch (IOException e) {

    }
```

e-Macao-16-3-123

Appending to a File

```
try {
        BufferedWriter out = new BufferedWriter(new
                        FileWriter("filename", true));
        out.write("aString");
        out.close();
    } catch (IOException e) {
    }
```

e-Macao-16-3-124

Serializing an Object

```
Object object =  new javax.swing.JButton("push");
try {
    // Serialize to a file
    ObjectOutputStream out = new

  ObjectOutputStream(new
              FileOutputStream("filename.ser"));
    out.writeObject(object);
    out.close();
    // Serialize to a byte array
    ByteArrayOutputStream bos = new

  ByteArrayOutputStream() ;
    out = new ObjectOutputStream(bos) ;
    out.writeObject(object);
    out.close();
    //  Get the bytes of the serialized object
     byte[] buf = bos.toByteArray();
   } catch (IOException e) {
```

e-Macao-16-3-125

Deserializing an Object

```
try {
     // Deserialize from a file
     File file = new File("filename.ser");
     ObjectInputStream in = new
                          ObjectInputStream(new
                          FileInputStream(file));
     // Deserialize the object
     javax.swing.JButton button =
            (javax.swing.JButton) in.readObject();
     in.close();

   } catch (ClassNotFoundException e) {

   } catch (IOException e) {

   }
```

e-Macao-16-3-126

Lab Work: Reading and Writing

Based on the code snippets, write a program that

1) reads text from standard input
2) copies the content of one file and writes to another file
3) appends "`This is emacao training`" to the end a text file
4) a program that joins series of files together

e-Macao-16-3-127

Stream Chaining

Stream chaining is a way of connecting several stream classes together to get the data in the form required.

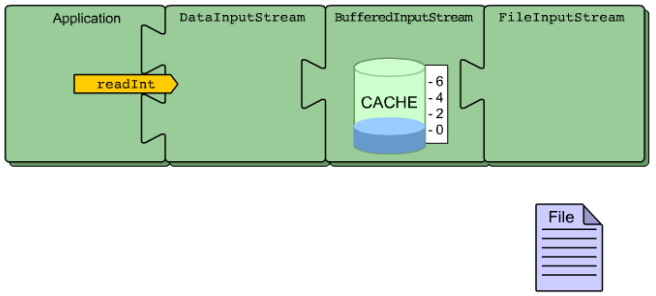
Each class performs a specific task on the data and forwards it to the next class in the chain.

The output produced by one component becomes the input to the next component in the chain.

Consider this.

e-Macao-16-3-128

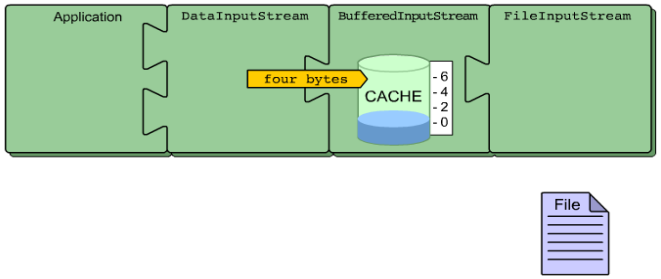
Example: Stream Chaining 1



When an application invokes a method like `readInt` on a `DataInputStream` object, what happens beneath the surface?

e-Macao-16-3-129

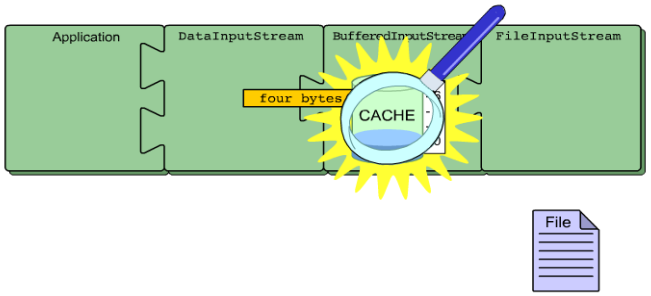
Example: Stream Chaining 2



The `DataInputStream` object requests four bytes (the representation for an `int` type) from the `BufferedInputStream` object (a reference to which is contained within the `DataInputStream` object).

e-Macao-16-3-130

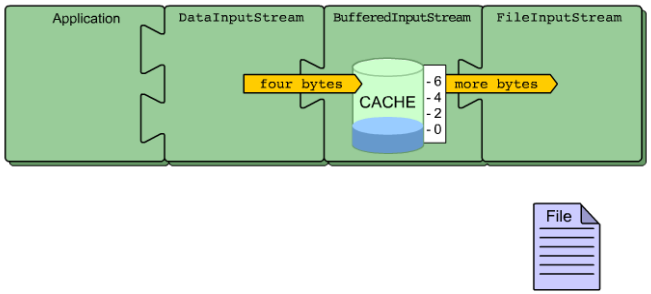
Example: Stream Chaining 3



The `BufferedInputStream` object inspects its internal cache. If the cache does not contain four bytes of leftover data from a previous read, the `BufferedInputStream` object will request additional bytes from the `FileInputStream` object (a reference to which is contained within the `BufferedInputStream` object).

e-Macao-16-3-131

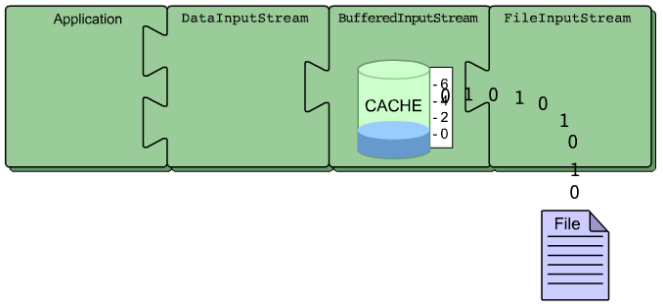
Example: Stream Chaining 4



The `BufferedInputStream` object inspects its internal cache. If the cache does not contain four bytes of leftover data from a previous read, the `BufferedInputStream` object will request additional bytes from the `FileInputStream` object (a reference to which is contained within the `BufferedInputStream` object).

e-Macao-16-3-132

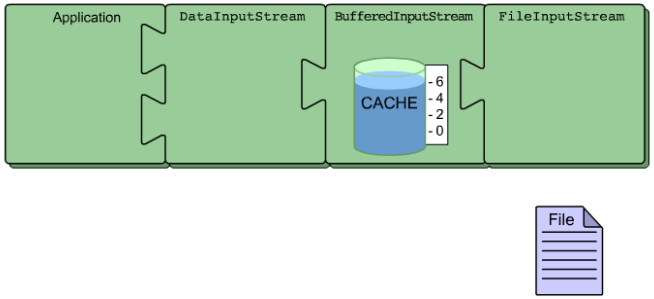
Example: Stream Chaining 5



The `FileInputStream` object reads the requested numbers of bytes from the file and returns them to the `BufferedInputStream` object, which then caches the data in its internal storage buffer.

e-Macao-16-3-133

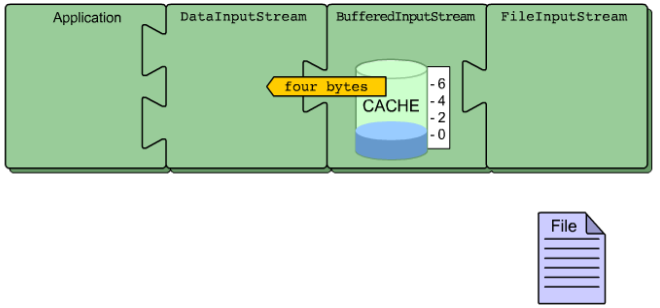
Example: Stream Chaining 6



The `FileInputStream` object reads the requested numbers of bytes from the file and returns them to the `BufferedInputStream` object, which then caches the data in its internal storage buffer.

e-Macao-16-3-134

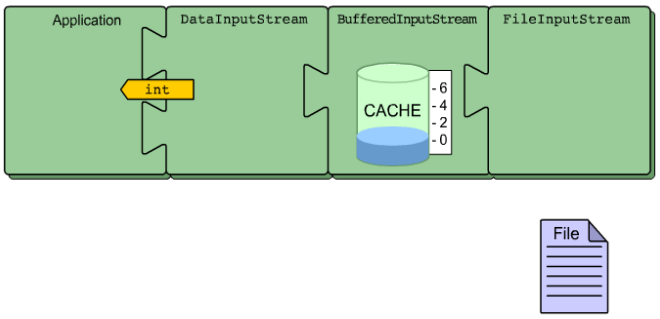
Example: Stream Chaining 7



The `BufferedInputStream` object extracts the four bytes requested by the `DataInputStream` object from the cache and returns them to the calling method.

e-Macao-16-3-135

Example: Stream Chaining 8

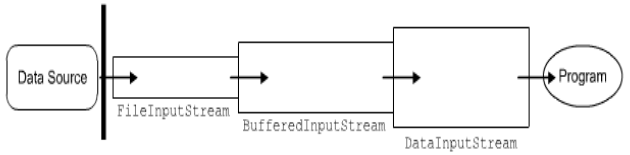


The `DataInputStream` object returns the four bytes to the application in the form of an `int` type.

e-Macao-16-3-136

IntputStream Chain

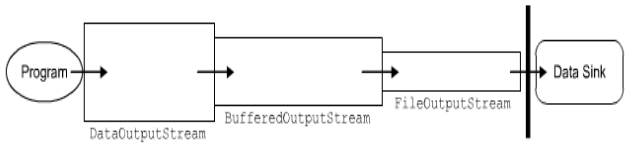
```
FileInputStream theFile = new FileInputStream( "input.dat" ) ;
BufferedInputStream theBuffer = new BufferedInputStream ( theFile ) ;
DataInputStream theData = new DataInputStream( theBuffer ) ;
```



e-Macao-16-3-137

OutputStream Chain

```
FileOutputStream theFile = new FileOutputStream( "output.dat" ) ;
BufferedOutputStream theBuffer = new BufferedOutputStream ( theFile ) ;
DataOutputStream theData = new DataOutputStream( theBuffer ) ;
```



e-Macao-16-3-138

File Operations

There are three non stream classes in `java.io` package.

1) `File` Class - represents a file on the local system

2) `FilenameFilter` class - is an interface used to filter a list of filenames

Each will be considered in details.

c

e-Macao-16-3-139

File Class

Represents a file on the local filesystem.

Usage:
1) to identify a file
2) obtain information about the file
3) and even change information about the file

Constructors:

1) `File(File parent, String child)`
2) `File(String pathname)`
3) `File(String parent, String child)`
4) `File(URI uri)`

e-Macao-16-3-140

Example: File 1

```
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("filename here");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "writeable" : "not  writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
```

e-Macao-16-3-141

Example: File 2

```
    p("is " + (f1.isDirectory()?"" :"not a directory"));
    p(f1.isFile() ?"normal file":" a named pipe");
    p(f1.isAbsolute() ? "absolute" : "not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

e-Macao-16-3-142

Directories

A directory is a `File` that contains a list of other files and directories.

When you create a `File` object and it is a directory, the `isDirectory()` method will return true.

In this case you can call `list()` on that object to extract the list of other files and directories inside.

The form of `list()` is

```
String[] list();
```

The list of file is returned in an array of `String` objects.

e-Macao-16-3-143

Example: Directories 1

```
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
```

e-Macao-16-3-144

Example: Directories 2

```
      }
    } else {
      System.out.print(dirname + " is not a”);

      System.out.println(“directory");
    }
  }
}
```

e-Macao-16-3-145

Creating Directories

Another two useful `File` utility methods are:

1) `mkdir()` – creates a directory , returning `true` on success and `false` on failure.

   Failure indicates that the path specified in the `File` object already exists, or that the directory cannot be created because the entire path does not exist yet.

2) `mkdirs()` – to create directories for which no path exists, it creates both a directory and all the parents of the directory

e-Macao-16-3-146

FilenameFilter

To limit the number of files returned by the `list()` method to include only those files that match a certain type of filename pattern, or filter, use the second form of `list()`.

The second form of `list()` is:

```
String[] list(FilenameFilter ffobj);
```

`FilenameFilter` defines only a single method, `accept()`, which is called once for each file in a list.

The `accept()` method returns `true` for files in the directory specified by directory that should be included in the list and returns `false` if otherwise.

e-Macao-16-3-147

Example: FilenameFilter 1

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

e-Macao-16-3-148

Example: FilenameFilter 2

Here is a modified version of directory listing program. Now it display only classes that use `.html` extension.

```
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

e-Macao-16-3-149

Stream Benefits

The Streaming interface to I/O in Java provides:

1) A clean abstraction for complex and often cumbersome task.
2) The composition of filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.
3) Java programs written to adhere to the abstract, high level-level `InputStream`,`OutputStream`, `Reader` and `Writer` classes will function properly in the future even when new and improved concrete stream classes are invented.
4) Serialization of object is expected to play an increasingly important role in Java Programming in the future.

e-Macao-16-3-150

Lab Work: Input and output

1) Write a program that reads the content of `"C:\Documents and Settings\All Users"` on your local system.
2) Write a program that counts the total number of directories and files you have in the path.
3) Archive at least one of the subdirectories of `All Users` folder and save it in zip format in your folder on the network.
4) Study and use `java.util.zip` package by referring to the API documentation for the appropriate classes to use in this exercise.

e-Macao-16-3-151

Project Exercise 2

1) Implement the client component of your software architecture which satisfies your use case model. Provide a web interface for users and desktop interface for back office processing. Implementation should be carried out using Java swing library.
2) Check for the consistency between your implementation and your design class diagrams (in your design model). For instance, are all your design classes implemented?
3) Check for the consistency between the dynamic aspect of your architecture (instance level collaboration diagrams) and your implementation
4) Update your implementation model indicating the implementing artifacts for your client component.

Note: Provide appropriate version control for all artifacts (models and codes)

A.3. Networking

Networking

e-Macao-16-3-153

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-154

Outline: Networking

Presents the network programming in Java language.

Main points:

1) Review the basic network concepts and Java Implementation.
2) Discuss the usage of java.net package.
3) Introduce the Secure Socket.
4) Introduce the New I/O API.
5) Introduce the Java implementation for UDP protocol.

e-Macao-16-3-155

Why Java

Why use Java for Networking?

a) Java was the first programming language designed from the ground up with networking in mind.

b) Java provides easy solutions to two crucial problem for Internet networking — platform independence and security.

c) It is far easier to write network programs in Java than in almost any other language.

- In the fully functional applications, very little code is devoted to networking.

e-Macao-16-3-156

Network programs with Java 1

Examples that a Network Program can do:

a) Server-Client

Examples: RssOwl (http://rssowl.sourceforge.net/)

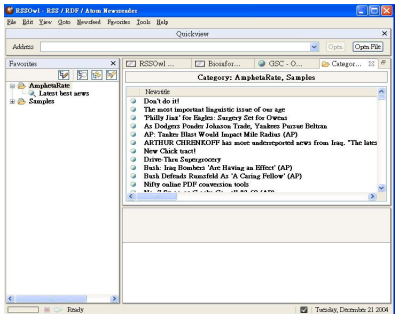
b) Peer-to-Peer

Examples: LimeWire (http://limewire.org/)

Azureus (http://azureus.sourceforge.net/)

e-Macao-16-3-157

Network programs with Java 2



http://rssowl.sourceforge.net/

- RssOwl combines news from different sources and allows the user to browse using a modern graphical user interface.
- Unlike a web browser, this program can continuously update the data in real time.

e-Macao-16-3-158

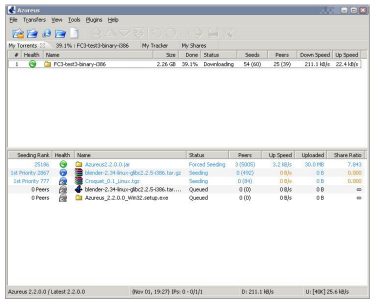
Network programs with Java 3

http://www.limewire.org/

- LimeWire enables the clients to query each other and transfer files among themselves.
- LimeWire is an open source pure Java application that uses a Swing GUI and standard Java networking classes.

e-Macao-16-3-159

Network programs with Java 4



http://azureus.sourceforge.net

- Azureus is one of the BitTorrent clients written in pure Java.
- BitTorrent is designed to serve files that can be referenced from known keys
- Downloaders can sharing a file while they're still downloading it.

e-Macao-16-3-160

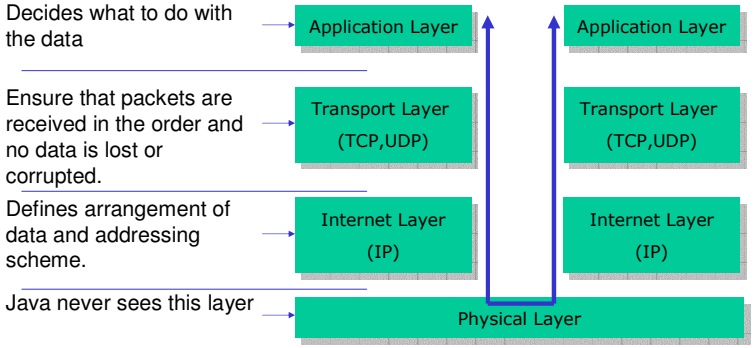
Concepts for network program

Important concepts needed for writing network program in Java

1) Communication protocols: TCP and UDP
2) Ports and Internet Addresses
3) Sockets
4) Uniform Resource Locator (URL)
5) Uniform Resource Identifier (URI)
6) Streams and Threads (Covered in previous sessions)
7) Classes in java.net and java.io packages

e-Macao-16-3-161

Network Layers



e-Macao-16-3-162

TCP and UDP

Java only supports TCP (Transmission Control Protocol), UDP (User Datagram Protocol) and application layer protocols built on top of these.

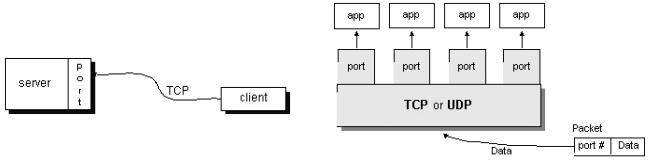
Characteristics of TCP and UDP :

TCP	UDP
• Provides the ability to acknowledge receipt of IP packets and request retransmission of lost or corrupted packets.	• Is an unreliable protocol that does not guarantee that packets will arrive at their destination.
• Allows the received packets to be put back together in the order they were sent.	• Allow the receiver to detect corrupted packets but does not guarantee that packets are delivered in the correct order.
• Requires a lot of overhead.	• Requires less overhead and faster.
• Supported classes in java.net : URL, URLConnection, Socket, and ServerSocket	• Supported classes in java.net : DatagramPacket, DatagramSocket, and MulticastSocket

e-Macao-16-3-163

Ports

Each port from the server can be treated by the clients as a separate machine offering different services.



Port numbers are represented by 16-bit numbers. (0 to 65,535)

The port numbers ranging from 0 - 1023 reserved for use by well-known services such as HTTP and FTP and other system services.

e-Macao-16-3-164

Sockets

You can reach required service via its network and port IDs. what then?

a) If you are a client
   - you need an API that will allow you to send messages to that service and read replies from it

b) If you are a server
   - you need to be able to create a port and listen at it.
   - you need to be able to read the message comes in and reply to it.

The **Socket** and **ServerSocket** are the Java client and server classes to do this.

e-Macao-16-3-165

Example : Sending Email 1

E-mail is sent by socket communication with port 25 on a computer system.

open a socket connected to port 25 on some system, and speak “mail protocol” to the daemon at the other end.

e-Macao-16-3-166

Example : Sending Email 2

```
import java.io.*;
import java.net.*;
public class SendEmail {
  public static void main(String args[]) throws
   IOException {
     Socket sock;
     DataInputStream dis;
     BufferedReader br;
     PrintStream ps;
     System.out.println(">>> Connect
                 mailhost.iist.unu.edu");
     sock = new Socket("mailhost.iist.unu.edu", 25);
     dis = new DataInputStream(sock.getInputStream());
```

e-Macao-16-3-167

Example : Sending Email 3

```
    br = new BufferedReader (new
       InputStreamReader(dis));
    ps = new PrintStream( sock.getOutputStream());
    System.out.println( br.readLine() );
    System.out.println(">>> Hello UNU/IISTT");
    ps.println("Hello UNU/IIST");
    System.out.println( br.readLine() );
    System.out.println(">>> Mail From:
     oluotes@yahoo.com");
    ps.println("MAIL FROM:milton_hm@hotmail.com");
    System.out.println( br.readLine() );
    String Addressee= "milton@iist.unu.edu";
```

e-Macao-16-3-168

Example : Sending Email 4

```
    System.out.println(">>> Rcpt to: " + Addressee);
    ps.println("RCPT TO: " + Addressee );
    System.out.println( br.readLine() );
    System.out.println(">>> Send \"data\"");
    ps.println("DATA");
    System.out.println( br.readLine() );
    System.out.println(">>>>>>>>>>");
    System.out.println(">>> This is the message\n that
     Java sent");
    System.out.println(">>> We are testing Socket
     Programming");
    System.out.println(">>> in eMacao Training
     program.");
    System.out.println(">>>>>>>>>>");
```

e-Macao-16-3-169

Example : Sending Email 5

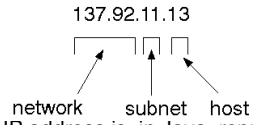
```
    ps.println("This is the message\n that Java sent");
     ps.println("We are testing Socket Programming");
     ps.println("in eMacao Training program.");
     System.out.println(">>> .");
     ps.println(".");
     System.out.println( br.readLine() );
     System.out.println(">>> QUIT");
     ps.println("QUIT");
     System.out.println( br.readLine() );
     ps.flush();
     sock.close();
   }
}
```

e-Macao-16-3-170

Internet Addressing

Internet address (IP address) is a unique number for identifying a device connected to the Internet.

The current standard is IPv4 which are four bytes long.



The hostname and IP address is, in Java, represented by java.net.InetAddress.

InetAddress is used by many other networking classes, including Socket, ServerSocket, URL, DatagramSocket, DatagramPacket, and more.

e-Macao-16-3-171

Example

The following program will print out the IP address of the www.iist.unu.edu

```
import java.net.*;
public class IISTByName {
  public static void main (String[] args) {
   try { InetAddress address =
     InetAddress.getByName   ("www.iist.unu.edu") ;
     System.out.println(address);
   } catch (UnknownHostException ex) {
       System.out.println("Could not find
          www.iist.unu.edu ");
   }
  }
}
```

e-Macao-16-3-172

InetAddress methods

Useful methods:

a) static InetAddress getByName(String host)
b) static InetAddress getLocalHost()
c) String getHostAddress(); // in dotted form
d) String getHostName();

e-Macao-16-3-173

The URL Class 1

The `java.net.URL` is an abstraction of a Uniform Resource Locator (URL).

URLs are composed of five pieces:

1. The scheme, also known as the protocol
2. The authority
3. The path
4. The query string
5. The fragment identifier, also known as the section or ref

<scheme>://<authority><path>?<query>#<fragment>

e-Macao-16-3-174

The URL Class 2

For example, given the URL :
http://www.ibiblio.org/javafaq/javabooks/index.html?isbn=123456789#toc

1. scheme : http
2. authority : www.ibiblio.org
3. path : /javafaq/books/javabooks/index.html
4. query string : isbn=123456789
5. fragment identifier : toc

e-Macao-16-3-175

The URL Class 3

The authority may further be divided into the user info, the host, and the port.

For example, in the URL http://admin@www.blackstar.com:8080/

1. user info : admin
2. host : www.blackstar.com
3. port : 8080

e-Macao-16-3-176

The URL Class 4

The java.net.URL class provides static methods for getting the above mentioned information:

a) getFile( )
b) getHost()
c) getPort( )
d) getProtocol()
e) getRef()
f) getQuery( )
g) getPath( )
h) getUserInfo( )
i) getAuthority( )

e-Macao-16-3-177

The URL Class 5

Unlike the InetAddress objects, you can construct instances of java.net.URL using one of its six constructors.

```
1) public URL(String url) throws
     MalformedURLException
2) public URL(String protocol, String hostname,
   String file) throws MalformedURLException
3) public URL(String protocol, String host, int port,
   String file) throws MalformedURLException
4) public URL(URL base, String relative) throws
   MalformedURLException
```

e-Macao-16-3-178

The URL Class 6

5) `public URL(URL base, String relative, URLStreamHandler handler) // 1.2 throws MalformedURLException`

6) `public URL(String protocol, String host, int port, String file, // 1.2 URLStreamHandler handler) throws MalformedURLException`

e-Macao-16-3-179

Example 1

The following program will test the protocol supported by the browser:

```
import java.net.*;
public class ProtocolTester {
    public static void testProtocol(String url) {
     try {
       URL u = new URL(url);
       System.out.println(u.getProtocol( ) + " is
        supported");
     }
     catch (MalformedURLException ex) {
       String protocol =
        url.substring(0, url.indexOf(':'));
       System.out.println(protocol + " is not
        supported");
     }
   }
}
```

e-Macao-16-3-180

Example 2

You can test it with the following Tester:

```
public class Tester {
  public static void main(String[] args) {
    ProtocolTester.testProtocol("http://www.adc.org");
    ProtocolTester.testProtocol("https://www.amazon.co
    m/exec/obidos/order2/");
    ProtocolTester.testProtocol("ftp://metalab.unc.edu
    /pub/languages/java/javafaq/");
```

e-Macao-16-3-181

Lab Work: URL 1

1) Write a program that will split the input URL into corresponding parts.

Given: java URLSplitter

```
http://www.unu.iist.edu/demoweb/html-
primer.html#A1.3.3.3
```

Output:

```
The output will be:
The URL is http://www.unu.iist.edu/demoweb/html-
primer.html#A1.3.3.3
The scheme is http
The user info is null
The host is www.unu.iist.edu
The port is -1

The path is /demoweb/html-primer.html
The ref is A1.3.3.3
The query string is null
```

e-Macao-16-3-182

Lab Work: URL 2

Try to test your program using the following URL:

```
ftp://mp3:mp3@138.247.121.61:21000/c%3a/
http://www.oreilly.com
http://www.ibiblio.org/nywc/compositions.phtml?categor
y=Piano
http://admin@www.blackstar.com:8080/
```

e-Macao-16-3-183

Retrieving Data from a URL

The URL class provides methods for retrieving data from a URL:

```
public InputStream openStream( ) throws IOException
public URLConnection openConnection( ) throws
IOException
public URLConnection openConnection(Proxy proxy)
throws IOException // 1.5
public Object getContent( ) throws IOException
public Object getContent(Class[] classes) throws
IOException // 1.3
```

e-Macao-16-3-184

Retrieving Data from a URL 1

Procedure to use the methods:

1) Create an URL object

   e.g. `URL u = new URL("http://www.iist.unu.edu");`

2) Open an InputStream object directly from the URL object

   e.g. `InputStream in = u.openStream( );`

3) Or open an URLConnection object from the URL object and then get an InputStream object from the URLConnection object .

   e.g.
   ```
   URLConnection uc = u.openConnection( );
      InputStream in = uc.getInputStream( );
   ```

4) In either case, you will have an InputStream. What's followed is the normal I/O procedure for getting data.

5) Don't forget to put the try catch block for catching the `MalformedURLException` and `IOException.`

e-Macao-16-3-185

Retrieving Data from a URL 2

What is the difference between using the openStream and openConnection method?

1) openStream method only give you the access to the raw data and cannot detect the encoding information.
2) openConnection method opens a socket to the specified URL and returns a URLConnection object.
3) The URLConnection object gives you access to everything sent by the server. You can access all the metadata specified by the protocol such as the scheme. The URLConnection class also lets you write data to as well as read from a URL.

e-Macao-16-3-186

Retrieving Data from a URL 3

The following methods are used to access the header fields and the contents after the connection is made to the remote object:

1)getContent

2)getHeaderField

3)getInputStream

4)getOutputStream

e-Macao-16-3-187

Retrieving Data from a URL 4

Certain header fields are accessed frequently. The methods:

1) getContentEncoding
2) getContentLength
3) getContentType
4) getDate
5) getExpiration
6) getLastModifed

e-Macao-16-3-188

Example : Reading from URL 1

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
  ublic static void main(String[] args) throws
  Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
        new InputStreamReader (yc.getInputStream()));
        String inputLine;
```

e-Macao-16-3-189

Example : Reading from URL 2

```
while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

e-Macao-16-3-190

Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is an abstraction of a URL.

Most URIs used in practice are URLs, but most specifications and standards such as XML are defined in terms of URIs

In Java 1.4 and later, URIs are represented by the `java.net.URI` class.

you should use the URL class when you want to download the content of a URL and the URI class when you want to use the URI for identification rather than retrieval.

When you need to do both, you may convert from a URI to a URL with the toURL( ) method, and in Java 1.5 you can also convert from a URL to a URI using the toURI( ) method of the URL class.

e-Macao-16-3-191

Uniform Resource Identifier (URI)

A URI reference has up to three parts: a scheme, a scheme-specific part, and a fragment identifier. The general format is:
scheme:*scheme-specific-part*:*fragment* .

Getter methods:

1) public String getScheme( )
2) public String getSchemeSpecificPart( )
3) public String getRawSchemeSpecificPart( )
4) public String getFragment( )
5) public String getRawFragment( )

e-Macao-16-3-192

Lab Work: URI

1) Write a program that will split the input URI into corresponding parts.
2) List the methods you can get form the URI class using the javadoc.

e-Macao-16-3-193

Networking Examples

Now you have the basic concepts for different components for Java networking. Let's try to start some simple experiments.

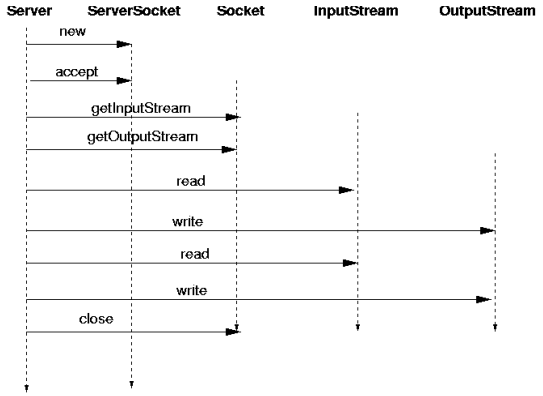
e-Macao-16-3-194

A Simple Example

e-Macao-16-3-195

Create a Server

How to create a server?



e-Macao-16-3-196

Echo Server 1

```
import java.io.*;
import java.net.*;
public class EchoServer {
  public static int MYECHOPORT = 8189;
    public static void main(String argv[]) {
      ServerSocket s = null;
      try {
          s = new ServerSocket(MYECHOPORT);
      } catch(IOException e) {
          System.out.println(e);
          System.exit(1);
      }
```

e-Macao-16-3-197

Echo Server 2

```
while (true) {
          Socket incoming = null;
          try {
            incoming = s.accept();
          } catch(IOException e) {
            System.out.println(e);
            continue;
          }
          try {
            incoming.setSoTimeout(10000); //10 seconds
          } catch(SocketException e) {
            e.printStackTrace();
          }
```

e-Macao-16-3-198

Echo Server 3

```
try {       handleSocket(incoming);
          } catch(InterruptedIOException e) {
            System.out.println("Time expired " + e);
          } catch(IOException e) {
            System.out.println(e);
          }
          try {
            incoming.close();
          } catch(IOException e) {
            // ignore
          }
      }
    }
```

e-Macao-16-3-199

Echo Server 4

```
public static void handleSocket(Socket incoming)
      throws IOException {
      BufferedReader reader =
          new BufferedReader(new InputStreamReader(
                        incoming.getInputStream()));
      PrintStream out =
          new PrintStream(incoming.getOutputStream());
      out.println("Hello. Enter BYE to exit");

      boolean done = false;
      while ( ! done) {
          String str = reader.readLine();
```

e-Macao-16-3-200

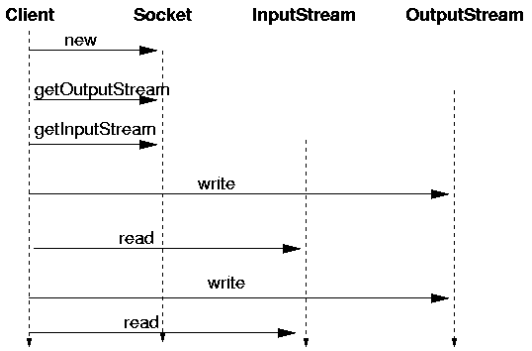
Echo Server 5

```
if (str == null) {
            done = true;
            System.out.println("Null received");
          }
          else {
            out.println("Echo: " + str);
            if (str.trim().equals("BYE"))
                done = true;
          }
            }
      incoming.close();
    }
}
```

e-Macao-16-3-201

Create a Client

How to create a client connected to a sever?



e-Macao-16-3-202

Echo Client 1

```
import java.io.*;
import java.net.*;

public class EchoClient {
  public static void main(String[] args) throws
      IOException {

            Socket echoSocket = null;
            PrintWriter out = null;
            BufferedReader in = null;
            BufferedReader stdIn = null;
```

e-Macao-16-3-203

Echo Client 2

```
try {
      echoSocket = new Socket("localhost", 8189);
      out = new
      PrintWriter(echoSocket.getOutputStream(), true);
      in = new BufferedReader(new
      InputStreamReader(echoSocket.getInputStream()));
      System.out.println (in.readLine());
      } catch (UnknownHostException e) {
            System.err.println("Don't know about host.");
            System.exit(1);
      } catch (IOException e) {
           System.err.println("Couldn't get I/O for "
            + "the connection to server.");
```

e-Macao-16-3-204

Echo Client 3

```
      System.exit(1);
}
try {
      stdIn = new BufferedReader(new InputStreamReader
            (System.in));
      String userInput;
      while ((userInput = stdIn.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
       }
      } catch (SocketException e) {
                  System.err.println("Socket closed");
      }
```

e-Macao-16-3-205

Echo Client 4

```
finally {
                  if (out != null)
                        out.close();
                  if (in != null)
                        in.close();
                  if (stdIn != null)
                        stdIn.close();
                  if (echoSocket != null)
                        echoSocket.close();
            }
      }
}
```

e-Macao-16-3-206

Lab Work: Chatting Program

1) Modify the previous Echo Sever and Echo Client examples to create a server-client chatting program.

   Hints:

   a) You need to create a server which will wait for the client to establish the connection. Then it will print a statement to the client's console notifying that a connection is made.

   b) Once the connection is made, both the server and client will be able read message from the counterpart and write message to it.

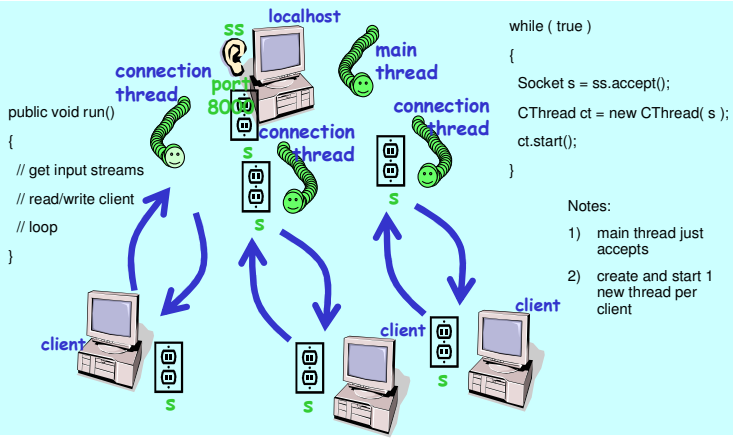
e-Macao-16-3-207

Exercise : Multiple Clients 1

1) Please modify the previous lab work. you need to make your server to be able to talk to multiple clients connected the server.

e-Macao-16-3-208

Exercise : Multiple Clients 2



e-Macao-16-3-209

Secure Sockets 1

Starting from JDK 1.4, Java Secure Sockets Extension (JSSE) is part of the standard distribution.

JSSE uses the Secure Sockets Layer (SSL) Version 3 and Transport Layer Security (TLS) protocols and their associated algorithms to secure network communications.

JSSE abstracts all the low-level details such as keys exchange, authentication, and data encryption. All you have to do is to send your data over the streams from the secured sockets obtained.

e-Macao-16-3-210

Secure Sockets 2

The Java Secure Socket Extension is divided into four packages:

1) javax.net.ssl :The abstract classes that define Java's API for secure network communication.
2) javax.net :The abstract socket factory classes used instead of constructors to create secure sockets.
3) javax.security.cert : A minimal set of classes for handling public key certificates that's needed for SSL in Java 1.1. (In Java 1.2 and later, the java.security.cert package should be used instead.)
4) com.sun.net.ssl : The concrete classes that implement the encryption algorithms and protocols in Sun's reference implementation of the JSSE.

e-Macao-16-3-211

Secure Client Sockets 1

Procedures to create a secure client socket:

1) get an instance of `SocketFactory` by invoking the static `SSLSocketFactory.getDefault()` method. e.g.

```
SocketFactory sf = SSLSocketFactory.getDefault();
```

2) use one of these five overloaded `createSocket( )` methods to build an SSLSocket:

```
1.public abstract Socket createSocket(String host,
  int port) throws IOException,
  UnknownHostException
2.public abstract Socket createSocket(InetAddress
  host, int port) throws IOException
```

e-Macao-16-3-212

Secure Client Sockets 2

```
3.public abstract Socket createSocket(String host,
  int port, InetAddress interface, int localPort)
  throws IOException, UnknownHostException
4.public abstract Socket createSocket(InetAddress
  host, int port, InetAddress interface, int
  localPort) throws IOException,
  UnknownHostException
5.public abstract Socket createSocket(Socket proxy,
  String host, int port, boolean autoClose) throws
  IOException
```

e-Macao-16-3-213

Secure Client Sockets 3

3) Once the socket has been created, you use it just like any other socket, through its getInputStream( ), getOutputStream(), and other methods.

   For example, if the following purchasing information is required to be sent over the network:

   a) Name: John Smith
   b) Product-ID: 67X-89
   c) Address: 1280 Deniston Blvd, NY NY 10003
   d) Card number: 4000-1234-5678-9017
   e) Expires: 08/05

   Using JSSE, the following code will do the work for you:

e-Macao-16-3-214

Secure Client Sockets 4

```
try {
   SSLSocketFactory factory
   = (SSLSocketFactory) SSLSocketFactory.getDefault( );
   Socket socket = factory.createSocket("localhost",
      7000);
   Writer out = new
      OutputStreamWriter(socket.getOutputStream( ),
      "ASCII");
   out.write("Name: John Smith\r\n");
```

e-Macao-16-3-215

Secure Client Sockets 5

```
   out.write("Product-ID: 67X-89\r\n");
   out.write("Address: 1280 Deniston Blvd, NY NY
      10003\r\n");
   out.write("Card number: 4000-1234-5678-9017\r\n");
   out.write("Expires: 08/05\r\n");
   out.flush( );
   out.close( );
   socket.close( );
} catch (IOException ex) {
     ex.printStackTrace( );
}
```

e-Macao-16-3-216

Configuring Secure Sockets

Methods are available for configuring how much and what kind of authentication and encryption is performed.

a) getSupportedCipherSuites() method tells you which combination of algorithms is available on a given socket

b) getEnabledCipherSuites() method tells you which suites this socket is willing to use

c) You can change the suites the client attempts to use via the setEnabledCipherSuites(String[] suites) method

- Sun's JDK 1.4 supports 23 cipher suites. For the list of the supported cipher suites, please check with JavaDoc.

d) There are still methods for handling handshaking and sessions, and I will open these for your further study.

e-Macao-16-3-217

Secure Server Sockets 1

Procedures to create a secure server socket:

1) get an instance of `ServerSocketFactory` by invoking the static `SSLSocketFactory.getDefault()` method. e.g.

```
ServerSocketFactory sf =
  SSLServerSocketFactory.getDefault();
```

2) use one of these three overloaded `createServerSocket( )` methods to build an SSLServerSocket:

```
1.public abstract ServerSocket
  createServerSocket(int port) throws IOException
2.public abstract ServerSocket
  createServerSocket(int port, int queueLength)
  throws IOException
3.public abstract ServerSocket
  createServerSocket(int port, int queueLength,
  InetAddress interface) throws IOException
```

e-Macao-16-3-218

Secure Server Sockets 2

3) Unlike creating the client socket, you need to do more to set up the encryption for the server socket.

   This setup varies between different JSSE implementations. In Sun's implementation, you may need to do the followings:

   1. Generate public keys and certificates using *keytool*.
   2. Pay money to have your certificates authenticated by a trusted third party such as Verisign.
   3. Create an SSLContext for the algorithm you'll use.
   4. Create a TrustManagerFactory for the source of certificate material you'll be using.

e-Macao-16-3-219

Secure Server Sockets 3

5. Create a KeyManagerFactory for the type of key material you'll be using.
6. Create a KeyStore object for the key and certificate database. (Sun's default is JKS.)
7. Fill the KeyStore object with keys and certificates; for instance, by loading them from the filesystem using the pass phrase they're encrypted with.
8. Initialize the KeyManagerFactory with the KeyStore and its pass phrase.
9. Initialize the context with the necessary key managers from the KeyManagerFactory, trust managers from the TrustManagerFactory, and a source of randomness. (The last two can be null if you're willing to accept the defaults.)

e-Macao-16-3-220

Lab Work: Secure Sockets 1

1) Generate public keys and certificates using *keytool*
   a) *D:\JAVA>**keytool -genkey -alias ourstore -keystore jnp3e.keys***
   b) Answer some questions and please remember the password you entered. You will need it later.
   c) A file jnp3e.keys will be generated and protected by the password you entered.
2) If you don't want to pay for the digital ID for experiment, you can use the verified keystore file called testkeys, protected with the password "passphrase", included in SUN's JSSE implementation package.
3) Create a class named SecureServer
4) Import the necessary packages.

e-Macao-16-3-221

Lab Work: Secure Sockets 2

5) Define variables:
   a) int PORT – default port number
   b) String ALGORITHM – algorithm for setting the SSLContext ("SSL")
   c) String KEYFILE – in our case, "keyfiles"
   d) String PASSWORD – in our case, "passphrase"
6) Create the context using SSLContext.getInstance(arg) method. You need to pass the variable ALGORITHM as argument.
7) As accepted the default, we don't need to create theTrustManagerFactory
8) Create the KeyManagerFactory using the static method KeyManagerFactory.getInstance (arg). The Sun implementation will need "SunX509" as argument.

e-Macao-16-3-222

Lab Work: Secure Sockets 3

9) Create the KeyStore using the static method KeyStore.getInstance(arg) Use "JKS" as argument.
10) For security, every key store is encrypted with a pass phrase that must be provided before we can load it from disk. The pass phrase is stored as a char[] array so it can be wiped from memory quickly rather than waiting for a garbage collector. Of course using a string literal here completely defeats that purpose.
11) Use the load method from the KeyStore to load the key file. Check the JavaDoc for method usage.
12) Use the init method from the KeyManagerFactory to initialize the KeyManagerFactory. Check the JavaDoc for method usage.
13) Use the init method from theSSLContext to initialize the context. Check the JavaDoc for method usage.

e-Macao-16-3-223

Lab Work: Secure Sockets 4

14) The left is similar to what we did for creating the secure client socket. So try yourself.
15) Test your server with the secure client we created.

e-Macao-16-3-224

New I/O (NIO) API

Java introduce the new I/O (NIO) API in v1.4.

New features:

a) Buffers for data of primitive types
b) Character-set encoders and decoders
c) A pattern-matching facility based on Perl-style regular expressions
d) Channels, a new primitive I/O abstraction
e) A file interface that supports locks and memory mapping
f) A multiplexed, non-blocking I/O facility for writing scalable servers

e-Macao-16-3-225

Why NIO?

Allow Java programmers to implement high-speed I/O.

NIO deals with data in blocks which can be much faster than processing data by the (streamed) byte.

e-Macao-16-3-226

Buffers

In the NIO library, all data is handled with buffers.

A buffer is essentially an array. Generally, it is an array of bytes, but other kinds of arrays can be used.

A buffer also provides structured access to data and also keeps track of the system's read/write processes.

Types:

a) ByteBuffer
b) CharBuffer
c) ShortBuffer
d) IntBuffer
e) LongBuffer·
f) FloatBuffer
g) DoubleBuffer

e-Macao-16-3-227

Channels

Channel is like a stream in original I/O.

You can read a buffer from and write a buffer to a channel.

Unlike streams, channels are bi-directional.

e-Macao-16-3-228

Read from a file

Codes for reading from a file:

```
FileInputStream fin = new
FileInputStream( "readandshow.txt" );
FileChannel fc = fin.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
fc.read( buffer );
```

e-Macao-16-3-229

Write to a file

Codes for writing to a file:

```
FileOutputStream fout = new
FileOutputStream( "writesomebytes.txt" );
FileChannel fc = fout.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
for (int i=0; i<message.length; ++i) {
buffer.put( message[i] );
}
buffer.flip();//prepares the buffer to have the newly-
            //read data written to another channel
fc.write( buffer );
```

e-Macao-16-3-230

Lab Work: NIO

1) Refer to the example, please use the NIO to create a program to write a String to a text file and store in your computer.
2) Read the text file back and print the content on the screen.
3) You may need to use the WritableByteChannel as following:

```
WritableByteChannel wbc =
            Channels.newChannel(System.out);
```

e-Macao-16-3-231

Server with NIO

Channels and buffers are really intended for server systems that need to process many simultaneous connections efficiently.

Handling servers requires the new selectors that allow the server to find all the connections that are ready to receive output or send input.

The following example will demonstrate the basics.

e-Macao-16-3-232

Example : Create NIO Server 1

Codes for creating a server with NIO:

```
ServerSocketChannel serverChannel =
ServerSocketChannel.open( );

ServerSocket ss = serverChannel.socket( );

// bind the socket to a specific port

ss.bind(new InetSocketAddress(port_no));

SocketChannel clientChannel = serverChannel.accept( );

//make the client channel non-blocking to allow the
//server to process multiple simultaneous connections:

clientChannel.configureBlocking(false);
```

e-Macao-16-3-233

Example : Create NIO Server 2

```
//make the ServerSocketChannel non-blocking. A non-
//blocking accept( ) returns null almost immediately
//if there are no incoming connections. Be sure to
//check for that

serverChannel.configureBlocking(false);

//create a Selector that enables the program to
//iterate over all the connections that are ready to
//be processed

Selector selector = Selector.open( );

//use the channel's register() method to register each
//channel with the selector that monitors it and
//specify the operation. Such as:

serverChannel.register(selector,
SelectionKey.OP_ACCEPT);
```

e-Macao-16-3-234

Example : Create NIO Server 3

```
SelectionKey key = clientChannel.register(selector,
SelectionKey.OP_WRITE);
//check whether anything is ready to be acted on, call
//the selector's select( ) method. For a long-running
//server, this normally goes in an infinite loop:
while (true) {
    selector.select ( );
// process selected keys...
//selectedKeys( ) method returns a java.util.Set
//containing one SelectionKey object for each ready
//channel
```

e-Macao-16-3-235

Example : Create NIO Server 4

```
Set readyKeys = selector.selectedKeys( );
Iterator iterator = readyKeys.iterator( );
while (iterator.hasNext( )) {
    SelectionKey key = (SelectionKey)
    (iterator.next( ));
     // Remove key from set
    iterator.remove( );

     // You can obtain the channel using the channel()
    //methods of the SelectionKey.
   if (key.isAcceptable( )) { ServerSocketChannel
     server = (ServerSocketChannel ) key.channel( );
```

e-Macao-16-3-236

Example : Create NIO Server 5

```
    // operate on the channel...
  else if (key.isWritable( )) { SocketChannel client =
              (SocketChannel ) key.channel( );
  // write data to client...
  }
}
```

e-Macao-16-3-237

User Datagram Protocol

The User Datagram Protocol (UDP) is an alternative protocol for sending data over IP

UDP is very quick, but not reliable.

Java's implementation of UDP is split into two classes: DatagramPacket and DatagramSocket.

The DatagramPacket class stuffs bytes of data into UDP packets called datagrams and lets you unstuff datagrams that you receive.

A DatagramSocket sends as well as receives UDP datagrams.

e-Macao-16-3-238

Constructors for DatagramPacket

For receiving datagrams:

a) `public DatagramPacket(byte[] buffer, int length)`

b) `public DatagramPacket(byte[] buffer, int offset, int length)`

For sending datagrams:

a) `public DatagramPacket(byte[] data, int length, InetAddress destination, int port)`

b) `public DatagramPacket(byte[] data, int offset, int length, InetAddress destination, int port) // Java 1.2`

c) `public DatagramPacket(byte[] data, int length, SocketAddress destination, int port) // Java 1.4`

d) `public DatagramPacket(byte[] data, int offset, int length, SocketAddress destination, int port) //Java 1.4`

e-Macao-16-3-239

Constructors for DatagramSocket

For socket bound to an anonymous port:

a) `public DatagramSocket( ) throws SocketException`

For socket listen for incoming datagrams on a particular port :

a) `public DatagramSocket(int port) throws SocketException`

Others constructors:

a) `public DatagramSocket(int port, InetAddress interface) throws SocketException`

b) `public DatagramSocket(SocketAddress interface) throws SocketException // Java 1.4`

c) `protected DatagramSocket(DatagramSocketImpl impl) throws SocketException // Java 1.4`

e-Macao-16-3-240

Sending and Receiving

After constructed the DatagramPacket, you can send and receive datagram from it :

a) `public void send(DatagramPacket dp) throws IOException`

b) `public void receive(DatagramPacket dp) throws IOException`

e-Macao-16-3-241

Lab Work: UDP

1) Please write a simple client which sends an empty UDP packet to a specified host and port and reads a response packet from the same host.

A.4. Database Connectivity

Database Connectivity

e-Macao-16-3-243

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-244

Overview

We are going to consider the following under this section:

1) Introduction
2) JDBC Overview
3) Information Processing

e-Macao-16-3-245

Introduction

Buried within the term "enterprise" is the idea of a business taken wholistically.

An enterprise solution identifies common problem domains within a business and provides a shared infrastructure for solving those problems.

Example:

If your business is running a bank, your individual branches may have different business cultures, but those cultures do not alter the fact that they all deal with customers and accounts. Looking at this business from an enterprise perspective means abstracting away from irrelevant differences in the way the individual branches do things, and instead approaching the business from their common ground. It does not mean dismissing meaningful distinctions, such as the need for bilingual support in Macao SAR.

e-Macao-16-3-246

Enterprise Systems: What?

Enterprise Systems are Information systems that support many or all of the various parts of a firm.

They can also refer to many mission-critical applications which are mainframe-based (also referred to as legacy systems).

Also known as enterprise-wide information systems.

Information systems that allow companies to integrate information across operations on a company-wide basis.

e-Macao-16-3-247

Enterprise Systems: Types

Enterprise systems are broadly categorized into two:

1) Relational - Relational Database Management Systems (RDBMS)

2) Non-Relational
   a) Non Relational Databases (Legacy Database Systems)
   b) Legacy Systems (Older systems like old Cobol Applications)
   c) Enterprise Resource Planning
   d) Customer Relationship Management (CRM)
   e) Supply Chain Management

e-Macao-16-3-248

Enterprise Systems Integration

Enterprise Systems Integration is normally defined as the bringing together of:

1) People,
2) Processes and
3) Information

to work together in an harmonized way, and supported by appropriate information systems.

It is also the bringing together of both old and new applications to achieve the overall goal of an organization.

e-Macao-16-3-249

Reasons for Integration

There are many reasons why Enterprise information systems need to be integrated. Some are stated below:

1) need to persist and retrieve information from data repositories.

2) need to leverage existing systems and resources while adopting and developing new technologies and architectures

3) concern over the past years, that the mainframe was going away and that all legacy applications would be scrapped and completely rewritten.

e-Macao-16-3-250

Java Integration Mechanisms

Java provides some technologies to integrate Enterprise systems:

1) Java Database connectivity (JDBC) for connecting applications to relational Database systems

2) JavaIDL and Java Connector Architecture (JCA) for connecting to Non-Relational systems

The focus of this section is JDBC.

JavaIDL will be addressed later in the course while JCA will be addressed sometimes in the training.

e-Macao-16-3-251

Relational Database 1

Programming is all about data processing; data is central to everything you do with a computer.

Databases, like filesystems are nothing more than specialized tools for data storage.

Filesystems are good for storing and retrieving a single volume of information associated with a single virtual location.

In other words, when you want to save a WordPerfect document, a filesystem allows you to associate it with a location in a directory tree for easy retrieval later.

e-Macao-16-3-252

Relational Database 2

Databases provide applications with a more powerful data storage and retrieval system based on mathematical theories about data devised by Dr. E. F. Codd.

Conceptually, a relational database can be pictured as a set of spreadsheets in which rows from one spreadsheet can be related to rows from another.

Each spreadsheet in a database is called a table. As with a spreadsheet, a table is made up of rows and columns.

A database engine is a process instance of the software accessing your database. For example Oracle, mySQL, Sybase etc

Database engines use a standard query language to retrieve information from databases and is called Structured Query Language (SQL).

e-Macao-16-3-253

SQL

SQL is not much like any programming language you might be familiar with.

Instead, it is more of a structured English for talking to a database.

Characteristics:

1) SQL keywords are case-insensitive

2) table and column names may or may not be case-insensitive depending on your database engine

3) the space between words in a SQL statement is unimportant

4) have a newline after each word, several spaces, or just a single space

e-Macao-16-3-254

SQL Usage

With SQL you can ask the following question:

1) How do you get the data into the database?
2) And how do you get it out once it is in there?

Much of the simplest database access comes in the form of equally simple SQL statements.

Some of these commands are:

1) `Create`
2) `Insert`
3) `Select`
4) `Update`
5) `Delete`

e-Macao-16-3-255

Create Statement

SQL `CREATE` statement handles the creation of database entities.

The major database engines provide GUI utilities that allow you to create tables without issuing any SQL.

Uses:

1) To create database

Syntax:

```
CREATE DATABASE database_name
```

2) To create tables

Syntax:

```
CREATE TABLE table_name (
column_name column_type column_modifiers,
...,
column_name column_type column_modifiers)
```

e-Macao-16-3-256

Lab Work: Create Statement

1) Open the console and type

```
C:\mysql\bin>mysql -u root
```

2) Test your connection by typing

```
mysql>show databases;
```

3) Create `emacaoTraining` database

```
mysql>create database emacaoTraining;
```

4) Change to the database

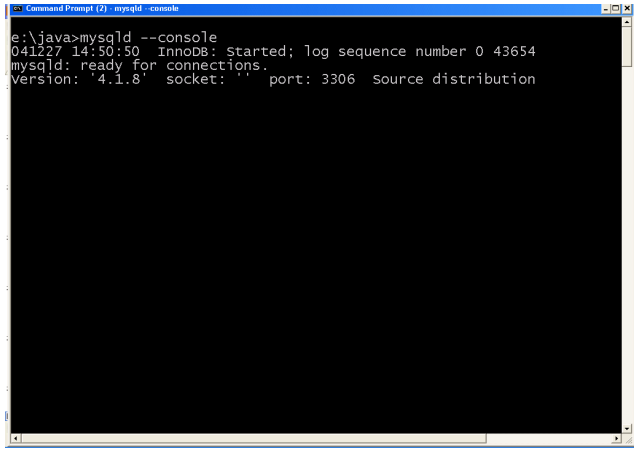
```
mysql>Use emacaoTraining;
```

5) Create license table

```
mysql>create table license (id int, name
varchar(40), sex varchar(10), date varchar(12),
licenseType varchar(10));
```

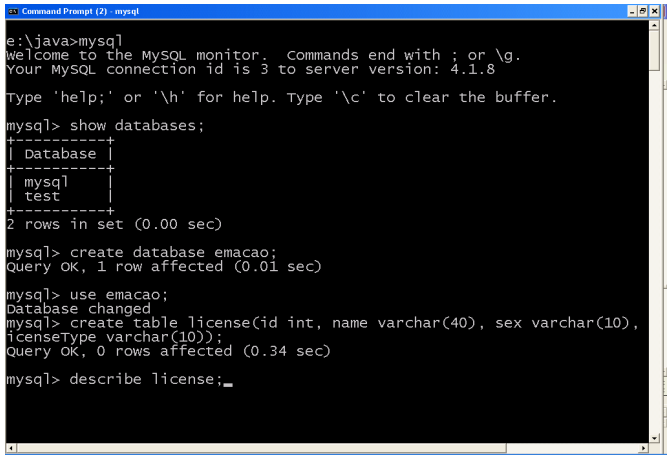
e-Macao-16-3-257

Lab Work: Console 1



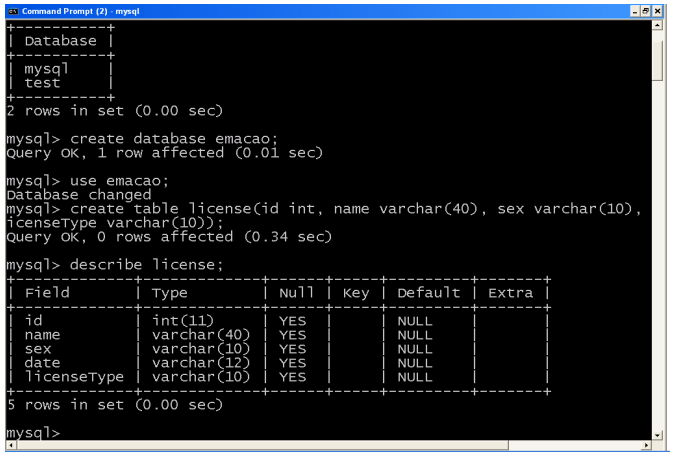
e-Macao-16-3-258

Lab Work: Console 2



e-Macao-16-3-259

Lab Work: Console 3



e-Macao-16-3-260

Insert Statement

With the tables in place, you use the INSERT statement to add data to them.

Its form is:

```
INSERT INTO table_name(column_name, ..., column_name)
VALUES (value, ..., value)
```

The first column name matches to the first value you specify, the second column name to the second value you specify, and so on for as many columns as you are inserting.

If you fail to specify a value for a column that is marked as `NOT NULL`, you will get an error on insert.

e-Macao-16-3-261

Lab Work: Insert Statement

1) Insert the following records into the license table.

a) `Id = 123`
`Name = Chong Gabriel`
`Sex = male`
`Date = 27/12/2004`
`LicenceType = Export`

b) `Id = 124`
`Name = martins Gabriel`
`Sex = Female`
`Date = 23/12/2004`
`LicenceType = Import`

e-Macao-16-3-262

Update Statement

The `UPDATE` statement enables you to modify data that you previously inserted into the database.

Its form is:

```
UPDATE table_name
SET column_name = value,
...,
column_name = value
WHERE column_name = value
```

This statement introduces the `WHERE` clause. It is used to help identify one or more rows in the database.

e-Macao-16-3-263

Lab Work: Update Statement

1) Change the `ID` and `Name` of the record with ID `124` to `126` and `Martins Leo Gabriel` respectively.

e-Macao-16-3-264

Select Statement

The most common SQL command you will use is the `SELECT` statement.

It allows you to select specific rows from the database based on search criteria.

It takes the following form:

```
SELECT column_name, ..., column_name
FROM table_name
WHERE column_name = value
```

e-Macao-16-3-265

Lab Work: Select Statement

1) Retrieve all records from the table.
2) Retrieve all records from the table where `ID` is `126`.

e-Macao-16-3-266

Delete Statement

The `DELETE` command looks a lot like the `UPDATE` statement.

Its syntax is:

```
DELETE FROM table_name WHERE column_name = value
```

Instead of changing particular values in the row, DELETE removes the entire row from the table.

e-Macao-16-3-267

Lab Work: Delete Statement

1) Remove all records from the table where `ID` is `125`.
2) Retrieve all records from the table
3) Remove all records from the table.

e-Macao-16-3-268

Database Programming

Database programming has traditionally been a technological Tower of Babel.

You are faced with dozens of available database products, and each one talks to your applications in its own private language.

If your application needs to talk to a new database engine, you have to teach it (and yourself) a new language.

As Java programmers, however, you should not worry about such translation issues.

Java is supposed to bring you the ability to "write once, compile once, and run anywhere," so it should bring it to you with database programming, as well.

e-Macao-16-3-269

JDBC Overview

JDBC API is a set of interfaces designed to insulate a database application developer from a specific database vendor.

It enables the developer to concentrate on writing the application - making sure that queries to the database are correct and that the data is manipulated as designed.

Sun developed a single API for database access—JDBC.

Three main design goals:

1) JDBC should be a SQL-level API.
2) JDBC should capitalize on the experience of existing database APIs.
3) JDBC should be simple.

e-Macao-16-3-270

JDBC and Developer

What does JDBC provide the developer?

1) the developer can write an application using the interface names and methods described in the API, regardless of how they were implemented in the driver

2) the developer writes an application using the interfaces described in the API as though they are actual class implementations

3) the driver vendor provides a class implementation of every interface in the API so that when an interface method is used, it is actually referring to an object instance of a class that implemented the interface.

e-Macao-16-3-271

JDBC and Driver Vendors

What do the driver vendors provide?

Driver vendors provide implementations of JDBC interfaces.

The JDBC API also enables developers to pass any string directly to the driver.

This makes it possible for developers to make use of custom features of their database without requiring that the application use ANSI SQL

With JBDC you can :

1) establish a connection with a database or access any tabular data source
2) send SQL statement
3) process the results

e-Macao-16-3-272

JDBC Structure

JDBC accomplishes its goals through a set of Java interfaces, each implemented differently by individual vendors.

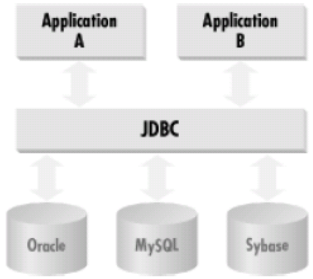
The set of classes that implement the JDBC interfaces for a particular database engine is called a JDBC driver.

In building a database application, you do not have to think about the implementation of these underlying classes at all.

The whole point of JDBC is to hide the specifics of each database and let you worry about just your application.

e-Macao-16-3-273

JDBC Architecture



e-Macao-16-3-274

JDBC Driver Type 1

JDBC Driver fit into one of the following:

1) Type 1:JDBC-ODBC Bridge plus ODBC Driver

2) Type 2:A native API partly Java technology-enabled

3) Type 3:Pure Java Driver for Database Middleware

4) Type 4:Direct-to-Database Pure Java Driver

e-Macao-16-3-275

Type 1: JDBC-ODBC Bridge 1

Type 1: JDBC-ODBC Bridge provides JDBC access via one or more Open Database Connectivity (ODBC) drivers.

Advantage:

1) a good approach for learning JDBC

2) may be useful for companies that already have ODBC drivers installed on each client machine

3) may be the only way to gain access to some low-end desktop databases

e-Macao-16-3-276

Type 1: JDBC-ODBC Bridge 2

Disadvantage:

1) Not for large-scale applications. Performance suffers because there's some overhead associated with the translation work to go from JDBC to ODBC.

2) doesn't support all the features of Java

3) user is limited by the functionality of the underlying ODBC driver

e-Macao-16-3-277

Type 2: Partial Java driver 1

Converts calls to the JDBC API into calls that connect to the client machine's application programming interface for a specific database, such as IBM, Informix, Oracle or Sybase.

Advantage:

1) Performance is better than that of Type 1, in part because the Type 2 driver contains compiled code that is optimized for the back-end database server's operating system.

e-Macao-16-3-278

Type 2: Partial Java driver 2

Disadvantage:

1) user needs to make sure the JDBC driver of the database vendor is loaded onto each client machine

2) must have compiled code for every operating system that the application will run on

3) best use is for controlled environments, such as an intranet

e-Macao-16-3-279

Type 3: Pure Java Middleware

Type 3: Pure Java driver for database middleware translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software.

Advantage:

1) used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them

2) Server-based, so no need for JDBC driver code on client machine

3) the back-end server component is optimized for the operating system that the database is running on

Disadvantage:

1) Needs some database-specific code on the middleware server.

e-Macao-16-3-280

Type 4: Direct-to-database Pure

Type 4: Direct-to-database pure Java driver converts JDBC calls into packets that are sent over the network in the proprietary format used by the specific database. Allows a direct call from the client machine to the database.

Advantage:

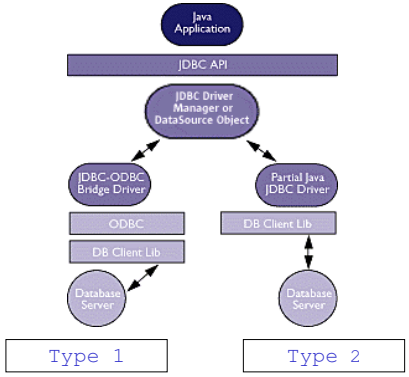
1) No need to install special software on client or server. Can be downloaded dynamically.

Disadvantage:

1) not optimized for server operating system, so the driver can't take advantage of operating system features

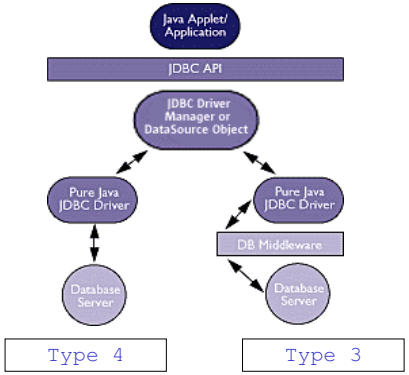
e-Macao-16-3-281

JDBC Driver Type 2



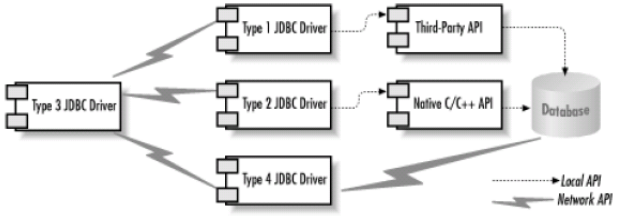
e-Macao-16-3-282

JDBC Driver Type 3



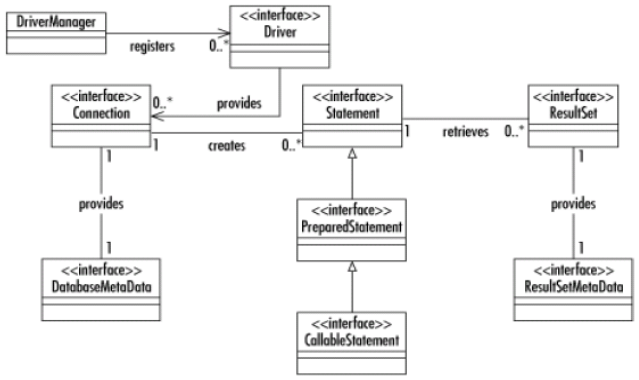
e-Macao-16-3-283

JDBC Drivers



e-Macao-16-3-284

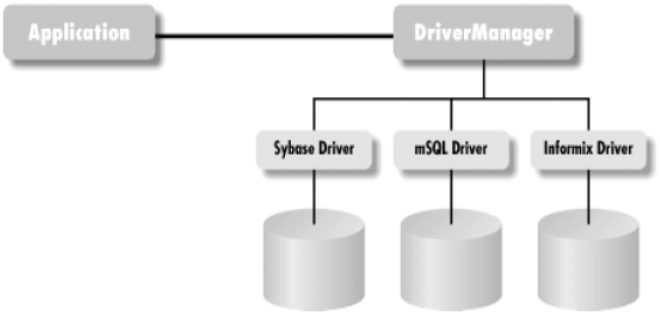
JDBC Class Diagram



e-Macao-16-3-285

Connecting to Database

JDBC shields an application from the specifics of individual database implementation.



e-Macao-16-3-286

Connection Troubles

The JDBC Connection process is the most difficult part of JDBC to get right.

There are generally two basic connection problems:

1) Connection fails with the message "Class not found"
   Solution: Set your JDBC driver in your `CLASSPATH`

2) Connection fails with the message "Driver not found"
   Solution: register the JDBC driver with the `DriverManager` class

e-Macao-16-3-287

Connection Process 1

When you write a Java database applet or application, the only driver-specific information JDBC requires from you is the database URL.

You can even have your application derive the URL at runtime - based on user input or applet parameters.

What happens when the URL and whatever properties the JDBC driver requires (generally a user ID and password) is passed?

1) the application will first request a `java.sql.Connection` implementation from the `DriverManager`

2) the `DriverManager` in turn will search through all of the known `java.sql.Driver` implementations for the one that connects with the URL you provided

e-Macao-16-3-288

Connection Process 2

3) if it exhausts all the implementations without finding a match, it throws an exception back to the application

4) once a `Driver` recognizes the URL, it creates a database connection using the properties specified

5) it then provides the `DriverManager` with a `java.sql.Connection` implementation representing that database connection

6) the `DriverManager` then passes that `Connection` object back to the application

7) the entire database connection process is handled by these two lines

```
Connection con = null;
con = DriverManager.getConnection(url, uid, password);
```

e-Macao-16-3-289

Connection Process 3

How does the JDBC `DriverManager` learn about a new driver implementation?

1) the `DriverManager` actually keeps a list of classes that implement the `java.sql.Driver` interface

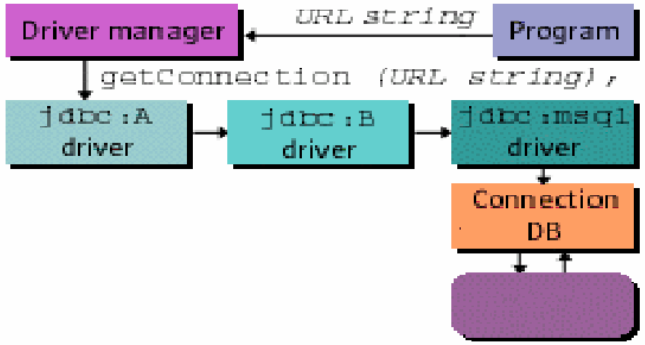
2) `Driver` implementations has to be registered for any potential database drivers it might require with the `DriverManager`

3) The act of instantiating a `Driver` class thus enters it in the `DriverManager`'s list

4) The process is called Driver Loading

e-Macao-16-3-290

Connection Process 4



e-Macao-16-3-291

Loading JDBC Drivers

There are three basic ways of loading the drivers:

1) explicitly call new to load your driver's implementation of `Driver`

2) use the `jdbc.drivers` property

```
>java -Djdbc.drivers=jdbc.odbc.JdbcOdbcDriver queryDB
```

3) load the class using `Class.forName`

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

e-Macao-16-3-292

Class for Creating a Connection 1

A class and two Interfaces are used for creating a connection to a database:

1) `java.sql.Driver`
   a) unless you are writing your own JDBC implementation, you should never have to deal with this class from your application

   b) a launching point for database connectivity by responding to `DriverManager` connection requests and providing information about the implementation in question

e-Macao-16-3-293

Class for Creating a Connection 2

2) `java.sql.DriverManager`
   a) Its main responsibility is to maintain a list of `Driver` implementations and present an application with one that matches a requested URL.
   b) has two methods `registerDriver()` and `deregisterDriver()`
   c) the methods allow `Driver` implementation to register and unregister itself with the `DriverManager`
   d) You can get an enumeration of registered drivers through the `getDrivers()` method

3) `java.sql.Connection`
   a) The Connection class represents a single logical database connection.

e-Macao-16-3-294

Example: Simple Connection 1

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class SimpleConnection {
 static public void main(String args[]) {
  Connection connection = null;
  // Process the command line
  if( args.length != 4 ) {
   System.out.print("Syntax: java SimpleConnection ");
   System.out.println("DRIVER URL UID PASSWORD");
   return;
  }
```

e-Macao-16-3-295

Example: Simple Connection 2

```
try { // load the driver
 Class.forName(args[0]).newInstance( );
}catch( Exception e ) {
 e.printStackTrace( );
 return;
}
try {
 connection = DriverManager.getConnection(args[1],
                                  args[2], args[3]);
 System.out.println("Connection successful!");
 // Do whatever queries or updates you want here!!!
}catch( SQLException e ) {
 e.printStackTrace( );
}
```

e-Macao-16-3-296

Example: Simple Connection 3

```
`finally {
   if( connection != null ) {
    try { connection.close( );
    }catch( SQLException e ) {
     e.printStackTrace( );
    }
   }
  }
 }
}
```

e-Macao-16-3-297

Lab Work: Creating Connection

1) Open `LicenseApp.java` file stored on the server
2) The program creates an interface for you to enter your license information into the database you created. Study the code and understand what it does.
3) Import appropriate packages into the program. Locate `connectToDB()`
4) Write a code to connect to the database you have created in mySQL using the following parameters

```
url = "jdbc:mysql://localhost/emacao"
username = root, Password = ""
Driver = "com.mysql.jdbc.Driver"
```

Note: set your `classpath` to the jar files provided along with the code. Print out the `Connection` object.

e-Macao-16-3-298

Database Access

The most basic kind of database access involves writing

1) updates- `INSERT`, `UPDATE`, or `DELETE`
2) queries – `SELECT`

With these you know ahead of time the type of statements you are sending to the database.

e-Macao-16-3-299

Database Access Steps

Accessing database involves:

1) creating a `Connection` object

2) generating implementation of `java.sql.Statement` tied to the database

3) use the statement to rollback or commit the statement object associated with that Connection

4) with the `Statement` object you can execute updates and queries

5) The result of executing queries and update is `java.sql.ResultSet`

6) `ResultSet` provides you with access to the data retrieved by a query.

e-Macao-16-3-300

Basic JDBC Classes

JDBC's most fundamental classes are :

1) `java.sql.Connection`
2) `java.sql.Statement`
3) `java.sql.ResultSet`

We have discussed (1), we now consider (2) and (3)

e-Macao-16-3-301

Statement

`Statement` class represents SQL statements.

It has three generic forms of statement execution methods:

1) `executeQuery(String query)`
   Usage: for any SQL calls that expect to return data from database

2) `executeUpdate(String query)`
   Usage: when SQL calls are not expected to return data from database
   It returns the number of row affected by `query`

3) `execute()`
   Usage: when you cannot determine whether SQL is an update or query
   return `true` if row is returned, use `getResultset()` to get the row
   otherwise returns `false`

e-Macao-16-3-302

Submitting a Query 1

Submitting a query involves

1) create a `Statement` object

```
try {
    Statement stmt = con.createStatement ();
} catch (SQLException e) {
    System.out.println (e.getMessage());
}
```

SQL exceptions occur when there is a database access error.

Errors are detected when a connection is broken or the database server goes down.

e-Macao-16-3-303

Submitting a Query 2

2) use the one the statement query method to submit the SQL statement to the database depending on the type of the SQL.

   JDBC does not attempt to interpret queries.

Example:

```
ResultSet rs = null;
rs = stmt.execteQuery("select * from license");
```

e-Macao-16-3-304

Example: Statement 1

```
import java.sql.*;
public class Update {
 public static void main(String args[]) {
  Connection connection = null;
  if( args.length != 2 ) {
   System.out.print("Syntax: <java Update [number]”);
   System.out.println(“[string]>");
   return;
  }
  try {
   String driver = "com.mysql.jdbc.Driver";
   Class.forName(driver).newInstance( );
   String url =“jdbc:mysql://localhost/emacao";
   con = DriverManager.getConnection(url, “root", "");
```

e-Macao-16-3-305

Example: Statement 2

```
   Statement s = con.createStatement( );
   String test_id = args[0];
   String test_val = args[1];
   int update_count =
   s.executeUpdate("INSERT INTO test (test_id, test_val)
    " + "VALUES(" + test_id + ", '" + test_val + "')");
   System.out.println(update_count + " rows inserted.");
   s.close( );
  }catch( Exception e ) {
   e.printStackTrace( );
  }
```

e-Macao-16-3-306

Example: Statement 3

```
  finally {
   if( con != null ) {
    try {
      con.close( );
    }catch( SQLException e ) {
     e.printStackTrace( );
    }
   }
  }
 }
}
```

e-Macao-16-3-307

Lab Work: Statement

1) Using the `LicenseApp.java` insert records into `license` table in `emacao` database

e-Macao-16-3-308

PreparedStatement

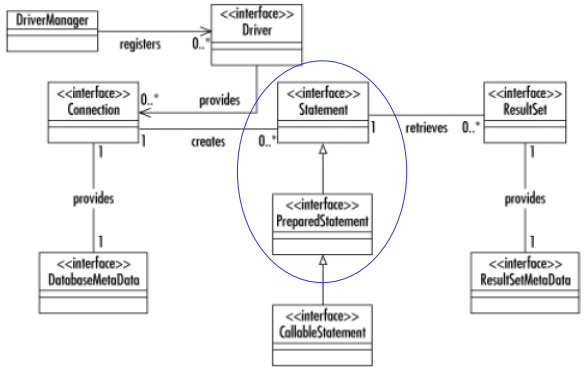
Prepared Statement is a precompiled SQL statement.

It is more efficient than calling the same SQL statement over and over.

The `PreparedStatement` class extends the `Statement` class by adding the capability of setting parameters inside of a statement.

e-Macao-16-3-309

PreparedStatement Inheritance



e-Macao-16-3-310

setXXX Methods 1

The `PreparedStatement` class extends the `Statement` class by adding the capability of setting parameters inside of a statement.

The `setXXX` methods are used to set SQL IN parameters values.

Must specify the types that are compatible with the defined SQL input type parameters.

For example, if the IN parameter has SQL type Integer, then you should use the `setInt` method

e-Macao-16-3-311

setXXX Methods 2

Method	SQL Types
setArrayLocator	Locator(<array>)
setASCIIStream	Uses an American Standards Code for Information Exchange (ASCII) stream to produce a LONGVARCHAR
setBigDecimal	NUMERIC
setBinaryStream	Uses a binary stream to produce a LONGVARBINARY
setBlobLocator	LOCATOR(BLOB)
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY or LONGVARBINARY (depending upon the size relative to the limits on VARBINARY)
setCharacterStream	Uses Java.io.Reader to produce a LONGVARCHAR
setClobLocater	LOCATOR(CLOB)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	The given Java technology object ("Javaobject") is converted to the target SQL Type before sent
setShort	SMALLINT
set String	VARCHAR OR LONGVARCHAR (depending upon the size relative to the driver's limits on VARCHAR)
setStructLocator	LOCATOR(<structure_type>)
setTime	TIME
setTimeStamp	TIMESTAMP
setUnicodeStream	Uses a Unicode stream to produce a LONGVARCHAR

e-Macao-16-3-312

Example: PreparedStatement

```
public boolean prepStatement(String name, String sex){
   String query = null;
   PreparedStatement prepStmnt = null;
   query = "update license set name = ?, sex = ? where
                                          id= 126";
   prepStmnt = con.preparedStatement (query);
   prepStmnt.setFloat(1, name);
  prepStmnt.setString(2, sex);
  Int rowsUpdate = prepStmnt.executeUpdate();
  return (rowUpdate > 0);
}
```

e-Macao-16-3-313

CallableStatement

`CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

`CallableStatement` class extends the `PreparedStatement` class, which provides the methods for setting IN parameters.

Methods for retrieving multiple results with a stored Procedure are supported with the `Statement.getMoreResults()` method.

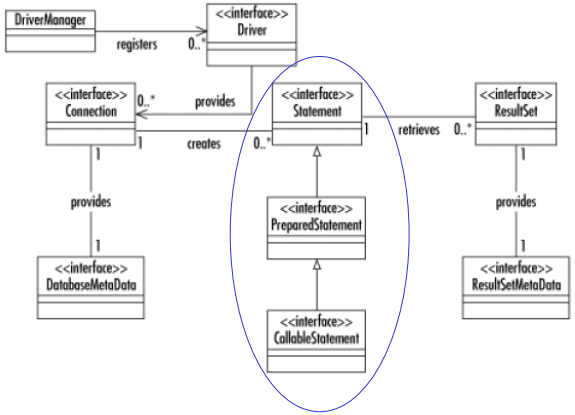
e-Macao-16-3-314

Example: CallableStatement

```
   int id= 126;
  CallableStatement callStm = null;
  String storProcName="{?=call return_license(?)}"
  querySales = con.prepareCall(storProcName);
  try {
       callStm.registerOutParameter(1, Type.VARCHAR);
       callStm.setInt(2, id);
       callStm.execute();
       String license = callStm.getString(1);
  } catch (SQLException e) {
       e.printStackTrace();
  }
```

e-Macao-16-3-315

CallableStatement Inheritance



e-Macao-16-3-316

Transaction Management

A transaction is a set of one or more statements that are executed together as a unit.

Either all of the statements are executed, or none of the statements is executed.

There are times when you do not want one statement to take effect unless another one also succeeds.

This is achieved through the `setAutoCommit()` method of `Connection` object.

e-Macao-16-3-317

Transaction Management

A transaction is a set of one or more statements that are executed together as a unit.

Either all of the statements are executed, or none of the statements is executed.

There are times when you do not want one statement to take effect unless another one also succeeds.

This is achieved through the `setAutoCommit()` method of `Connection` object.

The method takes a `boolean` value as a parameter.

e-Macao-16-3-318

Disabling Auto-commit Mode

When a connection is created, it is in auto-commit mode.

Each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode.

Example:

```
con.setAutoCommit(false);
```

e-Macao-16-3-319

Committing a Transaction

Once auto-commit mode is disabled, no SQL statements will be committed until you call the method commit explicitly.

This is achieved through the `commit()` method of connection objects.

All statements executed after the previous call to the `commit()` method will be included in the current transaction and will be committed together as a unit.

If you are trying to execute one or more statements in a transaction and get an `SQLException`, you should call the `rollback()` method to abort the transaction and start the transaction all over again.

e-Macao-16-3-320

Example: Transaction Commit

```
con.setAutoCommit(false);
PreparedStatement updateName =null;
String query = null;
Query="UPDATE license SET name = ? WHERE id = 126"
updateName= con.prepareStatement(query);
updateName.setString(1, name);
updateName.executeUpdate();
PreparedStatement updateSex = null;
query = "UPDATE test SET test_value =?”
updateSex = con.prepareStatement(query);
updateSex.setString(1, "Male");
updateSex.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

e-Macao-16-3-321

ResultSet

A `ResultSet` is one or more rows of data returned by a database query.

The class simply provides a series of methods for retrieving columns from the results of a database query

General form:

```
type gettype(int | String)
```

in which the argument represents either the column number or column name desired

can store values in the database as one type and retrieve them as a completely different type

e-Macao-16-3-322

ResultSet getXXX() Methods

Method	Java Type Returned
getArrayLocator	LOCATOR(<array>)
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBlobLocator	LOCATOR(BLOB)
getBoolean	boolean
getByte	byte
getBytes	byte[]
getCharacterStream	java.io.Reader
getClobLocator	LOCATOR(CLOB)
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getStructLocator	LOCATOR(<structure-type>)
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream or Unicode characters

e-Macao-16-3-323

SQL and Java Type Mapping

SQL TYPE	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

e-Macao-16-3-324

Getting the Next Record

`ResultSet` class handles only a single row from the database at any given time.

The class provides the `next( )` method for making it reference the next row of a result set.

If `next()` returns `true`, you have another row to process and any subsequent calls you make to the `ResultSet` object will be in reference to that next row.

If there are no rows left, it returns `false`.

e-Macao-16-3-325

Example: ResultSet

```
String query = "select * from license";
Statement stm = null;
stm = con.getStatement();
ResultSet rs = stm.executeQuery(query);
while(rs.next( )) {
    int a;
    String str;
    a = rs.getInt("id");
    if( rs.wasNull( ) ) {
      a = -1;
    }
    str = rs.getString("name");
    if( rs.wasNull( ) ) {
     str = null;
    }
}
```

e-Macao-16-3-326

SQL Null Versus Java null

SQL and Java have a serious mismatch in handling null values.

Java `ResultSet` has no way of representing a SQL NULL value for any numeric SQL column.

After retrieving a value from a `ResultSet`, it is therefore necessary to ask the `ResultSet` if the retrieved value represents a SQL NULL.

To avoid running into database oddities, however, it is recommended that you always check for SQL NULL.

Checking for SQL NULL involves a single call to the `wasNull( )` method in your `ResultSet` after you retrieve a value.

e-Macao-16-3-327

Example: wasNull()

```
rs.afterLast( );
while(rs.previous( )) {
    int a;
    String str;
    a = rs.getInt("test_id");
    if( rs.wasNull( ) ) {
      a = -1;
    }
    str = rs.getString("test_val");
    if( rs.wasNull( ) ) {
     str = null;
    }
}
```

e-Macao-16-3-328

Scrollable ResultSet 1

The single most visible addition to the JDBC API in its 2.0 specification is support for scrollable result sets.

Using scrollable result sets starts with the way in which you create statements.

The `Connection` class actually has two versions of `createStatement()`

1) the zero parameter version

Example:

```
Statement stm = con.createStatement();
```

e-Macao-16-3-329

Scrollable ResultSet 2

2) a two parameter version that supports the creation of `Statement` instances that generate scrollable `ResultSet` objects.

```
createStatement(int rsType,int rSConcurrency)
```

Parameters:

`rsType` - a result set type; one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

`rsConcurrency` - a concurrency type; one of `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

e-Macao-16-3-330

ResultSet Constants

JDBC defines three types of result sets:
1) TYPE_FORWARD_ONLY
2) TYPE_SCROLL_SENSITIVE
3) TYPE_SCROLL_INSENSITIVE

Out of these three `TYPE_FORWARD_ONLY` is the only type that is not scrollable.

The other two types are distinguished by how they reflect changes made to them.

`TYPE_SCROLL_INSENSITIVE` `ResultSet` is unaware of in-place edits made to modifiable instances.

`TYPE_SCROLL_SENSITIVE`, on the other hand, means that you can see changes made to the results if you scroll back to the modified row at a later time.

e-Macao-16-3-331

Result Set Navigation 1

When `ResultSet` is first created, it is considered to be positioned before the first row.

Positioning methods such as `next()` point a `ResultSet` to actual rows.

Your first call to `next( )`, for example, positions the cursor on the first row.

Subsequent calls to `next()` move the `ResultSet` ahead one row at a time.

With a scrollable ResultSet, however, a call to `next()` is not the only way to position a result set.

e-Macao-16-3-332

Result Set Navigation 2

The method `previous( )` works in an almost identical fashion to next( ).

While `next()` moves one row forward, `previous()` moves one row backward.

If it moves back beyond the first row, it returns `false`. Otherwise, it returns `true`.

Because a `ResultSet` is initially positioned before the first row, you need to move the `ResultSet` using some other method before you can call `previous()`.

e-Macao-16-3-333

Example: Result Set Navigation 1

```
import java.sql.*;
import java.util.*;
public class ReverseSelect {
 public static void main(String argv[]) {
  Connection con = null;
  try {
   String url = "jdbc:mysql://localhost/emacao";
   String driver = "com.mysql.jdbc.Driver";
   Statement stmt;
   ResultSet rs;
   Class.forName(driver).newInstance( );
   con = DriverManager.getConnection(url, "root", "");
   stmt =con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                  ResultSet.CONCUR_READ_ONLY);
   rs = stmt.executeQuery("SELECT * from license ORDER BY id");

   System.out.println("Got results:");
```

e-Macao-16-3-334

Example: Result Set Navigation 2

```
   rs.afterLast( );
   while(rs.previous( )) {
    int a;
    String str;
    a = rs.getInt("id");
    a = rs.wasNull( ) ? -1 : a;
    str = rs.getString("name");
    str = rs.wasNull( ) ? null : str;
    System.out.print("\tid= " + a);
    System.out.println("/str= '" + str + "'");
   }
   System.out.println("Done.");
  }catch( Exception e ) {
   e.printStackTrace( );
  }
```

e-Macao-16-3-335

Example: Result Set Navigation 3

```
  finally {
    if( con != null ) {
     try {
      con.close( );
     }catch( SQLException e ) {
      e.printStackTrace( );
     }
    }
   }
  }
}
```

e-Macao-16-3-336

Other Navigation Methods

JDBC 2.0 provides new methods to navigate around rows in result sets:

1) `beforeFirst( )`
2) `first( )`
3) `last( )`
4) `isBeforeFirst( )`
5) `isFirst( )`
6) `isLast( )`
7) `isAfterLast()`
8) `getRow( )`
9) `relative( )`
10) `absolute( )`

Except for `absolute()` and `relative()`, the names of the methods say exactly what they do. Each take integer arguments.

e-Macao-16-3-337

absolute() 1

For `absolute( )`, the argument specifies a row to navigate to.

Example:
A call to `absolute(5)` moves the ResultSet to row 5 unless there are four or fewer rows in the `ResultSet`.

A call to `absolute()` with a row number beyond the last row is therefore identical to a call to `afterLast()`

e-Macao-16-3-338

absolute() 2

You can also pass negative numbers to `absolute()`.

A negative number specifies absolute navigation backwards from the last row

Example:
`absolute(1)` is identical to `first()`, `absolute(-1)` is identical to `last(`

Similarly, `absolute(-3)` is the third to last row in the `ResultSet`. If there are fewer than three rows in the `ResultSet.`

e-Macao-16-3-339

relative()

The `relative()` method handles relative navigation through a `ResultSet`.

In other words, it tells the `ResultSet` how many rows to move forward or backward.

Example:

A value of 1 behaves just like `next()` and a value of -1 just like `previous()`.

e-Macao-16-3-340

Clean Up

The `Connection`, `Statement`, and `ResultSet` classes all have `close( )`.

It is always a good idea to close any instance of these objects when you are done with them.

It is useful to remember that closing a `Connection` implicitly closes all `Statement` instances associated with the Connection.

Similarly, closing a `Statement` implicitly closes `ResultSet` instances associated with it.

e-Macao-16-3-341

Example: Clean Up

```
try{
   // Connection, Statements here
}catch(SQLException ex){
   ex.printStarkTrace();
}finally {
    if( con != null ) {
     try {
       con.close( );
     }catch( SQLException e ) {
      e.printStackTrace( );
     }
    }
   }
  }
}
```

e-Macao-16-3-342

Lab Work: ResultSet

1) Using the `LicenseApp.java`, check if the data you are saving exists, if it exists update the record with the new values otherwise insert a new record.
2) Modify `LicenseApp.java` to implement previous and next buttons.

   Note: Previous and next buttons should be disabled whenever the pointer is at the beginning and end of the ResultSet respectively.

A.5. Architectures

Architectures

e-Macao-16-3-344

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

A.5.1. Distributed Objects

Distributed Objects

e-Macao-16-3-346

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-347

Overview

What options do I have for distributed application development?

Developers who program using the Java programming language can choose several solutions for creating distributed applications programs.

1) Java RMI technology
2) Java IDL technology (for CORBA programmers)
3) Enterprise JavaBeans technology

In this section we shall be talking about Java RMI and IDL technologies.

A.5.1.1 RMI

RMI

e-Macao-16-3-349

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-350

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

e-Macao-16-3-351

Introduction 1

Remote Method Invocation (RMI) technology was first introduced in JDK1.1.

RMI allows programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs.

RMI is based on a similar, earlier technology for procedural programming called remote procedure call (RPC)

e-Macao-16-3-352

Introduction 2

Disadvantages of RPC

a) RPC supports a limited set of data types

Therefore it is not suitable for passing and returning Java Objects

b) RPC requires the programmer to learn a special interface definition language (IDL) to describe the functions that can be invoked remotely

e-Macao-16-3-353

Introduction 3

The RMI architecture defines

a) How objects behave.

b) How and when exceptions can occur.

c) How memory is managed.

d) How parameters are passed to, and returned from, remote methods. The remote object model for Enterprise JavaBeans (EJB) is RMI-based.

e-Macao-16-3-354

Introduction 4

RMI is designed for Java-to-Java distributed applications.

RMI is simpler and easier to maintain than using socket.

Other options for creating Java-to-non-Java distributed applications are:

a) Java Interface Definition Language (IDL)

b) Remote Method Invocation (RMI) over Internet Inter-ORB Protocol (IIOP) -- RMI-IIOP.

e-Macao-16-3-355

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

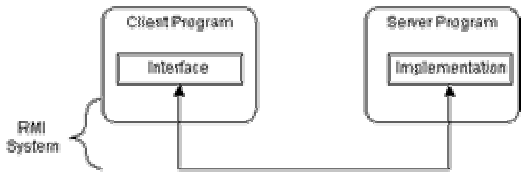
e-Macao-16-3-356

Architecture 1

RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

At client side, RMI uses interfaces to define behavior.

At server side, RMI uses classes to define implementation.



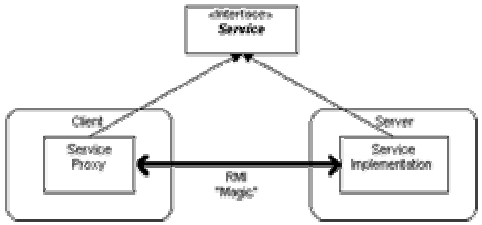
e-Macao-16-3-357

Architecture 2

The service interface is implemented by two classes.

a) The first one is at the server side which implements the behavior.

b) The second one is at the client side which acts as a proxy.

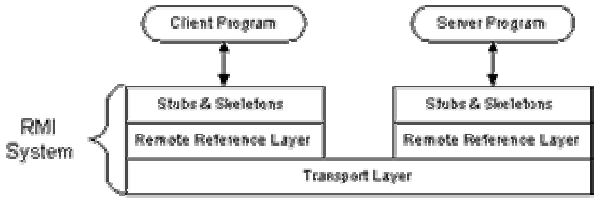


e-Macao-16-3-358

Layers

The RMI implementation is built from three abstraction layers.

a) The Stub and Skeleton layer
b) The Remote Reference Layer
c) The transport layer

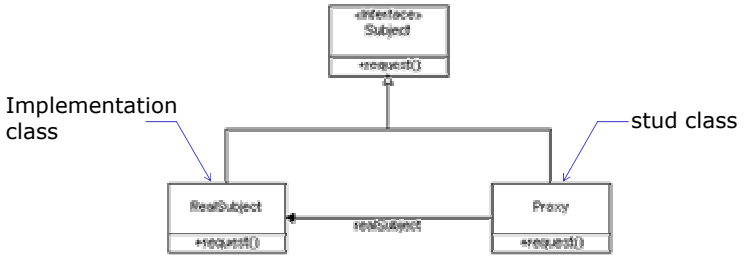


e-Macao-16-3-359

Stub and Skeleton Layer 1

The first layer lies beneath the view of the developer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

RMI uses the Proxy design pattern in this layer.



e-Macao-16-3-360

Stub and Skeleton Layer 2

A skeleton is a helper class that is generated for to communicate with the stub across the RMI link

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete.

You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

e-Macao-16-3-361

Remote Reference Layer

This layer provides a RemoteRef object that represents the link to the remote service implementation object.

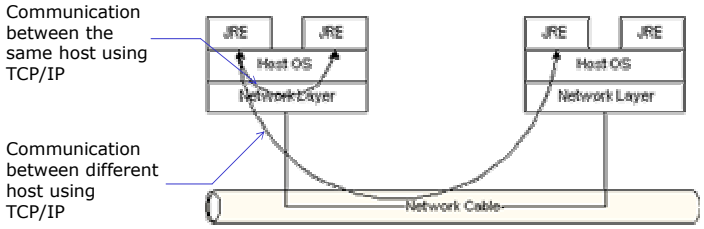
In JDK 1.1, only unicast, point-to-point connection is supported. Before a client can use a remote service, the remote service must be instantiated on the server and ran all the time.

In Java 2 SDK, client-server connection is added and activatable remote objects is supported . With the introduction of the RMI daemon, rmid, remote objects can be created and execute "on demand," rather than running all the time.

e-Macao-16-3-362

Transport Layer 1

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.



e-Macao-16-3-363

Transport Layer 2

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP).

JRMP is a proprietary, stream-based protocol.

Sun and IBM have jointly worked on another version of RMI, called RMI-IIOP(Remote Method Invocation over Internet Inter-ORB Protocol), which combines RMI-style ease of use with CORBA cross-language interoperability.

The remote object model for Enterprise Java Beans(EJBs) is RMI-based.

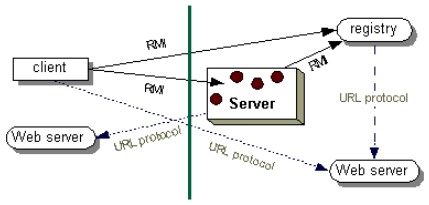
e-Macao-16-3-364

Naming Remote Objects

In RMI, clients find remote services by using a naming or directory service.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI).

RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.



e-Macao-16-3-365

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

e-Macao-16-3-366

Example : build a RMI system

In this example, we shall build a simple remote calculator service and use it from a client program.

A working RMI system is composed of several parts.

a) Interface definitions for the remote services
b) Implementations of the remote services
c) Stub and Skeleton files
d) A server to host the remote services
e) An RMI Naming service that allows clients to find the remote services
f) A class file provider (an `HTTP` or `FTP` server)
g) A client program that needs the remote services

e-Macao-16-3-367

Interface 1

The first step is to write and compile the Java code for the service interface.

All the interface has to extend the `java.rmi.Remote` interface and all the methods has to declare that it may throw a `RemoteException` object.

e-Macao-16-3-368

Interface 2

The interface may look like the following:

```
public interface Calculator extends
     java.rmi.Remote
{ public long add(long a, long b) throws
     java.rmi.RemoteException;
  public long sub(long a, long b) throws
     java.rmi.RemoteException;
  public long mul(long a, long b) throws
     java.rmi.RemoteException;
  public long div(long a, long b) throws
     java.rmi.RemoteException;
 }
```

e-Macao-16-3-369

Implement 1

The second step is to write the implementation for the remote service.

The implementation class may extend from the `java.rmi.server.UnicastRemoteObject` to link into the RMI system.

It must also provide an explicit default constructor throwing `RemoteException`. When this constructor calls super(), it activates code in `UnicastRemoteObject` that performs the RMI linking and remote object initialization.

e-Macao-16-3-370

Implement 2

```
public class CalculatorImpl extends
   java.rmi.server.UnicastRemoteObject implements
   Calculator {
        // Implementations must have an
        // explicit default constructor
        //  in order to declare the
        //   RemoteException exception
    public CalculatorImpl() throws
        java.rmi.RemoteException
    { super(); }
    public long add(long a, long b) throws
        java.rmi.RemoteException
    { return a + b; }
    ...
```

e-Macao-16-3-371

Lab Work: Implementation

1) Please write the rest of implementation for the Calculator interface.

Note: If your must extend some other classes other than extending from `UnicastRemoteObject`, the implementation may use the static `exportObject()` method of the `UnicastRemoteObject` instead.

Be careful that you may need to synchronize some portions of your remotely available method. But it is not necessary for this example.

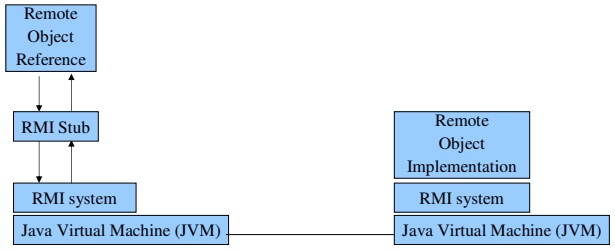
e-Macao-16-3-372

Stubs and Skeletons

To generate the Stub and Skeleton files, use the RMI complier, rmic as the following:

```
>rmic CalculatorImpl
```

The default option will create stubs/skeletons compatible with both JDK 1.1 and Java 2.



e-Macao-16-3-373

Host Server 1

Remote RMI services must be hosted in a server process. The following code is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;
public class CalculatorServer {
   public CalculatorServer() {
     try {
       Calculator c = new CalculatorImpl();

     Naming.rebind("rmi://localhost:1099/Calculator
     Service", c);
     } catch (Exception e) {
       System.out.println("Trouble: " + e);
     }
   }
```

e-Macao-16-3-374

Host Server 2

```
    public static void main(String args[]) {
      new CalculatorServer();
    }
}
```

e-Macao-16-3-375

Client 1

1) In the client's code, all you need to do is to lookup the object and use it's methods as local methods.
2) The client's code may look like the following:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {
    public static void main(String[] args) {
     try {
         Calculator c = (Calculator)Naming.lookup (
          "rmi://localhost/CalculatorService");
```

e-Macao-16-3-376

Client 2

```
            System.out.println( c.sub(4, 3) );
            System.out.println( c.add(4, 5) );
            System.out.println( c.mul(3, 6) );
            System.out.println( c.div(9, 3) );
        }
         catch (MalformedURLException murle) {}
         catch (RemoteException re){}
         catch (NotBoundException nbe){}
         catch (java.lang.ArithmeticException ae) {}
    }
}
```

e-Macao-16-3-377

Running the RMI System

1) Start up three consoles, one for the server, one for the client, and one for the RMIRegistry.
2) Type `rmiregistry` in the directory that contains the classes you have written.
3) In the server's console, type `java CalculatorServer` to start the server.
4) In the client's console, type java `CalculatorClient` to start the cleint program.
5) The output should look like:

```
1
9
18
3
```

e-Macao-16-3-378

Lab Work: RMI System

1) Please follow what we have discussed to develop a RMI server which hosts a service for calculating the square root of a number.

2) Compile your RMI server and generate the corresponding stub class.

3) Start rmi registry use command `rmiregistry <port number>`.
   a) make sure that the shell or window in which you will run the registry, either has no `CLASSPATH` set or has a `CLASSPATH` that does not include the path to any classes that you want downloaded to your client, including the stubs for your remote object implementation classes.
   b) If you start the `rmiregistry`, and it can find your stub classes in its `CLASSPATH`, it will ignore the server's `java.rmi.server.codebase` property, and as a result, your client(s) will not be able to download the stub code for your remote object.

e-Macao-16-3-379

Lab Work: RMI System

3) Start your RMI Server :

```
java  -classpath <classpath_of_server>
     -Djava.rmi.server.codebase=
      file:/<classpath_for_download>/ server
```

4) Create a client to test the RMI service.

e-Macao-16-3-380

Passing Parameters

All parameters passed from an RMI client to an RMI server must either be serializable or be a remote object.

For serializable:

a) Data is extracted from the local object and sent across the network to the remote server.
b) Object is then reconstructed in the remote server.
c) Any changes to the object in the RMIServer will not be reflected in the object held in the RMI client and vice versa.

For a remote object:

a) Stub information, not a copy of data, is actually sent over RMI.
b) Any call made to the parameter object become a remote calls back to the actual object.
c) Changes made in one JVM are reflected in the original JVM.

e-Macao-16-3-381

Conditions for serializability

If an object is to be serialized:

a) The class must be declared as public
b) The class must implement Serializable
c) The class must have a default (no-argument) constructor
d) All fields of the class must be serializable: either primitive types or serializable objects

e-Macao-16-3-382

Remote interfaces and class

A Remote class has two parts:

a) The interface (used by both client and server):
   1. Must be public
   2. Must extend the interface `java.rmi.Remote`
   3. Every method in the interface must declare that it throws `java.rmi.RemoteException` (other exceptions may also be thrown)

b) The class itself (used only by the server):
   1. Must implement a `Remote` interface
   2. Should extend `java.rmi.server.UnicastRemoteObject`
   3. May have locally accessible methods that are not in its `Remote` interface

e-Macao-16-3-383

Security

Your program should guarantee that the classes that get loaded do not perform operations that they are not allowed to perform.

A more conservative security manager than the default should be installed. The following code should be added to the main method of the server and client program:

```
if (System.getSecurityManager() == null{
  System.setSecurityManager(new
       RMISecurityManager());
}
```

e-Macao-16-3-384

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

e-Macao-16-3-385

Activatable Server

Enable server programs to wake up and start to run when they are needed.

Java RMI Activation System Daemon (`rmid`) is introduced to handle this task.

When a client requests a reference to the server from the `rmiregistry`, the `rmid` program, which holds the servers details, will be requested to start up the server and return the reference to the client. After that, the `rmiregistry` will be able to provide the reference of the server directly.

e-Macao-16-3-386

Activatable Server:Implementation

1) Subcalss the `java.rmi.activation.Activatable` class and implement the remote interface.
2) Implement the following constructor:

```
public Server(ActivationID id, MarshalledObject
    data) throws RemoteException{
     super(id,0);//register activatable object and
                 // export on anonymous port
     }
```

3) Create an activation description used by the `rmid` program.
4) Register the activation description with the `rmid` program.
5) Compile the activatable server with `javac` and `rmic` compiler.
6) The client program needs no modification.

e-Macao-16-3-387

Activatable Server:Setup 1

Before we can use the activatable server, you need to generate the activation description used by the `rmid` and register the description with the `rmid` program. We will group these processes into a utility program for illustration.

The structure of the utility program may look like this:

```
//1. Make the appropriate imports
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
public class SetupServer{
     public static void main(String args[]){
     try{
```

e-Macao-16-3-388

Activatable Server:Setup 2

```
//2. Declare for a security policy file
System.setSecurityManager(new RMISecurityManager());
Properties props = (Properties)System.getProperties();
props.put("java.security.policy",<location of
    security policy file>);
//3.Create an activation group description even there
//  is only one server
ActivationGroupDesc agd = new ActivationGroupDesc
    (props, null);
//4. Create a new activation group
ActivationGroupID agid =
    ActivationGroup.getSystem().registerGroup(agd);
```

e-Macao-16-3-389

Activatable Server:Setup 3

```
//5. Create the actual activation description
//   Don't miss the trailing slash (/)
String codebase = "file:/<location of server
     implementation file>/";
ActivationDesc desc = new ActivationDesc(agid,
"<name of the server>",codebase, null);
//6. Register the activation description to the rmid
//   program. Suppose the remote interface of the
//   server is RemoteInterface, the code will look
     like this:
RemoteInterface ref =
     (RemoteInterface)Activatable.register(desc);
```

e-Macao-16-3-390

Activatable Server:Setup 4

```
        //7. Bind the server in rmiregistry
        Naming.rebind("Server", ref);
        //8. Exit the setup program
        System.exit(0);
        }catch (Exception e){}
    }
}
```

e-Macao-16-3-391

Compile and Run 1

1) Compile all the classes use `javac`.
2) Run `rmic` on the implementation class
3) Start rmi registry use `rmiregistry`.
   a) make sure that the shell or window in which you will run the registry, either has no `CLASSPATH` set or has a `CLASSPATH` that does not include the path to any classes that you want downloaded to your client, including the stubs for your remote object implementation classes.
   b) If you start the `rmiregistry`, and it can find your stub classes in its `CLASSPATH`, it will ignore the server's `java.rmi.server.codebase` property, and as a result, your client(s) will not be able to download the stub code for your remote object.

e-Macao-16-3-392

Compile and Run 2

4) Start the activation daemon, `rmid`. Use `-J` option for a runtime flag.

```
rmid -J-Djava.security.policy=rmid.policy
```

The policy file may look like this:

```
grant{
permission com.sun.rmi.rmid.ExecOptionPermission
    "-Djava.*";
permission com.sun.rmi.rmid.ExecOptionPermission
    "-Dsun.*";
permission com.sun.rmi.rmid.ExecOptionPermission
    "-Dfile.*";
```

e-Macao-16-3-393

Compile and Run 3

```
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dpath.separator=*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Duser.*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dos.*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dline.separator=*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dawt.*";
```

e-Macao-16-3-394

Compile and Run 4

5) Running the setup program.

```
java
-Djava.security.policy
    =<full path of the policy file>
-Djava.rmi.server.codebase
    =file:/<location of the implementation stubs>/
<class name of the setup program>
```

e-Macao-16-3-395

Compile and Run 5

6) Running the client program.

```
java
-Djava.security.policy
    =<full path of the policy file>
<client name>
```

For testing purpose, use the following `security.policy`:

```
grant {
    permission java.security.AllPermission "", "";
};
```

e-Macao-16-3-396

Exercise: Activatable RMI

1) Write a remote interface called `HelloInterface`.
2) Define a method `getMessage (String s)` in it. This method has a return type as a String. Don't forget to throw the proper exception.
3) Create a class named Server which has to be a subclass of the `java.rmi.activation.Activatalbe` class.
4) Implement the `getMessage` method which will append "Hello" the argument and return it as a `String`.
5) Create the Setup program for the server.
6) Create a client program which should look up the activatable server and use the `getMessage` method of it.
7) Compile and generate the corresponding files.
8) Run the client and check the result.

e-Macao-16-3-397

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) summary

e-Macao-16-3-398

Summary

In this session, we cover the followings:

1) Architecture of RMI
2) Building RMI system including both client and server
3) Implementation for activatable RMI server

A.5.1.2 CORBA

CORBA

e-Macao-16-3-400

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-401

Overview

The main points in this section are:

1) The OMG and CORBA
2) basic concept of CORBA
2) CORBA Architecture
3) Object Request Broker (ORB)
4) Structure of a CORBA Application
5) Interface Definition Language

e-Macao-16-3-402

The OMG

1) Object Management Group
   a) Founded in 1989
   b) Not-for-Profit organization
   c) Vendor neutral
   d) ~800 member companies
2) Key Specifications
   a) UML
   b) CORBA

e-Macao-16-3-403

What is CORBA?

Defines a framework for object-oriented distributed applications.

Defined by a consortium of vendors under the direction of OMG.

Allows distributed programs in different languages and different platforms to interact as though they were in a single programming language on one computer.

Brings advantages of OO to distributed systems.

Allows you design a distributed application as a set of cooperating objects and to reuse existing objects.

e-Macao-16-3-404

Key CORBA Features

1) Application Development Transparencies
   a) Hardware/Language neutral
   b) Vendor neutral
   c) Object oriented paradigm

2) CORBA Interface Definition Language (IDL)

3) CORBAServices
   a) Naming
   b) Event
   c) Transaction
   d) Security

4) Interoperability

e-Macao-16-3-405

Object Request Broker (ORB)

1) A software component that mediates transfer of messages from a program to an object located on a remote host.

2) Hides underlying network communications from a programmer.

3) ORB allows you to create software objects whose member functions can be invoked by client programs located anywhere.

4) A server program contains instances of CORBA objects.

e-Macao-16-3-406

ORB: Conceptual View

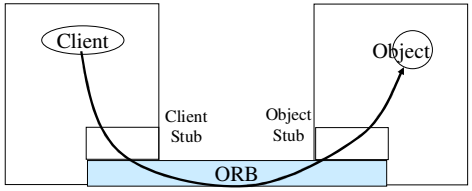
1) When a client invokes a member function on a CORBA object, the ORB intercepts the function call.

2) ORB directs the function call across the network to the target object.

3) The ORB then collects the results from the function call returns these to the function call.

e-Macao-16-3-407

Implementation Details



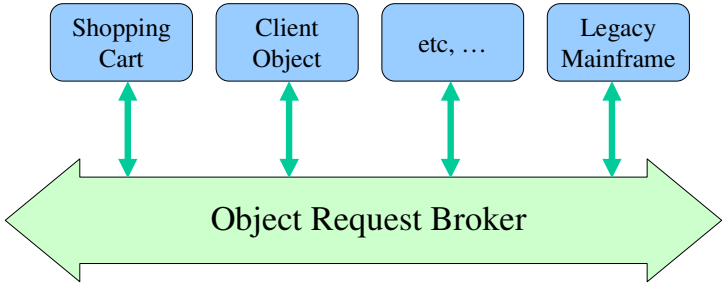
Access to the services provided by an Object

ORB : (Object-oriented middleware) Object Request Broker
ORB mediates transfer between client program and server object.

e-Macao-16-3-408

CORBA: A "Software Bus"

All CORBA objects connect to each other via ORB.



e-Macao-16-3-409

CORBA IDL

1) Interface Definition Language
   a) used to generate application code (stubs/skeletons)
   b) language neutral (Ada, C++, Java, …)

2) IDL is NOT a programming language
   a) lacks control structures
   b) provides no implementation details
   c) a specification

e-Macao-16-3-410

CORBA Objects and IDL

1) These are standard software objects implemented in any supported language including Java, C++ and Smalltalk.

2) Each CORBA object has a clearly defined interface specified in CORBA `interface definition language` (IDL).

3) The interface definition specifies the member functions available to the client without any assumption about the implementation of the object.

e-Macao-16-3-411

Client and IDL

1) To call a member function on a CORBA object the client needs only the object's IDL.

2) Client need not know the object's implementation, location or operating system on which the object runs.

e-Macao-16-3-412

Interface and Implementation

1) Interface and implementation can be in two different languages.

2) Interface abstracts and protects details (trade secrets) from client

3) Interface offers a means of expressing design without worrying about implementation.

4) Interface is separated from implementation

e-Macao-16-3-413

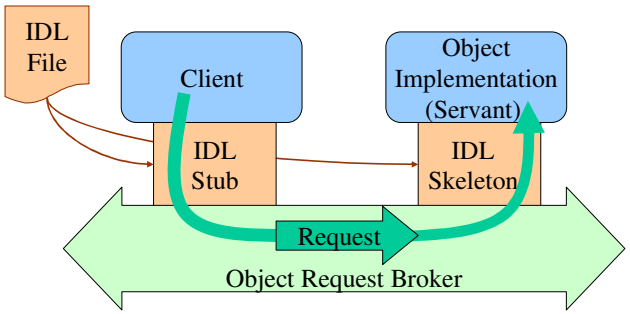
Example: CORBA IDL

```
module BankExample {
   interface Account {
      exception BadCheck {
         float fee;
        };
      float deposit(in float amount);
      float writeCheck(in float amount)
         raises (BadCheck);
    };
   interface AccountManager {
      Account openAccount(in string name);
    };
};
```

e-Macao-16-3-414

CORBA Application Diagram

Objects are identified by Interoperable Object References (IORs)



e-Macao-16-3-415

CORBA Development Steps

1) Design the Application
2) IDL Specification
3) IDL Compilation (Code Generation)
4) Write the Client & Server implementation specific code
5) Compile the source code
6) Run the application

A.5.1.3 JavaIDL

JavaIDL

e-Macao-16-3-417

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-418

Modules and Interfaces

- IDL modules
```
module MyStuff
{
 …
};
```
  - Provide a namespace to group a set of interfaces. Names are scoped using the "::" operator.
- IDL interface
```
interface Foo { };
```
- Java packages
```
package MyStuff;
…
```
  - Provide Internet-wide namespaces. Scoped using the "." operator.
- Java interface
```
public interface Foo
  {…};
```

e-Macao-16-3-419

IDL to Java: Parameters

1) Java uses pass-by-value for parameters (including parameters that are references)
2) IDL has `in`, `out` and `inout` types of parameters
3) The `in` parameter type maps to a normal Java parameter since it does not need to be changed
4) `out` and `inout` parameter types are passed via instances of Java `Holder` classes

e-Macao-16-3-420

Holder Classes

1) `Holder` classes encapsulate the real value of a parameter which can then be reassigned to
   a) a member "value"

```
// generated java

package Example;
public interface Modes {
  int operation (IntHolder outArg,
                 IntHolder inoutArg);
}
```

```
// user code

// select a target object
Example.Modes target = …;
// prepare to receive out
IntHolder outHolder = new IntHolder();
// set up the in side of the inout
IntHolder inoutHolder = new IntHolder (131);
// make the invocation
int result = target.operation (
                  outHolder, inoutHolder);
// use the values of holders
…outHolder.value…
…inoutHoulder.value…
```

e-Macao-16-3-421

Helper Classes

1) all user-defined IDL types have a Helper Java class
2) insert and extract Any
3) get CORBA::TypeCode of the type
4) narrow (for interfaces only)

e-Macao-16-3-422

IDL to Java: Attributes

- IDL attributes

```
attribute long assignable;
readonly attribute long
                fetchable;
```

- Java "`get`" and "`set`" methods

```
public int assignable();
public void assignable(int
                    val);
public int fetchable();
```

e-Macao-16-3-423

Basic Types

IDL Type	Java Type	Exception
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
octet	byte	
string	java.lang.String	CORBA::MARSHAL…
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

e-Macao-16-3-424

IDL to Java: Basic Types

- IDL char

```
const char MyChar = 'A';
```

- Java char

```
final public class MyChar
{
final public static char
value = (char)'A';
}
```

e-Macao-16-3-425

IDL to Java: Basic Types

- IDL octet

```
void foo(in octet x);
```

- Java byte

```
public void foo(byte x);
```

e-Macao-16-3-426

IDL to Java: Basic Types

- IDL boolean
```
const boolean truth =
TRUE;
```
- IDL constants `TRUE` and `FALSE`

- Java boolean
```
final public class truth
{
    final public static
boolean value = true;
}
```
- Java constants `true` and `false`

e-Macao-16-3-427

IDL to Java: Basic Types

- IDL string
```
const string MyString =
"Hello World";
```

- Java java.lang.String
```
final public class MyString
{
 final public static String
    value = "Hello World";
}
```

e-Macao-16-3-428

IDL to Java: Basic Types

- IDL integers
  - (unsigned) short
  - (unsigned) long
  - (unsigned) long long?

```
const unsigned short
MyUnsignedShort = 1580;
```

- Java integers
  - short
  - int
  - long

```
final public class
MyUnsignedShort
{
       final public
static short value =
(short)1580;
}
```

e-Macao-16-3-429

IDL to Java: Basic Types

- IDL floating-point float, double

```
const double MyDouble =
1.23456789;
```

- Java floating-point *float, double*

```
final public class
MyDouble
{
          final public
static
    double value =

    (double)1.23456789;
}
```

e-Macao-16-3-430

IDL to Java: Constructed Types

- IDL Enum

```
enum MyEnum
{none,first,second};
```

- Java class

```
final public class MyEnum
{
 final public static int
none = 0;
 final public static int
first = 1;
 final public static int
second = 2;
 final public static int
narrow(int i) throws
CORBA.BAD_PARAM {..};
}
```

- the narrow method is for checking enum values

e-Macao-16-3-431

IDL to Java: Constructed Types

- IDL *struct*

```
struct MyStruct
{
 long mylong;
 string mystring;
};
```

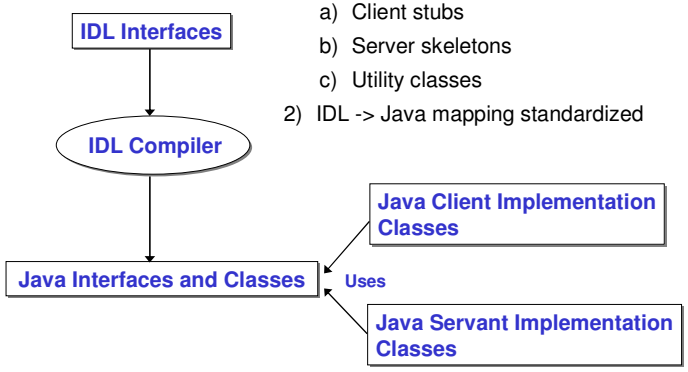
- Java *class*

```
final public class
MyStruct
{
    public MyStruct(int
_mylong, String
_mystring) {…};
    public MyStruct()
{…};

    public int mylong;
    public String
mystring;
}
```

e-Macao-16-3-432

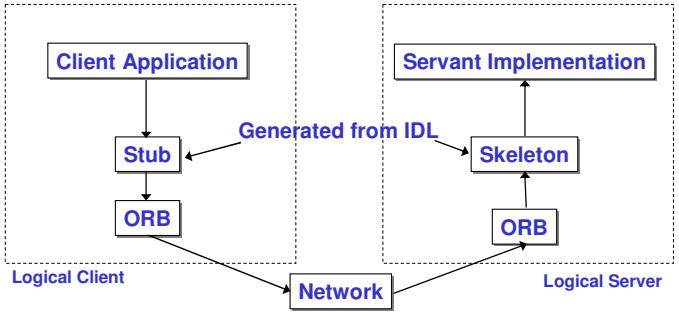
The Big Picture

1) Compiler outputs:
   a) Client stubs
   b) Server skeletons
   c) Utility classes
2) IDL -> Java mapping standardized



e-Macao-16-3-433

Big Picture (Invocation)



e-Macao-16-3-434

Example: File Transfer

This presents a file download CORBA application

The client request for a file and the server in turn sends the file to the client which then saves it on the local machine.

There are a number of steps involved:

1) Define an interface in IDL
2) Map the IDL interface to Java (done automatically)
3) Implement the interface
4) Develop the server
5) Develop a client
6) Run the naming service, the server, and the client.

e-Macao-16-3-435

Step 1: Define the IDL Interface 1

The first thing to do is to determine the operation that the server will support.

In this application, the client will invoke a method to download a file.

Here is the code.

```
interface FileInterface {
   typedef sequence<octet> Data;
   Data downloadFile(in string fileName);
};
```

Save this file as `FileInterface.idl`

e-Macao-16-3-436

Step 1: Define the IDL Interface 2

`Data` is a new type introduced using the `typedef` keyword.

A `sequence` in IDL is similar to an array except that a sequence does not have a fixed size

An `octet` is an 8-bit quantity that is equivalent to the Java type `byte`

The `downloadFile` method takes one parameter of type `string` that is declared `in`.

IDL defines three parameter-passing modes: `in` (for input from client to server), `out` (for output from server to client), and `inout` (used for both input and output).

e-Macao-16-3-437

Step 2: Map IDL to Java

Once you finish defining the IDL interface, you are ready to map the IDL interface to Java.

Java comes with the `idlj` compiler, which is used to map IDL definitions into Java declarations and statements.

The `idlj` compiler accepts options that allow you to specify if you wish to generate client stubs, server skeletons, or both.

let's compile the `FileInterface.idl` and generate both client and server-side files.

e-Macao-16-3-438

Step 3: Compile the IDL Interface

1) Compile the IDL Interface using:

```
prompt> idlj -oldImplBase -fall FileInterface.idl
```

2) IDL compilation produces many java constructs (interfaces and classes).

3) Each one is placed with a `<filename>.java`

e-Macao-16-3-439

Files Generated by IDL Compiler

1) Each file generated contains a Java interface or class scoped within a package.
2) This package is physically located in a directory of the same name according to Java conventions.

e-Macao-16-3-440

Client Side Files

1) `FileInterface.java` - an interface to provide a client a view of the methods in the IDL.

2) `_FileInterfaceStub.java` - a Java class that implements the methods defined in interface Grid. Provides functionality that allows client method invocations to be forwarded to a server.

e-Macao-16-3-441

Server Side Files

1) `_FileInterfaceImplBase.java` - an abstract Java class that allows server-side developers to implement the `FileInterface` interface.

2) `Other files: FileInterfaceHelper.java, FileInterfaceHolder.java, FileInterfaceOperations.java,`

e-Macao-16-3-442

Step 4: Implement the Interface 1

Provide an implementation to the `downloadFile()` method. This implementation is known as a servant.

```
import java.io.*;
public class FileServant extends _FileInterfaceImplBase
                                                    {
   public byte[] downloadFile(String fileName){
      File file = new File(fileName);
      byte buffer[] = new byte[(int)file.length()];
      try {
         BufferedInputStream input = new
         BufferedInputStream(new
                           FileInputStream(fileName));
         input.read(buffer,0,buffer.length);
         input.close();
```

e-Macao-16-3-443

Step 4: Implement the Interface 2

```
      } catch(Exception e) {
         System.out.println("FileServant Error:
                                    "+e.getMessage());
         e.printStackTrace();
      }
      return(buffer);
   }
}
```

e-Macao-16-3-444

Step 5: Develop the Server 1

The next step is developing the CORBA server.

Write `FileServer` class that implements a CORBA server that does the following:

1) Initializes the ORB
2) Creates a `FileServant` object
3) Registers the object in the CORBA Naming Service (COS Naming)
4) Prints a status message
5) Waits for incoming client requests

e-Macao-16-3-445

Step 5: Develop the Server 2

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class FileServer {
   public static void main(String args[]) {
      try{
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);
         // create the servant and register it with ORB
         FileServant fileRef = new FileServant();
         orb.connect(fileRef);
```

e-Macao-16-3-446

Step 5: Develop the Server 3

```
// get the root naming context
org.omg.CORBA.Object objRef =
 orb.resolve_initial_references("NameService");
NamingContext ncRef =
          NamingContextHelper.narrow(objRef);
// Bind the object reference in naming
NameComponent nc = new
          NameComponent("FileTransfer", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, fileRef);
System.out.println("Server started....");
```

e-Macao-16-3-447

Step 5: Develop the Server 4

```
         // Wait for invocations from clients
         java.lang.Object sync = new java.lang.Object();
         synchronized(sync){
            sync.wait();
         }
      } catch(Exception e) {
         System.err.println("ERROR: " + e.getMessage());
         e.printStackTrace(System.out);
      }
   }
}
```

e-Macao-16-3-448

Step 6: Develop the Client 1

The next step is developing the CORBA client.

Write `FileClient` class that implements a CORBA client that does the following:

1) Initializes the ORB
2) Retrieve the `FileTransfer` service from the naming server
3) Call the `downloadFile` method.

e-Macao-16-3-449

Step 6: Develop the Client 2

```
import java.io.*;
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class FileClient {
   public static void main(String argv[]) {
      try {
         // create and initialize the ORB
         ORB orb = ORB.init(argv, null);
         // get the root naming context
          org.omg.CORBA.Object objRef =
         orb.resolve_initial_references("NameService");
```

e-Macao-16-3-450

Step 6: Develop the Client 3

```
NamingContext ncRef =
          NamingContextHelper.narrow(objRef);
NameComponent nc = new
          NameComponent("FileTransfer", " ");
// Resolve the object reference in naming
NameComponent path[] = {nc};
FileInterfaceOperations fileRef =
     FileInterfaceHelper.narrow(ncRef.resolve(path));

if(argv.length < 1) {
    System.out.println("Usage: java FileClient
                                        filename");
}
```

e-Macao-16-3-451

Step 6: Develop the Client 4

```
    // save the file
    File file = new File(argv[0]);
    byte data[] = fileRef.downloadFile(argv[0]);
    BufferedOutputStream output = new
    BufferedOutputStream(new FileOutputStream("filenm"));
    output.write(data, 0, data.length);
    output.flush();
    output.close();
  }catch(Exception e) {
     System.out.println("FileClient Error: " +
                                        e.getMessage());
     e.printStackTrace();
  }
}}
```

e-Macao-16-3-452

Step 7: Run the Application

1) Running the the CORBA naming service.

```
prompt> tnameserv -ORBInitialPort 2500
```

2) Start the server

```
prompt> java FileServer -ORBInitialPort 2500
```

3) Run the client

```
prompt> java FileClient c:\hello.txt -ORBInitialHost
          mycomputerName  -ORBInitialPort  2500
```

e-Macao-16-3-453

Summary

1) We introduced general operation of CORBA.
2) Also details of specifying a client, server application, compiling them and registering and running.
3) You will have to configure your system before you try to do these steps.

e-Macao-16-3-454

Project Exercise

1) Implement the controller of your project as a distributed object by using either RMI or JavaIDL.
2) Device an approach through which the controller would locate a requested business object to handle a particular request and also invoke the appropriate operation requested for.
3) Persist your data using mySQL database engine.
4) See how you can make use of Java Message Service in your project.

A.5.2. Message Orientation

Message-Orientation

e-Macao-16-3-456

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

A.5.2.1 JavaMail

JavaMail

e-Macao-16-3-458

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-459

Overview

Email was the Internet's first killer application and still generates more Internet traffic than any protocol except HTTP.

One of the most frequently asked questions about Java is how to send email from a Java applet or application or how to send asynchronous messages between a Java application and homo-sapiens?

We shall be considering:

1) Introduction to JavaMail API
2) Protocols – SMTP, POP, IMAP MIME
3) Installation and configuration
4) Core Classes – Session, Message, Address, Authenticator, Transport, Store and Folder
5) Usage – sending and receiving email, processing HTML messages etc

e-Macao-16-3-460

What Is the JavaMail API?

The JavaMail API is a standard extension to Java that provides a class library for email clients.

Is an optional package (standard extension) for reading, composing, and sending electronic messages.

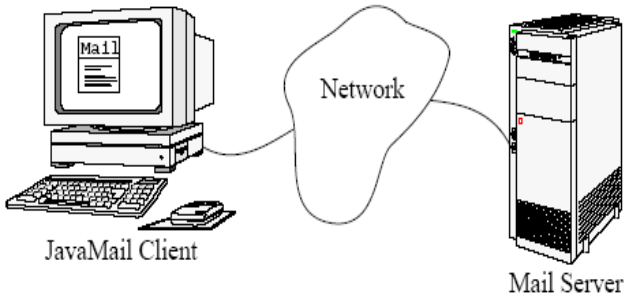
You use the package to create Mail User Agent (MUA) type programs, similar to Eudora, Pine, and Microsoft Outlook.

Purpose:

1) transporting
2) delivering and
3) Forwarding messages like sendmail or other Mail Transfer Agents

e-Macao-16-3-461

Mail Client and Server



e-Macao-16-3-462

Why Mail?

There are situation in which an application may need to send an email

1) an error situation occurs

2) when the next step in some workflow must be started

3) or in response to some events that has occurred

e-Macao-16-3-463

JavaMail Applications

There are several areas in which JavaMail is useful.
Some are discussed below:

1) A server-monitoring application such as Whistle Blower can periodically load pages from a web server running on a different host and email the webmaster if the web server has crashed.

2) An applet can use email to send data to any process or person on the Internet that has an email address, in essence using the web server's SMTP server as a simple proxy to bypass the usual security restrictions about whom an applet is allowed to talk to. In reverse, an applet can talk to an IMAP server on the applet host to receive data from many hosts around the Net.

3) A newsreader could be implemented as a custom service provider that treats NNTP as just one more means of exchanging messages.

e-Macao-16-3-464

Related Protocols 1

There are four protocols are commonly used with the API:

1) Simple Mail Transfer Protocol (SMTP)

2) Post Office Protocol (POP)

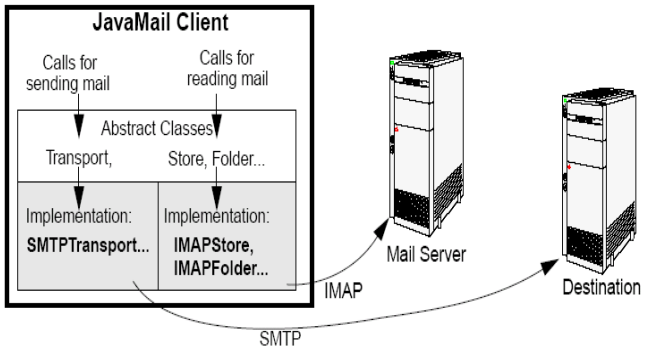
3) Internet Message Access Protocol (IMAP)

4) Multipurpose Internet Mail Extensions (MIME)

Each will be considered.

e-Macao-16-3-465

Related Protocols 2



e-Macao-16-3-466

SMTP

The Simple Mail Transfer Protocol (SMTP) is the mechanism for delivery of email.

In the context of JavaMail,

1) JavaMail-based program will communicate with company or Internet Service Provider's (ISP's) SMTP server.

2) The SMTP server will relay the message on to the SMTP server of the recipient to be acquired eventually by the user through POP or IMAP

e-Macao-16-3-467

POP

Post Office Protocol (POP) is the mechanism most people on the Internet use to get their mail.

It defines support for a single mailbox for each user.

Currently in version 3, also known as POP3

The ability to see how many new mail messages you have, are not supported by POP at all.

These capabilities are built into programs like Eudora or Microsoft Outlook, which remember things like the last mail received and calculate how many are new for you. So, when using the JavaMail API, if you want this type of information, you have to calculate it yourself.

e-Macao-16-3-468

IMAP

Internet Message Access Protocol (IMAP) more advanced protocol for receiving messages.

Currently in version 4, also known as IMAP4

Your mail server must support the protocol before you can use it.

You can't just change your program to use IMAP instead of POP and expect everything in IMAP to be supported.

Assuming your mail server supports IMAP, your JavaMail-based program can take advantage of users having multiple folders on the server and these folders can be shared by multiple users.

e-Macao-16-3-469

IMAP Drawbacks

It places a much heavier burden on the mail server requiring the server to receive the new messages, deliver them to users when requested, *and* maintain them in multiple folders for each user.

While this does centralize backups, as users' long-term mail folders get larger and larger, everyone suffers when disk space is exhausted.

But with POP, saved messages get offloaded from the mail.

e-Macao-16-3-470

MIME

MIME stands for Multipurpose Internet Mail Extensions

It is not a mail transfer protocol.

Instead, it defines the content of what is transferred.

For example:
1) format of the messages
2) attachments, and
3) etc

e-Macao-16-3-471

Installation

There are three versions of the JavaMail API commonly used today:

1) version 1.1.3
2) version 1.2
3) version 1.3.2

Version 1.3.2 is the latest.

The version of the JavaMail API you want to use affects what you download and install.

e-Macao-16-3-472

Installing JavaMail 1.3.2

1) Download `javamail-1_3_2.zip` from
   `http://java.sun.com/products/javamail`
2) Extract the zip file into a folder
3) set it in the `CLASSPATH` environment variable
4) Include the following archive files in the `CLASSPATH`
   a) `imap.jar`
   b) `mailapi.jar`
   c) `pop3.jar`
   d) `smtp.jar`

JavaMail needs a framework in order to complete its functions.

This framework is known as JavaBeans Activation Framework (JAF).

e-Macao-16-3-473

JAF

JavaBeans Activation Framework (JAF) is a standard extension that enables developers who use Java technology to take advantage of standard services:

1) to determine the type of an arbitrary piece of data,
2) encapsulate access to it,
3) discover the operations available on it,
4) and to instantiate the appropriate bean to perform the said operation(s).

It is the basic MIME-type support found in many browsers and mail tools.

e-Macao-16-3-474

Example: JAF

If a browser obtained a JPEG image JAF:

1) enables the browser to identify that stream of data as a JPEG image
2) and from that type, the browser could locate and instantiate an object that could manipulate, or view that image
3) discover the operations available on it,
4) and to instantiate the appropriate bean to perform the said operation(s).

e-Macao-16-3-475

Installing JAF

1) Download `jaf-1_0_2-upd.zip` from `http://java.sun.com/products/javabeans/glasgow/jaf.html`

2) extract the zip file into a folder

3) set it in the `CLASSPATH` environment variable

4) include `activation.jar` in the `CLASSPATH`

e-Macao-16-3-476

Installing JavaMail Using J2EE

JavaMail is bundled with J2EE

There is nothing special you have to do to use the basic JavaMail API.

Just make sure the `j2ee.jar` file is in your `CLASSPATH` and you are set.

Note: This will be deferred to J2EE courses!

e-Macao-16-3-477

Other Referencing Options

If you don't want to change the `CLASSPATH` environment variable:

1) copy the JAR files to your lib/ext directory under the Java Runtime environment (JRE) directory

2) for instance, `%JAVA_HOME%\lib\ext` on a Windows platform

e-Macao-16-3-478

Exercise

1) Download the latest version of the JavaMail API implementation.
2) Download the latest version of the JavaBeans Activation Framework.
3) Extract the zip files to a folder
4) Install the archive files.

e-Macao-16-3-479

Core Classes

There are seven core classes that make JavaMail API:

1) `Session`
2) `Message`
3) `Address`
4) `Authenticator`
5) `Transport`
6) `Store`
7) `Folder`

Each will be considered.

e-Macao-16-3-480

Session

It defines a basic mail session.

It is through this session that everything else works.

The `Session` object takes advantage of a `java.util.Properties` object to get information like mail server, username, password, and other information that can be shared across your entire application.

`Session` class is singleton

e-Macao-16-3-481

Session: Singleton 1

The constructors for the class are private.

An instance of the class can be created in four ways by calling the following methods of the class:

1) `getDefaultInstance(Properties props)`

2) `getDefaultInstance(Properties props,`
`Authenticator authenticator)`

3) `getInstance(Properties props)`

4) `getInstance(Properties props,`
`Authenticator authenticator)`

e-Macao-16-3-482

Session: Singleton 2

Each method returns either a default or new `Session` object.

The (1) and (2) methods get the default instance if one exists and if not a new session object is created.

The (3) and (4) create a new instance.

`props` is the `Properties` object that holds relevant properties

`authenticator` is `Authenticator` object used to call back to the application when a user name and password is needed.

e-Macao-16-3-483

Session: Usage

1) Get a default instance

```
Properties props = new Properties();
// fill props with any information
Session session = Session.getDefaultInstance(props,
                                             null);
```

2) Create a unique session

```
Properties props = new Properties();
// fill props with any information
Session session = Session.getInstance(props, null);
```

In both cases here the `null` argument is an `Authenticator` object.

e-Macao-16-3-484

Message 1

This class models an email message. It is an abstract class.

Subclasses provide actual implementations.

Characteristics:

1) Message implements the `Part` interface.
2) Direct subclass is `MimeMessage`
3) Message contains a set of attributes and a "content".
4) Messages within a folder also have a set of flags that describe its state within the folder.

e-Macao-16-3-485

Message 2

Message defines some new attributes in addition to those defined in the Part interface.

These attributes specify meta-data for the message - i.e., addressing and descriptive information about the message.

Message objects are obtained either from a Folder or by constructing a new Message object of the appropriate subclass.

Messages that have been received are normally retrieved from a folder named "INBOX".

Message is an abstract class, you cannot work with it. Use the subclasses.

e-Macao-16-3-486

MimeMessage

`MimeMessage` is the direct subclass of `Message`

It is an email message that understands MIME types and headers.

Message headers are restricted to US-ASCII characters only, though non-ASCII characters can be encoded in certain header fields.

Once you have your `Session` object, then you can create the message to send.

e-Macao-16-3-487

Creating a Message

1) pass along the `Session` object to the `MimeMessage` constructor.

```
MimeMessage message = new MimeMessage(session);
```

2) set its parts, as `Message` implements the `Part` interface (with `MimeMessage` implementing `MimePart`).

```
message.setContent("Hello", "text/plain");
```

3) If, however, you know you are working with a `MimeMessage` and your message is plain text, then use `setText()` method

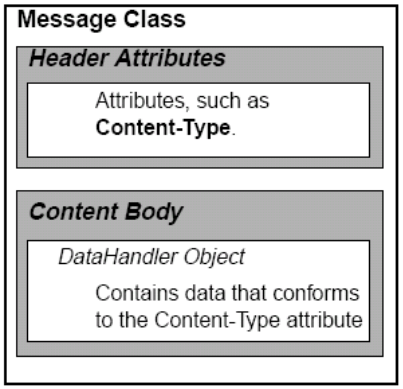
```
message.setText("Hello");
```

4) set the subject using the `setSubject()` method

```
message.setSubject("First");
```

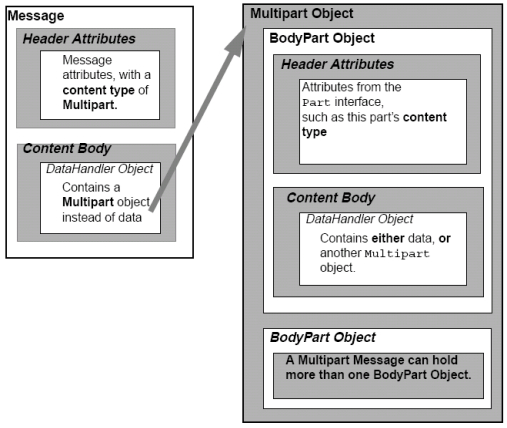
e-Macao-16-3-488

Simple Message



e-Macao-16-3-489

Multipart Message



e-Macao-16-3-490

Address

Once you've created the `Session` and the `Message`, as well as filled the message with content, it is time to address your letter with an `Address`.

This is done using `Address Class`.

Characteristics:

1) like `Message`, `Address` is an abstract class, hence use the subclass

2) you use the `javax.mail.internet.InternetAddress` class

e-Macao-16-3-491

Creating an Address 1

1) To create an address with just the email address, pass the email address to the constructor

```
Address address = new InternetAddress("xx@server.com");
```

2) If you want a name to appear next to the email address

```
Address address = new InternetAddress(xx@server.com,
                                      "Mr. Gabriel");
```

e-Macao-16-3-492

Creating an Address 2

Once you have created the Addresses you connect them to a message in one of two ways:

1) For identifying the sender, you use the `setFrom()` and `setReplyTo()` methods.

```
message.setFrom(address);
```

or If your message needs to show multiple from addresses, use the `addFrom()` method

```
Address address[] = ...;
message.addFrom(address);
```

e-Macao-16-3-493

Creating an Address 3

2) For identifying the message recipients, you use the `addRecipient()` method.

This requires a `Message.RecipientType` besides the address.

The three predefined types of address are:

a) `Message.RecipientType.TO`
b) `Message.RecipientType.CC`
c) `Message.RecipientType.BCC`

e-Macao-16-3-494

Creating an Address 4

```
…

Address toAddress = new
             InternetAddress("president@server.com");

Address ccAddress = new
             InternetAddress("first.lady@server.com");

message.addRecipient(Message.RecipientType.TO,
                                          toAddress);
message.addRecipient(Message.RecipientType.CC,
                                          ccAddress);

…
```

e-Macao-16-3-495

Authenticator

`Authenticator` Class provide access to protected resources (mail server) via a username and password

To use the `Authenticator`, you subclass the abstract class and return a `PasswordAuthentication` instance from the `getPasswordAuthentication()` method.

Example:

```
Properties props = new Properties();
// fill props with any information
Authenticator auth = new MyAuthenticator();
Session session = Session.getDefaultInstance(props,
                                             auth);
```

e-Macao-16-3-496

Transport 1

The final part of sending a message is to use the `Transport` class.

This class speaks the protocol-specific language for sending the message (usually SMTP).

It's an abstract class and works something like `Session`.

There are two ways of sending a message:

1) You can use the default version of the class by just calling the static `send()` method:

```
Transport.send(message);
```

e-Macao-16-3-497

Transport 2

2) You can get a specific instance from the session for your protocol, pass along the username and password (blank if unnecessary), send the message, and close the connection:

```
message.saveChanges(); // implicit with send()
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message,message.getAllRecipients(
                                                    ));
transport.close();
```

e-Macao-16-3-498

Transport 3

This latter way is better when you need to send multiple messages.

It will keep the connection with the mail server active between messages.

The basic `send()` mechanism makes a separate connection to the server for each method call.

e-Macao-16-3-499

Store and Folder 1

Getting messages starts similarly to sending messages:

1) Get a Session Object

2) You connect to a `Store`, quite possibly with a username and password or Authenticator.

3) Like Transport, you tell the `Store` what protocol to use

```
//Store store = session.getStore("imap");
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

e-Macao-16-3-500

Store and Folder 2

4) Get a `Folder`, which must be opened before you can read messages from it:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
```

5) Get its content with `getContent()` or write its content to a stream with `writeTo()`. The `getContent()` method only gets the message content, while `writeTo()` output includes headers.

```
System.out.println(((MimeMessage)message).getContent());
```

e-Macao-16-3-501

Store and Folder 3

6) Once you're done reading mail, close the connection to the folder and store.

```
folder.close(aBoolean);
store.close();
```

The boolean passed to the `close()` method of folder states whether or not to update the folder by removing deleted messages.

e-Macao-16-3-502

Using JavaMail API

We are going to demonstrate the usage of the API with the following:

1) sending messages
2) fetching messages
3) deleting Messages and Flags
4) authenticating Yourself
5) replying to Messages
6) forwarding Messages
7) working with attachments – sending and getting
8) processing HTML Messages – sending and including images

e-Macao-16-3-503

Sending Messages

This involves three steps:

1) getting a session
2) creating and filling a message
3) Send the message using the static `Transport.send()` method

You can specify your SMTP server by setting the `mail.smtp.host` property for the `Properties` object passed when getting the Session

e-Macao-16-3-504

Example: Sending Messages 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.Internet.*;
…
String host = "smtp.macau.ctm.net";
String from = "gab@iist.unu.edu";
String to = "milton@iist.unu.edu";

// Get system properties
Properties props = System.getProperties();

// Setup mail server
props.put("mail.smtp.host", host);
```

e-Macao-16-3-505

Example: Sending Messages 2

```
// Get session
Session session = Session.getDefaultInstance(props,
                                             null);
// Define message
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));
message.setSubject("Hello JavaMail");
message.setText("Welcome to JavaMail");

// Send message
Transport.send(message);
```

e-Macao-16-3-506

Lab Work: Sending Messages

1) Starting with the skeleton code, get the system Properties.
2) Add the name of your SMTP server to the properties for the `mail.smtp.host` key.
3) Get a Session object based on the Properties.
4) Create a MimeMessage from the session.
5) Set the from field of the message.
6) Set the to field of the message.
7) Set the subject of the message.
8) Set the content of the message.
9) Use a Transport to send the message.
10) Compile and run the program, passing your SMTP server, from address, and to address on the command line.

e-Macao-16-3-507

Lab Work : Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class MailExample {
public static void main (String args[]) throws

   Exception {
   String host = args[0];
   String from = args[1];
   String to = args[2];
   // Get system properties
   // Setup mail server
   // Get session
   // Define message
```

e-Macao-16-3-508

Lab Work : Skeleton Code 2

```
            // Set the to address
            // Set the subject
            // Set the content
            // Send message
  }

}
```

e-Macao-16-3-509

Fetching Messages

Reading messages involves five steps:

1) getting a session
2) get and connect to an appropriate store for your mailbox
3) open the appropriate folder
4) get your message(s)
5) and close the connection when done.

e-Macao-16-3-510

Example: Fetching Messages 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.Internet.*;
…
String host = ...;
String username = ...;
String password = ...;

// Create empty properties
Properties props = new Properties();

// Get session
Session session = Session.getDefaultInstance(props,
                                             null);
```

e-Macao-16-3-511

Example: Fetching Messages 2

```
// Get the store
Store store = session.getStore("pop3");
store.connect(host, username, password);
// Get folder
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);

// Get directory
Message message[] = folder.getMessages();

for (int i=0, n=message.length; i<n; i++) {
  System.out.print(i + ": " + message[i].getFrom()[0]);
  System.out.println("\t" + message[i].getSubject());
}
```

e-Macao-16-3-512

Example: Fetching Messages 3

```
// Close connection
folder.close(false);
store.close();
```

This code snippet displays the subjects of the messages.

To display the whole message:
1) you can prompt the user after seeing the from and subject fields,

2) and then call the message's `writeTo()` method if they want to see it

e-Macao-16-3-513

Example: Displaying Content 1

```
BufferedReader reader = new BufferedReader (
                  new InputStreamReader(System.in));

// Get directory
Message message[] = folder.getMessages();
for (int i=0, n=message.length; i<n; i++) {
  System.out.print(i + ": " + message[i].getFrom()[0]);
  System.out.println("\t" + message[i].getSubject());

  System.out.print("Do you want to read message? ");
  System.out.println("[YES to read/QUIT to end]");
  String line = reader.readLine();
}
```

e-Macao-16-3-514

Example: Displaying Content 2

```
  if ("YES".equals(line)) {
    message[i].writeTo(System.out);
  } else if ("QUIT".equals(line)) {
    break;
  }
}
```

e-Macao-16-3-515

Lab Work: Fetching Messages 1

1) Starting with the skeleton code, get or create a Properties object.
2) Get a Session object based on the Properties.
3) Get a Store for your email protocol, either pop3 or imap.
4) Connect to your mail host's store with the appropriate username and password.
5) Get the folder you want to read. More than likely, this will be the INBOX.
6) Open the folder read-only.
7) Get a directory of the messages in the folder. Save the message list in an array variable named message.
8) For each message, display the from field and the subject.
9) Display the message content when prompted.

e-Macao-16-3-516

Lab Work: Fetching Messages 2

10) Close the connection to the folder and store.
11) Compile and run the program, passing your mail server, username, and password on the command line. Answer YES to the messages you want to read. Just hit ENTER if you don't. If you want to stop reading your mail before making your way through all the messages, enter QUIT.

e-Macao-16-3-517

Lab Work : Skeleton Code 1

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class GetMessageExample {
  public static void main (String args[]) throws
                    Exception{
   String host = args[0];
   String username = args[1];
   String password = args[2];
   // Create empty properties
   // Get session
```

e-Macao-16-3-518

Lab Work : Skeleton Code 2

```
   // Get the store
   // Connect to store
   // Get folder
   // Open read-only
  BufferedReader reader = new BufferedReader ( new

      InputStreamReader(System.in));
  // Get directory
  for (int i=0, n=message.length; i<n; i++) {
     // Display from field and subject
     System.out.print("Do you want to read message?");
           System.out.println("[YES to read/QUIT to
  end]");
     String line = reader.readLine();
```

e-Macao-16-3-519

Lab Work : Skeleton Code 3

```
      if ("YES".equals(line)) {
         // Display message content
       } else if ("QUIT".equals(line)) {
           break;
       }
     } // Close connection
   }
}
```

e-Macao-16-3-520

Flags

The `Flags` class represents the set of flags on a `Message`. `Flags` are composed of predefined system flags, and user defined flags.

A System flag is represented by the `Flags.Flag` inner class.

1) `Flags.Flag.ANSWERED`
2) `Flags.Flag.DELETED`
3) `Flags.Flag.DRAFT`
4) `Flags.Flag.FLAGGED`
5) `Flags.Flag.RECENT`
6) `Flags.Flag.SEEN`
7) `Flags.Flag.USER`

Use the `getPermanentFlags()` method of `Folder` class to find out what flags are supported

A User defined flag is represented as a `String`.

e-Macao-16-3-521

Deleting Messages

To delete messages, you set the message's DELETED flag:

```
message.setFlag(Flags.Flag.DELETED, true);
```

Open up the folder in READ_WRITE mode first though:

```
folder.open(Folder.READ_WRITE);
```

Then, when you are done processing all messages, close the folder, passing in a `true` value to expunge the deleted messages.

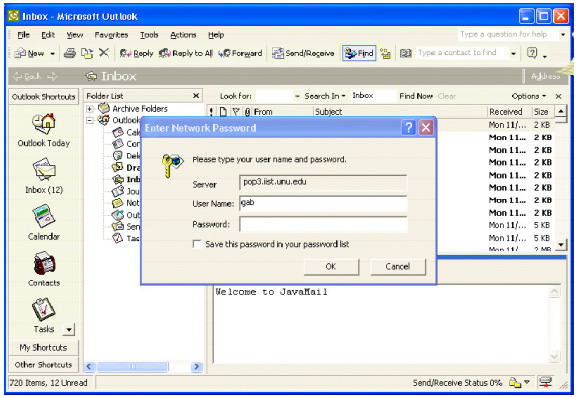
To unset a flag, just pass `false` to the `setFlag()` method.

To see if a flag is set, check with `isSet()`.

e-Macao-16-3-522

Authentication 1

How do you achieve something like this using JavaMail?



e-Macao-16-3-523

Authentication 2

Use an `Authenticator` to prompt for username and password when needed.

Instead of connecting to the `Store` with the host, username, and password, you configure the `Properties` to have the host, and tell the `Session` about your custom `Authenticator` instance.

Example:

```
Properties props = System.getProperties();
props.put("mail.pop3.host", host);

// Setup authentication, get session
Authenticator auth = new PopupAuthenticator();
Session session = Session.getDefaultInstance(props,
                                             auth);
```

e-Macao-16-3-524

Authentication 3

```
// Get the store
Store store = session.getStore("pop3");
store.connect();
```

e-Macao-16-3-525

PopupAuthenticator 1

```
import javax.mail.*;
import javax.swing.*;
import java.util.*;

public class PopupAuthenticator extends Authenticator {

  public PasswordAuthentication
                       getPasswordAuthentication(){
    String username, password;
    String result = JOptionPane.showInputDialog("Enter
                            'username,password'");
    StringTokenizer st = new StringTokenizer(result,
                                             ",");
```

e-Macao-16-3-526

PopupAuthenticator 2

```
    username = st.nextToken();
    password = st.nextToken();
    return new PasswordAuthentication(username,
                                      password);
  }
}
```

e-Macao-16-3-527

Replying to Messages

The `Message` class includes a `reply()` method to configure a new message with the proper recipient and subject, adding "Re: " if not already there.

This does not add any content to the message, only copying the from or reply-to header to the new recipient.

The method takes a `boolean` parameter indicating whether to reply to only the sender (`false`) or reply to all (`true`).

Example:

```
MimeMessage reply = (MimeMessage)message.reply(false);
reply.setFrom(new InternetAddress("xxx@server.com"));
reply.setText("Thanks");
Transport.send(reply);
```

e-Macao-16-3-528

Lab Work: Replying to Messages

1) The skeleton code already includes the code to get the list of messages from the folder and prompt you to create a reply.
2) When answered affirmatively, create a new MimeMessage from the original message.
3) Set the from field to your email address.
4) Create the text for the reply. Include a canned message to start. When the original message is plain text, add each line of the original message, prefix each line with the "> " characters.
5) Set the message's content, once the message content is fully determined. Send the message.
6) Compile and run the program, passing your mail server, SMTP server, username, password, and from address on the command line. Answer YES to the messages you want to send replies. Just hit ENTER if you don't. If you want to stop going through your mail before making your way through all the messages, enter QUIT.

e-Macao-16-3-529

Lab Work: Skeleton Code 1

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class ReplyExample {
   public static void main (String args[]) throws
                                          Exception {
      String host = args[0];
   String sendHost = args[1];
      String username = args[2];
      String password = args[3];
      String from = args[4];
```

e-Macao-16-3-530

Lab Work: Skeleton Code 2

```
      // Create empty properties
      Properties props = System.getProperties();
      props.put("mail.smtp.host", sendHost);
      // Get session
      Session session = Session.getDefaultInstance
                                    (props, null);
      // Get the store
      Store store = session.getStore("pop3");
      store.connect(host, username, password);
      // Get folder
      Folder folder = store.getFolder("INBOX");
      folder.open(Folder.READ_ONLY);
```

e-Macao-16-3-531

Lab Work: Skeleton Code 3

```
BufferedReader reader = new BufferedReader
      ( new InputStreamReader(System.in));
// Get directory
Message message[] = folder.getMessages();
for (int i=0, n=message.length; i<n; i++) {
      System.out.println(i + ": " +
            message[i].getFrom()[0] + "\t" +
                  message[i].getSubject());
      System.out.println("Do you want to reply to
            the message? [YES to reply/QUIT to
                                    end]");
      String line = reader.readLine();
```

e-Macao-16-3-532

Lab Work: Skeleton Code 4

```
      if ("YES".equals(line)) {
            // Create a reply message
            // Set the from field
            // Create the reply content, copying
            //over the original if text
            // Set the content
            // Send the message
      }else if ("QUIT".equals(line)) {
            break;
      }
   }
```

e-Macao-16-3-533

Lab Work: Skeleton Code 5

```
      // Close connection
      folder.close(false);
      store.close();
   }
}
```

e-Macao-16-3-534

Message Parts

A mail message can be made up of multiple parts

Each part is a `BodyPart`, or more specifically, a `MimeBodyPart` when working with MIME messages.

The different body parts get combined into a container called `Multipart` or, again, more specifically a `MimeMultipart`.

e-Macao-16-3-535

Forwarding Message 1

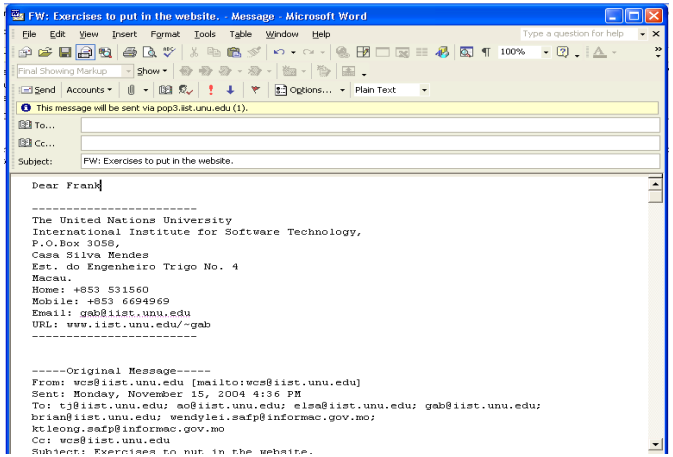
To forward a message:

1) you create one part for the text of your message
2) and a second part with the message to forward,
3) and combine the two into a multipart.
4) Then you add the multipart to a properly addressed message and send it.

To copy the content from one message to another, just copy over its `DataHandler`, a class from the JavaBeans Activation Framework.

e-Macao-16-3-536

Forwarding Message 2



e-Macao-16-3-537

Example: Forwarding Message 1

```
// Create the message to forward
Message forward = new MimeMessage(session);

// Fill in header
forward.setSubject("Fwd: " + message.getSubject());
forward.setFrom(new InternetAddress(from));
forward.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Here you go with the original
                                 message:\n\n");
```

e-Macao-16-3-538

Example: Forwarding Message 2

```
// Create a multi-part to combine the parts
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);

// Create and fill part for the forwarded content
messageBodyPart = new MimeBodyPart();
messageBodyPart.setDataHandler(message.getDataHandler());
// Add part to multi part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
forward.setContent(multipart);
// Send message
Transport.send(forward);
```

e-Macao-16-3-539

Working with Attachments

Attachments are resources associated with a mail message, usually kept outside of the message like a text file, spreadsheet, or image.

With JavaMail you can:

1) attach resources to your mail message with the JavaMail API

2) and get those attachments when you receive the message

e-Macao-16-3-540

Sending Attachments

To send an attachment with your mail

1) Create a new `MimeBodyPart`

2) Create a `DataSource` object. A `DataSource` object is part of JAF defined in `javax.activation` package.

3) Wrap the `DataSource` object in a `DataHandler`. This will allow us to pass the `DataHandler` to the body part object.

e-Macao-16-3-541

Example: Sending Attachments 1

```
// Define message
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));
message.setSubject("Hello JavaMail Attachment");

// Create the message part
BodyPart messageBodyPart = new MimeBodyPart();

// Fill the message
messageBodyPart.setText("Pardon Ideas");

Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
```

e-Macao-16-3-542

Example: Sending Attachments 2

```
// Part two is attachment
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new
                          DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Put parts in message
message.setContent(multipart);

// Send the message
Transport.send(message);
```

e-Macao-16-3-543

Lab Work: Sending Attachment 1

1) The skeleton code already includes the code to get the initial mail session.
2) From the session, get a Message and set its header fields: to, from, and subject.
3) Create a BodyPart for the main message cotent and fill its content with the text of the message.
4) Create a Multipart to combine the main content with the attachment. Add the main content to the multipart.
5) Create a second BodyPart for the attachment.
6) Get the attachment as a DataSource.
7) Set the DataHandler for the message part to the data source. Carry the original filename along.
8) Add the second part of the message to the multipart.

e-Macao-16-3-544

Lab Work: Sending Attachment 2

9) Set the content of the message to the multipart.
10) Compile and run the program, passing your SMTP server, from address, to address, and filename on the command line. This will send the file as an attachment.

e-Macao-16-3-545

Lab Work: Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class AttachExample {
   public static void main (String args[]) throws
  Exception {
     String host = args[0];
     String from = args[1];
     String to = args[2];
     String filename = args[3];
     // Get system properties
     Properties props = System.getProperties();
```

e-Macao-16-3-546

Lab Work: Skeleton Code 2

```
  // Setup mail server
     props.put("mail.smtp.host", host);
  // Get session
     Session session = Session.getInstance(props,
                                                null);
     // Define message
     // Create the message part
     // Fill the message
     // Create a Multipart
     // Add part one //
     // Part two is attachment //
     // Create second body part
```

e-Macao-16-3-547

Exercise : Skeleton Code 3

```
            // Get the attachment
      // Set the data handler to the attachment
      // Set the filename
      // Add part two
            // Put parts in message
      // Send the message
  }
}
```

e-Macao-16-3-548

Getting Attachments

The content of your message is a `Multipart` object when it has attachments.

You then need to process each `Part`, to get the main content and the attachment(s).

Parts marked with a disposition of `Part.ATTACHMENT` from `part.getDisposition()` are clearly attachments.

However, attachments can also come across with no disposition (and a non-text MIME type) or a disposition of `Part.INLINE`.

Just get the original filename with `getFileName()` and the input stream with `getInputStream()`.

e-Macao-16-3-549

Example: Getting Attachments

```
Multipart mp = (Multipart)message.getContent();

for (int i=0, n=multipart.getCount(); i<n; i++) {
  Part part = multipart.getBodyPart(i));
  String disposition = part.getDisposition();

  if ((disposition != null) &&

  ((disposition.equals(Part.ATTACHMENT) ||

      (disposition.equals(Part.INLINE))) {
    saveFile(part.getFileName(), part.getInputStream());
  }
}
```

e-Macao-16-3-550

Lab Work: Implementation

1) Please write the rest of implementation for the Calculator interface.

   Note: If your must extend some other classes other than extending from `UnicastRemoteObject`, the implementation may use the static `exportObject()` method of the `UnicastRemoteObject` instead.

   Be careful that you may need to synchronize some portions of your remotely available method. But it is not necessary for this example.

e-Macao-16-3-551

Attachment: General Case

The code above covers the simplest case where message parts are flagged appropriately.

To cover all cases, handle when the disposition is `null` and get the MIME type of the part to handle accordingly.

```
if (disposition == null) {
  // Check if plain
  MimeBodyPart mbp = (MimeBodyPart)part;
  if (mbp.isMimeType("text/plain")) {
          // Handle plain
  } else {
          // Special non-attachment cases here of
          // image/gif, text/html, ...
  }... }
```

e-Macao-16-3-552

Sending HTML Messages

To send a HTML file as the message and let the mail reader worry about fetching any embedded images or related pieces

1) use the `setContent()` method of Message
2) passing along the content as a `String` and setting the content type to `text/html`.

Example:

```
String htmlText = "<H1>Hello</H1>" +
  "<imgsrc=\"http://www.jguru.com/images/logo.gif\">";
message.setContent(htmlText, "text/html"));
```

e-Macao-16-3-553

Including Images in HTML

if you want your HTML content message to be complete, with embedded images included as part of the message:

1) you must treat the image as an attachment

2) and reference the image with a special cid URL, where the cid is a reference to the Content-ID header of the image attachment.

3) tell the `MimeMultipart` that the parts are related by setting its subtype in the constructor (or with `setSubType()`)

4) and set the Content-ID header for the image to a random string which is used as the src for the image in the `<img>` tag.

e-Macao-16-3-554

Example: Including Images 1

```
String file = ...;

// Create the message
Message message = new MimeMessage(session);

// Fill its headers
message.setSubject("Embedded Image");
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));
// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
String htmlText = "<H1>Hello</H1>" + "<img
                            src=\"cid:mememe\">";
```

e-Macao-16-3-555

Example: Including Images 2

```
messageBodyPart.setContent(htmlText, "text/html");

// Create a related multi-part to combine the parts
MimeMultipart multipart = new MimeMultipart("related");
multipart.addBodyPart(messageBodyPart);

// Create part for the image
messageBodyPart = new MimeBodyPart();

// Fetch the image and associate to part
DataSource fds = new FileDataSource(file);
messageBodyPart.setDataHandler(new DataHandler(fds));
messageBodyPart.setHeader("Content-ID","<memememe>");
```

e-Macao-16-3-556

Example: Including Images 3

```
// Add part to multi-part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
message.setContent(multipart);
```

e-Macao-16-3-557

Lab Work: Sending HTML

1) The skeleton code already includes the code to get the initial mail session, create the main message, and fill its headers (to, from, subject).
2) Create a BodyPart for the HTML message content.
3) Create a text string of the HTML content. Include a reference in the HTML to an image (<img src="…">) that is local to the mail message.
4) Set the content of the message part. Be sure to specify the MIME type is text/html.
5) Create a Multipart to combine the main content with the attachment. Be sure to specify that the parts are related. Add the main content to the multipart.
6) Create a second BodyPart for the attachment.
7) Get the attachment as a DataSource, and set the DataHandler for the message part to the data source.

e-Macao-16-3-558

Lab Work:

8) Set the Content-ID header for the part to match the image reference specified in the HTML.
9) Add the second part of the message to the multipart, and set the content of the message to the multipart.
10) Send the message.
11) Compile and run the program, passing your SMTP server, from address, to address, and filename on the command line. This will send the images as an inline image within the HTML text.

e-Macao-16-3-559

Lab Work: Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class HtmlImageExample {
   public static void main (String args[]) throws
                                    Exception {
      String host = args[0];
      String from = args[1];
      String to = args[2];
      String file = args[3];
```

e-Macao-16-3-560

Lab Work: Skeleton Code 2

```
            // Get system properties
      Properties props = System.getProperties();
            // Setup mail server
      props.put("mail.smtp.host", host);
      // Get session
      Session session = ession.getDefaultInstance(props,

      null);
      // Create the message
      Message message = new MimeMessage(session);
      // Fill its headers
      message.setSubject("Embedded Image");
      message.setFrom(new InternetAddress(from));
```

e-Macao-16-3-561

Lab Work: Skeleton Code 3

```
      message.addRecipient(Message.RecipientType.TO,
      new InternetAddress(to));
      // Create your new message part
      // Set the HTML content, be sure it references
       //the attachment
      // Set the content of the body part
      // Create a related multi-part to combine the
   parts
      // Add body part to multipart
      // Create part for the image
      // Fetch the image and associate to part
      // Add a header to connect to the HTML
      // Add part to multi-part
      // Associate multi-part with message
      // Send message  } }
```

A.5.2.2 Java Message Service

Java Message Service

e-Macao-16-3-563

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-564

Overview

1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

e-Macao-16-3-565

Introduction 1

Information systems are increasingly based on distributed architectures

Needs for integrating existing stand-alone systems are increasing

Middleware is an attempt to ease distributed system development, and try to embedded complexity of communication between programs such as:

a) Different data representations & encodings
b) Different transport protocols
c) Different programming languages, ...

e-Macao-16-3-566

Introduction 2

Types of middleware

1) Procedure-oriented
   a) Client/Server e.g. RPC

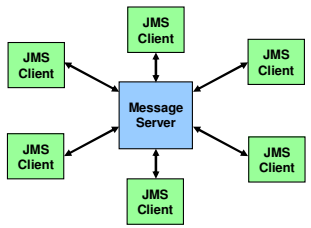
2) Object-oriented
   a) Distributed Objects e.g. CORBA, RMI

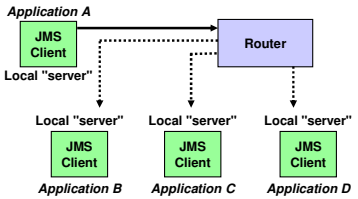
3) Message Oriented Middlewares(MOMs)
   a) Asynchronous messaging e.g. JMS

e-Macao-16-3-567

What is Messaging ?

1) A method of peer-to-peer communication between software components or applications.
2) Enables distributed communication that is loosely coupled; differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.





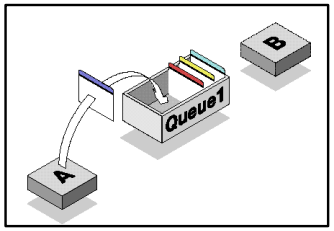
e-Macao-16-3-568

Reliable Messaging With Queues

MOMs provide asynchronous messaging

If one party is unavailable, messaging subsystem is still available and functional

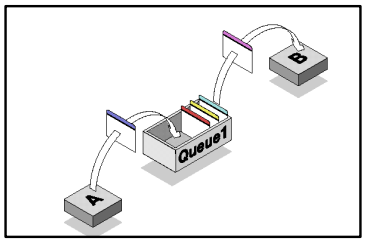
Queues exist independent from the applications

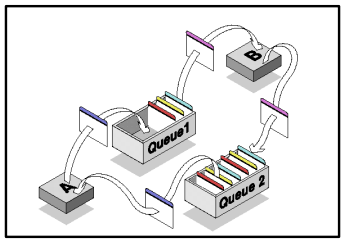


e-Macao-16-3-569

Queuing Basics 1

Queues are uni-directional, but multiple queues may be used to provide bi-directional messaging

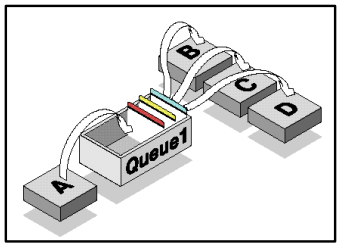


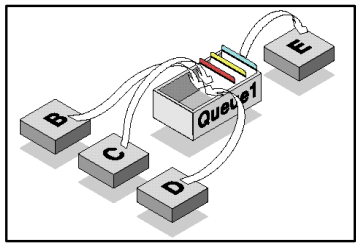


e-Macao-16-3-570

Queuing Basics 2

Queues can work in different models and make one to many and many to one relations possible





e-Macao-16-3-571

The Java Messaging Service 1

A J2EE API to access MOM products from Java

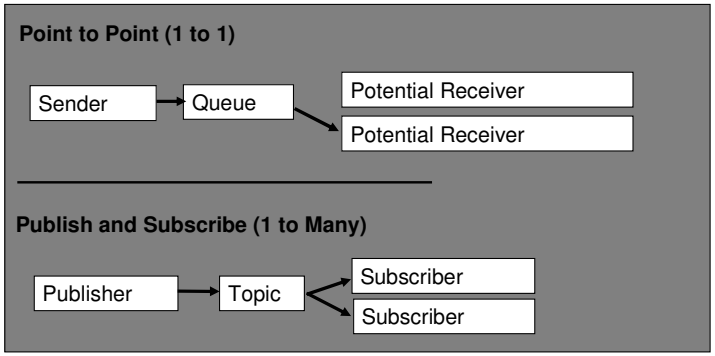
Vendor-neutral API for higher-interoperability

Has two models:

1) Publish and Subscribe
   a) 0 or more recipients
   b) Messages passed between publishers and subscribers via topics
   c) Message can be subscribed to in a durable manner
   d) Message are consumed at least once
2) Point-to-Point
   a) One recipient only
   b) Messages are consumed at most once and only once

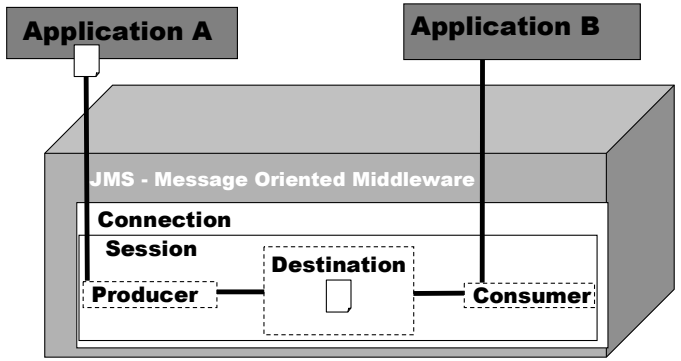
e-Macao-16-3-572

The Java Messaging Service 2



e-Macao-16-3-573

Producer and Consumer



e-Macao-16-3-574

The Promises of JMS

1) "Messaging for the masses"
   a) Could have similar impact that SQL had on databases
2) First enterprise messaging API to achieve wide industry support
3) Simplifies development of enterprise applications
4) Leverages existing enterprise-proven messaging systems
5) Easy to write portable messaging based business applications

e-Macao-16-3-575

Limitations of the JMS

JMS does not address

a) Security
b) Load Balancing
c) Fault Tolerance
d) Error Notification (apart from Exceptions)
e) Administration API
f) Transport protocol for messaging

e-Macao-16-3-576

Overview

1) introduction
2) JMS messaging model
3) JMS programming model and implementation
4) advance configuration
5) summary

e-Macao-16-3-577

JMS API Concepts 1

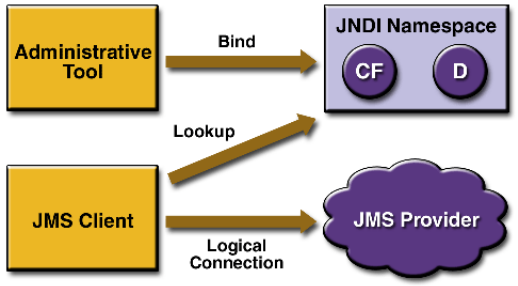
A JMS application is composed of the following parts:

1) JMS Provider
   a) messaging system that implements JMS and administrative functionality, e.g. IBM's MQSeries and JBossMQ message server.
2) JMS Clients
   a) Java programs that send/receive messages
3) Messages
   a) Items of information sent between JMS clients.
4) Administered Objects
   a) preconfigured JMS objects created by an admin for the use of clients
   b) `ConnectionFactory`, `Destination` (`Queue` or `Topic`)

e-Macao-16-3-578

JMS API Concepts 2

Interaction between different parts of JMS:



e-Macao-16-3-579

JMS Messaging Domains

Point-to-Point (PTP)

a) built around the concept of message queues
b) each message has only one consumer

Publish-Subscribe systems

a) uses a "topic" to send and receive messages
b) each message has multiple consumers

e-Macao-16-3-580

Point-to-Point Messaging 1

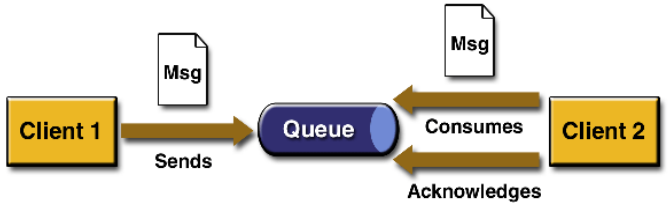
Each message is addressed to a specific queue

Receiving clients extract messages from the queue(s)

Queues retain all messages sent to them until:

a) the messages are consumed

b) the messages expire



e-Macao-16-3-581

Point-to-Point Messaging 2

Characteristics of Point-to-Point Messaging :

a) Each message has only one consumer.

b) A sender and a receiver of a message have no timing dependencies.

c) The receiver acknowledges the successful processing of a message.

d) Should be used when every message send must be processed successfully by one consumer.

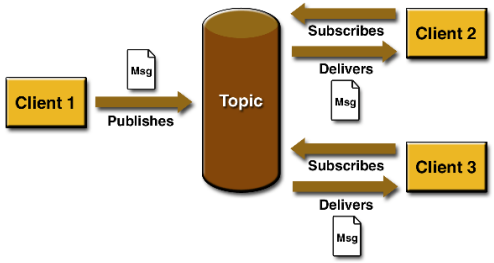
e-Macao-16-3-582

Publish/Subscribe Messaging 1

Clients address messages to a topic.

The system takes care of distributing the messages.

Topics retain messages only as long as it takes to distribute them to current subscribers.



e-Macao-16-3-583

Publish/Subscribe Messaging 2

Characteristics of Pub/Sub Messaging :

a) Each message may have multiple consumers.

b) A client that subscribes to a topic can consume only messages published after the client has created a subscription.

c) The subscriber must continue to be active in order for it to consume messages.

d) Exception for time dependency is allowed for durable subscription. (Will be discussed later)

e-Macao-16-3-584

JMS Message

A JMS message has three parts:

1) Message Header
   a) used for identifying and routing messages
   b) contains vendor-specified values, but could also contain application-specific data
   c) typically name/value pairs
2) Message Properties (optional)
   a) act like additional headers
3) Message Body(optional)
   a) contains the data
   b) five different message body types in the JMS specification

e-Macao-16-3-585

JMS Header

Automatically assigned headers
JMSDestination
JMSDeliveryMode
JMSMessageID
JMSTimestamp
JMSExpiration
JMSRedelivered
JMSPriority

Developer-assigned headers
JMSReplyTo
JMSCorrelationID
JMSType

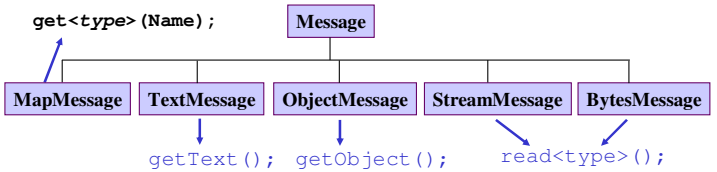
e-Macao-16-3-586

JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble,getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes,writeString,readString
StreamMessage	stream of primitive values	writeString,writeDouble,writeLong,readString
ObjectMessage	serialize object	setObject,getObject

e-Macao-16-3-587

Accessing JMS Message



e-Macao-16-3-588

Messages Consumption

In the JMS Specification, messages can be consumed in either of two ways:

Synchronously

a) A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
b) The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.

Asynchronously

a) A client can register a *message listener* with a consumer.
b) Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

e-Macao-16-3-589

Overview

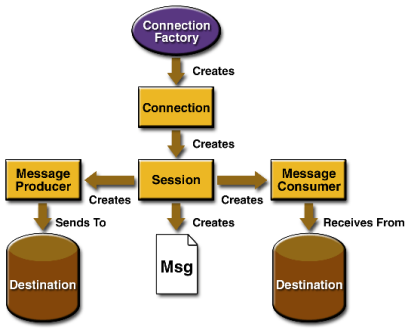
1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

e-Macao-16-3-590

JMS API Programming Model

The basic building blocks of a JMS application:

1) Administered objects
2) Sessions
3) Message producers
4) Message consumers
5) Messages



e-Macao-16-3-591

JMS Client Setup Procedure

A typical pub/sub JMS client executes the following setup procedure:

1) Use `JNDI` to find a `ConnectionFactory` object
2) Use `JNDI` to find one or more Destination objects
3) Use the `ConnectionFactory` to create a JMS Connection
4) Use the Connection to create one or more JMS Sessions
5) Use a Session and the Destinations to create the `TopicPublisher` and `TopicSubscriber` needed
6) Enable the Connection to start delivering messages to `TopicSubscriber`

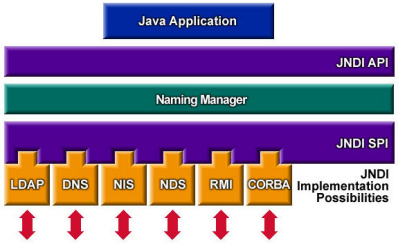
e-Macao-16-3-592

What is JNDI

Java Naming and Directory Interface (JNDI) is an integral component of J2EE technology

JNDI is an application programming interface (API) that provides directory and naming services to Java applications.

JNDI is defined to be independent of any specific naming or directory service implementation. A variety of services can be accessed in a common way.



e-Macao-16-3-593

JNDI package

Following are the JNDI packages:

1) javax.naming
2) javax.naming.directory
3) javax.naming.event
4) javax.naming.ldap
5) javax.naming.spi

e-Macao-16-3-594

Obtain JNDI Connection 1

1) Instantiate an Properties object:

```
Properties env = new Properties();
```

2) Specify the JNDI properties specific to the vendor:

```
env.put("java.naming.factory.initial",
        "org.jnp.interfaces.NamingContextFactory");
env.put("java.naming.provider.url",
        "jnp://localhost:1099");
env.put("java.naming.factory.url.pkgs",
        "org.jboss.naming:org.jnp.interfaces");
```

3) Obtain JNDI Connection

```
Context jndi=new InitialContext(env);
```

e-Macao-16-3-595

Obtain JNDI Connection 2

If a file named jndi.properties is in the classpath of the client program, you can use the following setting:

```
Context jndi = new
        InitialContext(System.getProperties());
```

This can remove the vendor specific code from the client program.

e-Macao-16-3-596

Setup Using JNDI

1) Use `JNDI` to find a `ConnectionFactory` object :

```
TopicConnectionFactory conFactory =
     (TopicConnectionFactory)jndi.lookup
          ("ConnectionFactory");
```

2) Use `JNDI` to find one or more Destination objects :

```
Topic myTopic =
     (Topic)jndi.lookup(topicName);
```

remark: In JBoss, the topic name can be found in the file :

```
<Jboss_Home>\server\default\deploy\jms\jbossmq-
destinations-service.xml
```

e-Macao-16-3-597

Setup Connection and Session

1) Use `ConnectionFactory` to create a JMS Connection

```
TopicConnection connection =
          conFactory.createTopicConnection();
```

2) Use the Connection to create one or more JMS Sessions

```
TopicSession pubSession =
            connection.createTopicSession(false,
              Session.AUTO_ACKNOWLEDGE);
TopicSession subSession =
            connection.createTopicSession(false,
              Session.AUTO_ACKNOWLEDGE);
```

e-Macao-16-3-598

Message Publisher

1) Creating TopicPublisher or MessageProducer

```
// TopicPublisher publisher =
//          pubSession.createPublisher();
MessageProducer producer=
   pubSession.createProducer(myTopic);
```

2) Send a message

```
TextMessage m=pubSession.createTextMessage();
m.setText("just another message");
producer.send(m);
// publisher.publish(m);
```

3) Closing the connection

```
connection.close();
```

e-Macao-16-3-599

Message Subscriber

1) Creating subscriber

```
TopicSubscriber subscriber =
            subSession.createSubscriber(myTopic);
or
MessageConsumer
```

2) Set a JMS message listener

```
subscriber.setMessageListener(<Message Listener>);
```

e-Macao-16-3-600

Message Listener 1

A Message Listener is a class implements interface `javax.jms.MessageListener` and has to implement the `onMessage(javax.jms.Message message)` method

Example of `onMessage` method:

```
public void onMessage(Message message) {
   TextMessage msg = null;
   try {
       if (message instanceof TextMessage) {
           msg = (TextMessage) message;
           System.out.println("Reading message: " +
             msg.getText());
       }
```

e-Macao-16-3-601

Message Listener 2

```
   else {
          System.out.println("Message of wrong type: "
       +  message.getClass().getName());
         }
   } catch (JMSException e) {
          System.out.println("JMSException in
          onMessage(): " + e.toString());
      } catch (Throwable t) {
          System.out.println("Exception in onMessage():"
          + t.getMessage());
      }
   }
```

e-Macao-16-3-602

Start and Close the Connection

Enable the Connection to start delivering messages to `TopicSubscriber`

```
connection.start();
```

Stop the Connection before ending the client program.

```
connection.close();
```

Both methods throws `javax.jms.JMSException`

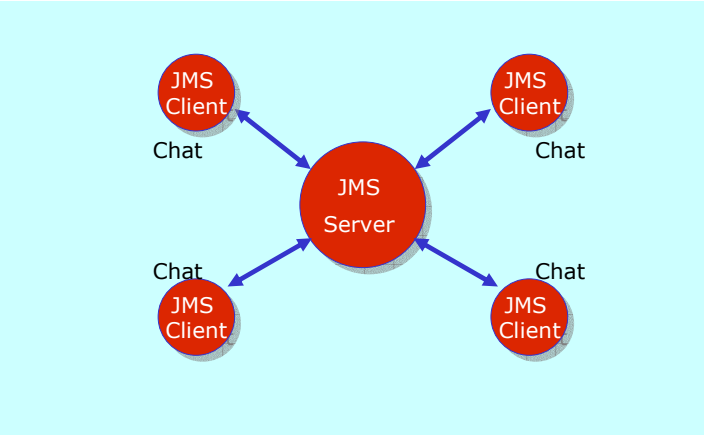
e-Macao-16-3-603

Lab Work: A JMS Chat Client 1

1) According to the procedure we discussed, write a pub/sub chatting program using JMS.
   a) Use JBoss as the JMS server.
   b) Create a topic "emacao" in JBoss. You can modify the file `<JBoss Home>\server\default\deploy\jms\jbossmq-destinations-service.xml` for creating a topic.
   c) Execute the program from the command line:
      1. `Java Chat topic/emacaoD3 username`
      2. Note: for JBoss, the default JNDI name for a topic is `topic/<topic name>`
      3. and for a queue is `queue/<queue name>`.

e-Macao-16-3-604

Lab Work: A JMS Chat Client 2



e-Macao-16-3-605

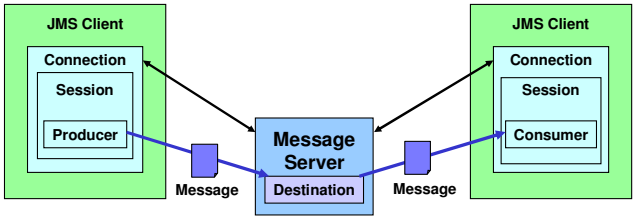
Point-to-Point Messaging 1

Point-to-Point (PTP) application is built around the concept of message queues, sender and receivers.

a) Each message is addressed to a specific queue and the receiving clients extract messages from the queues established to hold their messages.
b) Each message has only one consumer.
c) A sender and receiver have no time dependencies.
d) The receiver acknowledges the successful processing of a message.
e) Use PTP when every message you send must be processed successfully by one consumer

e-Macao-16-3-606

Point-to-Point Messaging 2



e-Macao-16-3-607

Message Queue Sender

1) Performs a Java Naming and Directory Interface (`JNDI`) API lookup of the `QueueConnectionFactory` and `Queue`.
2) Creates a `QueueConnection` and a `QueueSession`.
3) Creating Queue Sender

```
javax.jms.QueueSender
        sender=session.createSender(<queue name>);
```

4) Send a message

```
Message m=session.createTextMessage();
m.setText("just another message");
sender.send(m);
```

5) Closing the connection

```
connection.close();
```

e-Macao-16-3-608

Message Queue Receiver

1) Performs a Java Naming and Directory Interface (`JNDI`) API lookup of the `QueueConnectionFactory` and `Queue`.
2) Creates a `QueueConnection` and a `QueueSession`.
3) Creating Queue Receiver
   ```
   javax.jms.QueueReceiver
     queueReceiver=session.createReceiver(<queue name>);
   ```
4) Starts the connection, causing message delivery to begin
5) Receives the messages sent to the queue until the end-of-message-stream
   - `Message m = queueReceiver.receive();`
   - `Message m = queueReceiver.receive(0);`
6) Closing the connection
   ```
   connection.close();
   ```

e-Macao-16-3-609

Timed Synchronous Receive

If you do not want your program to consume system resources unnecessarily, do one of the following:

1) Call the receive method with a timeout argument greater than 0:
   ```
   Message m = queueReceiver.receive(1); // 1 ms
   ```
2) Call the receiveNoWait method, which receives a message only if one is available:
   ```
   Message m = queueReceiver.receiveNoWait();
   ```
3) The `receive()` method is also available for the `TopicSubscriber` and will negate the use of the `onMessage()` callback.

e-Macao-16-3-610

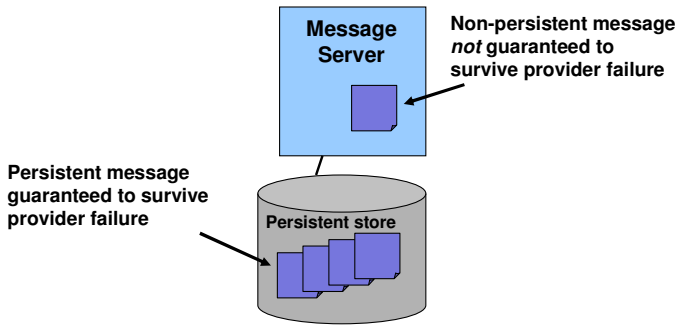
Basic Reliability Mechanisms

1) Specifying message persistence.
   a) You can specify that messages are persistent, meaning that they must not be lost in the event of a provider failure.
2) Controlling message acknowledgment.
   a) You can specify various levels of control over message acknowledgment while creating a session.
3) Setting message priority levels while sending a message.
   a) JMS API defines ten levels of priority from 0 (lowest) to 9 (highest)
4) Allowing messages to expire.
   a) You can specify an expiration time for messages while sending a message.

e-Macao-16-3-611

Persistent Messages

Messages can be marked as persistent.
The implementation of the storage mechanism is up to the JMS provider.



e-Macao-16-3-612

Acknowledgement Modes

Session.AUTO_ACKNOWLEDGE	Indicates that that the message-responses are automitcally generated. Provide guarantee for single delivery to the JMS destination.
Session.DUP_OK_ACKNOWLEGE	Use this mode only when your application can tolerate duplicate message. Reduce the overhead for checking the once-and-only-only delivery.
Session.CLIENT_ACKNOWLEGE	Client needs to call msg.acknowledge() in order to indicate the receive of message.

e-Macao-16-3-613

Exercise: Message Queue

1) Create Point-to-Point messaging program
   a) Create a queue in JBoss with a name qex.
   b) According to our discussion, please create a JMS client for sending message to the qex queue.
   c) Create a Queue Receiver for the qex queue.
   d) Try to stop the receiver and use the sender to send some message. Restart the receiver and check if it receive the message.

e-Macao-16-3-614

Overview

1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

e-Macao-16-3-615

Temporary Topics 1

Is a topic that is dynamically created by the JMS provider, using the `createTemporaryTopic()` of the `TopicSession` object.

Is a topic associated with the connection that belongs to the `TopicSesson` that created it.

It lasts only as long as its associated client connection is active.

Topic identity is transferred using the `JMSReplyTo` header.

e-Macao-16-3-616

Temporary Topics 2

Procedure to create a temporary topic:

1) After a session,`mySession`, is created, the client can create a dynamic topic:

```
javax.jmx.Topic tempTopic =
        mySession.createTemporaryTopic();
```

2) Create a message for the subscriber to reply to:

```
javax.jmx.TextMessage message =
        mySession.createTextMessage();
```

3) Set up the `JMSReplyTo` destination

```
message.setJMSReplyTo(tempTopic);
```

e-Macao-16-3-617

Temporary Topics 3

When a client needs to respond to the message, it can use the `JMSReplyTo` Desination:

```
public void onMessage(javax.jms.Message amessage){
 ...
   TextMessage message = (TextMessage)amessage;
   javax.jms.Topic tempTopic =
     (javax.jms.Topic)message.getJMSReplyTo();
```

e-Macao-16-3-618

Lab Work: Temporary Topic 1

1) Write two JMS clients to simulate the following scenario :

a) An event organizer is constantly promoting events for its agents. It will publish the event message to and deliver to all the subscribed agents.

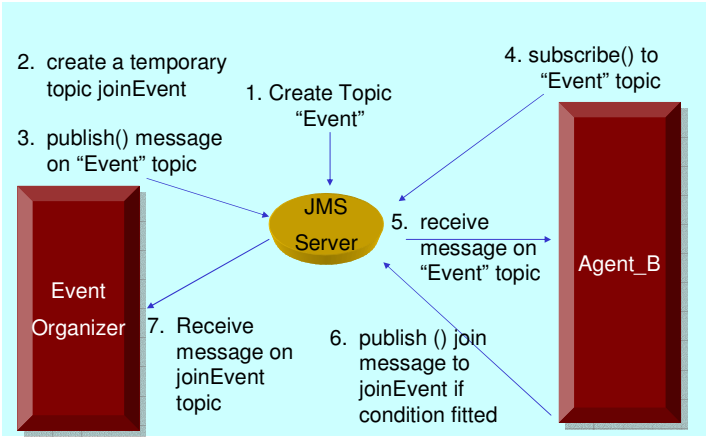
b) After received the message, the Agents' program will evaluate the event according to certain criteria and decide to either joining the event or not. In the exercise, you can make up your own criteria such as cost or date.

c) If the agent decided to join the event, it's program will automatically send a message back to the organizer.

d) Your organizer's program required to create a temporary topic and attached it as the destination for the agents to reply to.

e-Macao-16-3-619

Lab Work: Temporary Topic 2



e-Macao-16-3-620

Durable Subscriptions

By default a subscriber gets only messages published on a topic while a subscriber is alive

Durable subscription retains messages until they are received by a subscriber or expire

You can use the `createDurableScubsriber` method of the `java.jms.TopicSession` to create a durable subscription:

```
...
javax.jms.TopicSubscriber subscriber =
    session.createDurableSubscriber(tempTopic,"subsc
    ription name");
...
```

e-Macao-16-3-621

Unsubscribing

In order to explicitly unsubscribe a subscription, you can use the follow methods:

For nondurable subscription:

```
...
        subcriber.close();
```

For durable subscription:

```
...
        session.unsubscribe
            ("<subscription name>");
```

e-Macao-16-3-622

Lab Work: Durable Subscription

1) Write a JMS client which will send a single message to the topic emacaoD3.
2) Write a JMS Topic subscriber which will subscribe to the topic emacaoD3 and print out all the message received.
3) Modify the Topic subscriber to make it a durable subscriber. What is the difference between this and the subscriber created in item 2?
4) How to let the client created in item 3 to unsubscribe to the topic?

A.6. Summary

Summary

e-Macao-16-3-624

Course Outline

1) Introduction
2) streams
3) Networking
4) Database connectivity
5) architectures
   a) distributed objects
      a) rmi
      b) corba
      c) JavaIDL
   b) message-orientation
      a) Java mail
      b) Java message service
6) summary

e-Macao-16-3-625

Summary

In this course, we covered the following topics:

1) Streams
2) Networking in Java
3) Database connectivity
4) Architectures
    a) distributed objects
        a) rmi
        b) corba
        c) JavaIDL
    b) message-orientation
        a) Java mail
        b) Java message service

e-Macao-16-3-626

Acknowledgements

1) We would like to thank Dr. Tomasz Janowski and Dr. Adegboyega Ojo for their valuable comments.

2) We would also like to thank all the members of eMacao team for their support.

B. Assessment

B.1. Set 1

1	What will be displayed if you try to run the following code and there is no file called Hello.txt in the current directory? `import java.io.*;` `public class Mine {` `  public static void main(String argv[]){` `  Mine m=new Mine();` `  System.out.println(m.amethod());` `  }` `  public int amethod() {` `  try {` `  FileInputStream dis=new FileInputStream("Hello.txt");` `  }catch (FileNotFoundException fne) {` `  System.out.println("No such file found");` `  return -1;` `  }catch(IOException ioe) {` `  }` `  return 0;` `  }` `}`		tick
	a	No such file found	
	b	No such file found ,-1	X
	c	No such file found, Doing finally, -1	
	d	0	

2	Which of the following statements can be used to change the current working directory to a new directory called “DirName” using an instance of the File class called “FileName”?		tick
	a	FileName.chdir("DirName")	
	b	FileName.cd("DirName")	
	c	FileName.cwd("DirName")	
	d	The File class does not support directly changing the current directory.	X

3	Which of the following operations cannot be performed using the File class?		tick
	a	Change the current directory	X
	b	Return the name of the parent directory.	
	c	Delete a file	
	d	Find if a file contains text or binary information.	X

4	Which of the following is not a benefit of distributed programming?		tick
	a	Separation of concern	
	b	Ease of maintenance	

	C	Balance resource loading	
	d	Object serialization	X

5		Which of the following classes is not part of the java.io package?	tick
	a	File	
	b	PushInpuStream	X
	c	InputStream	
	d	RandomAccessFile	

6		Which of the following would read in characters encoded in BIG-5 from a file?	tick
	a	`BufferedReader br = new BufferedReader(new FileReader(“Fileaname”));`	
	b	`BufferedInputStream bis = new BufferedInputStream(new FileReader(new InputStreamReader(“filename”)));`	
	c	`BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream(“filename”)));`	x
	d	`FileReader fr = new FileReader(“fielname”);`	

7		What will happen when you try to compile and run the following code?	tick
		`import java.io.*;` `public class URLConnectionReader {` `  public static void main(String[] args)` `  throws Exception {` `  URL yahoo = new URL("http://www.yahoo.com/");` `  URLConnection yc = yahoo.openConnection();` `  BufferedReader in = new BufferedReader(new InputStreamReader  (yc.getInputStream()));` `  String inputLine` `  while ((inputLine = in.readLine()) != null)` `  System.out.println(inputLine);` `  in.close();` `  }` `}`	
	a	Retrieves URL data from the specified location	
	b	Read and does not display the content of the stream	
	c	Compile time error occurs	x
	d	Prints www.yahoo.com	

8		Which of the following statements are required for database connection?	tick
	a	`DriverManager.registerDriver(Driver driver);`	x
	b	`class.forname(Driver driver).newIntsance();`	x
	c	`DriverManager.getConnection(String url, String ui, String pwd);`	X
	d	`Statement stm = con.createStatement();`	

9	Creating an RMI system involves the following actions except		tick
	a	Implementing java.rmi.Remote interface	
	b	Hosting the RMI service in a server process	
	c	Extending java.rmi.server.UnicastRemoteObject	
	d	Generating skeletons for the client using rmic	x

10	Which of the following is a function of an Object Request Broker?		tick
	a	Implements IDL module on the server	
	b	Mediates between two CORBA objects	x
	c	Prevents function calls across the network to the target object	
	d	Acts as a proxy for the client CORBA object	

11	What will happen when you try to compile and run the following code?		tick

```
import java.io.*;
public class Extension implements FilenameFilter {
  String ext;
  public OnlyExt(String ext){
    this.ext=ext;
  }
  public Boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}

import java.io.*;
class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new Extension("class");
    String s[] = f1.list(only);
    for (int i=0; i < s.length; i++){
        System.out.println(s[i]);
    }
  }
}
```

	a	Prints out the list of classes with .class extension	
	b	Compile time error occurs	x
	c	Prints out the list of classes	
	d	Exception is thrown: `IllegalArgumentException`	

12	The following methods are defined by the `URL` class except		tick
	a	`boolean sameFile(URL other)`	

	B	`Int getPort()`	
	c	`String Userinfo()`	x
	d	`String getFile()`	

13	Which of the following APIs need to be implemented by a developer?		tick
	a	`java.sql.Driver`	
	b	`java.sql.Connection`	
	c	`com.oracle.jdbc.OracleResultSet`	
	d	none	x

14	Which of these message types are not supported by JMS API?		tick
	a	`TextMessage`	
	b	`XMLMessage`	x
	C	`HashMapMessage`	x
	d	`ByteMessage`	

15	Which of the following statements are appropriate for deleting a message from an INBOX folder?		tick
	a	`message.setFlag(Flags.Flag.DELETED, true);` `folder.open();`	
	b	`message.setFlag(Flags.Flag.DELETED, true);` `folder.open(Folder.READ_WRITE);`	x
	c	`message.setFlag(Flags.Flag.DELETED, true);` `store.open(Folder.READ_WRITE);`	
	d	`message.setFlag(Flags.Flag.DELETED, true);` `folder.open(Folder.DELETED);`	

B.2. Set 2

1	What will happen when you try to compile and run the following code?		tick
	`import java.io.*;` `public class URLConnectionReader {` `  public static void main(String[] args)` `  throws Exception {` `  URL yahoo = new URL("http://www.yahoo.com/");` `  URLConnection yc = yahoo.openConnection();` `  BufferedReader in = new BufferedReader(new` `  InputStreamReader(yc.getInputStream()));` `  String inputLine;` `  while ((inputLine = in.readLine()) != null)` `  System.out.println(inputLine);` `  in.close();` `  }` `}`		
	a	It retrieves URL data from the specified location	x
	b	It reads but does not display the content of the stream	
	c	Compile time error occurs	
	d	It prints www.yahoo.com	

2	Which of the following operations can be performed using the File class?		tick
	a	Changing the current directory	x
	b	Returning the name of the parent directory.	
	c	Deleting a file	
	d	Checking if a file contains text or binary information.	

3	Which of the following statements are not required for database connection?		tick
	a	`DriverManager.registerDriver(Driver driver);`	
	b	`Class.forname(Driver driver).newIntsance();`	
	c	`DriverManager.getConnection(String url, String ui, String pwd);`	
	d	`Statement stm = con.createStatement();`	x

4	Creating an RMI system involves the following actions except		tick
	a	Implementing java.rmi.Remote interface	x
	b	Hosting the RMI service in a server process	x
	c	Extending java.rmi.server.UnicastRemoteObject	x
	d	Generating skeletons for the client using rmic	

5	Which of the following is a function of an Object Request Broker?		tick
	a	Implements IDL module on the server	
	b	Mediates between two CORBA objects	x
	c	Prevents function calls across the network to the target object	
	d	Acts as a proxy for the client CORBA object	

6	Which of the following APIs need to be implemented by a developer?		tick
	a	`java.sql.Driver`	
	b	`java.sql.Connection`	
	c	`com.oracle.jdbc.OracleResultSet`	x
	d	none	

7	What will be displayed if you try to run the following code and there is no file called Hello.txt in the current directory?		tick

```
import java.io.*;
public class Mine {
    public static void main(String argv[]){
        Mine m=new Mine();
        System.out.println(m.amethod());
    }
    public int amethod() {
        try {
           FileInputStream dis=new
                            FileInputStream("Hello.txt");
        }catch (FileNotFoundException fne) {
            System.out.println("No such file found");
            return -1;
        }catch(IOException ioe) {
        } finally{
            System.out.println("Doing finally");
        }
        return 0;
    }
}
```

	a	“No such file found” is displayed	
	b	“No such file found ,-1” is displayed	x
	c	“No such file found, Doing finally, -1” is displayed	
	d	0 is displayed	

8	Which of the following classes encodes chars for output?		tick
	a	`java.io.OutputStream`	

	B	java.io.OutputStreamWriter	x
	c	Java.io.BufferedOutputStream	
	d	Java.io.EncodeWriter	

9	Which of these message types are not supported by JMS API?		tick
	a	TextMessage	
	b	XMLMessage	x
	c	HashMapMessage	x
	d	ByteMessage	

10	Which of the following classes are not part of the java.io package?		tick
	a	File	
	b	PushbackInpuStream	x
	c	Inputstream	
	d	RandomAccessFile	

11	What will happen when you try to compile and run the following code?		tick

```
import java.io.*;
public class Extension implements FilenameFilter {
  String ext;
  public OnlyExt(String ext){
    this.ext=ext;
  }
  public Boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}

import java.io.*;
class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new Extension("class");
    String s[] = f1.list(only);
    for (int i=0; i < s.length; i++){
        System.out.println(s[i]);
    }
  }
}
```

	a	The list of classes with .class extension is displayed	
	b	Compile time error occurs	x
	c	The list of classes is displayed	
	d	Exception is thrown: IllegalArgumentException	

12	Which of the following statements can be used to change the current working directory to a new directory called "DirName" using an instance of the File class called "FileName"?		tick
	a	`FileName.chdir("DirName")`	
	b	`FileName.cd("DirName")`	
	c	`FileName.cwd("DirName")`	
	d	The File class does not support directly changing the current directory.	x

13	Which of the following is not a benefit of distributed programming?		tick
	a	Separation of concerns	
	b	Persistence	x
	c	Resource Load balancing	
	d	Ease of Maintenance	

14	The following methods are defined by the `URL` class except		tick
	a	`boolean sameFile(URL other)`	x
	b	`Int getPort()`	
	c	`String Userinfo()`	
	d	`String getFile()`	x

15	Which of the following statements must be executed before you can read a message from an INBOX?		tick
	a	`Transport.send(message);`	
	b	`Session session = Session.getInstance(props, null);` `MimeMessage message = new MimeMessage(session);`	
	c	`Store store = session.getStore("imap");`	
	d	`folder.open(Folder.READ_ONLY);`	x

Distributed Programming

Gabriel Oteniya and Milton Chau Keng Fong

UNU-IIST

The Course

1) objectives - what do we intend to achieve?
2) outline - what content will be taught?
3) resources - what teaching resources will be available?
4) organization - duration, major activities, daily schedule

Course Objectives

1) learn the fundamental concepts of distributed programming for enterprise application development
2) learn the various distributed programming architectures and how to apply them
3) lean the importance of distributed computing and outline the factors to consider when designing a distributed system
4) presents different Distributed Architecture

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Outline: Introduction

Presents an overview of the distributed programming.

Main points:

1) what are distributed systems?.
2) why distributed programming?.
3) nature and design considerations.
4) types of networks.
5) distributed architectures.

Outline: Stream

Presents the `java.io` package

Main points:

1) what is a stream?
2) types of Streams
3) characteristics of Streams
4) working with streams
5) bridging Streams
6) stream chaining

Outline: Networking

Presents the network programming in Java language.

Main points:

1) review the basic network concepts and Java Implementation
2) discuss the usage of java.net package.
3) introduce the Secure Socket.
4) introduce the New I/O API.
5) introduce the Java implementation for UDP protocol.

Outline: Database Connectivity

Presents JDBC API from the basics of SQL to the more esoteric features of advanced JDBC.

Main points:

1) introduction to Database and Structured Query Language

2) JDBC architecture

3) JDBC core interfaces

4) query processing

5) transaction Management

Outline: Message Orientation

Presents how to build a loosely coupled application using messaging mechanism.

Main points:

1) JavaMail – to provide asynchronous communication between application components and human users.
2) Java Message Service – to provide asynchronous/synchronous communication software components

Outline: Distributed Objects

This section basically addresses:

1) Remote Method Invocation (RMI)
2) Common Object Request Broker Architecture (CORBA)
3) Interface Definition Language (IDL)
4) IDL to Java mapping (JavaIDL)

Outline: Summary

Revision of the material introduced during the course.

How this course provides a foundation for the remaining courses:

1) Java XML processing
2) Java Web Services
3) J2EE web components
4) J2EE business components

Course Resources

1) books
   a) Distributed Programming with Java, Qusay H. Mahmoud, Manning Publisher 2000
   b) Java in Distributed Systems: Concurrency, Distribution and Persistence, Marko Boger, 2001
   c) Developing Distributed and E-commerce Applications, 2nd edition, Darrel Ince, 2nd edition, Pearson Addison Westly, 2004.
   d) Java Message Service (O'Reilly Java Series), Richard Monson-Haefel, David Chappell
2) tools
   a) mySQL Database engine
   b) JBoss 4.0.1
   c) JBossMQ
   d) Hermes 1.8 (JBossMQ Browser)

Course Logistics

1) duration - 36 hours
2) activities - lecture (hands-on), development
3) sessions/day - morning 09:00–13:00 and afternoon 14:30–16:30
4) number of sessions - 6 morning and 6 afternoon
5) style – interactive, Lab work and tutorial

Course Prerequisite

1) some experience in object-oriented programming:
   a) C++
   b) Delphi
   c) Java Programming Language
   d) any other object-oriented language
2) basic understanding of TCP/IP networking concepts

Introduction

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Distributed System

Distributed system can be defined as a combination of several computers with separate memory, linked over a network, and on which it is possible to run a distributed applications.

Characteristics:

1) capable of communicating over a network

2) the network is usually stable

3) fail-safe

4) each device has a permanent identification within the network

Hence, it is a collection of independent computers, interconnected via a network, capable of collaborating on a task.

Distributed Application

A distributed application consist of several parts of a program communicating with each other, which cooperate to carry out a common task.

For example, client server application.

Typically, but not necessarily, the parts of the application are distributed across several computers.

The distribution can also be simulated on one computer.

In this case, however, information is not transmitted via a common memory or address space, but with the aid of techniques of remote communication.

Distributed Programming

Distributed programming is a model in which processing occurs in many different places (or nodes) around a network.

Characteristics:

1) processing can occur whenever it makes the most sense

2) carried out on a distributed system

3) making calls to other address spaces possibly on different machines

4) tasks are handled in parallel

Why Distributed Programming?

1) balance resource loading

2) lower cost of development since clients can access remote codes for services

3) separation of concerns

4) Platform independence

Design Considerations

In general, three aspects need to be put into consideration:

1) Concurrency – actual or apparent parallelism of control flows

   issues: how to manage both heavy and light weight processes

2) Distribution – is the logical and spatial distance of objects from each other

   Issue: how these object can locate, access and communicate with each other

3) Persistence – is the long-term storage of data or objects on non-volatile media

   issues: how to persist data and objects. Persistence achives the distribution of data or objects in time.

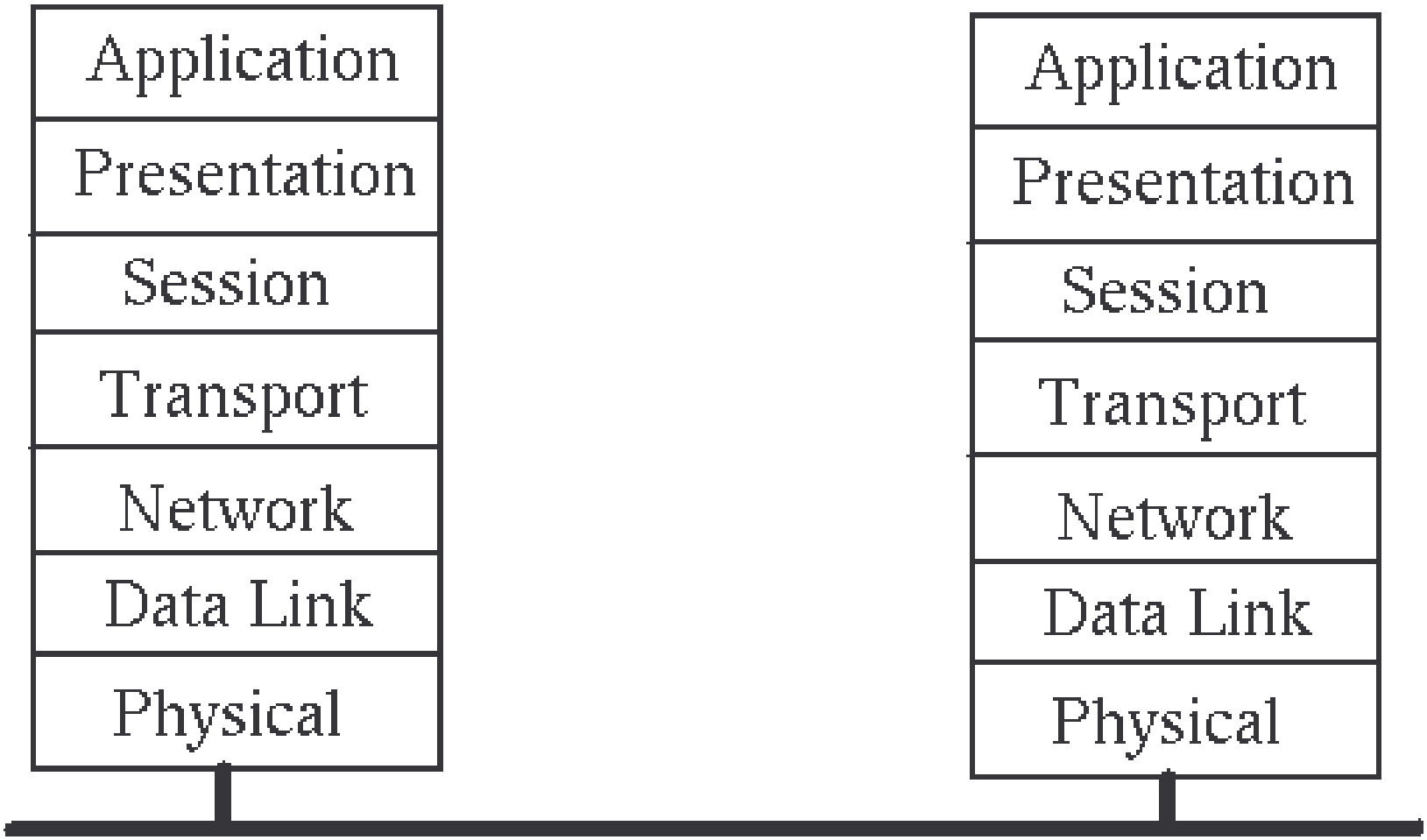
Protocol Layers

1) Communications betweens processes takes place using agreed conventions - protocols

2) Network communications requires protocols to cover high-level application communication all the way down to wire communication

3) Complexity handled by encapsulation in protocol layers

ISO OSI Protocol



OSI layers

1) Network layer provides switching and routing technologies

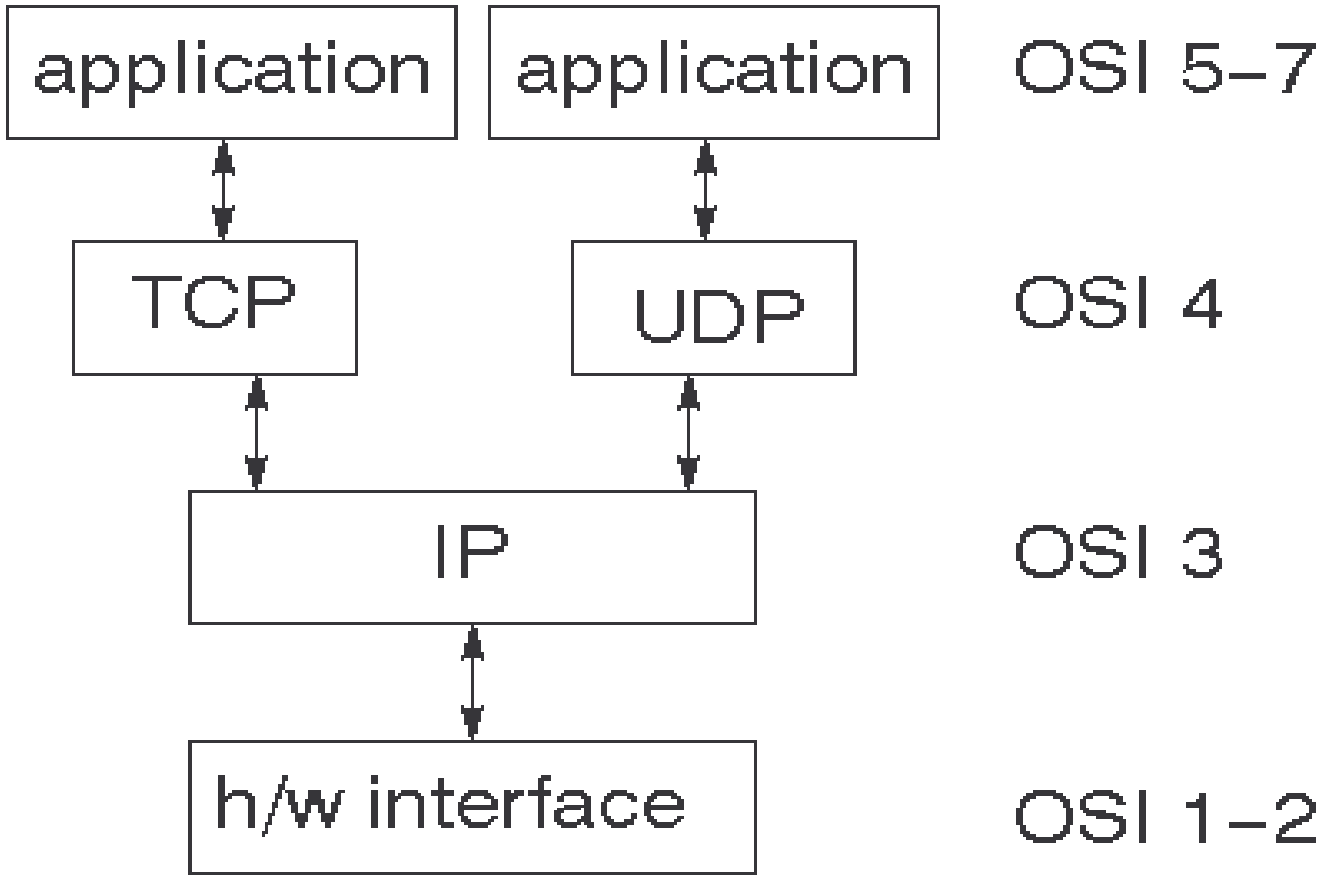
2) Transport layer provides transparent transfer of data between end systems and is responsible for end-to-end error recovery and flow control

3) Session layer establishes, manages and terminates connections between applications.

4) Presentation layer provides independence from differences in data representation (e.g. encryption)

5) Application layer supports application and end-user processes

TCP/IP Protocol



Port and Socket

1) port
   a) conduit into a computer through which information flows and assigned a unique number
   b) usually port numbers 0 to 1023 are reserved for special purposes (e.g. HTTP – 80, FTP – 21, SMTP – 25
   c) TCP/IP-based computer is identified by a pair of IP address and Port number

2) socket
   a) a socket is one end of a process that an application is using to communicate
   b) defined by two addresses: the IP address of the host computer; and the port address of the application or process running on the host

Connection Models

There are two types of connection models:

1) Connection oriented
2) Connectionless

Connection oriented transports may be established on top of connectionless ones –TCP over IP

Connectionless transports my be established on top of connection oriented ones – HTTP over TCP

Connection oriented

1) A single connection is established for the session

2) Two-way communications flow along the connection

3) When the session is over, the connection is broken

4) The analogy is to a phone conversation

5) An example is TCP

Connectionless

1) In a connectionless system, messages are sent independent of each other

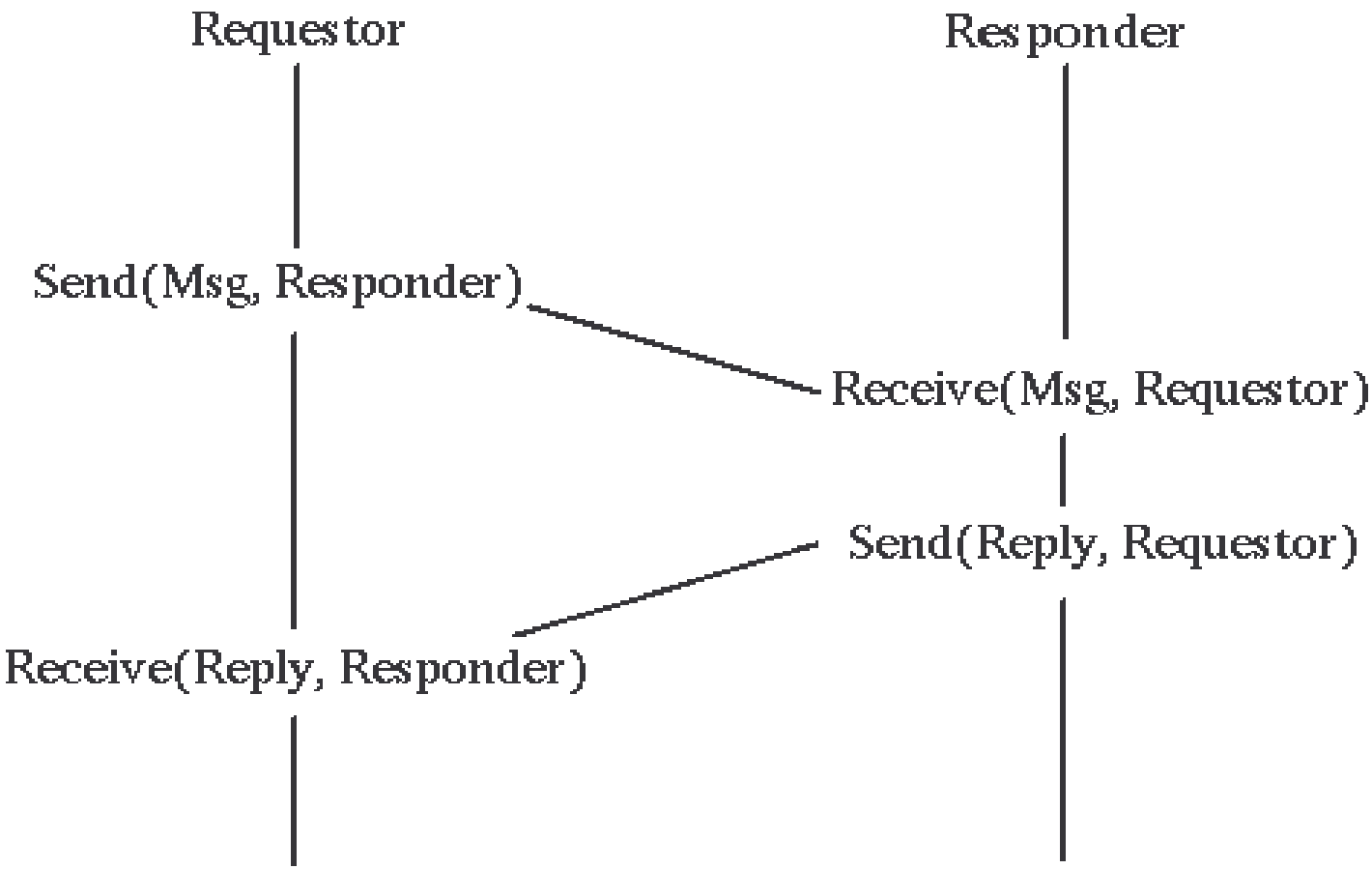
2) Ordinary mail is the analogy

3) Connectionless messages may arrive out of order

4) An example is the IP protocol

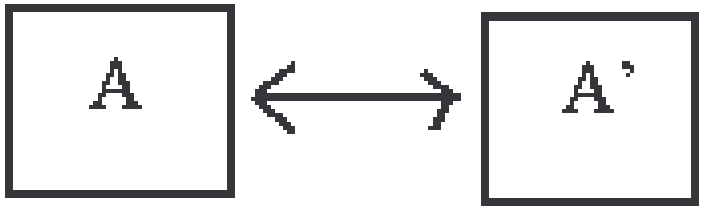
Communications Model

Message passing

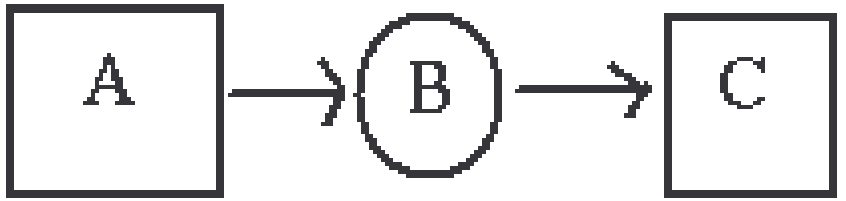


Distributed Computing Models

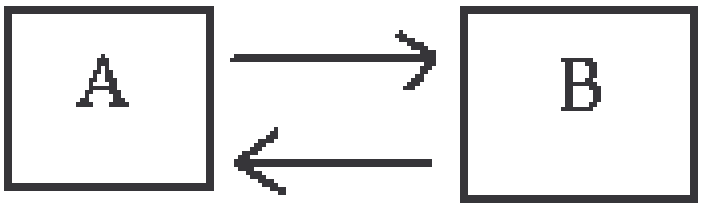
peer–to–peer



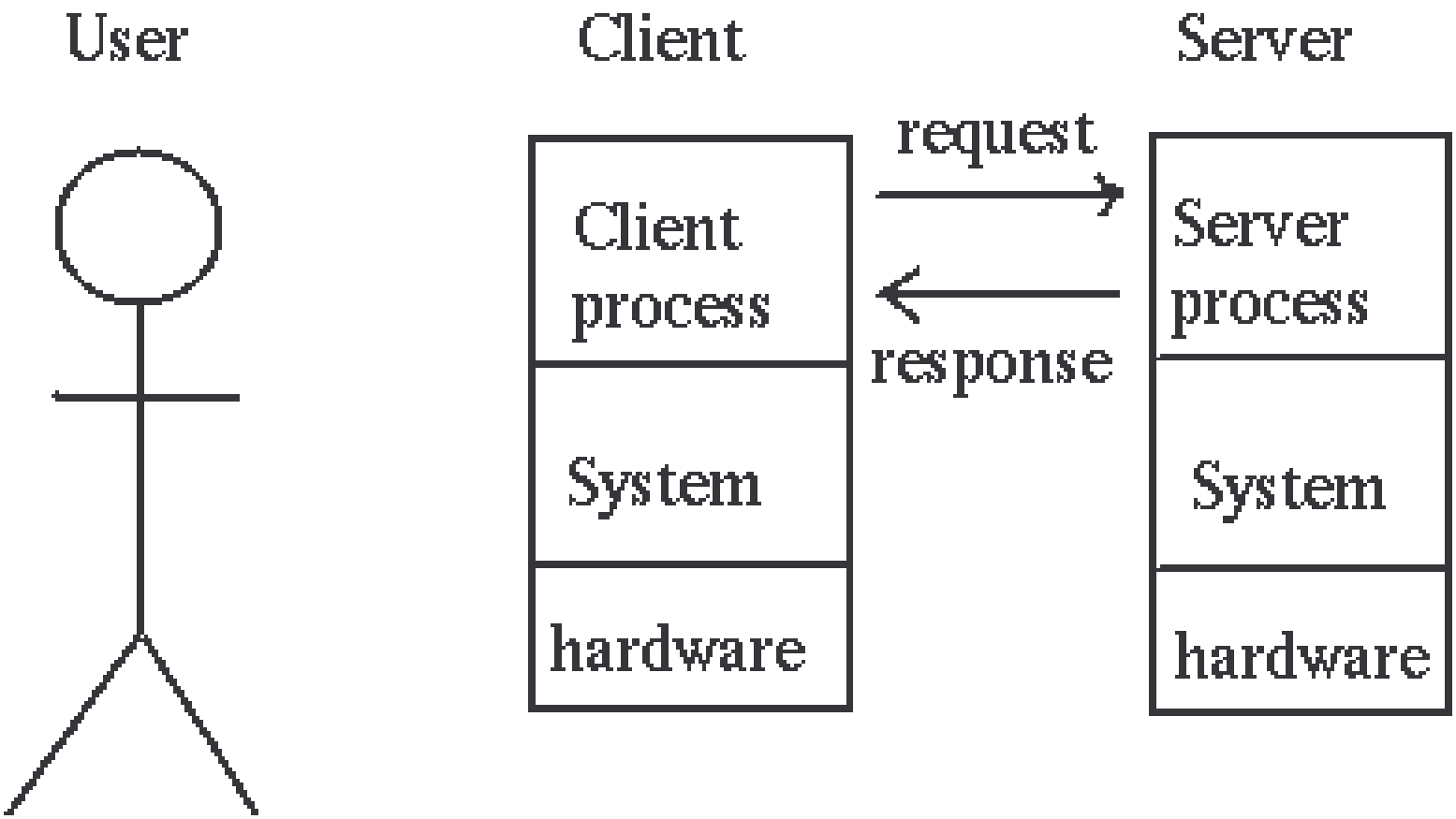
filter



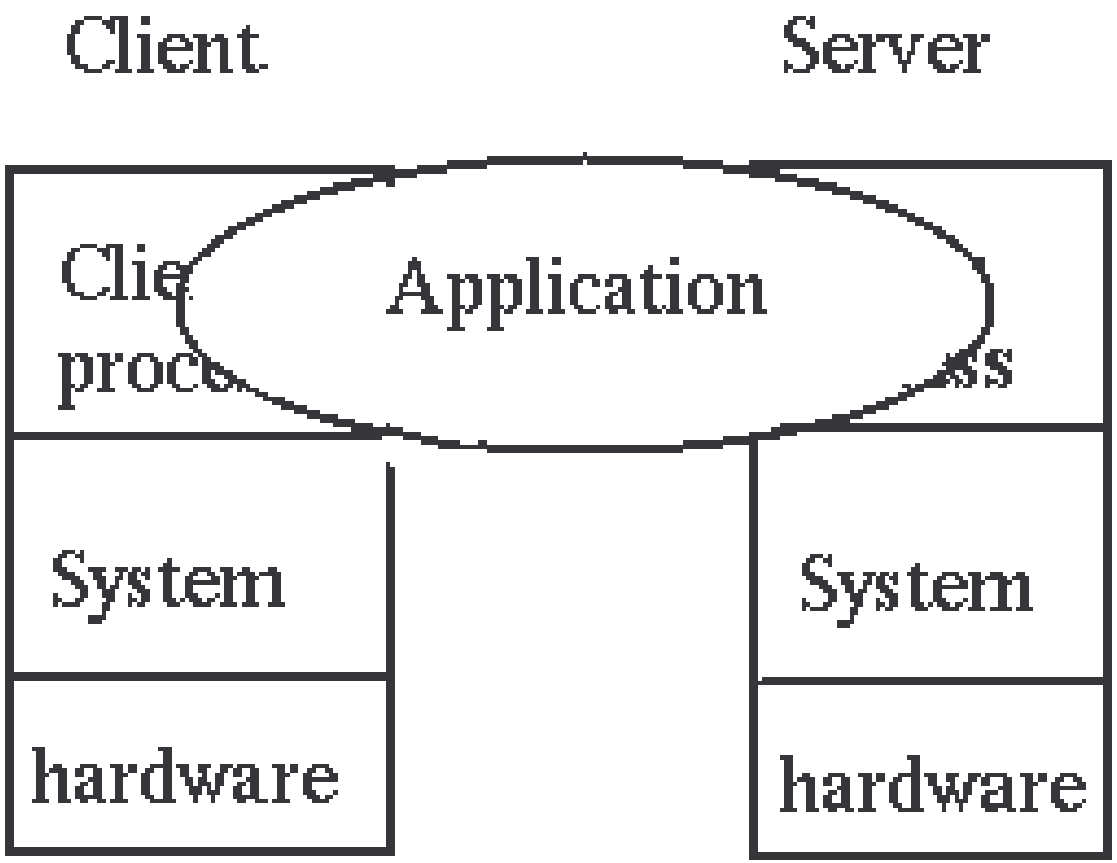
client–server



Client/Server System

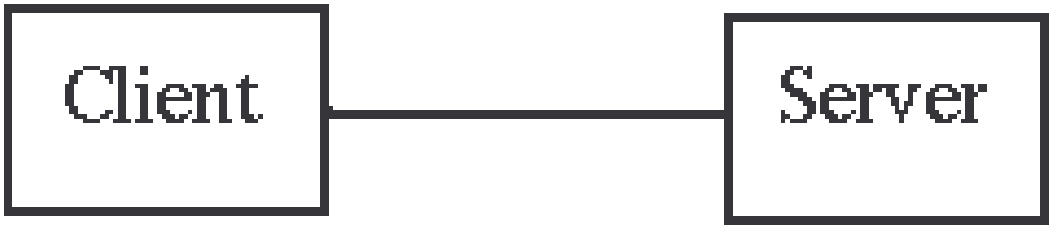


Client/Server Application

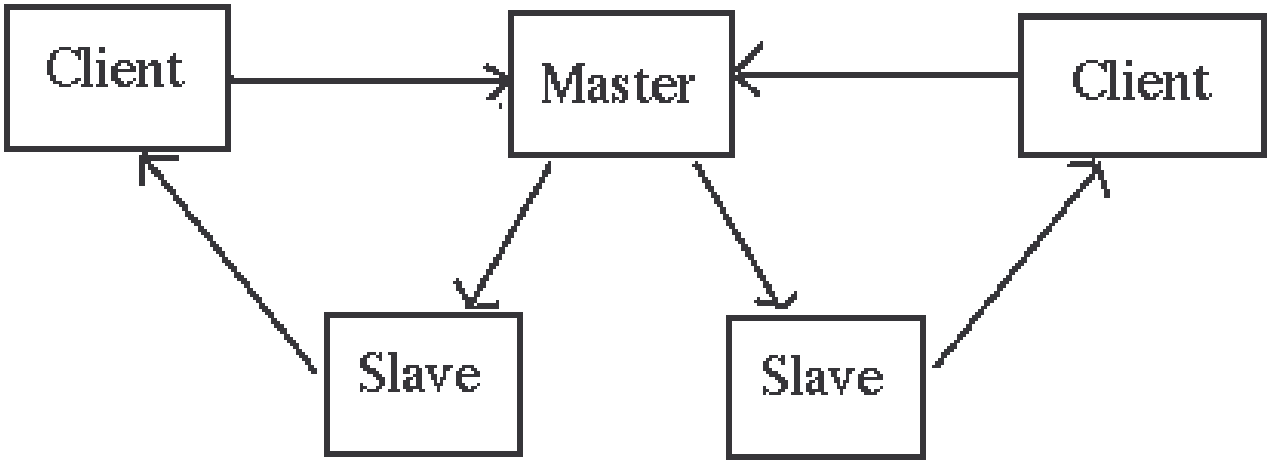


Server Distribution 1

Single client, single server

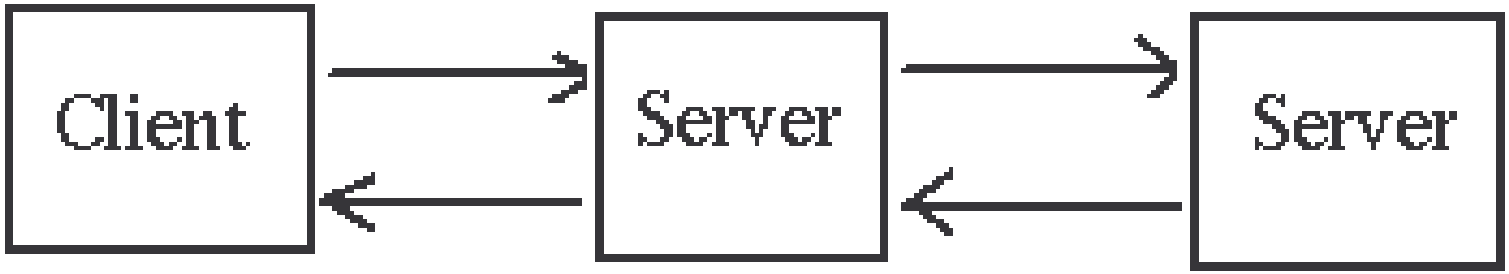


multiple clients, single server



Server Distribution 2

single client, multiple servers



multiple clients, multiple servers

Component Distribution

Every distribution is made up of three components:

1) Presentation component
2) Application logic
3) Data access

Middleware 1

1) intermediate layers between client and server

2) what exactly is it?

    a) a vague term that covers all the distributed software needed to support interactions between client and server

3) where does the middleware start and where does it end?

    a) It starts with the API set on the client side that is used to invoke a service, and it covers the transmission of the request over the network and the resulting response”

Middleware 2

1) The network services include things like TCP/IP

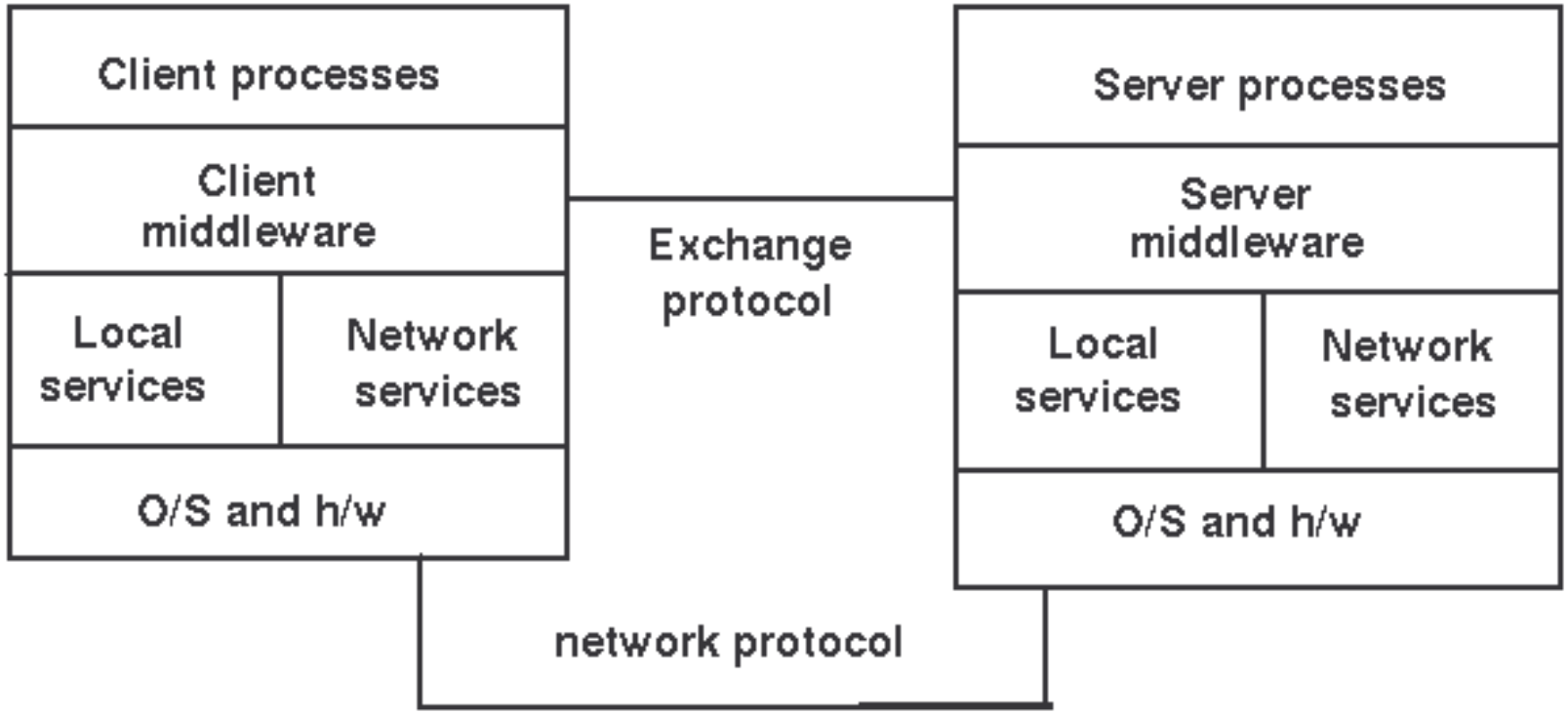
2) The middleware layer is application-independent s/w using the network services

3) Examples of middleware are: DCE, RPC, Corba

4) Middleware may only perform one function (such as RPC) or many (such as DCE)

Middleware Model

The middleware model is



Example: Middleware

1) Primitive services such as terminal emulators, file transfer, email

- Basic services such as RPC

1) Integrated services such as DCE, Network O/S

- Distributed object services such as CORBA, OLE/ActiveX
- Mobile object services such as RMI, Jini
- World Wide Web

Middleware Functions

1) Initiation of processes at different computers

- Session management

1) Directory services to allow clients to locate servers

- remote data access
- Concurrency control to allow servers to handle multiple clients
- Security and integrity
- Monitoring
- Termination of processes both local and remote

Project Exercise 1

1) Describe a typical distributed system in use in your agency
2) Which of the following distributed architecture models best represents the distributed system described in question 1?
3) List the different components of the systems listed in 1
4) Provide a model of the system described in question 1 using a UML deployment diagram showing the various components listed in question two as well as the nodes hosting these components.
5) Identify the possible points of failures in the distributed system using the model presented in question 4.

Streams

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Overview

1) what is a stream?
2) types of Streams
3) characteristics of Streams
4) working with streams
5) bridging Streams
6) stream chaining

Introduction

Most programs use data in one form or another, whether as input, output, or both.

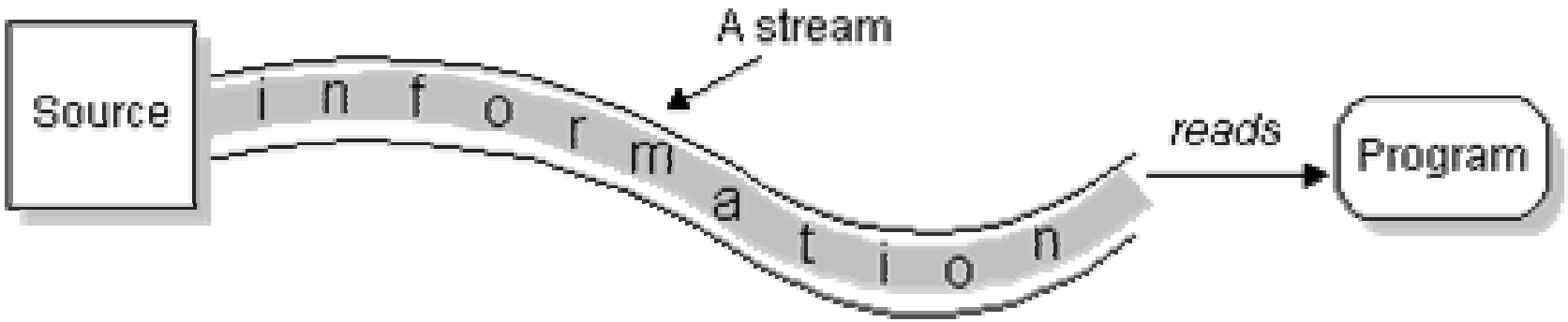
The sources of input and output can vary between a local file, a socket on the network, a database, variables in memory, or another program.

Even the type of data can vary between objects, characters, multimedia, and others.

Reading Data

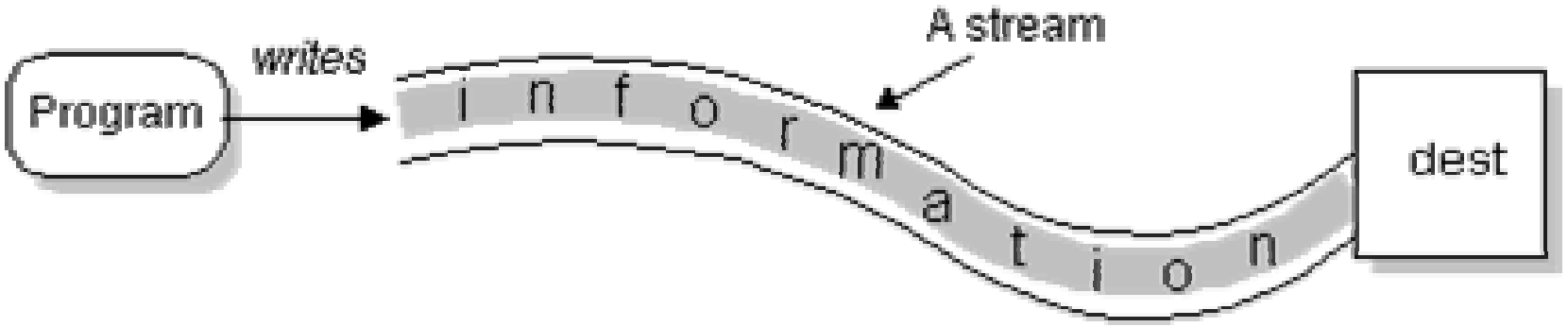
To bring data into a program, a Java program:

1) opens a stream to a data source, such as a file or remote socket
2) and reads the information serially



Writing Data

On the flip side, a program can open a stream to a data source and write to it in a serial fashion.



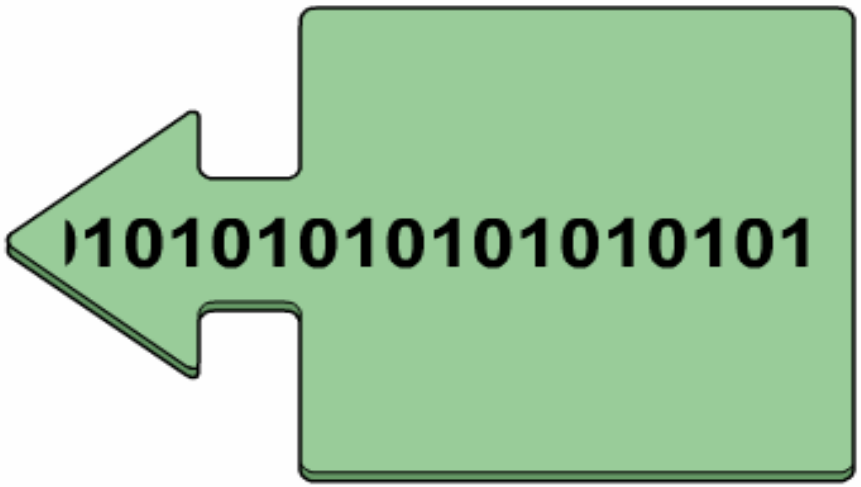
Reading and Writing Data

The concept of serially reading from, and writing to different data sources is the same.

For that very reason, once you understand the top level classes the remaining classes are straightforward to work with.

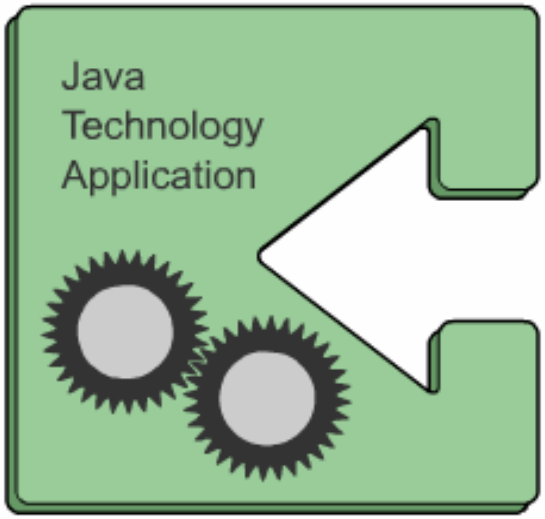
These classes are stored in the `java.io` package.

Streams and Data Sources 1



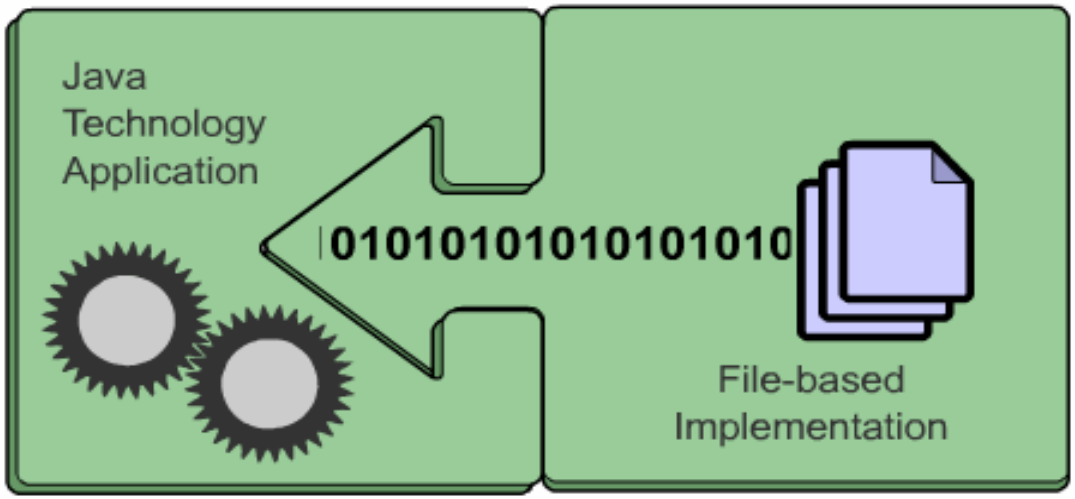
Applications often need to read data to and write data from other sources. Streams provide a means for reading and writing sequences of bytes.

Streams and Data Sources 2



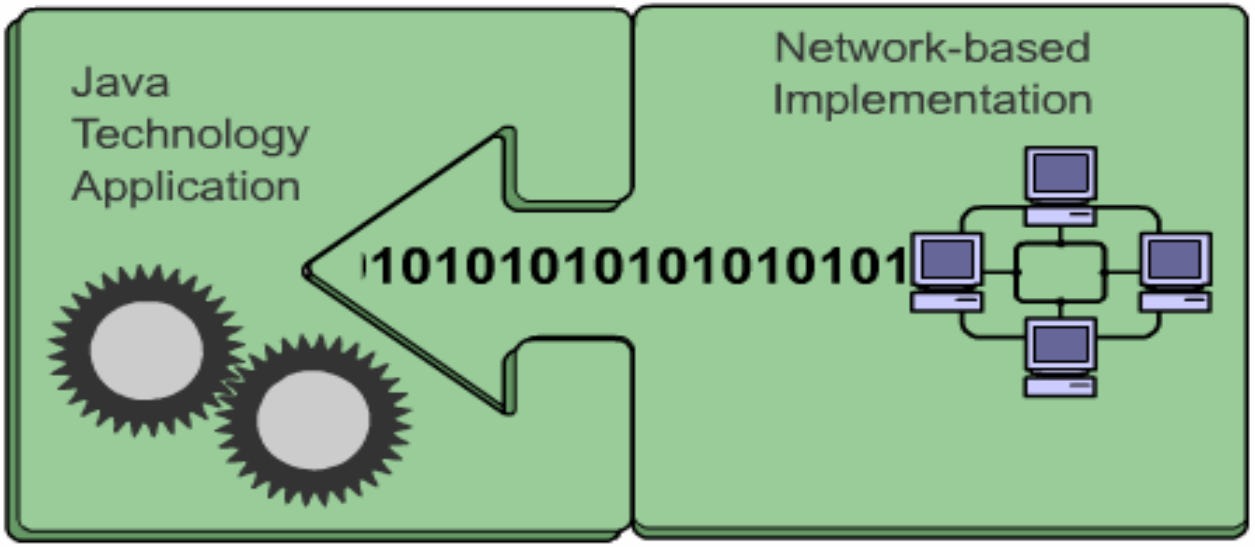
Streams are often used for character values, although this is not the only possibility. The stream concept gives consistent access to any source of data. The source of the data might be a file.

Streams and Data Sources 3



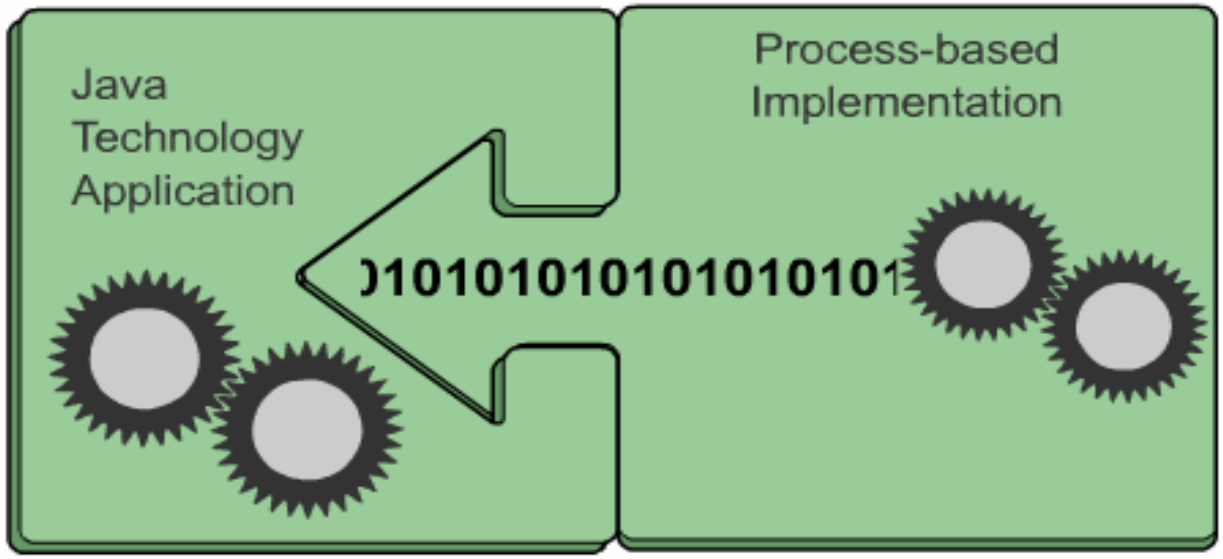
Streams are often used for character values, although this is not the only possibility. The stream concept gives consistent access to any source of data. The source of the data might be a file.

Streams and Data Sources 4



The source of this data might be a network.

Streams and Data Sources 5



The source of this data might even be another process.

Reading and Writing Algorithms

No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same:

Reading:

```
open a stream
while more information
    read information
close the stream
```

Writing:

```
open a stream
while more information
    write information
close the stream
```

Example: Reading Text from File

```
try {
      BufferedReader in = new BufferedReader(new
                                  FileReader("file"));
      String str;
      while ((str = in.readLine()) != null) {
            process(str);
      }
      in.close();
} catch (IOException e) {
      e.printStacktrace();
}
```

Lab Work: Reading a File

1) Based on the code snippet write a program to read a file and displays the content to the console.

Stream Types

There are two categories of streams:

1) 8-bit `byte` streams
2) 16-bit Unicode `character` streams

Prior to JDK 1.1, the input and output classes (mostly found in the `java.io` package) only supported 8-bit `byte` streams.

The concept of 16-bit Unicode `character` streams was introduced in JDK 1.1.

Stream Support

Support for `byte` streams are provided by:

1) `java.io.InputStream` abstract class
2) `java.io.OutputStream` abstract class
3) and their subclasses.

While the support for `character` streams are provided by:

1) `java.io.Reader` abstract class
2) `java.io.Writer` abstract class
3) and their subclasses.

Character versus Byte 1

Most of the functionality available for `byte` streams is also provided for `character` streams.

Methods for character streams generally accept parameters of data type `char` while methods for byte streams accept `byte`.

The names of the methods in both sets of classes are almost identical except for the suffix

1) character-stream classes end with the suffix Reader or Writer

2) byte-stream classes end with the suffix InputStream and OutputStream

Character versus Byte 2

For example:

1) to read files using `character` streams you would use
`java.io.FileReader` class.

2) to read files using `byte` streams you would use
`java.io.FileInputStream`.

Unless you are working with binary data, such as image and sound files, you should use readers and writers (`character` streams) to read and write the data.

Why?

Character versus Byte 3

`Character` streams are always preferred to `byte` streams when reading and writing information because:

1) They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).

2) They are easier to internationalize because they are not dependent upon a specific character encoding.

3) They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

I/O Streams Organization

`java.io` is a large collection of classes, consisting of over 50 classes.

For the purpose of understanding the relationships that exist among these classes, they are categorized using the following criteria:

1) Data flow
2) Function and
3) Type of Data they process

Data Flow

Stream classes that channel data into a program are called input streams.

Example:

1) `FileInputStream`
2) `FileReader`
3) `ObjectInputStream`
4) `PipedInputStream`
5) `etc`.

Stream classes that channel data out of a program are called output streams.

Example:

1) `FileOutStream`
2) `FileWriter`
3) `ObjectOutputStream`
4) `PipedOutputStream`
5) `etc`.

Function

Streams can also be grouped by the function they perform.

There are two categories:

1) Node or Data sink Streams - nature of the resource at the other end of the stream, For example,
   a) `FileInputStream` reads byte data from a file,
   b) `PipedWriter` writes `character` data to a pipe (Thread)

2) Process or Filter Streams - type of processing performed on the contents of the stream,
   For example,
   a) `BufferedReader` to buffer reading to reduce disk/network access
   b) `ObjectOutputStream` for object serialization

Data Sink Streams

`CharArrayReader, CharArrayWriter`	For reading from or writing to character buffers in memory
`FileReader, FileWriter`	For reading from or writing to files
`PipedReader, PipedWriter`	Used to forward the output of one thread as the input to another thread
`StringReader StringWriter`	For reading from or writing to strings in memory
`ByteArrayInputStream ByteArrayOutputStream`	For reading from or writing to byte buffers in memory
`FileInpuStream, FileOutputStream`	For reading from or writing bytes to files
`PipedInputStream, PipedOutputStream`	Used to forward the output of one thread as the input to another thread

Example: Data Sink Streams

Displays contents of a file.

```
import java.io.*;
public class Type{
   public static void main(String args[]) throws
                                                Exception{
      FileReader fr = new FileReader(args[0]);
      PrintWriter pw = new PrintWriter(System.out,
                                                   true);
      char c[] = new char[4096];
      int read = 0;
      while ((read = fr.read(c)) != -1)
             pw.write(c, 0, read);
      fr.close(); pw.close();
   }}
```

Filter Streams

`BufferedReader , BufferedWriter`	For buffered reading/writing to reduce disk/network access for more efficiency
`InputStreamReader , OutputStreamWriter`	Provide a bridge between byte and character streams
`SequenceInputStream`	Concatenates multiple input streams.
`ObjectInputStream , ObjectOutputStream`	Use for object serialization.
`DataInputStream , DataOutputStream`	For reading/writing raw bytes to Java native data types.
`PushbackReader`	Allows to "peek" ahead in a stream by one character.
`LineNumberReader`	For reading while keep tracking of the line number.

Example: Filter Streams

Displays contents of many files

```
import java.io.*;
class cat  {
  public static void main (String args[]) {
      String thisLine;
      for (int i=0; i < args.length; i++) {
      try {
       BufferedReader br = new BufferedReader(new
                                FileReader(args[i]));
       while ((thisLine = br.readLine()) != null) {
         System.out.println(thisLine);
       }
      }catch (IOException e) {
       System.err.println("Error: " + e);
      }}}}
```

Data Type

At the Simplest level, the `java.io` package can be decomposed into classes that process either of two types of data:

1) Byte Stream
2) Character Stream

Fundamental Stream Classes

	Byte Stream	Character Stream
Read Data	`InputStream`	`Reader`
Write data	`OutputStream`	`Writer`

Byte Stream Classes

They process raw bytes.

They come in two basic forms:

1) InputStreams – channel `byte` data into the program

   Example:

   `java.io.FileInputStream`

2) OutputStreams – channel `byte` data from the program

   Example:

   `java.io.FileOutputStream`

Byte-stream classes end with the suffix InputStream and OutputStream

Byte-Stream Parent Classes

`InputStream` and `OutputStream` are the abstract parent classes for byte-stream based classes in the `java.io` package.

Usage:

1) `InputStream` classes are used to read 8-bit byte streams and

2) `OutputStream` classes are used to write to 8-bit byte streams.

Methods for reading and writing to streams:

```
int read()
int read(byte[] b)
int read(byte[] b, int offset, int length)
int write(int b)
int write(byte[] b)
int write(byte[] b, int offset, int length)
```

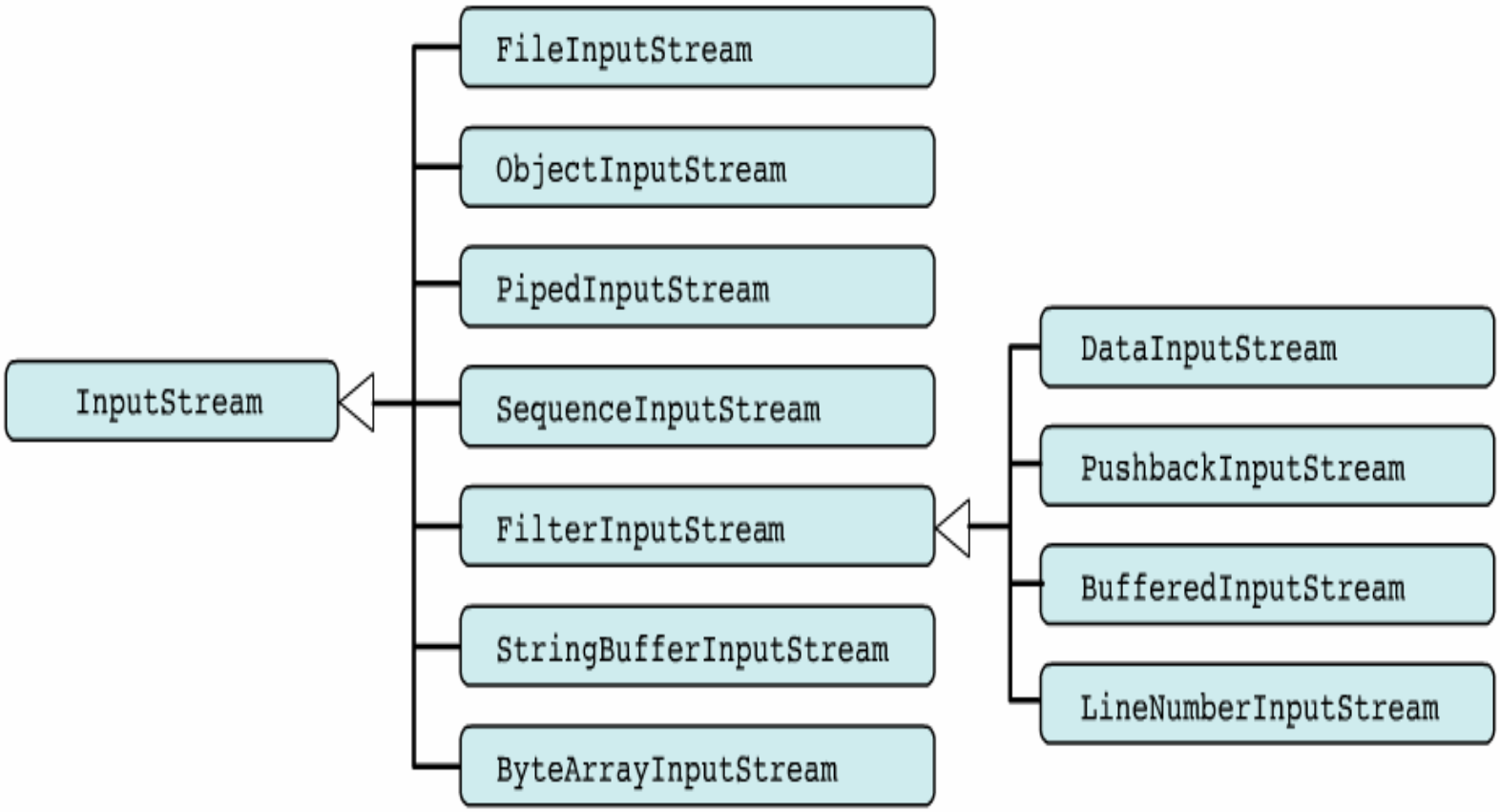
InputStream

This abstract class provides the core methods used to read bytes from an input node.

The methods are:

```
int read()
int read(byte[] b)
int read(byte[] b, int offset, int length)
void close()
int available()
long skip( long l)
boolean markSupported()
void mark( int i)
void reset()
```

InputStream Hierarchy



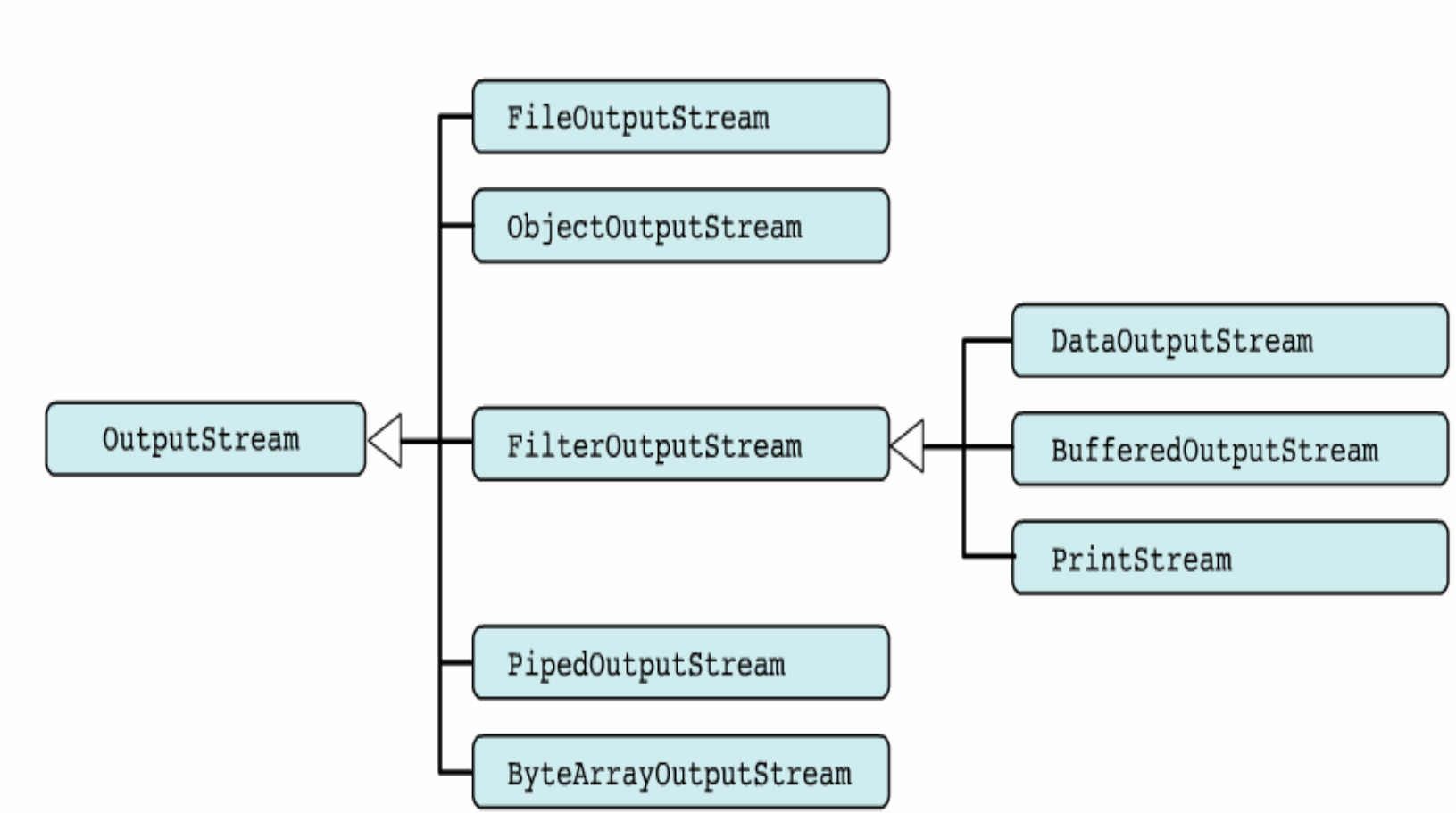
OutputStream

This abstract class provides the core methods used to write bytes to an output node.

The methods are:

```
int write(int b)
int write(byte[] b)
int write(byte[] b, int offset, int length)
void close()
void flush()
```

OutputStream Hierarchy



Data Sink Byte-Stream

Classes that take byte input from different types of nodes (file, pipe, byte array, etc)

Example:

```
FileInputStream
PipedInputStrean
```

Classes that send byte output to different types of node (file, pipe, byte array, etc)

Example:

```
FileOutputStream
PipedOutputStream
```

Example: Data Sink Byte-Stream

1) `FileInputStream`/`FileOutputStream`

   Usage: is meant for reading/writing streams of raw bytes such as image data to/from files

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new
BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

2) `ByteArrayInputStream`/`ByteArrayOutputStream`

   Usage: are useful for holding data when the underlying data type is irrelevant to the purpose of the application

```
byte[] c
...
ByteArrayInputSteam r = new ByteArrayInputSteam(c);
```

Example: Data Sink Byte-Stream

3) `PipedInputStream`/`PipedOutputStrea`

Usage: read from or write to pipes. Often used to exchange data between threads.

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

Or

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

Or

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
pi.connect(po);
```

Example: FileInputStream 1

```
import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) throws
   Exception {
    int size;
    InputStream f =
      new FileInputStream("FileInputStreamDemo.java");

    System.out.print("Total Available Bytes: " )
     System.out.println((size = f.available()));
    int n = size/40;
    System.out.println("First " + n +
          " bytes of the file one read() at a time");
    for (int i=0; i < n; i++) {
      System.out.print((char) f.read());
    }
```

Example: FileInputStream 2

```
System.out.println("\nStill Available: " +
                                   f.available());
System.out.println("Reading the next " + n +
                    " with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
  System.err.println("couldn't read " + n + "
                                     bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size =
                                   f.available()));
System.out.println("Skipping half of remaining
                           bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " +
                                   f.available());
```

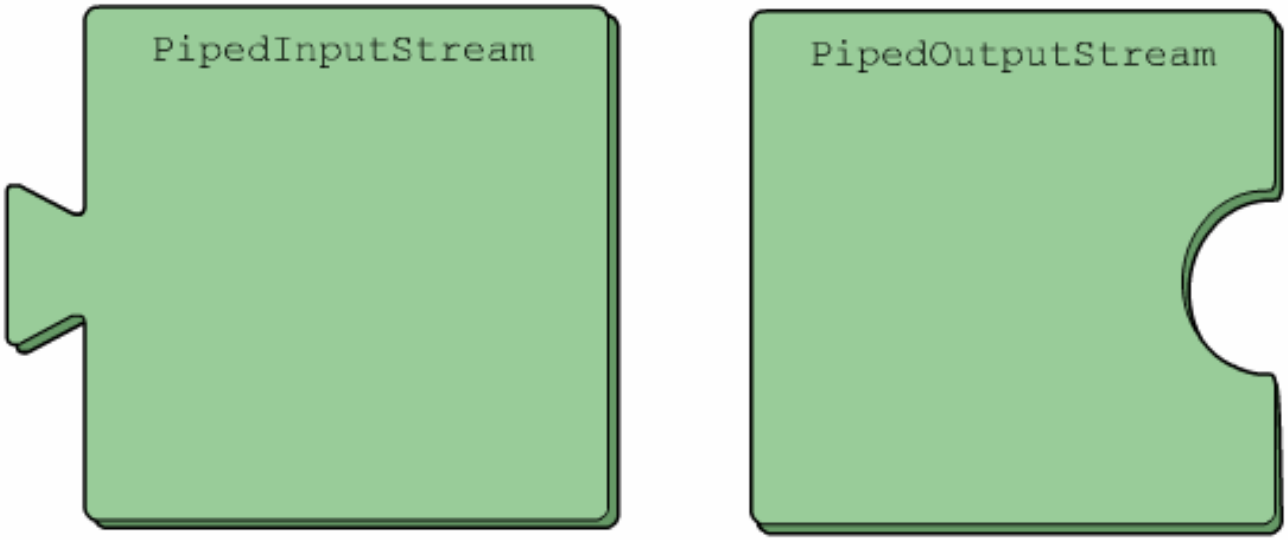
Example: FileInputStream 3

```
    System.out.println("Reading " + n/2 + " into the
                                    end of array");
    if (f.read(b, n/2, n/2) != n/2) {
      System.err.println("couldn't read " + n/2 + "
                                      bytes.");
    }
    System.out.println(new String(b, 0, b.length));
    System.out.println("\nStill Available: " +
                                    f.available());
    f.close();
  }
}
```

Lab Work: Reading a File

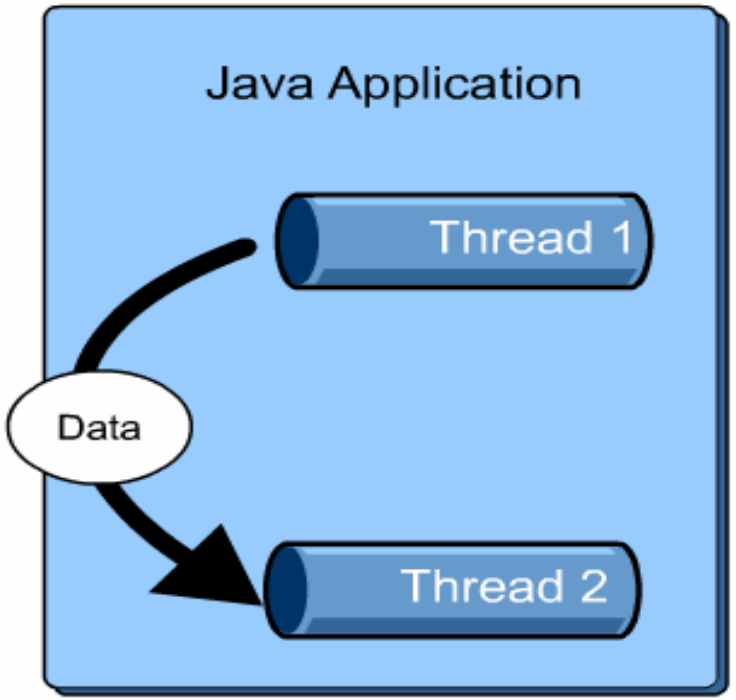
1) Write a program that reads an MP3 file.

PipedInput/Output Stream 1



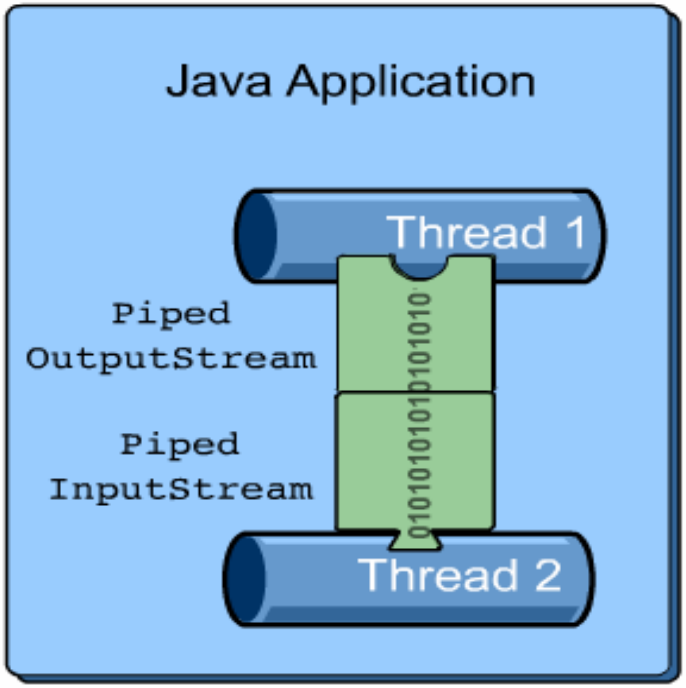
The `PipedInputStream` and `PipedOutputStream` classes are designed to be used as a pair. They allow a mechanism for communication among threads, although they have other uses, too.

PipedInput/Output Stream 2



To illustrate how the piped streams can be used, suppose you have an application which must transfer data between two separate threads.

PipedInput/Output Stream 3



The `PipedInputStream` /`PipedOutputStream` are created as a pair, so that what is written into the output stream can be read from the input stream. The effect is much like a pipeline that carries data from one point in an application to another.

Example: Piped Stream 1

Shows how to exchange data between two threads

```
import java.io.*;

class ReadThread extends Thread implements Runnable {
      InputStream pi = null;
      OutputStream po = null;
      String process = null;
      ReadThread( String process, InputStream pi,
                                    OutputStream po) {
            this.pi = pi;
            this.po = po;
            this.process = process;
      }
```

Example: Piped Stream 2

```
public void run() {
      int ch;
      byte[] buffer = new byte[512];
      int bytes_read;
      try {
            for(;;) {
                  bytes_read = pi.read(buffer);
                  if (bytes_read == -1) { return; }
                  po.write(buffer, 0, bytes_read);
            }
      } catch (Exception e) {
                  e.printStackTrace();
      } finally {  }
   }

}
```

Example: Piped Stream 3

```
class SystemStream {
 public static void main( String [] args) {
    try {
      int ch;
      while (true) {
        PipedInputStream  in = new PipedInputStream();
        PipedOutputStream out = new PipedOutputStream(
                                                 in );
        FileOutputStream writeOut = new
                                FileOutputStream("out");

        ReadThread rt = new ReadThread("reader",
                                      System.in, out );
        ReadThread wt = new ReadThread("writer",  in,
                                          System.out );
        rt.start();
        wt.start();
      }
```

Example: Piped Stream 4

```
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Filter Byte-Stream

They convert bytes to primitive data

They write primitive data

Example:

```
BufferedInputStream/BufferedOutputStream
DataInputStream/DataoutputStream
```

Example: Filter Byte-Stream 1

1) `BufferedInputStream/BufferedOutputStream`

   Usage:

   These classes buffer data emanating from an InputStream object or in route to an OutputStream object.

Benefits:

a) Improved performance - Buffered streams cache data to reduce the need to access slower transmission media.

b) Simplicity - Buffered streams manage the data cache themselves, so you do not have to.

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

Example: Filter Byte-Stream 2

2) `DataInputStream/DataoutputStream`

Usage:

These classes transform bytes emanating from an InputStream type into primitives (such as int, long, or double) or primitives in route to an OutputStream type into bytes.

Attach a `DataInputStream` filter to an InputStream object when you need to read primitives from a stream.

Attach a `DataOutputStream` filter to an OutputStream object when you need to write primitives to a stream.

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

Example: BufferedInputStream 1

```
import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) throws
  IOException {
    String s = "This is a &copy; copyright symbol " +

      "but this is &copy not.\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new


      ByteArrayInputStream(buf);
    BufferedInputStream f = new BufferedInputStream(in);
    int c;
    boolean marked = false;
```

Example: BufferedInputStream 2

```
while ((c = f.read()) != -1) {
  switch(c) {
  case '&':
    if (!marked) {
      f.mark(32);
      marked = true;
    } else {
      marked = false;
    }
    break;
  case ';':
    if (marked) {
      marked = false;
      System.out.print("(c)");
    } else
```

Example: BufferedInputStream 3

```
      System.out.print((char) c);
      break;
    case ' ':
      if (marked) {
        marked = false;
        f.reset();
        System.out.print("&");
      } else
        System.out.print((char) c);
      break;
  default:
    if (!marked)
        System.out.print((char) c);
    break;
  }}}}
```

Serialization

Serialization is a process of writing an object to a byte stream.

Writing an Object

```
FileOutputStream out = new FileOutputStream("tmp");
ObjectOutput objOut = new ObjectOutputStream(out);
objOut.writeObject(Color.red);
```

Reading an Object

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream objIn = new ObjectInputStream(in);
Color c = (Color)objIn.readObject();
```

Object Serialization 1

Provides a way for objects to be written as a stream of bytes and then later recreated from that stream of bytes.

The job of an `ObjectInputStream` class is to convert collections of bytes into objects.

Sending an object over a stream was a cumbersome process. How?

Essentially, you had to decompose the object into its constituent parts, sending each to the stream individually, and then reconstruct the object manually at the other end of the stream.

Process is cumbersome. Solution?

Object Serialization 2

The introduction of new interface to the `java.io` package, the `Serializable` interface

The `Serializable` interface eliminates the drawbacks of sending objects across streams.

Each object to be sent has to implement this interface

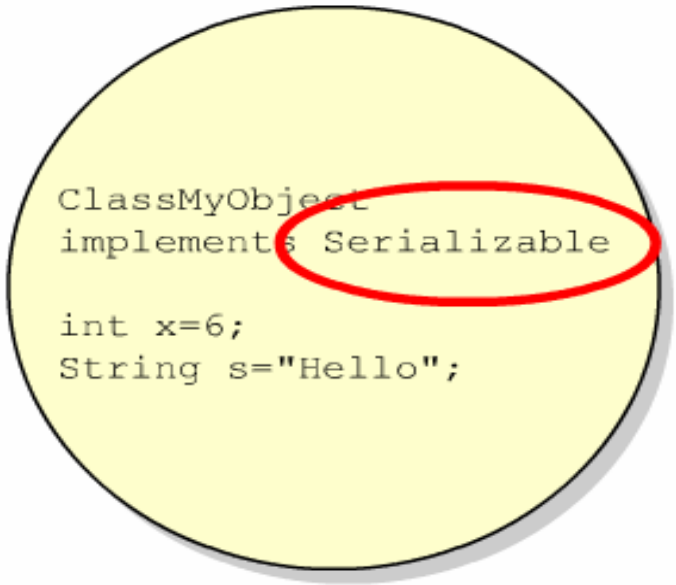
```
import java.io.* ;
class Date implements Serializable {
      int m, d, y ;
      public Date( int m, int d, int y ) {
                  this.m = m ; this.d = d ; this.y = y ;
      }
}
```

Object Serialization 3


```
MyObject:field x, type int, value 6:field s, type String value "Hello"
```

Serialization takes the state (that is the instance variables) of an object and represents them as a sequence of bytes

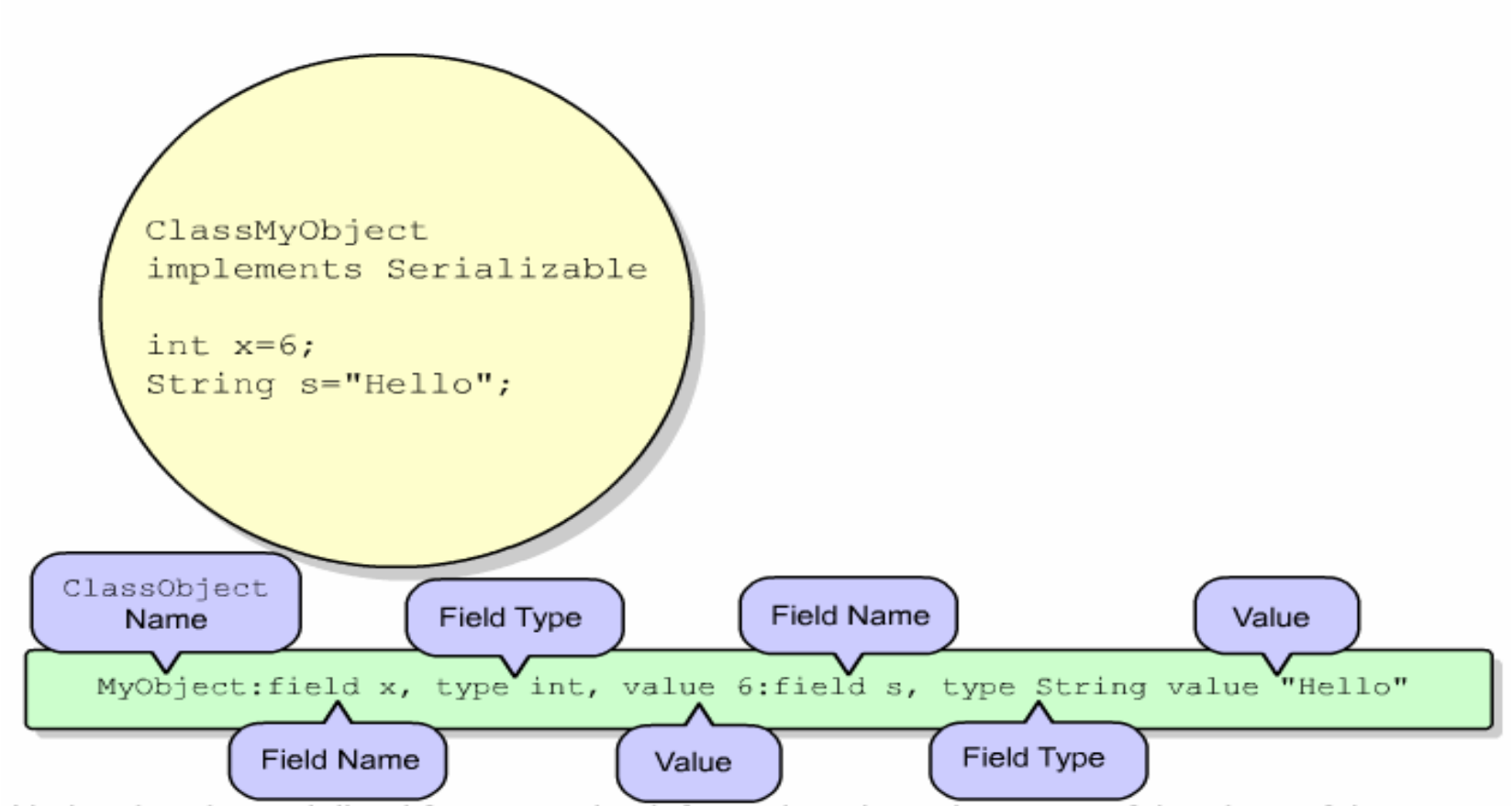
Object Serialization 4




```
MyObject:field x, type int, value 6:field s, type String value "Hello"
```

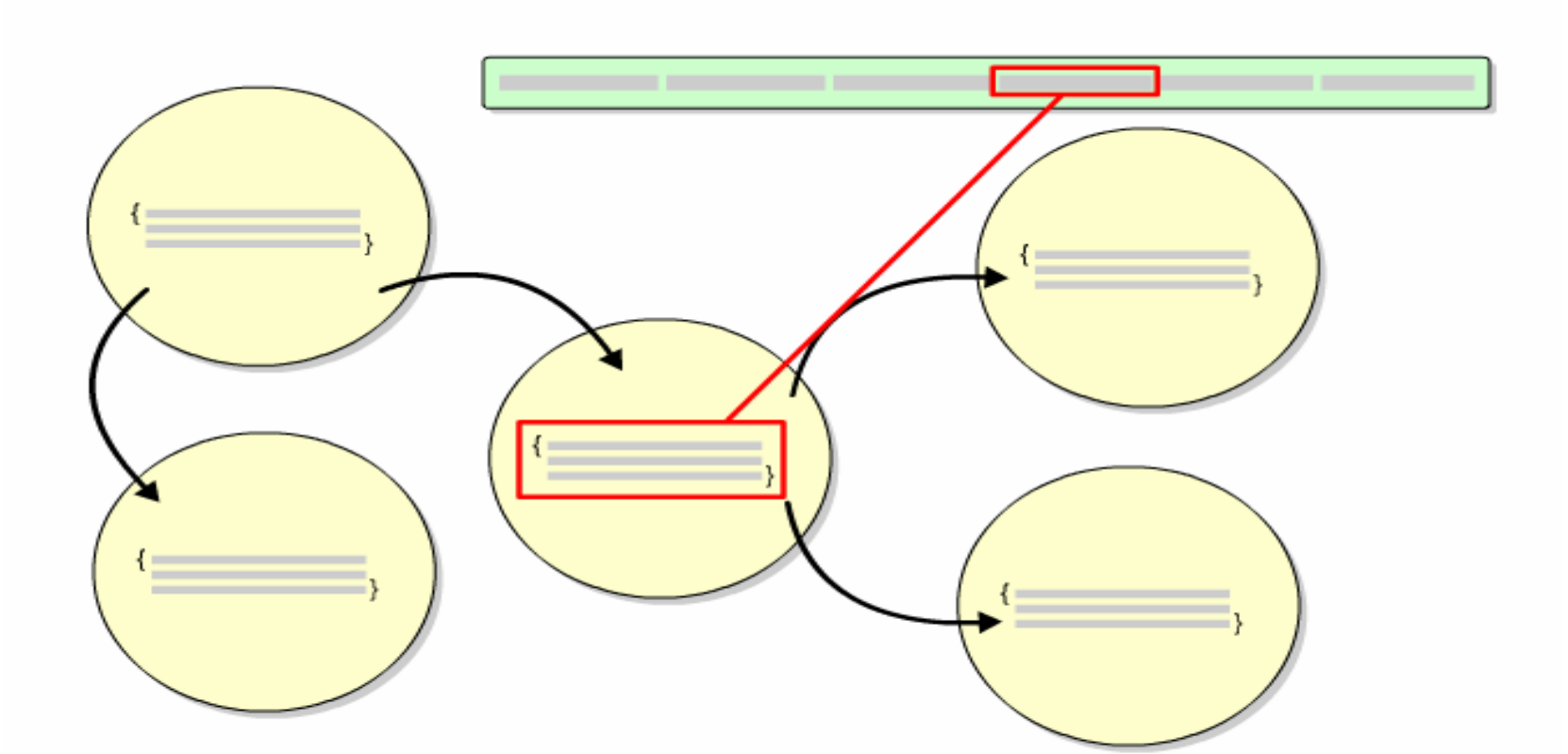
It is important that the class that defines the object declares that the object is serializable. Otherwise, this process does not work.

Object Serialization 5



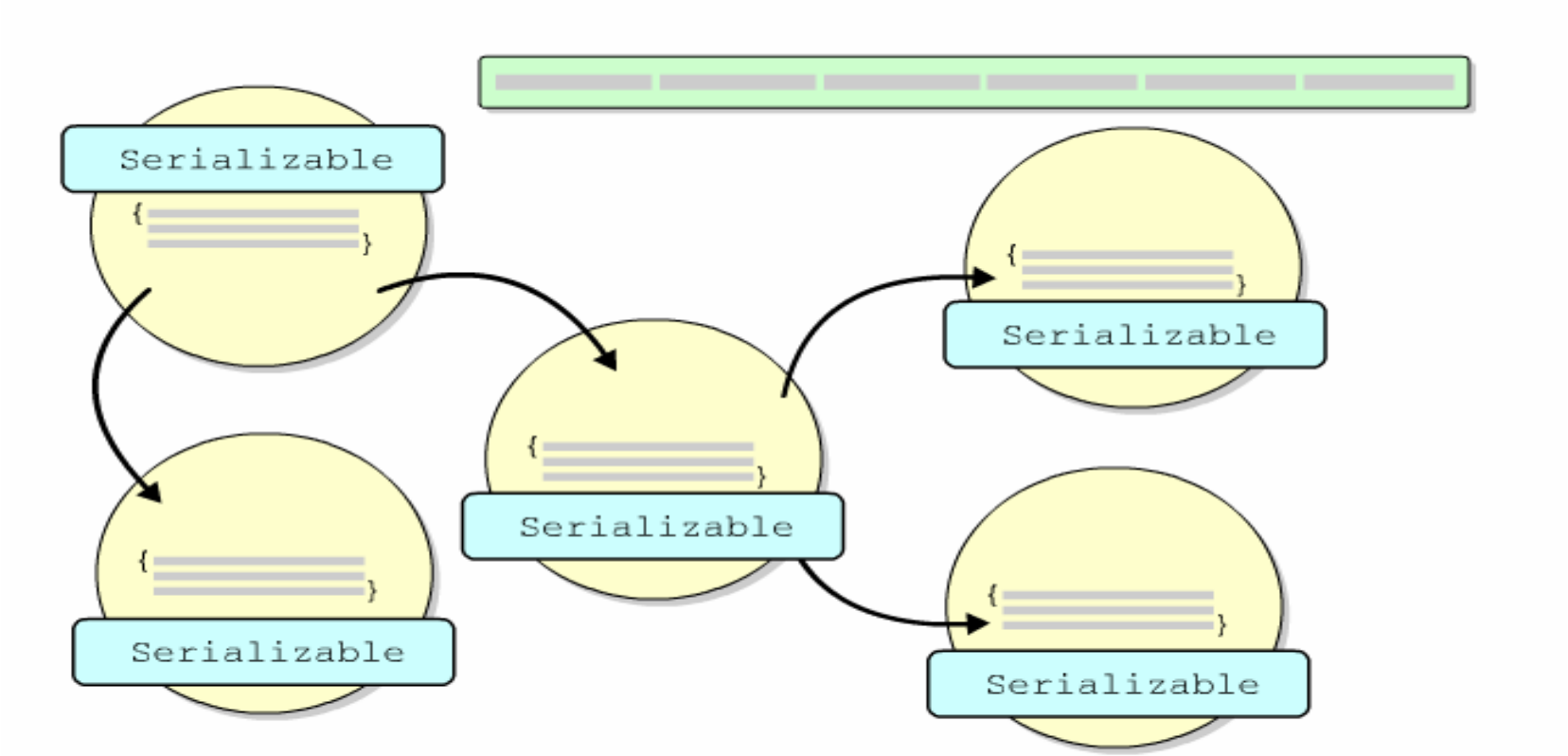
Notice that the serialized form contains information about the name of the class of the object, the names and types of the fields, as well as the values themselves.

Object Serialization 6



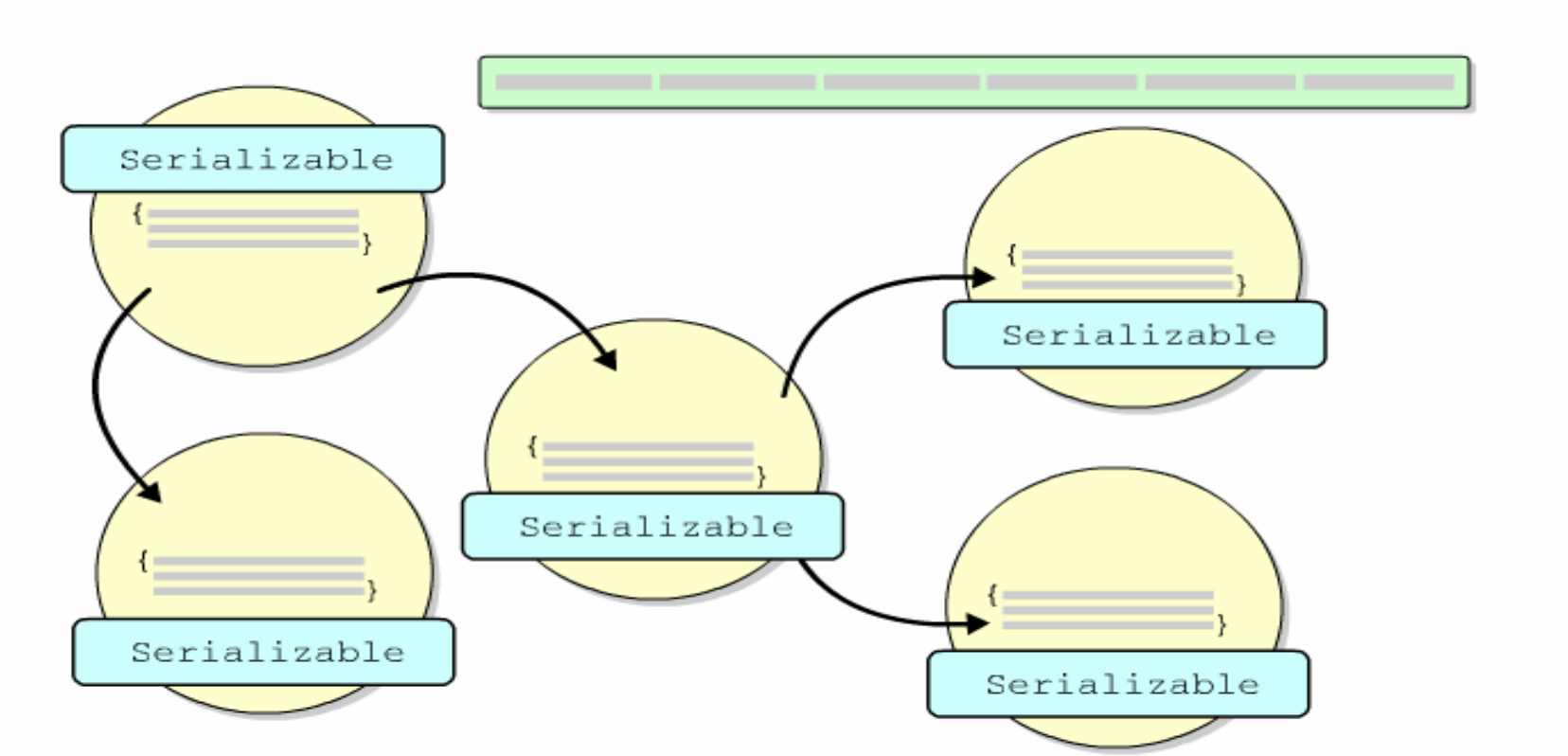
Most objects have references to other objects within them, and sometimes those reference objects also have other references to other objects. All the objects that are reachable through these references form the object graph.

Object Serialization 7



Normally, when an object is serialized, all the objects in the object graph must be serialized. This means that all the objects in the object graph must implement `Serializable` as well. Otherwise, the serialization will fail.

Object Serialization 8



Normally, when an object is serialized, all the objects in the object graph must be serialized. This means that all the objects in the object graph must implement `Serializable` as well. Otherwise, the serialization will fail.

Character Stream Classes

They process 16 bits Unicode characters.

They come in two basic forms:

1) Reader – channel `character` data into the program

   Example:

   ```
   java.io.FileReader
   ```

2) Writer – channel `character` data from the program

   Example:

   ```
   java.io.FileReader
   ```

Character-stream classes end with the suffix Reader and Writer

Character Parent Classes

`Reader` and `Reader` are the abstract parent classes for byte-stream based classes in the `java.io` package.

Usage:

1) `Reader` classes are used to read 16-bit character streams and

2) `Writer` classes are used to write to 16-bit character streams.

Methods for reading and writing to streams:

```
int read()
int read(char[] c)
int read(char[] c, int offset, int length)
int write(int c)
int write(char[] c)
int write(char[] c, int offset, int length)
```

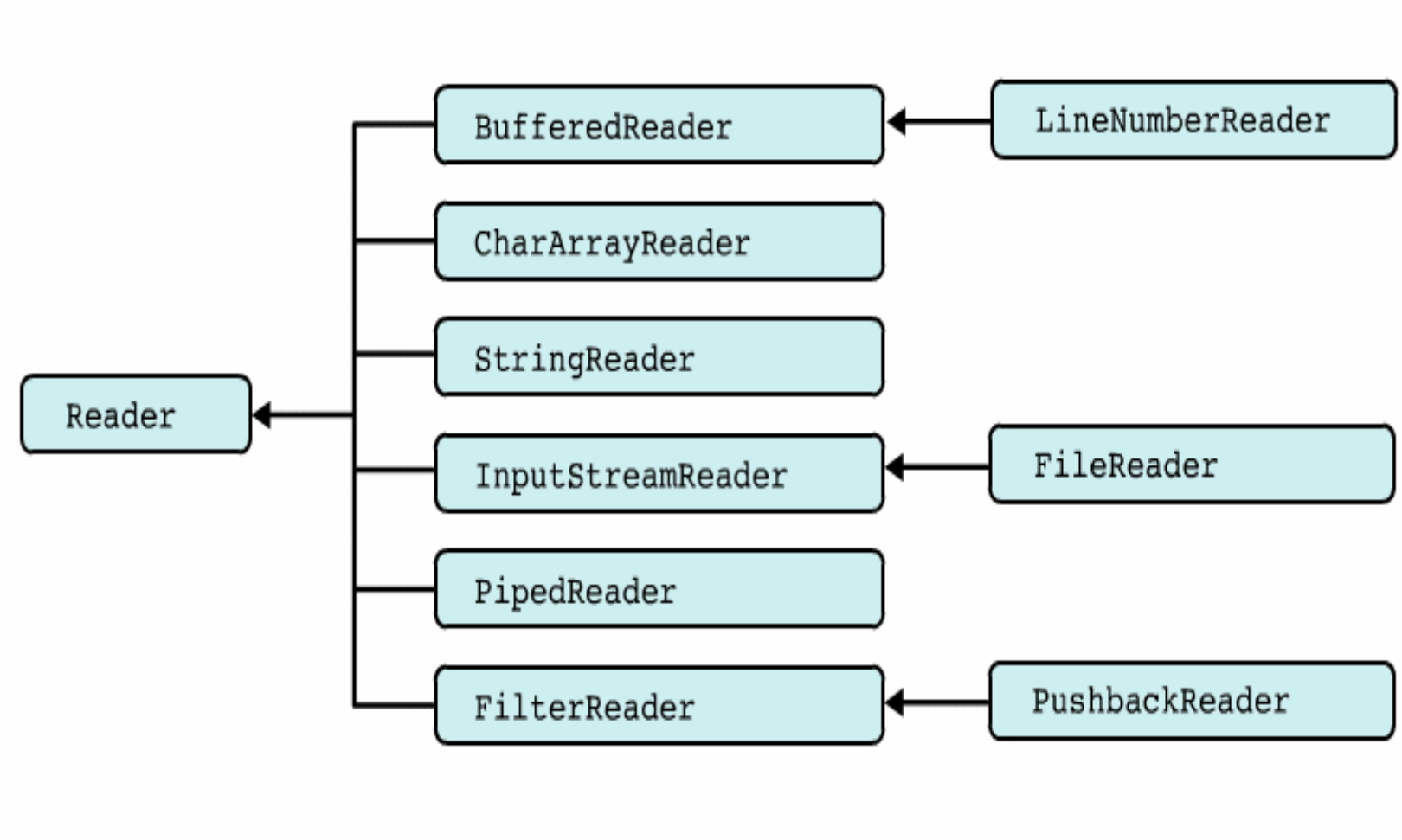
Reader

This abstract class provides the core methods used to read characters from an input node.

The methods are:

```
int read()
int read(char[] c)
int read(char[] c, int offset, int length)
void close()
long skip( long l)
boolean markSupported()
void mark( int i)
void reset()
```

Reader Hierarchy



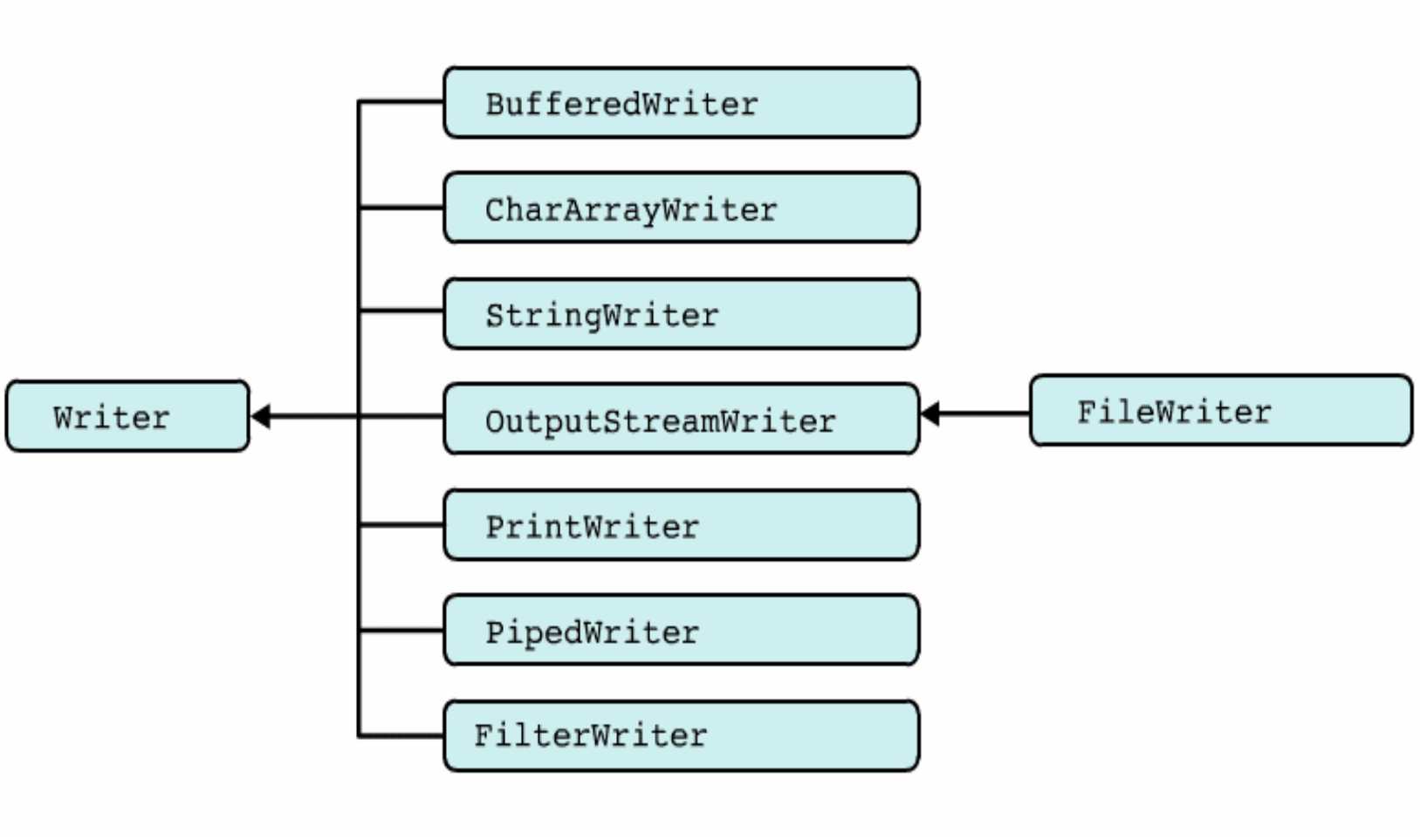
Writer

This abstract class provides the core methods used to write characters to an output node.

The methods are:

```
int write(int b)
int write(char[] c)
int write(char[] c, int offset, int length)
void close()
void flush()
```

Writer Hierarchy



Data Sink Character-Stream

Classes that take input from different types of nodes (file, pipe, char array, etc)

Example:

```
FileReader
PipedReader
```

Classes that send output to different types of node (file, pipe, char array, etc)

Example:

```
FileWriter
PipedWriter
```

Example: Node Character-Stream

1) `FileReader`/`FileWriter`

   Usage: is meant for reading/writing streams of character data to/from files

```
File f = new File("mydata.txt");
FileReader fis = new FileReader(f);
```

2) `CharArrayReader`/`CharArrayWriter`

   Usage: are useful for holding data when the underlying data type is irrelevant to the purpose of the application

```
char[] c
...
CharArrayReader r = new CharArrayReader(c);
```

Bridging Streams 1

To bridge the gap between the `byte` and `character` stream classes, JDK 1.1 and JDK 1.2 provide the `java.io.InputStreamReader` and `java.io.OutputStreamWriter` classes.

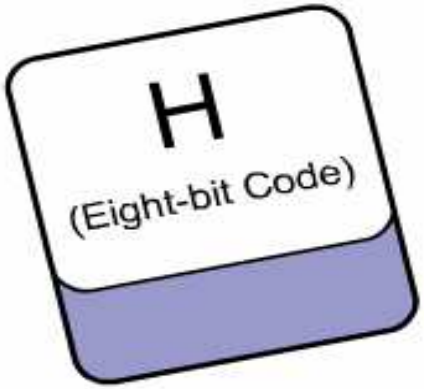
Usage:

The only purpose of these classes is to convert byte data into character-based data according to a specified (or the platform default) encoding.

For example, the static data member "`in`" in the "`System`" class is essentially a handle to the Standard Input (stdin) device. If you want to wrap this inside the `java.io.BufferedReader` class that works with character-streams, you use `InputStreamReader` class as follows:

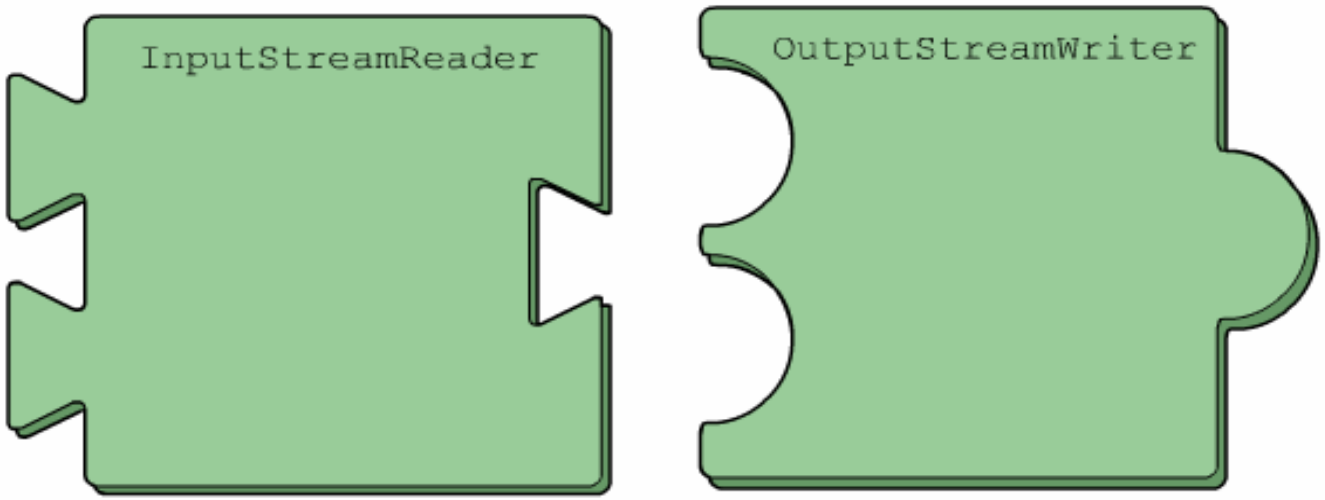
```
BufferedReader in = new BufferedReader(new
                        InputStreamReader(System.in));
```

Bridging Streams 2



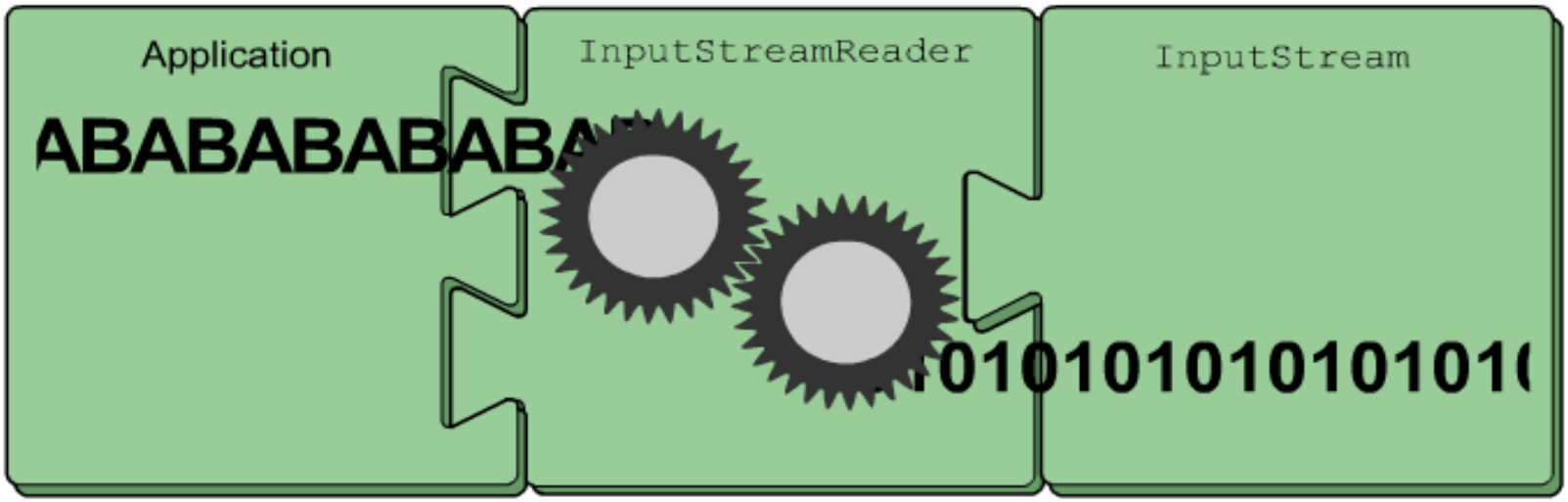
When a key is pressed, most platforms produce an 8bit code to represent a character.
There are many different character sets that a platform may use to encode characters.
Internally, the JVM uses 16bit Unicode characters.

Bridging Streams 3



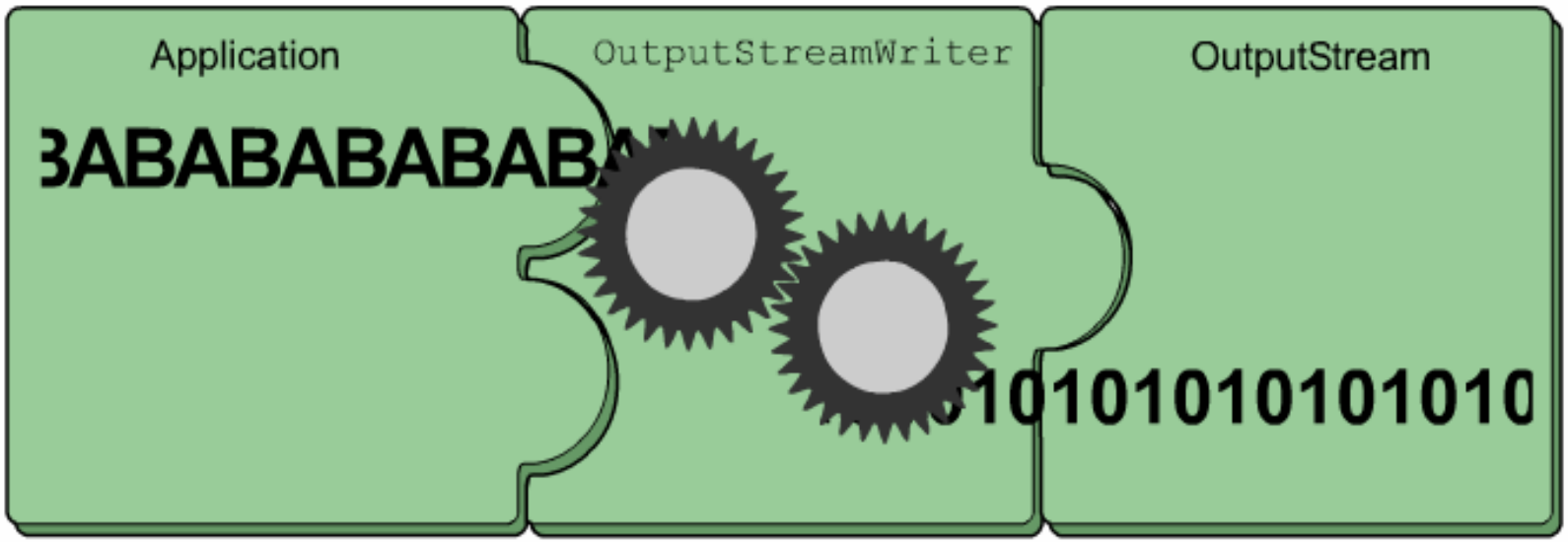
When characters enter or leave the program, they must be converted between these distinct encoding formats. This is the job of the `InputStreamReader` and `OutputStreamWriter` classes.

Bridging Streams 4



This is the `InputStreamReader` object. This class uses a `Reader` type object on the left hand side, but can plug into an `InputStream` type object on the right hand side. This allows it to read data that are encoded in a platorm-specific way from an `InputStream` object and convert those data into 16 bit unicode characters that can be read from the `Reader` interface by the client program.

Bridging Streams 5



When an application wants to write data out to the local platform, the `OutputStreamWriter` performs the inverse conversion, taking Unicode characters and mapping them to bytes in the local encoding format.

Summary: Filter Streams

Type	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Converting between bytes and character	InputStreamReader OuptutStreamWriter	
Object serialization		ObjectInputStream ObjectOutputStream
Data conversion		DataInputStream DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

Reading Text from Standard Input

```
try {
        BufferedReader in = new BufferedReader(new
                            InputStreamReader(System.in));
        String str = "";
        while (str != null) {
            System.out.print("> prompt ");
            str = in.readLine();
            process(str);
        }
    } catch (IOException e) {
    }
```

Reading Text from a File

```
try {
        BufferedReader in = new BufferedReader(new
                            FileReader("infilename"));
        String str;
        while ((str = in.readLine()) != null) {
            process(str);
        }
        in.close();
    } catch (IOException e) {
    }
```

Writing to a File

```
try {
        BufferedWriter out = new BufferedWriter(new
                             FileWriter("outfilename"));
        out.write("aString");
        out.close();
    } catch (IOException e) {

    }
```

Appending to a File

```
try {
        BufferedWriter out = new BufferedWriter(new
                             FileWriter("filename", true));
        out.write("aString");
        out.close();
    } catch (IOException e) {
    }
```

Serializing an Object

```
Object object =  new javax.swing.JButton("push");
try {
    // Serialize to a file
    ObjectOutputStream out = new

  ObjectOutputStream(new
                FileOutputStream("filename.ser"));
    out.writeObject(object);
    out.close();
    // Serialize to a byte array
    ByteArrayOutputStream bos = new

  ByteArrayOutputStream() ;
    out = new ObjectOutputStream(bos) ;
    out.writeObject(object);
    out.close();
    //  Get the bytes of the serialized object
     byte[] buf = bos.toByteArray();
   } catch (IOException e) {
```

Deserializing an Object

```
try {
      // Deserialize from a file
      File file = new File("filename.ser");
      ObjectInputStream in = new
                             ObjectInputStream(new
                             FileInputStream(file));
      // Deserialize the object
      javax.swing.JButton button =
              (javax.swing.JButton) in.readObject();
      in.close();

    } catch (ClassNotFoundException e) {

    } catch (IOException e) {

    }
```

Lab Work: Reading and Writing

Based on the code snippets, write a program that

1) reads text from standard input
2) copies the content of one file and writes to another file
3) appends "`This is emacao training`" to the end a text file
4) a program that joins series of files together

Stream Chaining

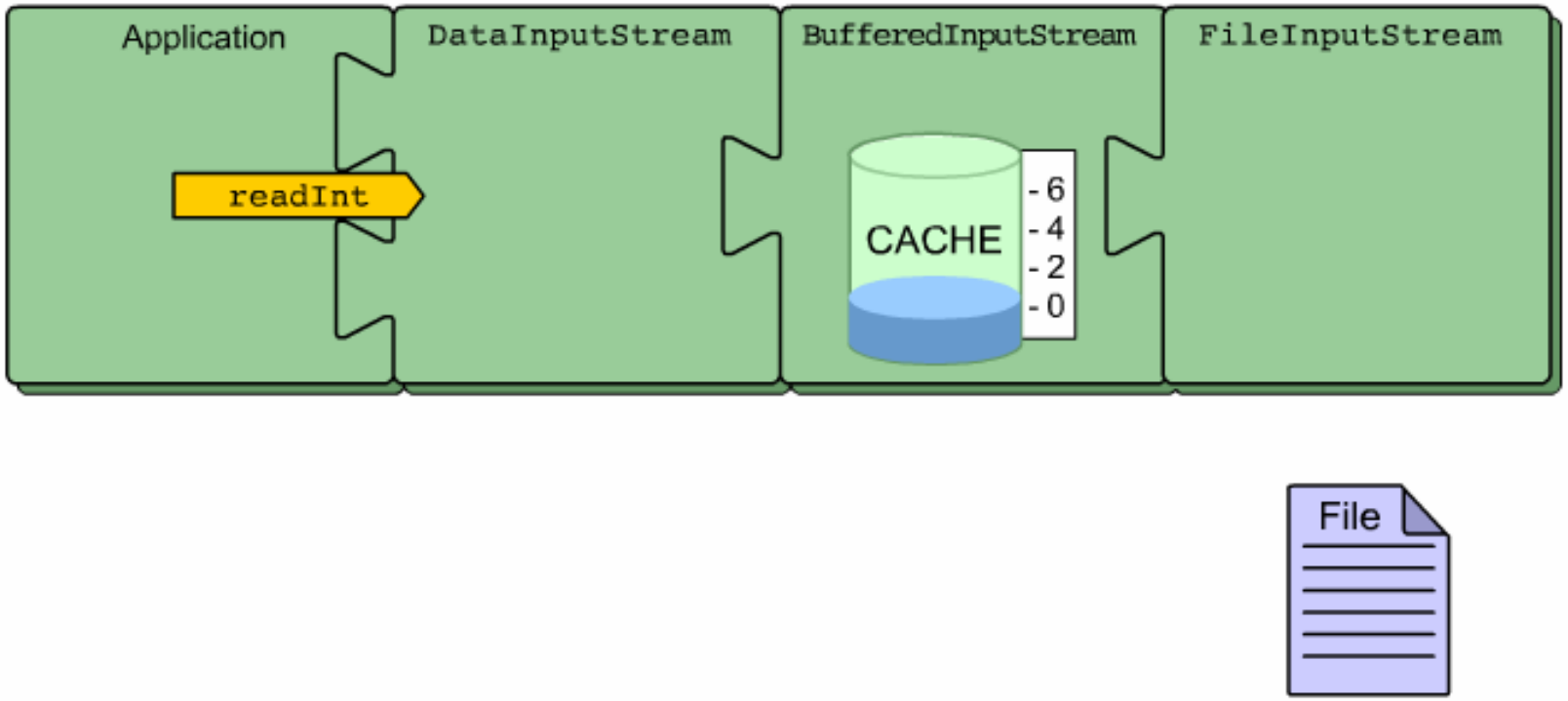
Stream chaining is a way of connecting several stream classes together to get the data in the form required.

Each class performs a specific task on the data and forwards it to the next class in the chain.

The output produced by one component becomes the input to the next component in the chain.

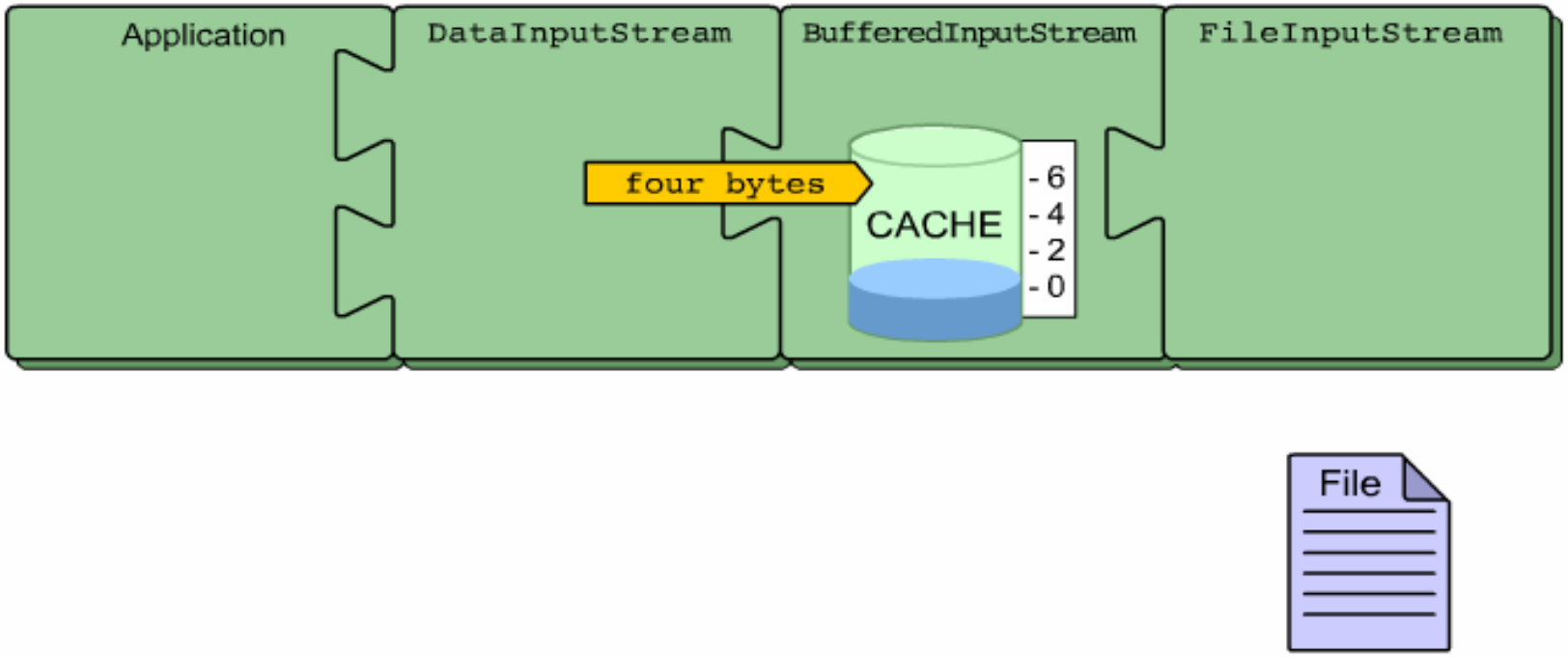
Consider this.

Example: Stream Chaining 1



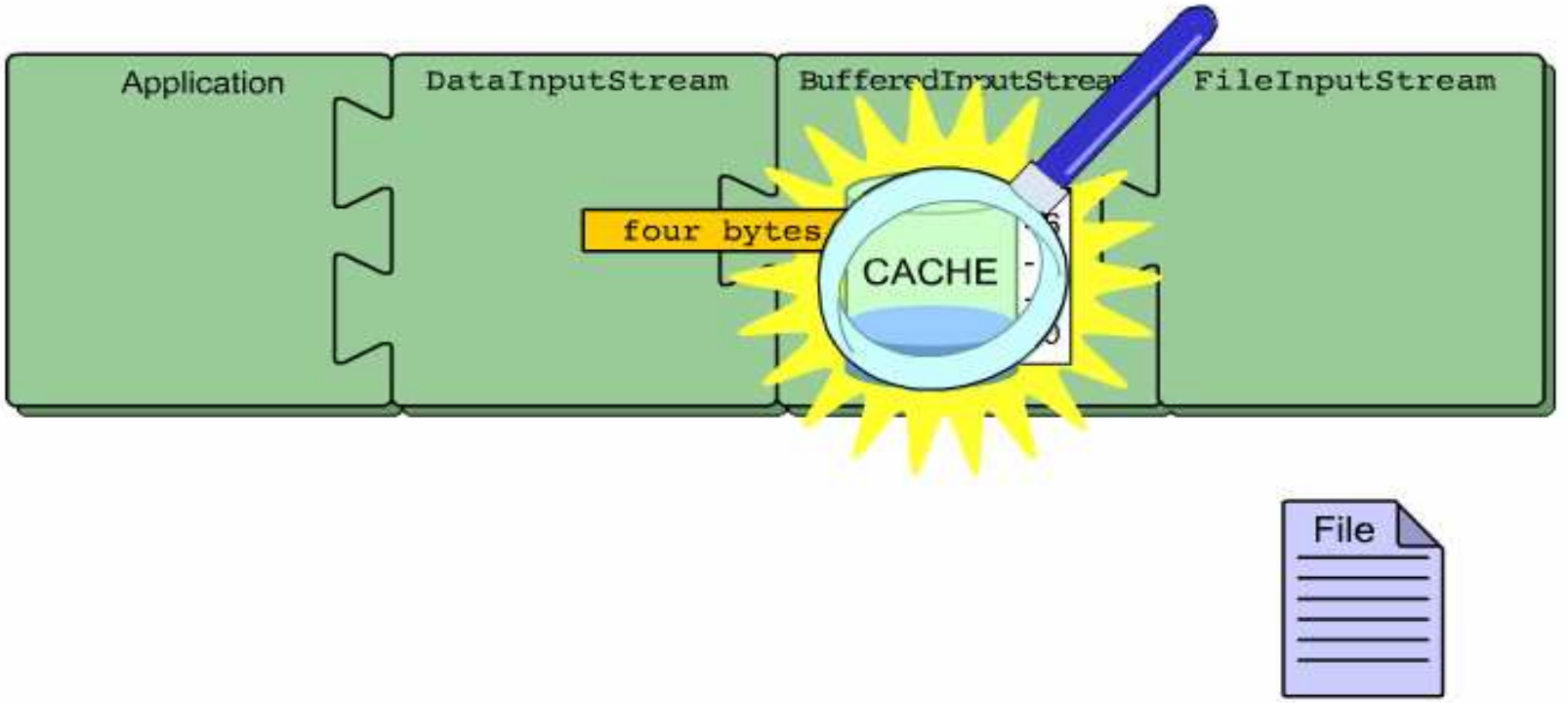
When an application invokes a method like `readInt` on a `DataInputStream` object, what happens beneath the surface?

Example: Stream Chaining 2



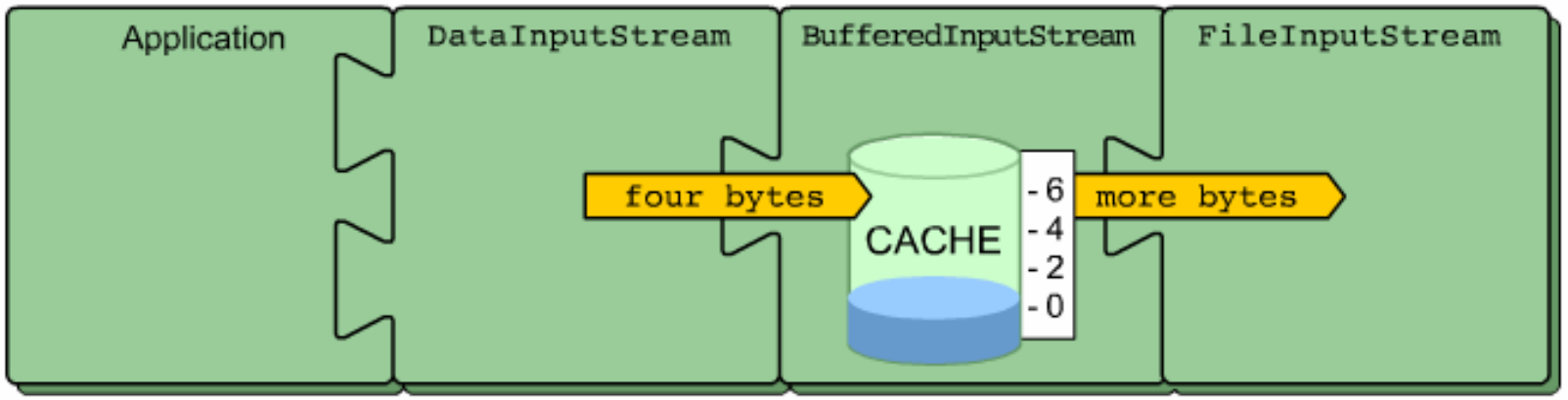
The `DataInputStream` object requests four bytes (the representation for an `int` type) from the `BufferedInputStream` object (a reference to which is contained within the `DataInputStream` object).

Example: Stream Chaining 3



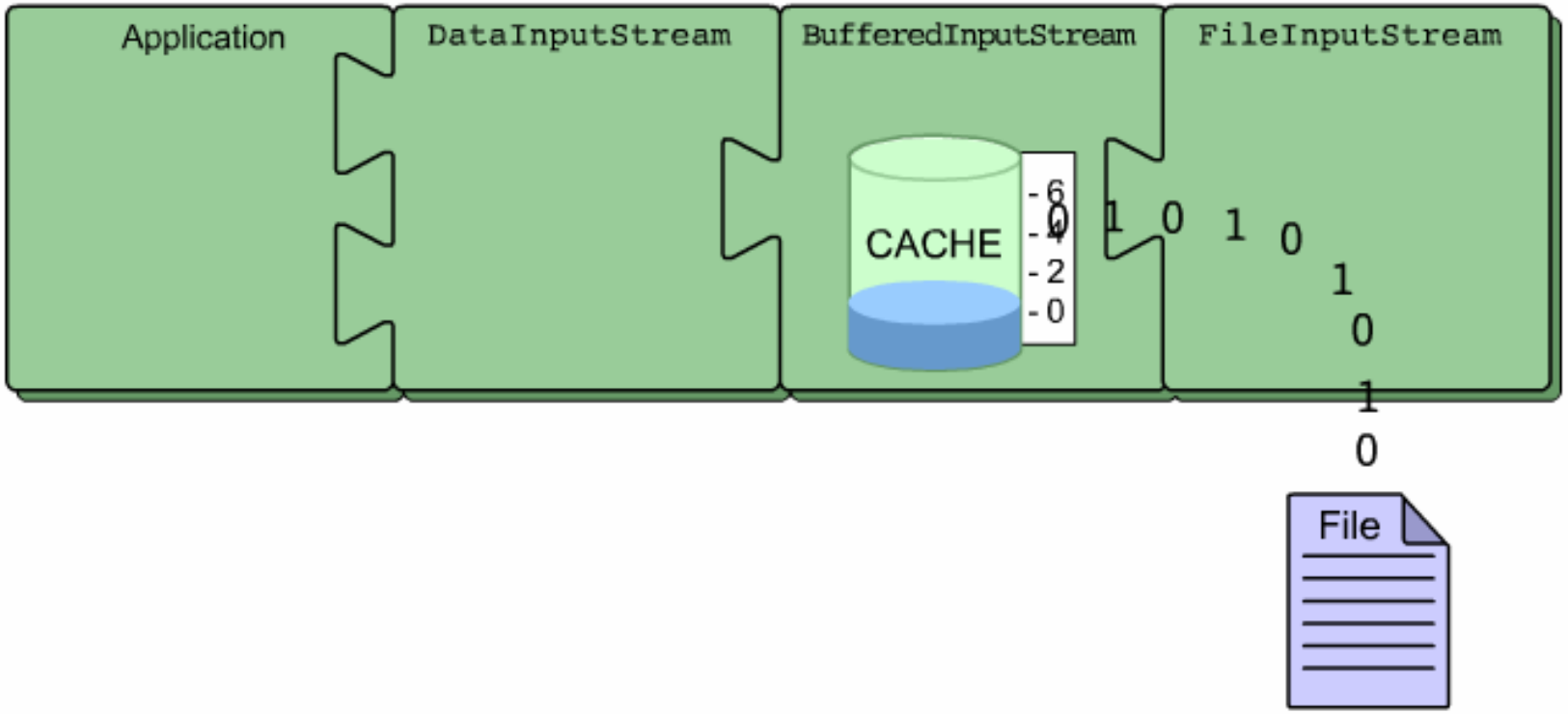
The `BufferedInputStream` object inspects its internal cache. If the cache does not contain four bytes of leftover data from a previous read, the `BufferedInputStream` object will request additional bytes from the `FileInputStream` object (a reference to which is contained within the `BufferedInputStream` object).

Example: Stream Chaining 4



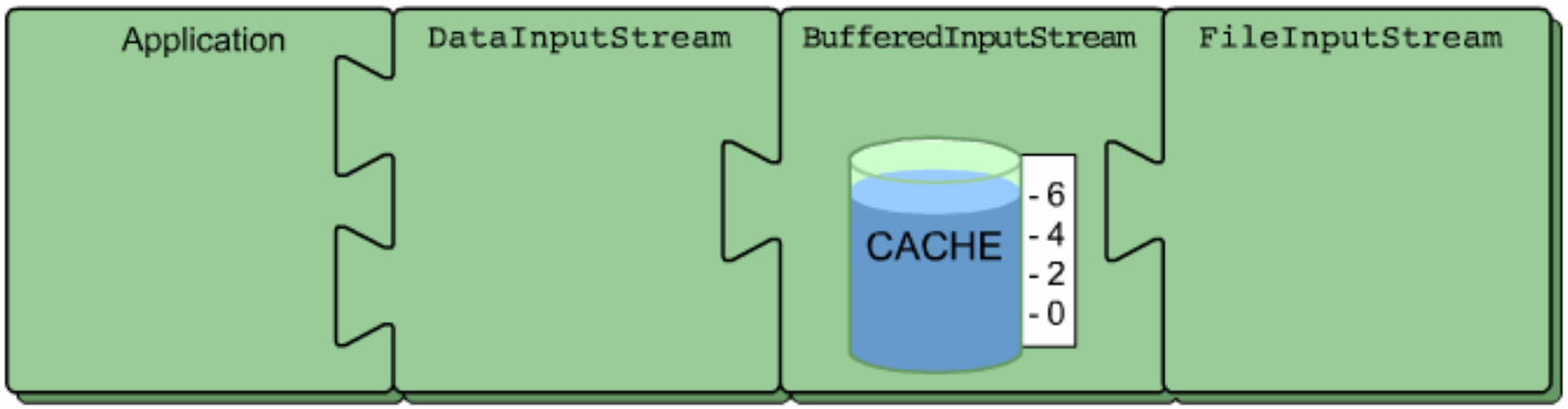
The `BufferedInputStream` object inspects its internal cache. If the cache does not contain four bytes of leftover data from a previous read, the `BufferedInputStream` object will request additional bytes from the `FileInputStream` object (a reference to which is contained within the `BufferedInputStream` object).

Example: Stream Chaining 5



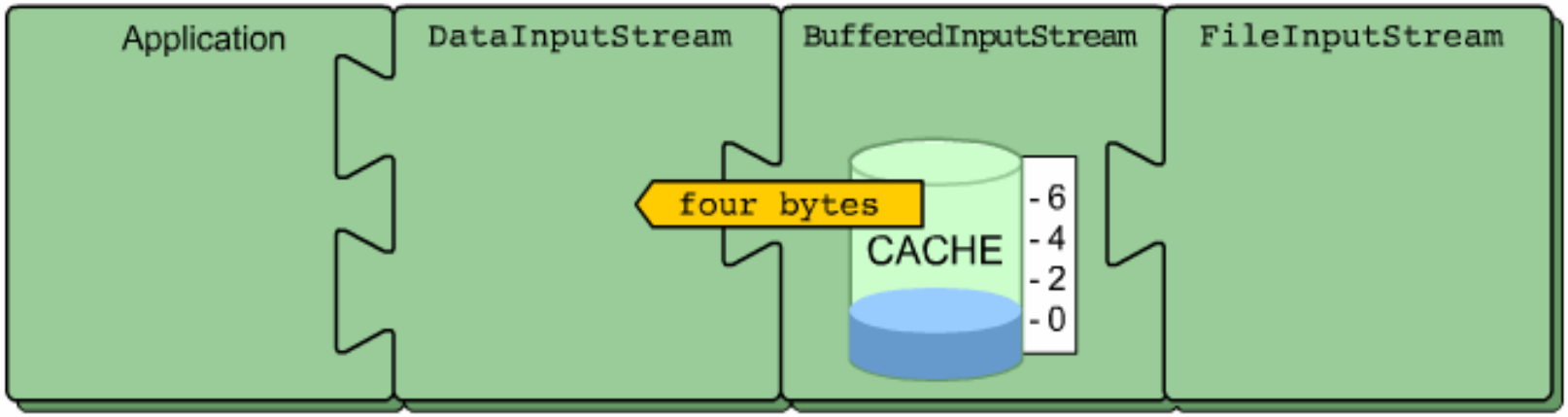
The `FileInputStream` object reads the requested numbers of bytes from the file and returns them to the `BufferedInputStream` object, which then caches the data in its internal storage buffer.

Example: Stream Chaining 6



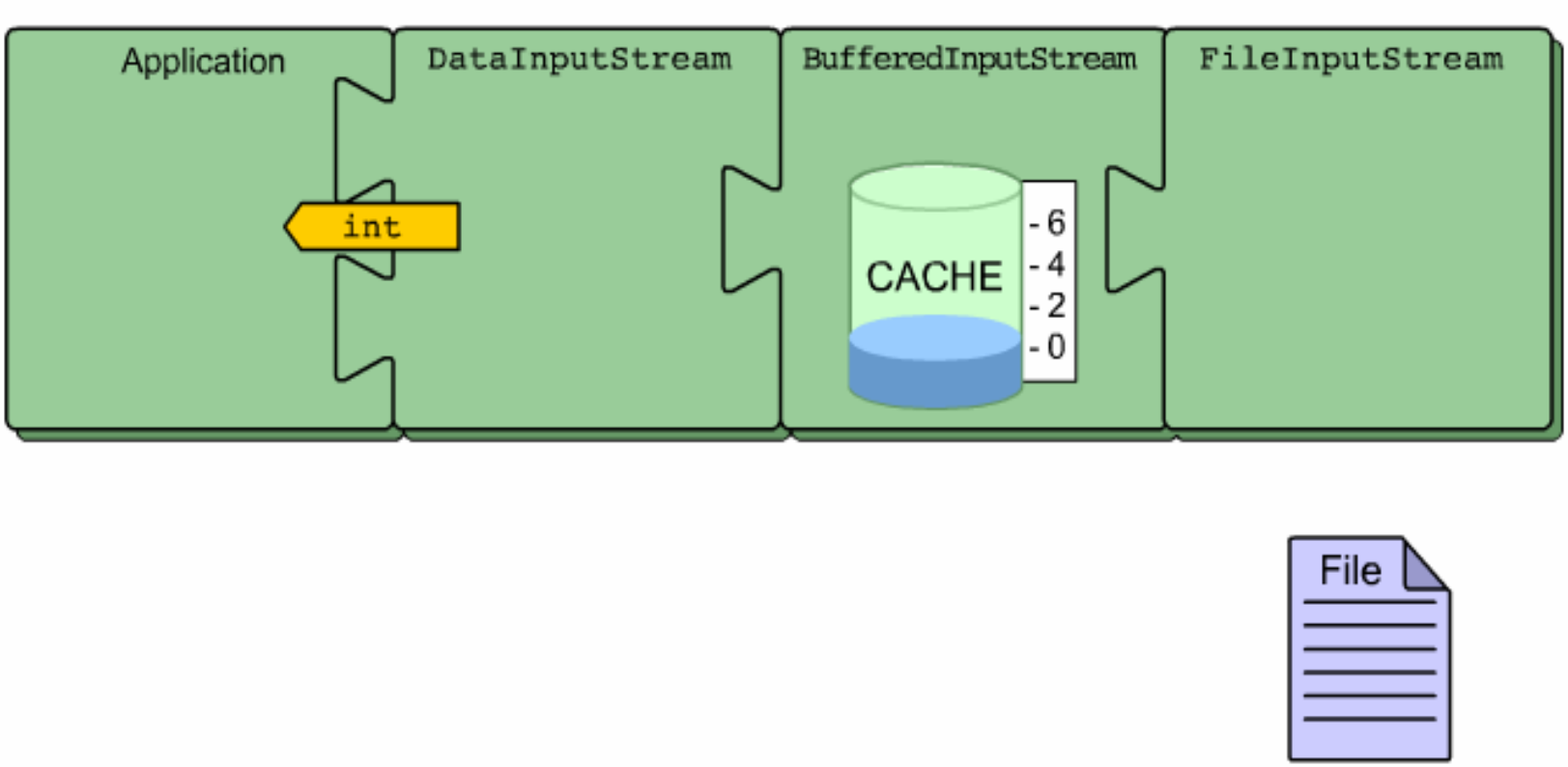
The `FileInputStream` object reads the requested numbers of bytes from the file and returns them to the `BufferedInputStream` object, which then caches the data in its internal storage buffer.

Example: Stream Chaining 7



The `BufferedInputStream` object extracts the four bytes requested by the `DataInputStream` object from the cache and returns them to the calling method.

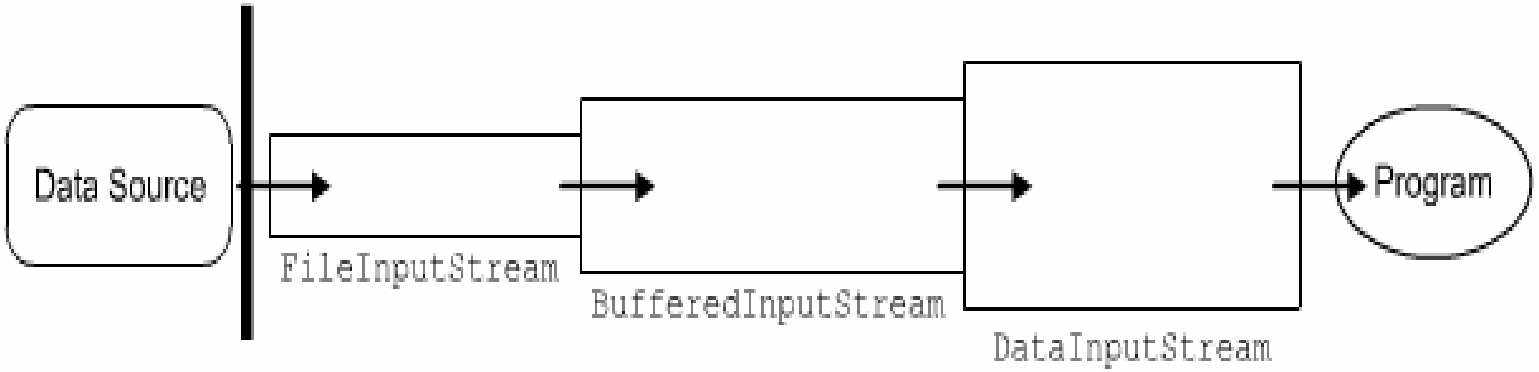
Example: Stream Chaining 8



The `DataInputStream` object returns the four bytes to the application in the form of an `int` type.

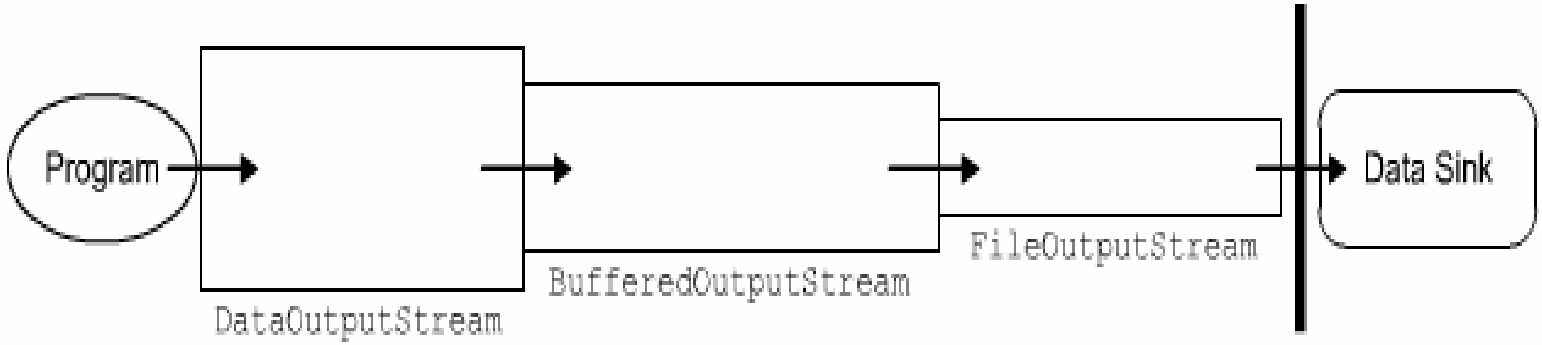
IntputStream Chain

```
FileInputStream theFile = new FileInputStream( "input.dat" ) ;
BufferedInputStream theBuffer = new BufferedInputStream ( theFile ) ;
DataInputStream theData = new DataInputStream( theBuffer ) ;
```



OutputStream Chain

```
FileOutputStream theFile = new FileOutputStream( "output.dat" ) ;
BufferedOutputStream theBuffer = new BufferedOutputStream ( theFile ) ;
DataOutputStream theData = new DataOutputStream( theBuffer ) ;
```



File Operations

There are three non stream classes in `java.io` package.

1) `File` Class - represents a file on the local system

2) `FilenameFilter` class - is an interface used to filter a list of filenames

Each will be considered in details.

File Class

Represents a file on the local filesystem.

Usage:

1) to identify a file
2) obtain information about the file
3) and even change information about the file

Constructors:

1) `File(File parent, String child)`
2) `File(String pathname)`
3) `File(String parent, String child)`
4) `File(URI uri)`

Example: File 1

```
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File(“filename here");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "writeable" : "not  writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
```

Example: File 2

```
    p("is " + (f1.isDirectory()?"" :"not a directory"));
    p(f1.isFile() ?"normal file":" a named pipe");
    p(f1.isAbsolute() ? "absolute" : "not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

Directories

A directory is a `File` that contains a list of other files and directories.

When you create a `File` object and it is a directory, the `isDirectory()` method will return true.

In this case you can call `list()` on that object to extract the list of other files and directories inside.

The form of `list()` is

```
String[] list();
```

The list of file is returned in an array of `String` objects.

Example: Directories 1

```
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
```

Example: Directories 2

```
        }
      } else {
        System.out.print(dirname + " is not a”);

        System.out.println(“directory");
      }
    }
}
```

Creating Directories

Another two useful `File` utility methods are:

1) `mkdir()` – creates a directory , returning `true` on success and `false` on failure.

   Failure indicates that the path specified in the `File` object already exists, or that the directory cannot be created because the entire path does not exist yet.

2) `mkdirs()` – to create directories for which no path exists, it creates both a directory and all the parents of the directory

FilenameFilter

To limit the number of files returned by the `list()` method to include only those files that match a certain type of filename pattern, or filter, use the second form of `list()`.

The second form of `list()` is:

```
String[] list(FilenameFilter ffobj);
```

`FilenameFilter` defines only a single method, `accept()`, which is called once for each file in a list.

The `accept()` method returns `true` for files in the directory specified by directory that should be included in the list and returns `false` if otherwise.

Example: FilenameFilter 1

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

Example: FilenameFilter 2

Here is a modified version of directory listing program. Now it display only classes that use `.html` extension.

```
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

Stream Benefits

The Streaming interface to I/O in Java provides:

1) A clean abstraction for complex and often cumbersome task.
2) The composition of filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.
3) Java programs written to adhere to the abstract, high level-level `InputStream`,`OutputStream`, `Reader` and `Writer` classes will function properly in the future even when new and improved concrete stream classes are invented.
4) Serialization of object is expected to play an increasingly important role in Java Programming in the future.

Lab Work: Input and output

1) Write a program that reads the content of "`C:\Documents and Settings\All Users`" on your local system.
2) Write a program that counts the total number of directories and files you have in the path.
3) Archive at least one of the subdirectories of `All Users` folder and save it in zip format in your folder on the network.
4) Study and use `java.util.zip` package by referring to the API documentation for the appropriate classes to use in this exercise.

Project Exercise 2

1) Implement the client component of your software architecture which satisfies your use case model. Provide a web interface for users and desktop interface for back office processing. Implementation should be carried out using Java swing library.
2) Check for the consistency between your implementation and your design class diagrams (in your design model). For instance, are all your design classes implemented?
3) Check for the consistency between the dynamic aspect of your architecture (instance level collaboration diagrams) and your implementation
4) Update your implementation model indicating the implementing artifacts for your client component.

Note: Provide appropriate version control for all artifacts (models and codes)

Networking

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Outline: Networking

Presents the network programming in Java language.

Main points:

1) Review the basic network concepts and Java Implementation.
2) Discuss the usage of java.net package.
3) Introduce the Secure Socket.
4) Introduce the New I/O API.
5) Introduce the Java implementation for UDP protocol.

Overview

1) introduction
2) basic network concepts and Java Implementation
2) Implementation
3) JMS programming model and implementation
4) advance configuration
5) summary

Why Java

Why use Java for Networking?

a) Java was the first programming language designed from the ground up with networking in mind.

b) Java provides easy solutions to two crucial problem for Internet networking — platform independence and security.

c) It is far easier to write network programs in Java than in almost any other language.

- In the fully functional applications, very little code is devoted to networking.

Network programs with Java 1

Examples that a Network Program can do:

a) Server-Client

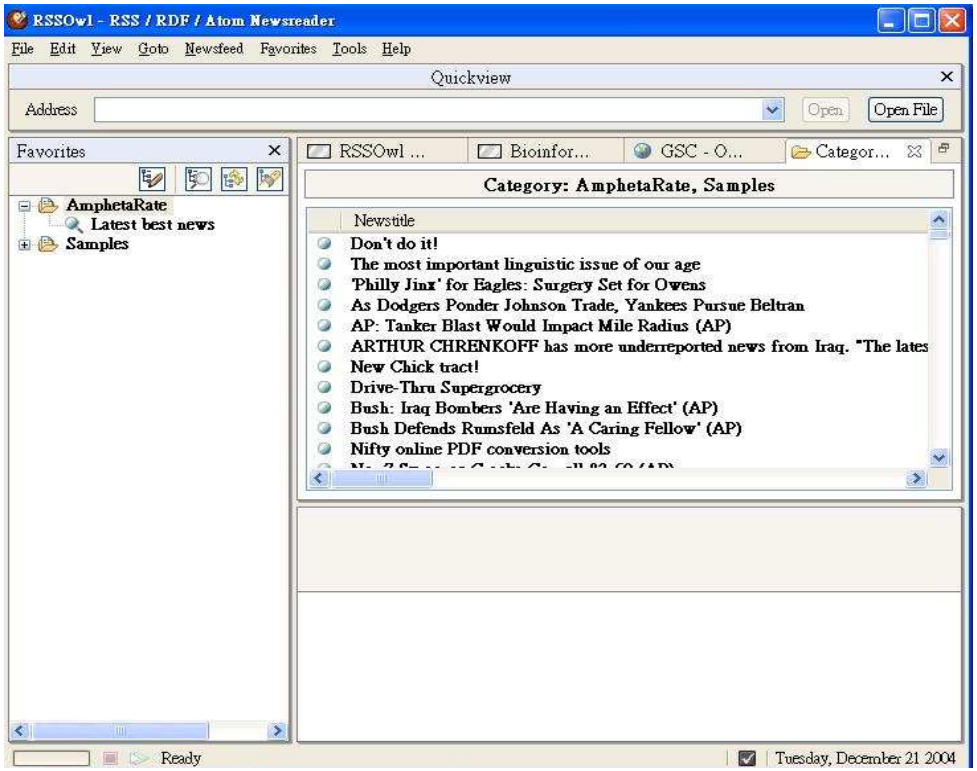
Examples: RssOwl (http://rssowl.sourceforge.net/)

b) Peer-to-Peer

Examples: LimeWire (http://limewire.org/)

Azureus (http://azureus.sourceforge.net/)

Network programs with Java 2



http://rssowl.sourceforge.net/

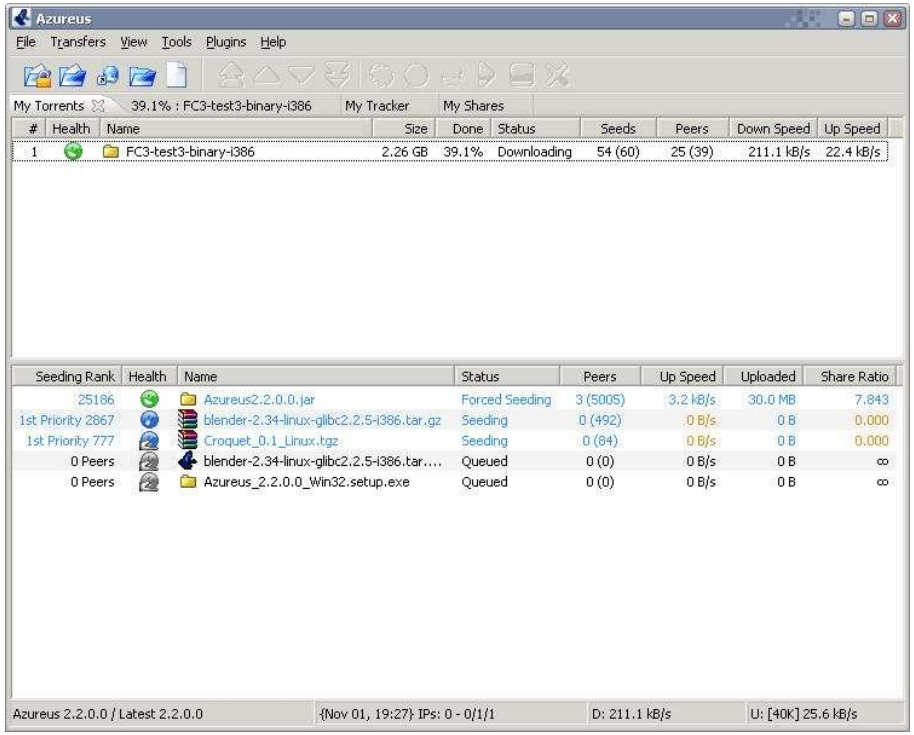
- RssOwl combines news from different sources and allows the user to browse using a modern graphical user interface.
- Unlike a web browser, this program can continuously update the data in real time.

Network programs with Java 3

http://www.limewire.org/

- LimeWire enables the clients to query each other and transfer files among themselves.
- LimeWire is an open source pure Java application that uses a Swing GUI and standard Java networking classes.

Network programs with Java 4



http://azureus.sourceforge.net

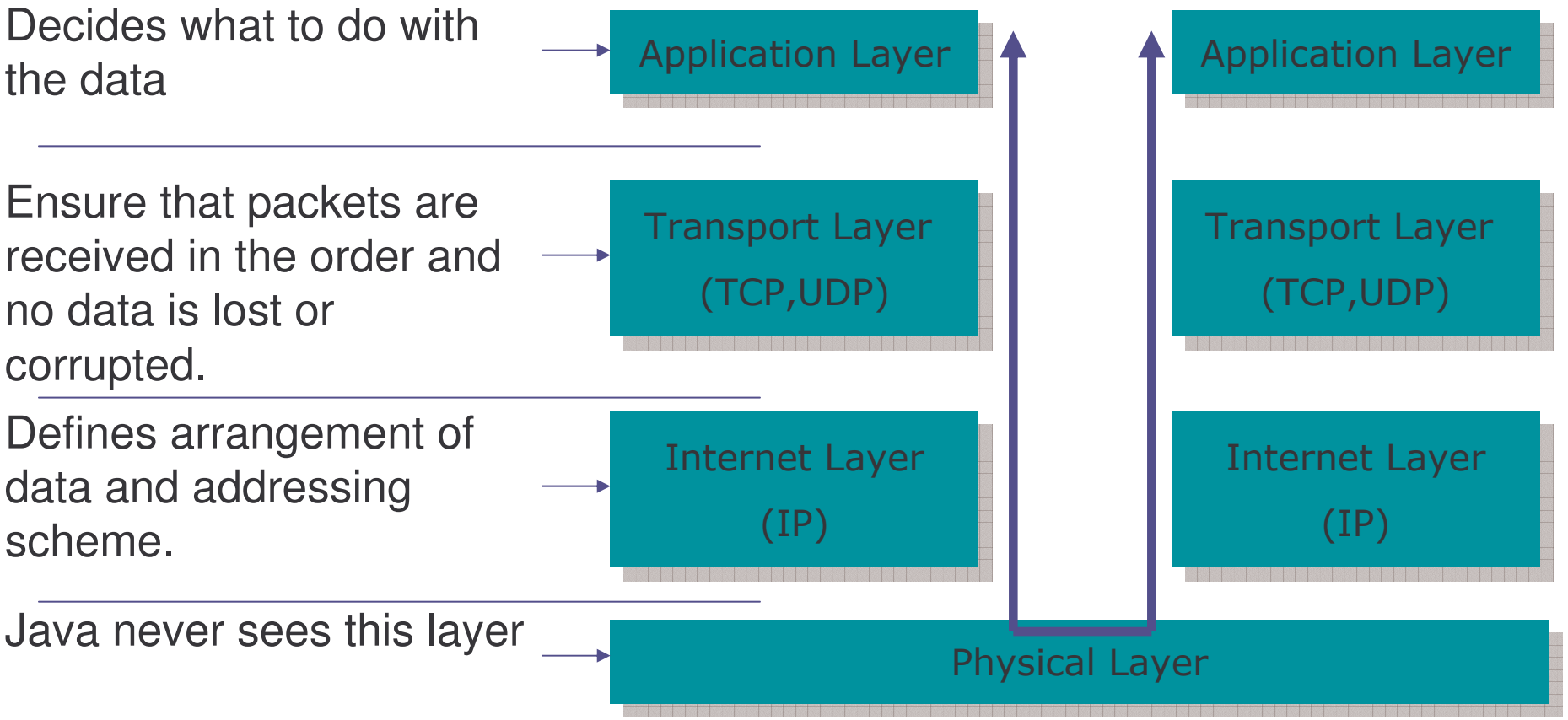
- Azureus is one of the BitTorrent clients written in pure Java.
- BitTorrent is designed to serve files that can be referenced from known keys
- Downloaders can sharing a file while they're still downloading it.

Concepts for network program

Important concepts needed for writing network program in Java

1) Communication protocols: TCP and UDP
2) Ports and Internet Addresses
3) Sockets
4) Uniform Resource Locator (URL)
5) Uniform Resource Identifier (URI)
6) Streams and Threads (Covered in previous sessions)
7) Classes in java.net and java.io packages

Network Layers



TCP and UDP

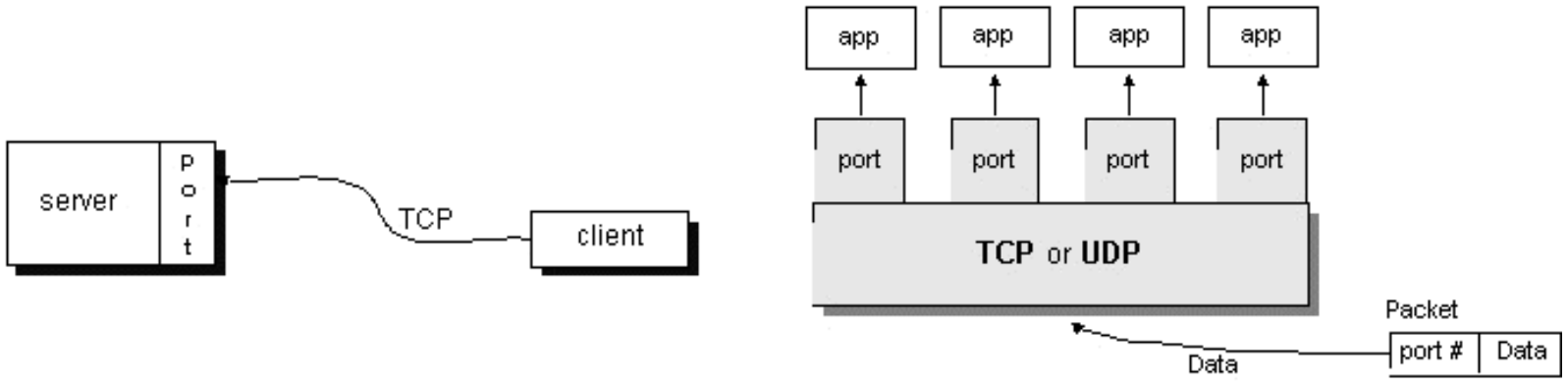
Java only supports TCP (Transmission Control Protocol), UDP (User Datagram Protocol) and application layer protocols built on top of these.

Characteristics of TCP and UDP :

TCP	UDP
• Provides the ability to acknowledge receipt of IP packets and request retransmission of lost or corrupted packets. • Allows the received packets to be put back together in the order they were sent. • Requires a lot of overhead. • Supported classes in java.net : URL, URLConnection, Socket, and ServerSocket	• Is an unreliable protocol that does not guarantee that packets will arrive at their destination. • Allow the receiver to detect corrupted packets but does not guarantee that packets are delivered in the correct order. • Requires less overhead and faster. • Supported classes in java.net : DatagramPacket, DatagramSocket, and MulticastSocket

Ports

Each port from the server can be treated by the clients as a separate machine offering different services.



Port numbers are represented by 16-bit numbers. (0 to 65,535)

The port numbers ranging from 0 - 1023 reserved for use by well-known services such as HTTP and FTP and other system services.

Sockets

You can reach required service via its network and port IDs. what then?

a) If you are a client
   - you need an API that will allow you to send messages to that service and read replies from it

b) If you are a server
   - you need to be able to create a port and listen at it.
   - you need to be able to read the message comes in and reply to it.

The **Socket** and **ServerSocket** are the Java client and server classes to do this.

Example : Sending Email 1

E-mail is sent by socket communication with port 25 on a computer system.

open a socket connected to port 25 on some system, and speak “mail protocol” to the daemon at the other end.

Example : Sending Email 2

```
import java.io.*;
import java.net.*;
public class SendEmail {
  public static void main(String args[]) throws
   IOException {
      Socket sock;
      DataInputStream dis;
      BufferedReader br;
      PrintStream ps;
      System.out.println(">>> Connect
                  mailhost.iist.unu.edu");
      sock = new Socket("mailhost.iist.unu.edu", 25);
      dis = new DataInputStream(sock.getInputStream());
```

Example : Sending Email 3

```
 br = new BufferedReader (new
    InputStreamReader(dis));
 ps = new PrintStream( sock.getOutputStream());
System.out.println( br.readLine() );
System.out.println(">>> Hello UNU/IISTT");
ps.println("Hello UNU/IIST");
System.out.println( br.readLine() );
System.out.println(">>> Mail From:
 oluotes@yahoo.com");
ps.println("MAIL FROM:milton_hm@hotmail.com");
System.out.println( br.readLine() );
String Addressee= "milton@iist.unu.edu";
```

Example : Sending Email 4

```
System.out.println(">>> Rcpt to: " + Addressee);
ps.println("RCPT TO: " + Addressee );
System.out.println( br.readLine() );
System.out.println(">>> Send \"data\"");
ps.println("DATA");
System.out.println( br.readLine() );
System.out.println(">>>>>>>>>>");
System.out.println(">>> This is the message\n that
 Java sent");
System.out.println(">>> We are testing Socket
 Programming");
System.out.println(">>> in eMacao Training
 program.");
System.out.println(">>>>>>>>>>");
```

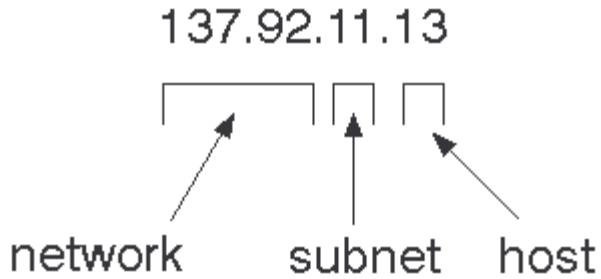
Example : Sending Email 5

```
    ps.println("This is the message\n that Java
    sent");
     ps.println("We are testing Socket Programming");
     ps.println("in eMacao Training program.");
     System.out.println(">>> .");
     ps.println(".");
     System.out.println( br.readLine() );
     System.out.println(">>> QUIT");
     ps.println("QUIT");
     System.out.println( br.readLine() );
     ps.flush();
     sock.close();
   }
}
```

Internet Addressing

Internet address (IP address) is a unique number for identifying a device connected to the Internet.

The current standard is IPv4 which are four bytes long.



The hostname and IP address is, in Java, represented by java.net.InetAddress.

InetAddress is used by many other networking classes, including Socket, ServerSocket, URL, DatagramSocket, DatagramPacket, and more.

Example

The following program will print out the IP address of the www.iist.unu.edu

```
import java.net.*;
public class IISTByName {
  public static void main (String[] args) {
   try { InetAddress address =
     InetAddress.getByName   ("www.iist.unu.edu") ;
     System.out.println(address);
   } catch (UnknownHostException ex) {
       System.out.println("Could not find
         www.iist.unu.edu ");
   }
  }
}
```

InetAddress methods

Useful methods:

a) static InetAddress getByName(String host)

b) static InetAddress getLocalHost()

c) String getHostAddress(); // in dotted form

d) String getHostName();

The URL Class 1

The `java.net.URL` is an abstraction of a Uniform Resource Locator (URL).

URLs are composed of five pieces:

1. The scheme, also known as the protocol
2. The authority
3. The path
4. The query string
5. The fragment identifier, also known as the section or ref

<scheme>://<authority><path>?<query>#<fragment>

The URL Class 2

For example, given the URL :

http://www.ibiblio.org/javafaq/javabooks/index.html?isbn=123456789#toc

1. scheme : http
2. authority : www.ibiblio.org
3. path : /javafaq/books/javabooks/index.html
4. query string : isbn=123456789
5. fragment identifier : toc

The URL Class 3

The authority may further be divided into the user info, the host, and the port.

For example, in the URL http://admin@www.blackstar.com:8080/

1. user info : admin
2. host : www.blackstar.com
3. port : 8080

The URL Class 4

The java.net.URL class provides static methods for getting the above mentioned information:

a) getFile( )
b) getHost()
c) getPort( )
d) getProtocol()
e) getRef()
f) getQuery( )
g) getPath( )
h) getUserInfo( )
i) getAuthority( )

The URL Class 5

Unlike the InetAddress objects, you can construct instances of java.net.URL using one of its six constructors.

1) `public URL(String url) throws MalformedURLException`

2) `public URL(String protocol, String hostname, String file) throws MalformedURLException`

3) `public URL(String protocol, String host, int port, String file) throws MalformedURLException`

4) `public URL(URL base, String relative) throws MalformedURLException`

The URL Class 6

5) `public URL(URL base, String relative, URLStreamHandler handler) // 1.2 throws MalformedURLException`

6) `public URL(String protocol, String host, int port, String file, // 1.2 URLStreamHandler handler) throws MalformedURLException`

Example 1

The following program will test the protocol supported by the browser:

```
import java.net.*;
public class ProtocolTester {
    public static void testProtocol(String url) {
     try {
       URL u = new URL(url);
       System.out.println(u.getProtocol( ) + " is
        supported");
     }
     catch (MalformedURLException ex) {
       String protocol =
        url.substring(0, url.indexOf(':'));
       System.out.println(protocol + " is not
        supported");
     }
   }
}
```

Example 2

You can test it with the following Tester:

```
public class Tester {
  public static void main(String[] args) {
    ProtocolTester.testProtocol("http://www.adc.org");
    ProtocolTester.testProtocol("https://www.amazon.co
    m/exec/obidos/order2/");
    ProtocolTester.testProtocol("ftp://metalab.unc.edu
    /pub/languages/java/javafaq/");
```

Lab Work: URL 1

1) Write a program that will split the input URL into corresponding parts.

Given: java URLSplitter

```
http://www.unu.iist.edu/demoweb/html-
primer.html#A1.3.3.3
```

Output:

```
The output will be:
The URL is http://www.unu.iist.edu/demoweb/html-
primer.html#A1.3.3.3
The scheme is http
The user info is null
The host is www.unu.iist.edu
The port is -1

The path is /demoweb/html-primer.html
The ref is A1.3.3.3
The query string is null
```

Lab Work: URL 2

Try to test your program using the following URL:

```
ftp://mp3:mp3@138.247.121.61:21000/c%3a/
http://www.oreilly.com
http://www.ibiblio.org/nywc/compositions.phtml?categ
ory=Piano
http://admin@www.blackstar.com:8080/
```

2) What is the difference between file and path?

Retrieving Data from a URL

The URL class provides methods for retrieving data from a URL:

```
public InputStream openStream( ) throws IOException

public URLConnection openConnection( ) throws
IOException

public URLConnection openConnection(Proxy proxy)
throws IOException // 1.5

public Object getContent( ) throws IOException

public Object getContent(Class[] classes) throws
IOException // 1.3
```

Retrieving Data from a URL 1

Procedure to use the methods:

1) Create an URL object

   e.g. `URL u = new URL("http://www.iist.unu.edu");`

2) Open an InputStream object directly from the URL object

   e.g. `InputStream in = u.openStream( );`

3) Or open an URLConnection object from the URL object and then get an InputStream object from the URLConnection object .

   e.g.
   ```
   URLConnection uc = u.openConnection( );
   InputStream in = uc.getInputStream( );
   ```

4) In either case, you will have an InputStream. What's followed is the normal I/O procedure for getting data.

5) Don't forget to put the try catch block for catching the `MalformedURLException` and `IOException`.

Retrieving Data from a URL 2

What is the difference between using the openStream and openConnection method?

1) openStream method only give you the access to the raw data and cannot detect the encoding information.
2) openConnection method opens a socket to the specified URL and returns a URLConnection object.
3) The URLConnection object gives you access to everything sent by the server. You can access all the metadata specified by the protocol such as the scheme. The URLConnection class also lets you write data to as well as read from a URL.

Retrieving Data from a URL 3

The following methods are used to access the header fields and the contents after the connection is made to the remote object:

1)getContent

2)getHeaderField

3)getInputStream

4)getOutputStream

Retrieving Data from a URL 4

Certain header fields are accessed frequently. The methods:

1) getContentEncoding
2) getContentLength
3) getContentType
4) getDate
5) getExpiration
6) getLastModifed

Example : Reading from URL 1

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
  ublic static void main(String[] args) throws
   Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
        new InputStreamReader (yc.getInputStream()));
        String inputLine;
```

Example : Reading from URL 2

```
while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is an abstraction of a URL.

Most URIs used in practice are URLs, but most specifications and standards such as XML are defined in terms of URIs

In Java 1.4 and later, URIs are represented by the `java.net.URI` class.

you should use the URL class when you want to download the content of a URL and the URI class when you want to use the URI for identification rather than retrieval.

When you need to do both, you may convert from a URI to a URL with the toURL( ) method, and in Java 1.5 you can also convert from a URL to a URI using the toURI( ) method of the URL class.

Uniform Resource Identifier (URI)

A URI reference has up to three parts: a scheme, a scheme-specific part, and a fragment identifier. The general format is:
scheme:*scheme-specific-part*:*fragment* .

Getter methods:

1) public String getScheme( )
2) public String getSchemeSpecificPart( )
3) public String getRawSchemeSpecificPart( )
4) public String getFragment( )
5) public String getRawFragment( )

Lab Work: URI

1) Write a program that will split the input URI into corresponding parts.
2) List the methods you can get form the URI class using the javadoc.

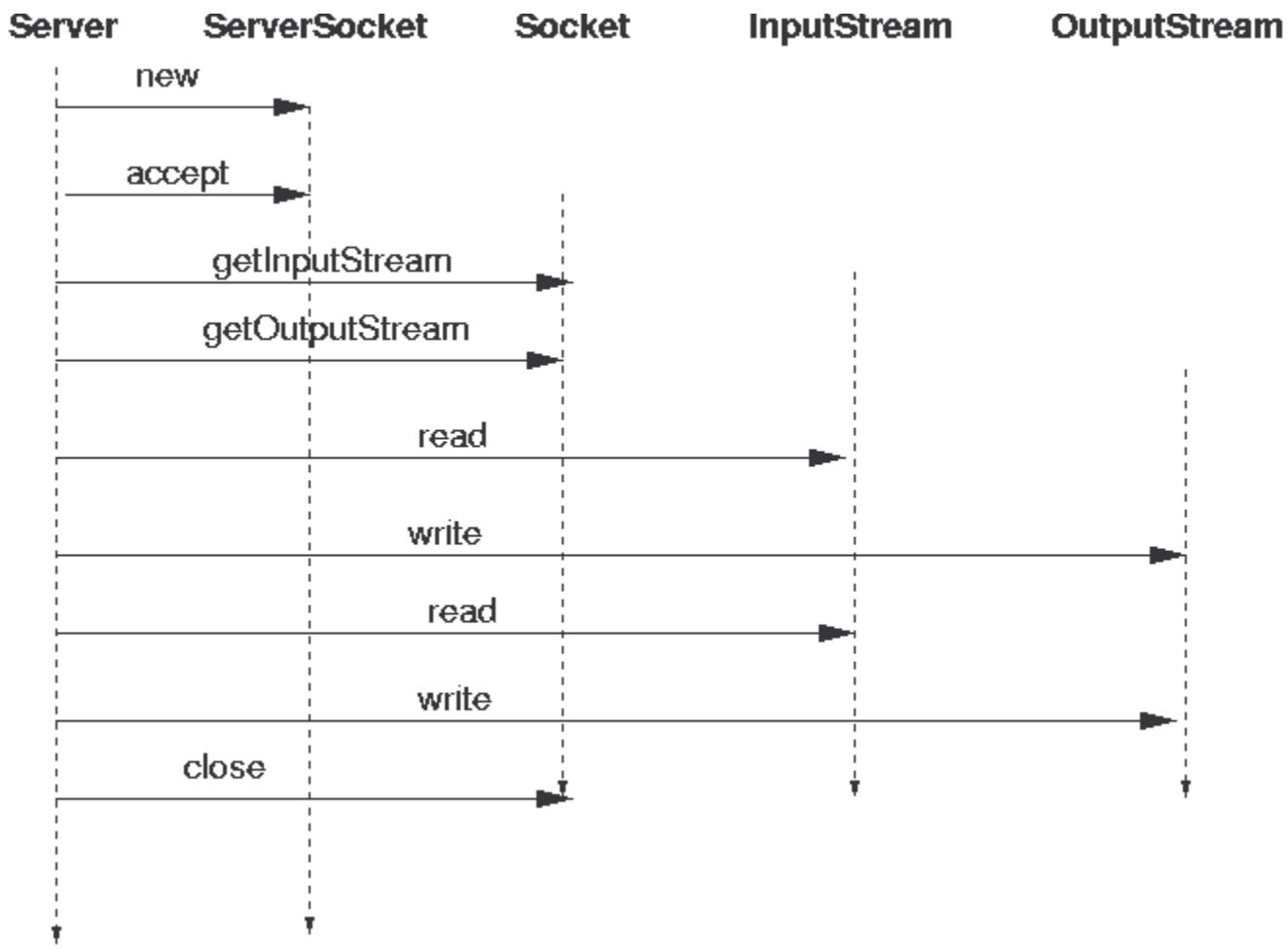
Networking Examples

Now you have the basic concepts for different components for Java networking. Let's try to start some simple experiments.

A Simple Example

Create a Server

How to create a server?



Echo Server 1

```
import java.io.*;
import java.net.*;
public class EchoServer {
  public static int MYECHOPORT = 8189;
    public static void main(String argv[]) {
      ServerSocket s = null;
      try {
          s = new ServerSocket(MYECHOPORT);
      } catch(IOException e) {
          System.out.println(e);
          System.exit(1);
      }
```

Echo Server 2

```
while (true) {
        Socket incoming = null;
        try {
          incoming = s.accept();
        } catch(IOException e) {
          System.out.println(e);
          continue;
        }
        try {
          incoming.setSoTimeout(10000); //10 seconds
        } catch(SocketException e) {
          e.printStackTrace();
        }
```

Echo Server 3

```
try {         handleSocket(incoming);
            } catch(InterruptedIOException e) {
              System.out.println("Time expired " + e);
            } catch(IOException e) {
              System.out.println(e);
            }
            try {
              incoming.close();
            } catch(IOException e) {
              // ignore
            }
      }
    }
```

Echo Server 4

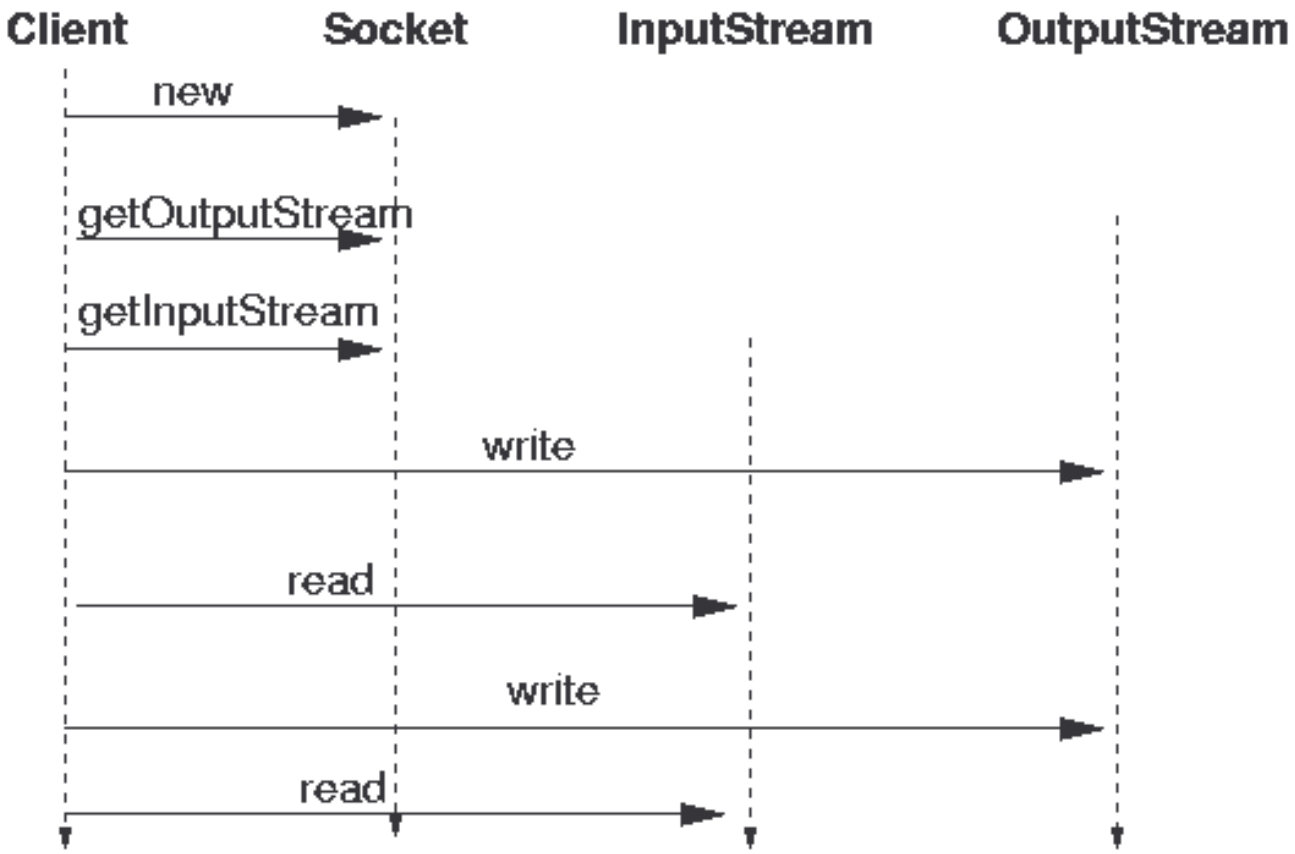
```
public static void handleSocket(Socket incoming)
      throws IOException {
      BufferedReader reader =
          new BufferedReader(new InputStreamReader(
                        incoming.getInputStream()));
      PrintStream out =
          new PrintStream(incoming.getOutputStream());
      out.println("Hello. Enter BYE to exit");

      boolean done = false;
      while ( ! done) {
          String str = reader.readLine();
```

Echo Server 5

```
if (str == null) {
            done = true;
            System.out.println("Null received");
          }
          else {
            out.println("Echo: " + str);
            if (str.trim().equals("BYE"))
                done = true;
          }
            }
      incoming.close();
    }
}
```

Create a Client

How to create a client connected to a sever?



Echo Client 1

```
import java.io.*;
import java.net.*;


public class EchoClient {
  public static void main(String[] args) throws
      IOException {


            Socket echoSocket = null;
            PrintWriter out = null;
            BufferedReader in = null;
            BufferedReader stdIn = null;
```

Echo Client 2

```
try {
        echoSocket = new Socket("localhost", 8189);
        out = new
        PrintWriter(echoSocket.getOutputStream(), true);
        in = new BufferedReader(new
        InputStreamReader(echoSocket.getInputStream()));
        System.out.println (in.readLine());
        } catch (UnknownHostException e) {
                System.err.println("Don't know about host.");
                System.exit(1);
        } catch (IOException e) {
                System.err.println("Couldn't get I/O for "
                + "the connection to server.");
```

Echo Client 3

```
        System.exit(1);
}
try {
        stdIn = new BufferedReader(new InputStreamReader
                (System.in));
        String userInput;
        while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
         }
        } catch (SocketException e) {
                        System.err.println("Socket closed");
        }
```

Echo Client 4

```
finally {
                    if (out != null)
                        out.close();
                    if (in != null)
                        in.close();
                    if (stdIn != null)
                        stdIn.close();
                    if (echoSocket != null)
                        echoSocket.close();
                }
        }
}
```

Lab Work: Chatting Program

1) Modify the previous Echo Sever and Echo Client examples to create a server-client chatting program.

   Hints:

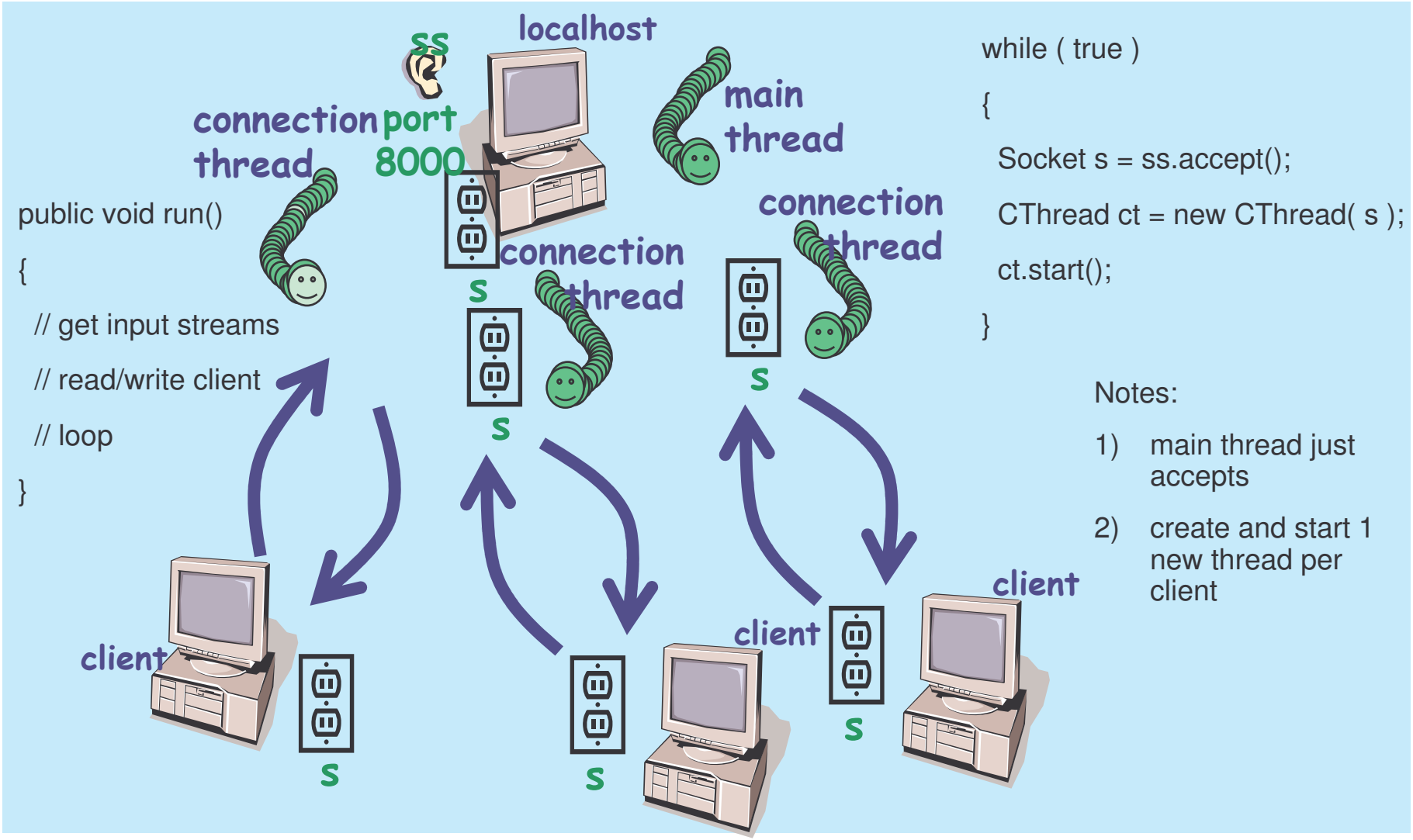
   a) You need to create a server which will wait for the client to establish the connection. Then it will print a statement to the client's console notifying that a connection is made.

   b) Once the connection is made, both the server and client will be able read message from the counterpart and write message to it.

Exercise : Multiple Clients 1

1) Please modify the previous lab work. you need to make your server to be able to talk to multiple clients connected the server.

Exercise : Multiple Clients 2



Secure Sockets

Starting from JDK 1.4, Java Secure Sockets Extension (JSSE) is part of the standard distribution.

JSSE uses the Secure Sockets Layer (SSL) Version 3 and Transport Layer Security (TLS) protocols and their associated algorithms to secure network communications.

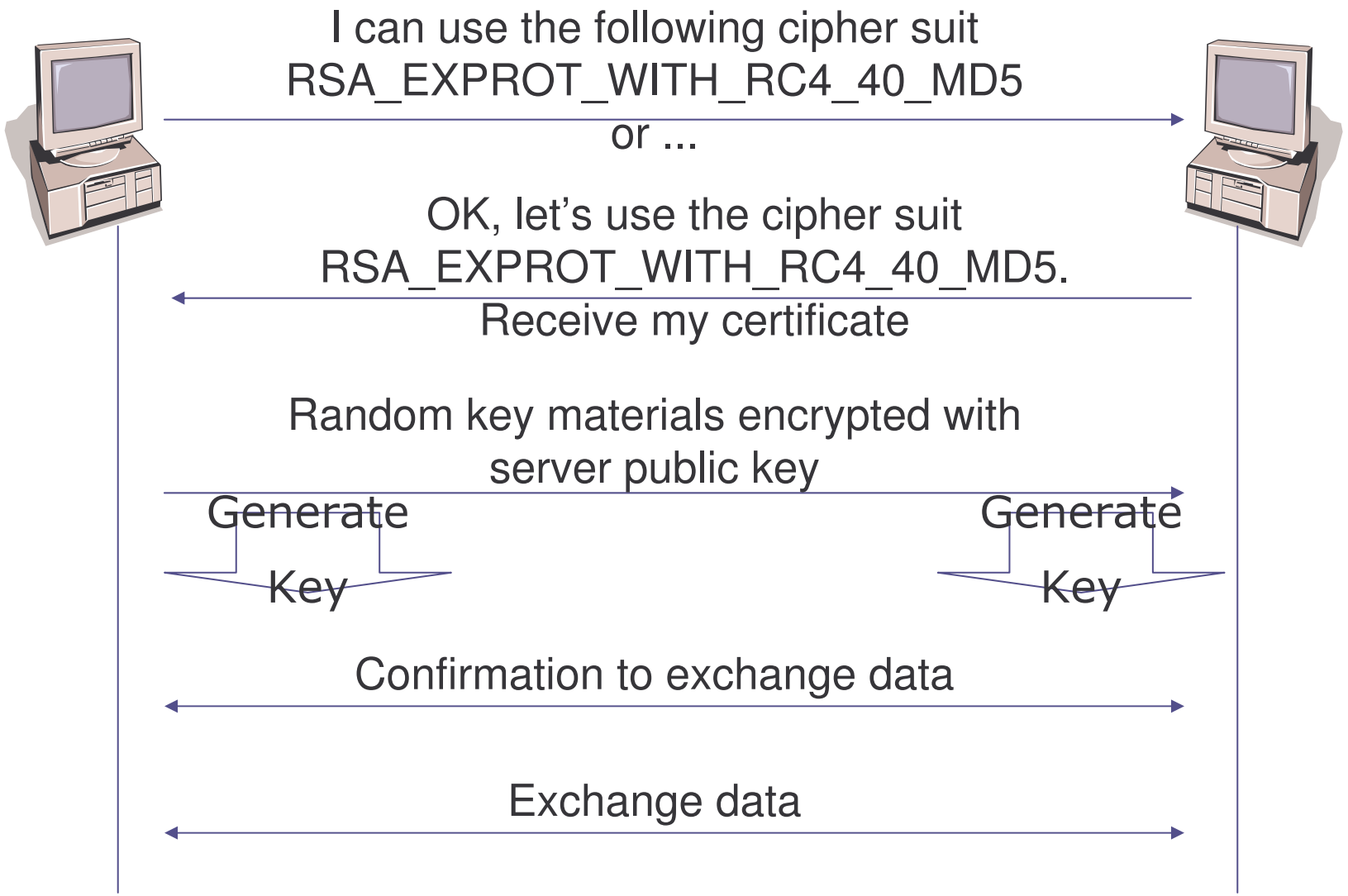
JSSE abstracts all the low-level details such as keys exchange, authentication, and data encryption. All you have to do is to send your data over the streams from the secured sockets obtained.

Java Secure Socket Extension

The Java Secure Socket Extension is divided into four packages:

1) javax.net.ssl :The abstract classes that define Java's API for secure network communication.
2) javax.net :The abstract socket factory classes used instead of constructors to create secure sockets.
3) javax.security.cert : A minimal set of classes for handling public key certificates that's needed for SSL in Java 1.1. (In Java 1.2 and later, the java.security.cert package should be used instead.)
4) com.sun.net.ssl : The concrete classes that implement the encryption algorithms and protocols in Sun's reference implementation of the JSSE.

SSL Handshake



Secure Client Sockets 1

Procedures to create a secure client socket:

1) get an instance of `SocketFactory` by invoking the static `SSLSocketFactory.getDefault()` method. e.g.

```
SocketFactory sf = SSLSocketFactory.getDefault();
```

2) use one of these five overloaded `createSocket( )` methods to build an SSLSocket:

```
1.public abstract Socket createSocket(String host,
  int port) throws IOException,
  UnknownHostException

2.public abstract Socket createSocket(InetAddress
  host, int port) throws IOException
```

Secure Client Sockets 2

3. `public abstract Socket createSocket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException`

4. `public abstract Socket createSocket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException`

5. `public abstract Socket createSocket(Socket proxy, String host, int port, boolean autoClose) throws IOException`

Secure Client Sockets 3

3) Once the socket has been created, you use it just like any other socket, through its getInputStream( ), getOutputStream(), and other methods.

   For example, if the following purchasing information is required to be sent over the network:

   a) Name: John Smith

   b) Product-ID: 67X-89

   c) Address: 1280 Deniston Blvd, NY NY 10003

   d) Card number: 4000-1234-5678-9017

   e) Expires: 08/05

Using JSSE, the following code will do the work for you:

Secure Client Sockets 4

```
try {
    SSLSocketFactory factory
    = (SSLSocketFactory) SSLSocketFactory.getDefault( );
    Socket socket = factory.createSocket("localhost",
       7000);
    Writer out = new
       OutputStreamWriter(socket.getOutputStream( ),
       "ASCII");
    out.write("Name: John Smith\r\n");
```

Secure Client Sockets 5

```
    out.write("Product-ID: 67X-89\r\n");
    out.write("Address: 1280 Deniston Blvd, NY NY
       10003\r\n");
    out.write("Card number: 4000-1234-5678-9017\r\n");
    out.write("Expires: 08/05\r\n");
    out.flush( );
    out.close( );
    socket.close( );
} catch (IOException ex) {
     ex.printStackTrace( );
}
```

Configuring Secure Sockets

Methods are available for configuring how much and what kind of authentication and encryption is performed.

a) getSupportedCipherSuites() method tells you which combination of algorithms is available on a given socket

b) getEnabledCipherSuites() method tells you which suites this socket is willing to use

c) You can change the suites the client attempts to use via the setEnabledCipherSuites(String[] suites) method

- Sun's JDK 1.4 supports 23 cipher suites. For the list of the supported cipher suites, please check with JavaDoc.

d) There are still methods for handling handshaking and sessions, and I will open these for your further study.

Secure Server Sockets 1

Procedures to create a secure server socket:

1) get an instance of `ServerSocketFactory` by invoking the static `SSLSocketFactory.getDefault()` method. e.g.

```
ServerSocketFactory sf =
  SSLServerSocketFactory.getDefault();
```

2) use one of these three overloaded `createServerSocket( )` methods to build an SSLServerSocket:

```
1.public abstract ServerSocket
  createServerSocket(int port) throws IOException

2.public abstract ServerSocket
  createServerSocket(int port, int queueLength)
  throws IOException

3.public abstract ServerSocket
  createServerSocket(int port, int queueLength,
  InetAddress interface) throws IOException
```

Secure Server Sockets 2

3) Unlike creating the client socket, you need to do more to set up the encryption for the server socket.

   This setup varies between different JSSE implementations. In Sun's implementation, you may need to do the followings:

   1. Generate public keys and certificates using *keytool*.
   2. Pay money to have your certificates authenticated by a trusted third party such as Verisign.
   3. Create an SSLContext for the algorithm you'll use.
   4. Create a TrustManagerFactory for the source of certificate material you'll be using.

Secure Server Sockets 3

5. Create a KeyManagerFactory for the type of key material you'll be using.
6. Create a KeyStore object for the key and certificate database. (Sun's default is JKS.)
7. Fill the KeyStore object with keys and certificates; for instance, by loading them from the filesystem using the pass phrase they're encrypted with.
8. Initialize the KeyManagerFactory with the KeyStore and its pass phrase.
9. Initialize the context with the necessary key managers from the KeyManagerFactory, trust managers from the TrustManagerFactory, and a source of randomness. (The last two can be null if you're willing to accept the defaults.)

Lab Work: Secure Sockets 1

1) Please try to run through the process for setting up a secure socket server as following and implement it in java code.
   a) Generate public keys and certificates using *keytool*
      - *D:\JAVA>**keytool -genkey -alias ourstore -keystore jnp3e.keys***
      - Answer some questions and please remember the password you entered. You will need it later.
      - A file jnp3e.keys will be generated and protected by the password you entered.
   b) If you don't want to pay for the digital ID for experiment, you can use the verified keystore file called testkeys, protected with the password "passphrase", included in SUN's JSSE implementation package.
   c) Create a class named SecureServer.

Lab Work: Secure Sockets 2

d) Import the necessary packages.

e) Define variables:

- int PORT – default port number
- String ALGORITHM – algorithm for setting the SSLContext (“SSL”)
- String KEYFILE – in our case, “keyfiles”
- String PASSWORD – in our case, “passphrase”

f) Create the context using SSLContext.getInstance(arg) method. You need to pass the variable ALGORITHM as argument.

g) As accepted the default, we don’t need to create theTrustManagerFactory

Lab Work: Secure Sockets 3

h) Create the KeyManagerFactory using the static method KeyManagerFactory.getInstance (arg). The Sun implementation will need "SunX509“ as argument.

i) Create the KeyStore using the static method KeyStore.getInstance(arg) Use “JKS” as argument.

j) For security, every key store is encrypted with a pass phrase that must be provided before we can load it from disk. The pass phrase is stored as a char[] array so it can be wiped from memory quickly rather than waiting for a garbage collector. Of course using a string literal here completely defeats that purpose.

k) Use the load method from the KeyStore to load the key file. Check the javadoc for method usage.

Lab Work: Secure Sockets 4

l) Use the init method from the KeyManagerFactory to initialize the KeyManagerFactory. Check the javadoc for method usage.

m) Use the init method from theSSLContext to initialize the context. Check the javadoc for method usage.

2) You have completed the setup process at this point. Create a secure server socket for this server at port 7000.

3) Test your server with the secure client.

New I/O (NIO) API

Java introduce the new I/O (NIO) API in v1.4.

New features:

a) Buffers for data of primitive types

b) Character-set encoders and decoders

c) A pattern-matching facility based on Perl-style regular expressions

d) Channels, a new primitive I/O abstraction

e) A file interface that supports locks and memory mapping

f) A multiplexed, non-blocking I/O facility for writing scalable servers

Why NIO?

Allow Java programmers to implement high-speed I/O.

NIO deals with data in blocks which can be much faster than processing data by the (streamed) byte.

NIO Components

1) Buffers
2) Channels
3) Selectors
4) Regular Expressions
5) Character Set Coding

Buffers

In the NIO library, all data is handled with buffers.

A buffer is essentially an array. Generally, it is an array of bytes, but other kinds of arrays can be used.

A buffer also provides structured access to data and also keeps track of the system's read/write processes.

Types:

a) ByteBuffer
b) CharBuffer
c) ShortBuffer
d) IntBuffer
e) LongBuffer·
f) FloatBuffer
g) DoubleBuffer

Channels

Channel is like a stream in original I/O.

You can read a buffer from and write a buffer to a channel.

Unlike streams, channels are bi-directional.

Read from a file

Codes for reading from a file:

```
FileInputStream fin = new
FileInputStream( "readandshow.txt" );

FileChannel fc = fin.getChannel();

ByteBuffer buffer = ByteBuffer.allocate( 1024 );

fc.read( buffer );
```

Write to a file

Codes for writing to a file:

```
FileOutputStream fout = new
FileOutputStream( "writesomebytes.txt" );

FileChannel fc = fout.getChannel();

ByteBuffer buffer = ByteBuffer.allocate( 1024 );

for (int i=0; i<message.length; ++i) {

buffer.put( message[i] );

}

buffer.flip();//prepares the buffer to have the newly-
              //read data written to another channel

fc.write( buffer );
```

Lab Work: NIO

1) Refer to the example, please use the NIO to create a program to write a String to a text file and store in your computer.
2) Read the text file back and print the content on the screen.
3) You may need to use the WritableByteChannel as following:

```
WritableByteChannel wbc =
                Channels.newChannel(System.out);
```

Server with NIO

Channels and buffers are really intended for server systems that need to process many simultaneous connections efficiently.

Handling servers requires the new selectors that allow the server to find all the connections that are ready to receive output or send input.

Asynchronous I/O

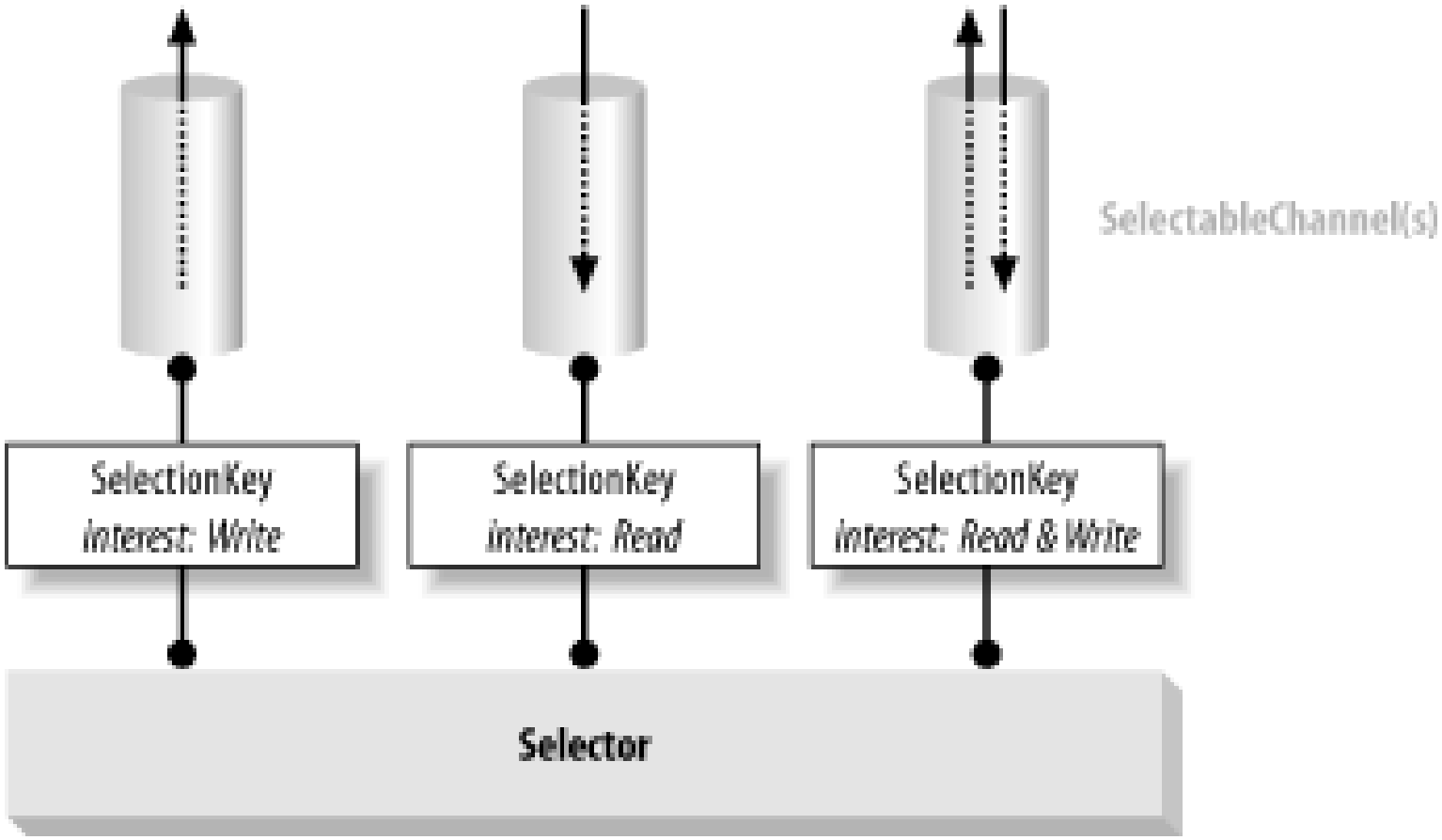
Asynchronous I/O is made possible in NIO with scalable sockets, which consist of the following components:

a) Selectable Channel - A channel that can be multiplexed

b) Selector - A multiplexor of selectable channel

c) Selection key - A token representing the registration of a selectable channel with a selector

Selectors, Keys and Channels



The Selection Process 1

1) Create a Selector and register channels with it

   Use `Open()` method to create a Selector.

   The `register()` method is on `SelectableChannel`, not `Selector`

2) Invoke `select()` method on the `Selector` object

3) Retrieve the Selected Set of keys from the Selector

   Selected set: Registered keys with non-empty Ready Sets

```
keys = selector.selectedKeys()
```

The Selection Process 2

4) Iterate over the Selected Set
    1) Check each key's Ready Set
    2) Remove the key from the Selected Set (`iterator.remove()`)
        1) Bits in the Ready Sets are never reset while the key is in the Selected Set
        2) The Selector never removes keys from the Selected Set – you must do so
    3) Service the channel (`key.channel()`) as appropriate (read, write, etc)

Example : NIO Server 1

Skeleton dodes for a simple server with NIO:

```
  //Open a ServerSocketChannel
  ServerSocketChannel serverChannel =
  ServerSocketChannel.open( );
//make the ServerSocketChannel non-blocking.
//necessary for asynchronous i/o
  serverChannel.configureBlocking(false);
  ServerSocket ss = serverChannel.socket( );
  // bind the socket to a specific port
  ss.bind(new InetSocketAddress(PORT_NO));
```

Example : Create NIO Server 2

```
//Open the selector

  Selector selector = Selector.open( );

//use the channel's register() method to register the
  //ServerSocketchannel with the selector.


  serverChannel.register(selector,
  SelectionKey.OP_ACCEPT);
```

Example : Create NIO Server 3

```
//check whether anything is ready to be acted on, call
//the selector's select( ) method. For a long-running
//server, this normally goes in an infinite loop:
while (true) {
    try {
      selector.select( );
    }
    catch (IOException ex) {
      ex.printStackTrace( );
      break;
    }
```

Example : Create NIO Server 4

```
 // process selected keys...

//selectedKeys( ) method returns a java.util.Set
//containing one SelectionKey object for each ready
//channel

 Set readyKeys = selector.selectedKeys( );
Iterator iterator = readyKeys.iterator( );
while (iterator.hasNext( )) {

    SelectionKey key = (SelectionKey)

    (iterator.next( ));

     // Remove key from set

    iterator.remove( );
```

Example : Create NIO Server 5

```
  // You can obtain the channel using the channel()
  //methods of the SelectionKey. Catch the IOExcepton.
try{
      if (key.isAcceptable( ))
        { ServerSocketChannel
          server = (ServerSocketChannel ) key.channel( );
           SocketChannel client = server.accept( );
        // Data manipulation
         System.err.println("Got connection from
  "+client.socket().getInetAddress().getHostName());
```

Example : Create NIO Server 6

```
                //Send some message to client

                ByteBuffer bb =
                ByteBuffer.allocateDirect(1024);

                byte[] message = "Hello... You are
                reaching NIO Server".getBytes();

                bb.put( message);

                bb.flip();

                client.write( bb);

                }

        } catch (IOException e) {   }

    }

}
```

User Datagram Protocol

The User Datagram Protocol (UDP) is an alternative protocol for sending data over IP

UDP is very quick, but not reliable.

Java's implementation of UDP is split into two classes: DatagramPacket and DatagramSocket.

The DatagramPacket class stuffs bytes of data into UDP packets called datagrams and lets you unstuff datagrams that you receive.

A DatagramSocket sends as well as receives UDP datagrams.

Constructors for DatagramPacket

For receiving datagrams:

a) `public DatagramPacket(byte[] buffer, int length)`

b) `public DatagramPacket(byte[] buffer, int offset, int length)`

For sending datagrams:

a) `public DatagramPacket(byte[] data, int length, InetAddress destination, int port)`

b) `public DatagramPacket(byte[] data, int offset, int length, InetAddress destination, int port) // Java 1.2`

c) `public DatagramPacket(byte[] data, int length, SocketAddress destination, int port) // Java 1.4`

d) `public DatagramPacket(byte[] data, int offset, int length, SocketAddress destination, int port) //Java 1.4`

Constructors for DatagramSocket

For socket bound to an anonymous port:

a) `public DatagramSocket( ) throws SocketException`

For socket listen for incoming datagrams on a particular port :

a) `public DatagramSocket(int port) throws SocketException`

Others constructors:

a) `public DatagramSocket(int port, InetAddress interface) throws SocketException`

b) `public DatagramSocket(SocketAddress interface) throws SocketException // Java 1.4`

c) `protected DatagramSocket(DatagramSocketImpl impl) throws SocketException // Java 1.4`

Sending and Receiving

After constructed the DatagramPacket, you can send and receive datagram from it :

a) `public void send(DatagramPacket dp) throws IOException`

b) `public void receive(DatagramPacket dp) throws IOException`

Lab Work: UDP

1) Write a EchoUDP server which will send back any message received from client.
2) Write a UDPclient which sends some message to the EchoUDP server for testing it.

Summary

In this session, we cover the followings:

1) Review the basic concepts for networking.
2) Discuss how to create simple server-client program.
3) Discuss the secure socket implementation in Java.
4) Introduce the NIO API.
5) Introduce the Java implementation for User Datagram Protocol.

Database Connectivity

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Overview

We are going to consider the following under this section:

1) Introduction
2) JDBC Overview
3) Information Processing

Introduction

Buried within the term "enterprise" is the idea of a business taken wholistically.

An enterprise solution identifies common problem domains within a business and provides a shared infrastructure for solving those problems.

Example:

If your business is running a bank, your individual branches may have different business cultures, but those cultures do not alter the fact that they all deal with customers and accounts. Looking at this business from an enterprise perspective means abstracting away from irrelevant differences in the way the individual branches do things, and instead approaching the business from their common ground. It does not mean dismissing meaningful distinctions, such as the need for bilingual support in Macao SAR.

Enterprise Systems: What?

Enterprise Systems are Information systems that support many or all of the various parts of a firm.

They can also refer to many mission-critical applications which are mainframe-based (also referred to as legacy systems).

Also known as enterprise-wide information systems.

Information systems that allow companies to integrate information across operations on a company-wide basis.

Enterprise Systems: Types

Enterprise systems are broadly categorized into two:

1) Relational - Relational Database Management Systems (RDBMS)

2) Non-Relational
   a) Non Relational Databases (Legacy Database Systems)
   b) Legacy Systems (Older systems like old Cobol Applications)
   c) Enterprise Resource Planning
   d) Customer Relationship Management (CRM)
   e) Supply Chain Management

Enterprise Systems Integration

Enterprise Systems Integration is normally defined as the bringing together of:

1) People,
2) Processes and
3) Information

to work together in an harmonized way, and supported by appropriate information systems.

It is also the bringing together of both old and new applications to achieve the overall goal of an organization.

Reasons for Integration

There are many reasons why Enterprise information systems need to be integrated. Some are stated below:

1) need to persist and retrieve information from data repositories.

2) need to leverage existing systems and resources while adopting and developing new technologies and architectures

3) concern over the past years, that the mainframe was going away and that all legacy applications would be scrapped and completely rewritten.

Java Integration Mechanisms

Java provides some technologies to integrate Enterprise systems:

1) Java Database connectivity (JDBC) for connecting applications to relational Database systems

2) JavaIDL and Java Connector Architecture (JCA) for connecting to Non-Relational systems

The focus of this section is JDBC.

JavaIDL will be addressed later in the course while JCA will be addressed sometimes in the training.

Relational Database 1

Programming is all about data processing; data is central to everything you do with a computer.

Databases, like filesystems are nothing more than specialized tools for data storage.

Filesystems are good for storing and retrieving a single volume of information associated with a single virtual location.

In other words, when you want to save a WordPerfect document, a filesystem allows you to associate it with a location in a directory tree for easy retrieval later.

Relational Database 2

Databases provide applications with a more powerful data storage and retrieval system based on mathematical theories about data devised by Dr. E. F. Codd.

Conceptually, a relational database can be pictured as a set of spreadsheets in which rows from one spreadsheet can be related to rows from another.

Each spreadsheet in a database is called a table. As with a spreadsheet, a table is made up of rows and columns.

A database engine is a process instance of the software accessing your database. For example Oracle, mySQL, Sybase etc

Database engines use a standard query language to retrieve information from databases and is called Structured Query Language (SQL).

SQL

SQL is not much like any programming language you might be familiar with.

Instead, it is more of a structured English for talking to a database.

Characteristics:

1) SQL keywords are case-insensitive

2) table and column names may or may not be case-insensitive depending on your database engine

3) the space between words in a SQL statement is unimportant

4) have a newline after each word, several spaces, or just a single space

SQL Usage

With SQL you can ask the following question:

1) How do you get the data into the database?
2) And how do you get it out once it is in there?

Much of the simplest database access comes in the form of equally simple SQL statements.

Some of these commands are:

1) `Create`
2) `Insert`
3) `Select`
4) `Update`
5) `Delete`

Create Statement

SQL `CREATE` statement handles the creation of database entities.

The major database engines provide GUI utilities that allow you to create tables without issuing any SQL.

Uses:

1) To create database

Syntax:

```
CREATE DATABASE database_name
```

2) To create tables

Syntax:

```
CREATE TABLE table_name (
column_name column_type column_modifiers,
...,
column_name column_type column_modifiers)
```

Lab Work: Create Statement

2) Open another console and type

```
C:\mysql\bin>mysql -u root
```

2) Test your connection by typing

```
mysql>show databases;
```

5) Create emacao database

```
mysql>create database emacao;
```

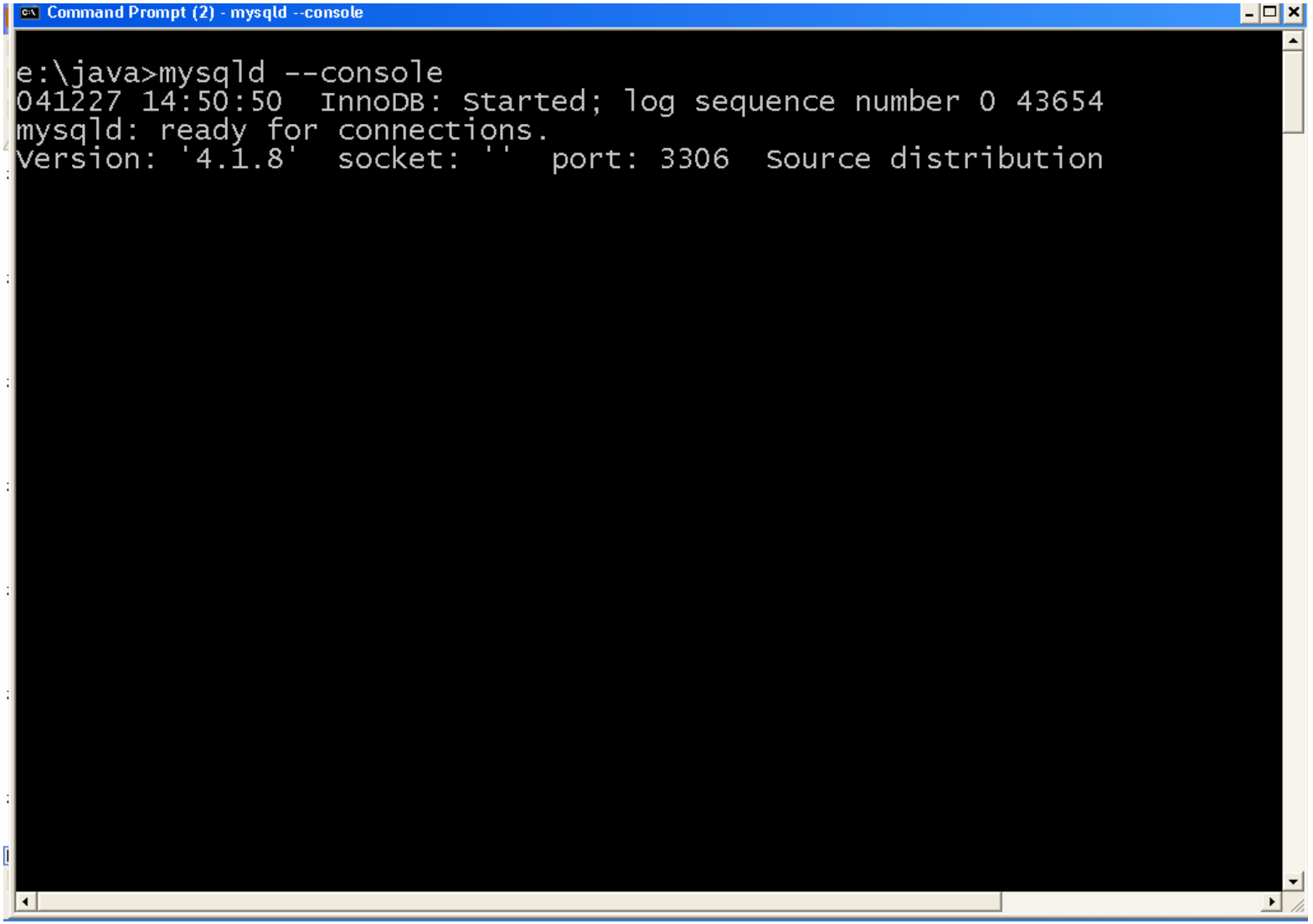
6) Change to the database

```
mysql>Use emacao;
```

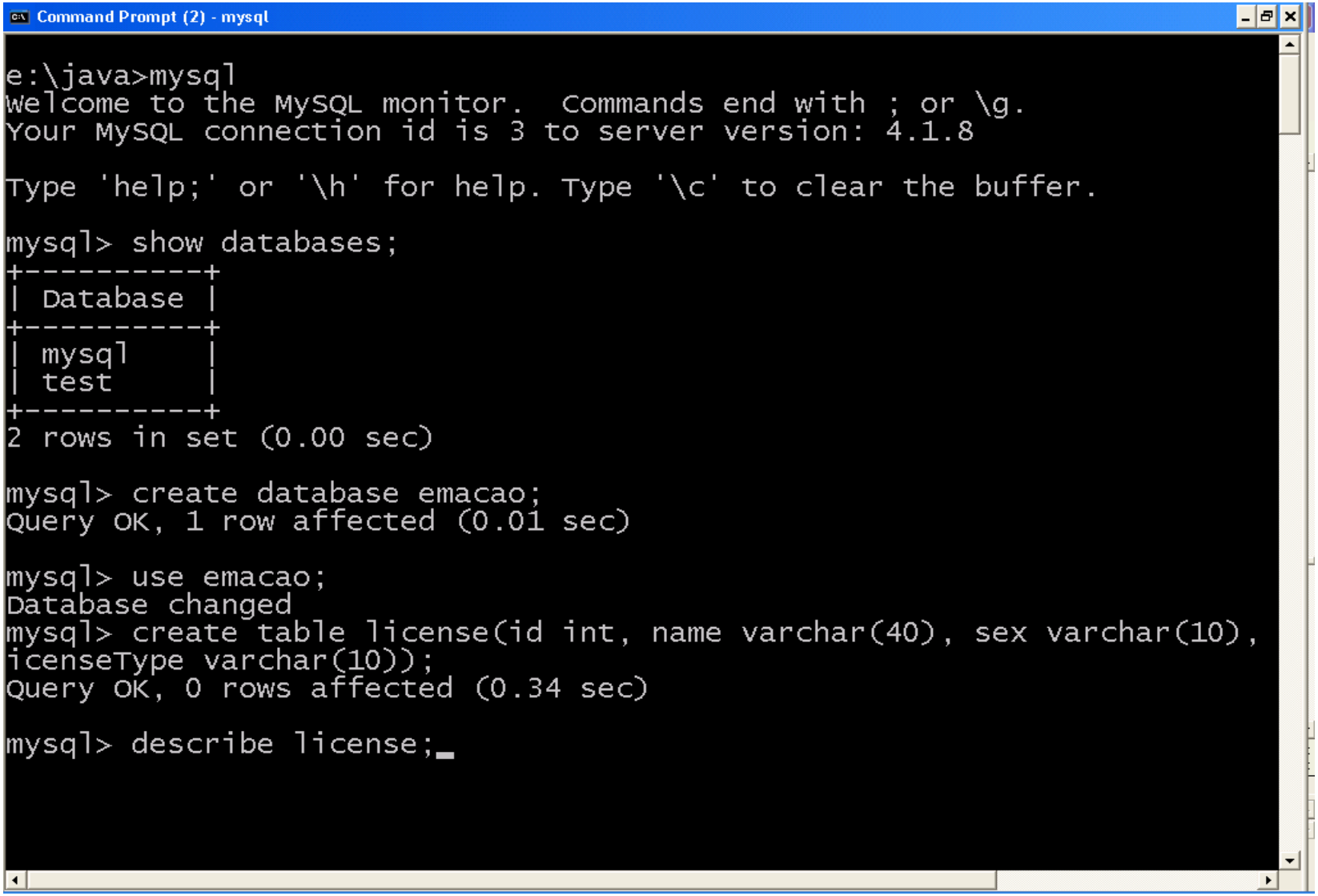
6) Create license table

```
mysql>create table license (id int, name
varchar(40), sex varchar(10), date varchar(12),
licenseType varchar(10));
```

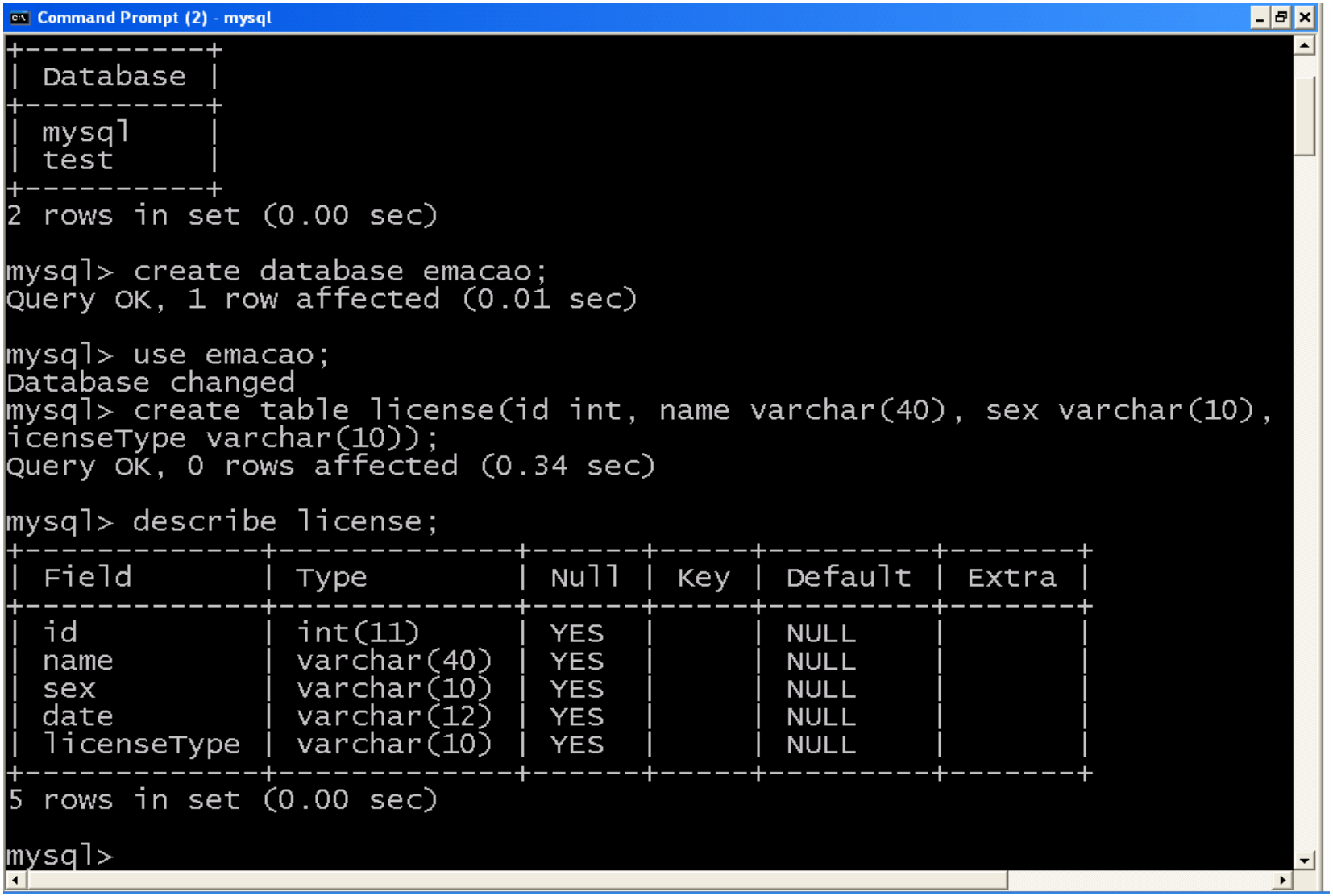
Lab Work: Console 1



Lab Work: Console 2



Lab Work: Console 3



Insert Statement

With the tables in place, you use the INSERT statement to add data to them.

Its form is:

```
INSERT INTO table_name(column_name, ..., column_name)
VALUES (value, ..., value)
```

The first column name matches to the first value you specify, the second column name to the second value you specify, and so on for as many columns as you are inserting.

If you fail to specify a value for a column that is marked as `NOT NULL`, you will get an error on insert.

Lab Work: Insert Statement

1) Insert the following records into the license table.

a) Id = 123

Name = Chong Gabriel

Sex = male

Date = 27/12/2004

LicenceType = Export

b) Id = 124

Name = martins Gabriel

Sex = Female

Date = 23/12/2004

LicenceType = Import

Update Statement

The `UPDATE` statement enables you to modify data that you previously inserted into the database.

Its form is:

```
UPDATE table_name
SET column_name = value,
...,
column_name = value
WHERE column_name = value
```

This statement introduces the `WHERE` clause. It is used to help identify one or more rows in the database.

Lab Work: Update Statement

1) Change the `ID` and `Name` of the record with ID `124` to `126` and `Martins Leo Gabriel` respectively.

Select Statement

The most common SQL command you will use is the `SELECT` statement.

It allows you to select specific rows from the database based on search criteria.

It takes the following form:

```
SELECT column_name, ..., column_name
FROM table_name
WHERE column_name = value
```

Lab Work: Select Statement

1) Retrieve all records from the table.
2) Retrieve all records from the table where `ID` is `126`.

Delete Statement

The `DELETE` command looks a lot like the `UPDATE` statement.

Its syntax is:

```
DELETE FROM table_name WHERE column_name = value
```

Instead of changing particular values in the row, DELETE removes the entire row from the table.

Lab Work: Delete Statement

1) Remove all records from the table where `ID` is `125`.
2) Retrieve all records from the table
3) Remove all records from the table.

Database Programming

Database programming has traditionally been a technological Tower of Babel.

You are faced with dozens of available database products, and each one talks to your applications in its own private language.

If your application needs to talk to a new database engine, you have to teach it (and yourself) a new language.

As Java programmers, however, you should not worry about such translation issues.

Java is supposed to bring you the ability to "write once, compile once, and run anywhere," so it should bring it to you with database programming, as well.

JDBC Overview

JDBC API is a set of interfaces designed to insulate a database application developer from a specific database vendor.

It enables the developer to concentrate on writing the application - making sure that queries to the database are correct and that the data is manipulated as designed.

Sun developed a single API for database access—JDBC.

Three main design goals:

1) JDBC should be a SQL-level API.
2) JDBC should capitalize on the experience of existing database APIs.
3) JDBC should be simple.

JDBC and Developer

What does JDBC provide the developer?

1) the developer can write an application using the interface names and methods described in the API, regardless of how they were implemented in the driver

2) the developer writes an application using the interfaces described in the API as though they are actual class implementations

3) the driver vendor provides a class implementation of every interface in the API so that when an interface method is used, it is actually referring to an object instance of a class that implemented the interface.

JDBC and Driver Vendors

What do the driver vendors provide?

Driver vendors provide implementations of JDBC interfaces.

The JDBC API also enables developers to pass any string directly to the driver.

This makes it possible for developers to make use of custom features of their database without requiring that the application use ANSI SQL

With JBDC you can :

1) establish a connection with a database or access any tabular data source
2) send SQL statement
3) process the results

JDBC Structure

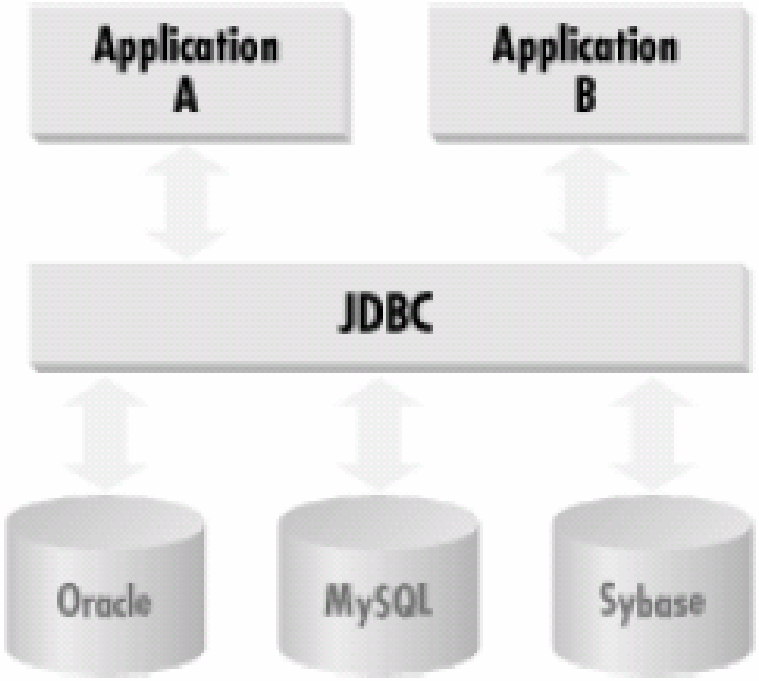
JDBC accomplishes its goals through a set of Java interfaces, each implemented differently by individual vendors.

The set of classes that implement the JDBC interfaces for a particular database engine is called a JDBC driver.

In building a database application, you do not have to think about the implementation of these underlying classes at all.

The whole point of JDBC is to hide the specifics of each database and let you worry about just your application.

JDBC Architecture



JDBC Driver Type 1

JDBC Driver fit into one of the following:

1) Type 1:JDBC-ODBC Bridge plus ODBC Driver

2) Type 2:A native API partly Java technology-enabled

3) Type 3:Pure Java Driver for Database Middleware

4) Type 4:Direct-to-Database Pure Java Driver

Type 1: JDBC-ODBC Bridge 1

Type 1: JDBC-ODBC Bridge provides JDBC access via one or more Open Database Connectivity (ODBC) drivers.

Advantage:

1) a good approach for learning JDBC

2) may be useful for companies that already have ODBC drivers installed on each client machine

3) may be the only way to gain access to some low-end desktop databases

Type 1: JDBC-ODBC Bridge 2

Disadvantage:

1) Not for large-scale applications. Performance suffers because there's some overhead associated with the translation work to go from JDBC to ODBC.

2) doesn't support all the features of Java

3) user is limited by the functionality of the underlying ODBC driver

Type 2: Partial Java driver 1

Converts calls to the JDBC API into calls that connect to the client machine's application programming interface for a specific database, such as IBM, Informix, Oracle or Sybase.

Advantage:

1) Performance is better than that of Type 1, in part because the Type 2 driver contains compiled code that is optimized for the back-end database server's operating system.

Type 2: Partial Java driver 2

Disadvantage:

1) user needs to make sure the JDBC driver of the database vendor is loaded onto each client machine

2) must have compiled code for every operating system that the application will run on

3) best use is for controlled environments, such as an intranet

Type 3: Pure Java Middleware 1

Type 3: Pure Java driver for database middleware translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software.

Advantage:

1) used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them

2) Server-based, so no need for JDBC driver code on client machine

3) the back-end server component is optimized for the operating system that the database is running on

Type 3: Pure Java Middleware 2

Type 3: Pure Java driver for database middleware translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software.

Advantage:

1) used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them

2) Server-based, so no need for JDBC driver code on client machine

3) the back-end server component is optimized for the operating system that the database is running on

Disadvantage:

1) Needs some database-specific code on the middleware server.

Type 4: Direct-to-database Pure

Type 4: Direct-to-database pure Java driver converts JDBC calls into packets that are sent over the network in the proprietary format used by the specific database. Allows a direct call from the client machine to the database.

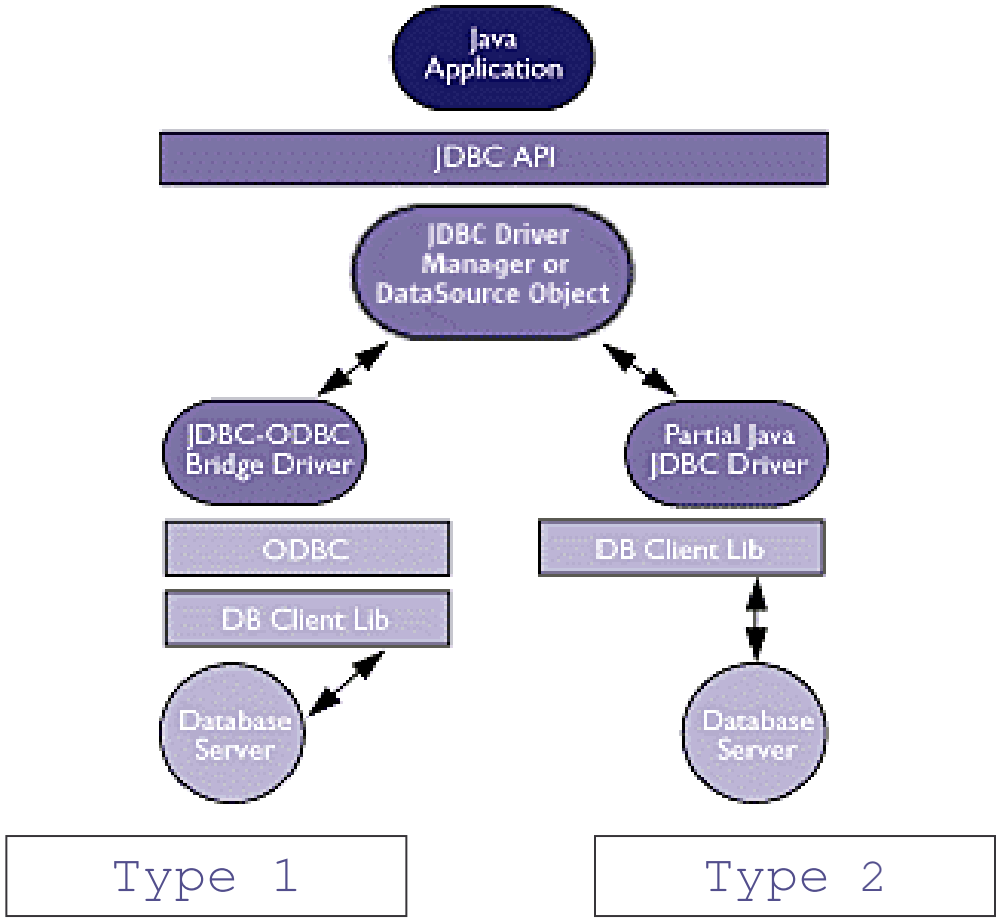
Advantage:

1) No need to install special software on client or server. Can be downloaded dynamically.

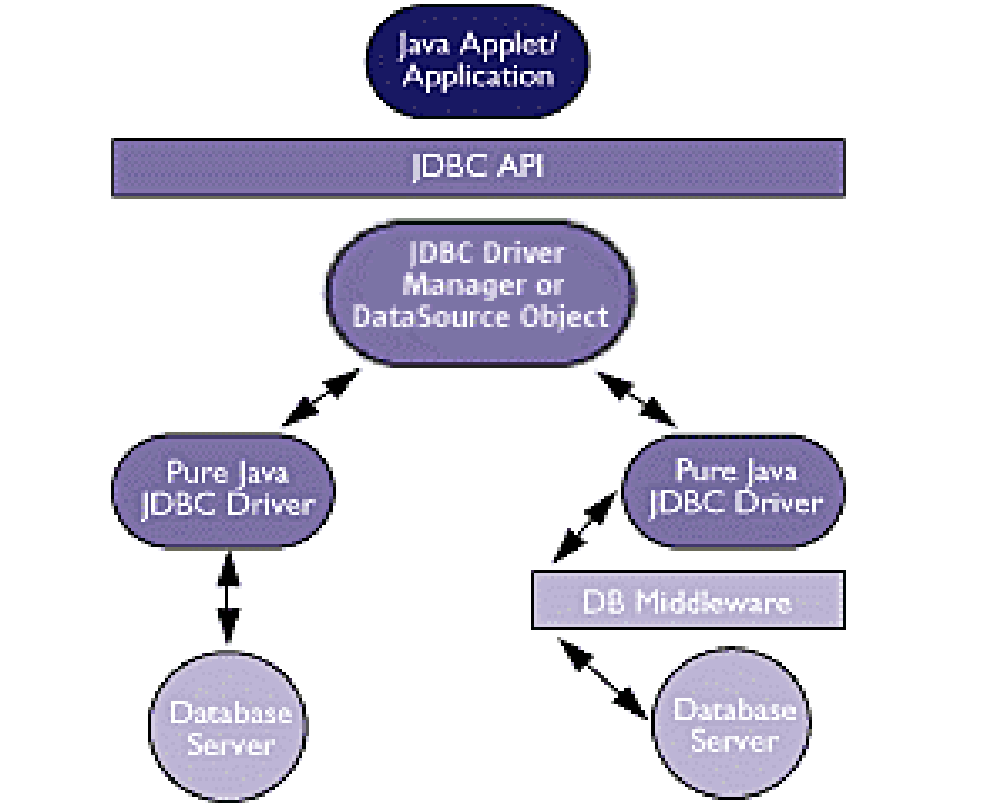
Disadvantage:

1) not optimized for server operating system, so the driver can't take advantage of operating system features

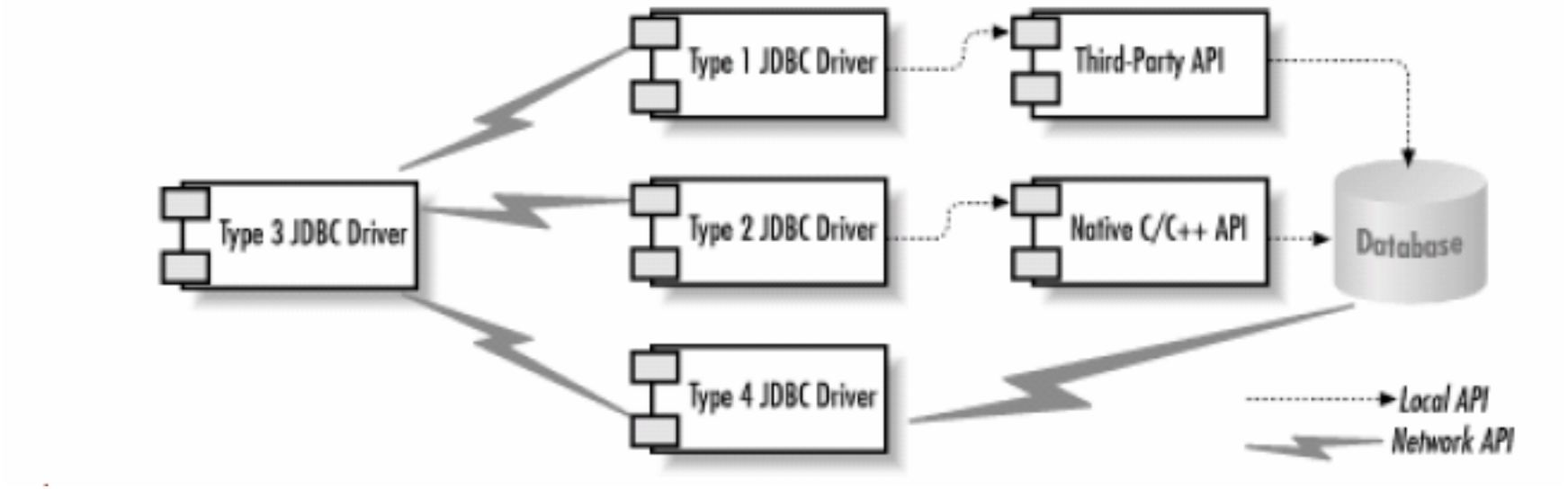
JDBC Driver Type 2



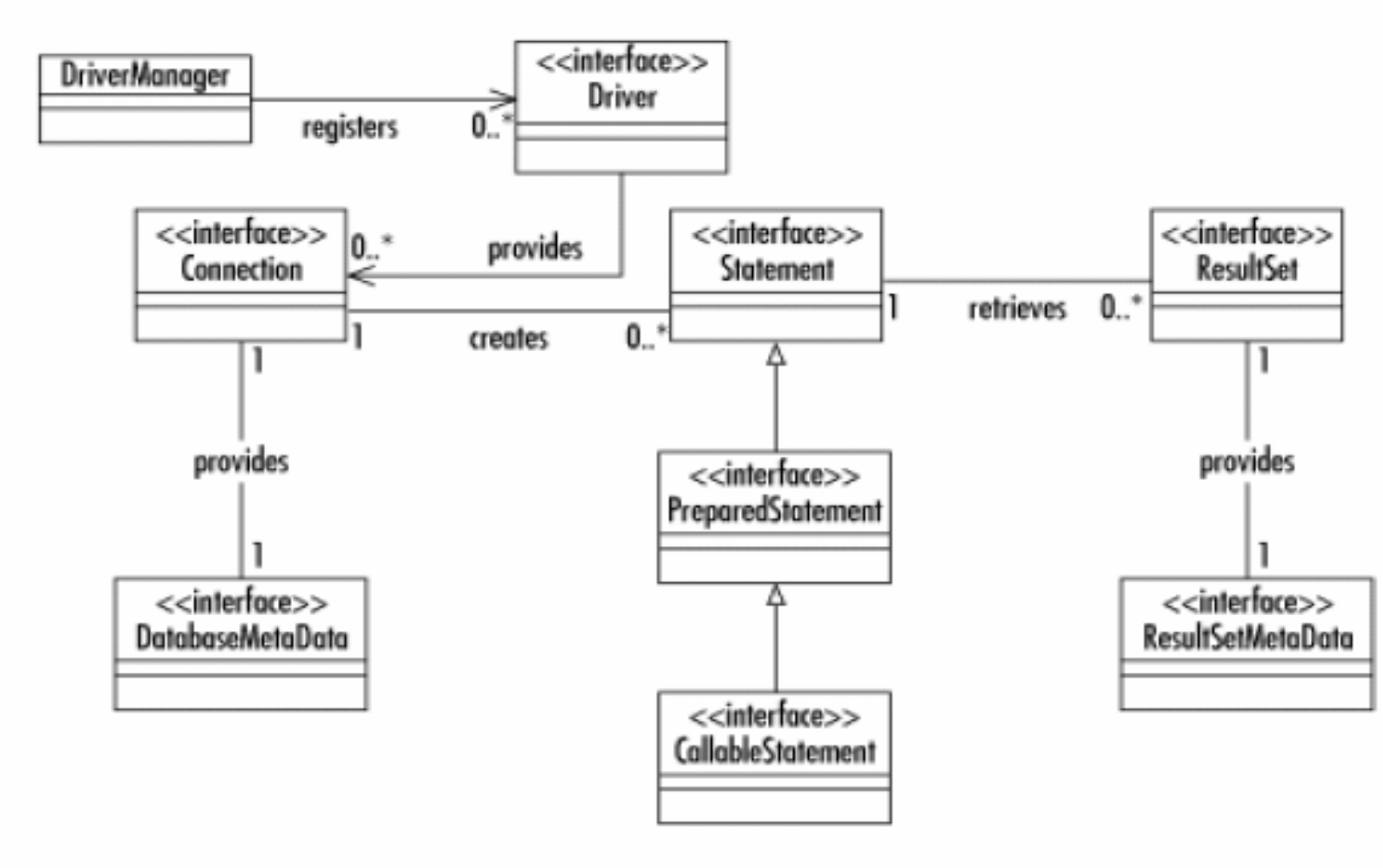
JDBC Driver Type 3



JDBC Drivers

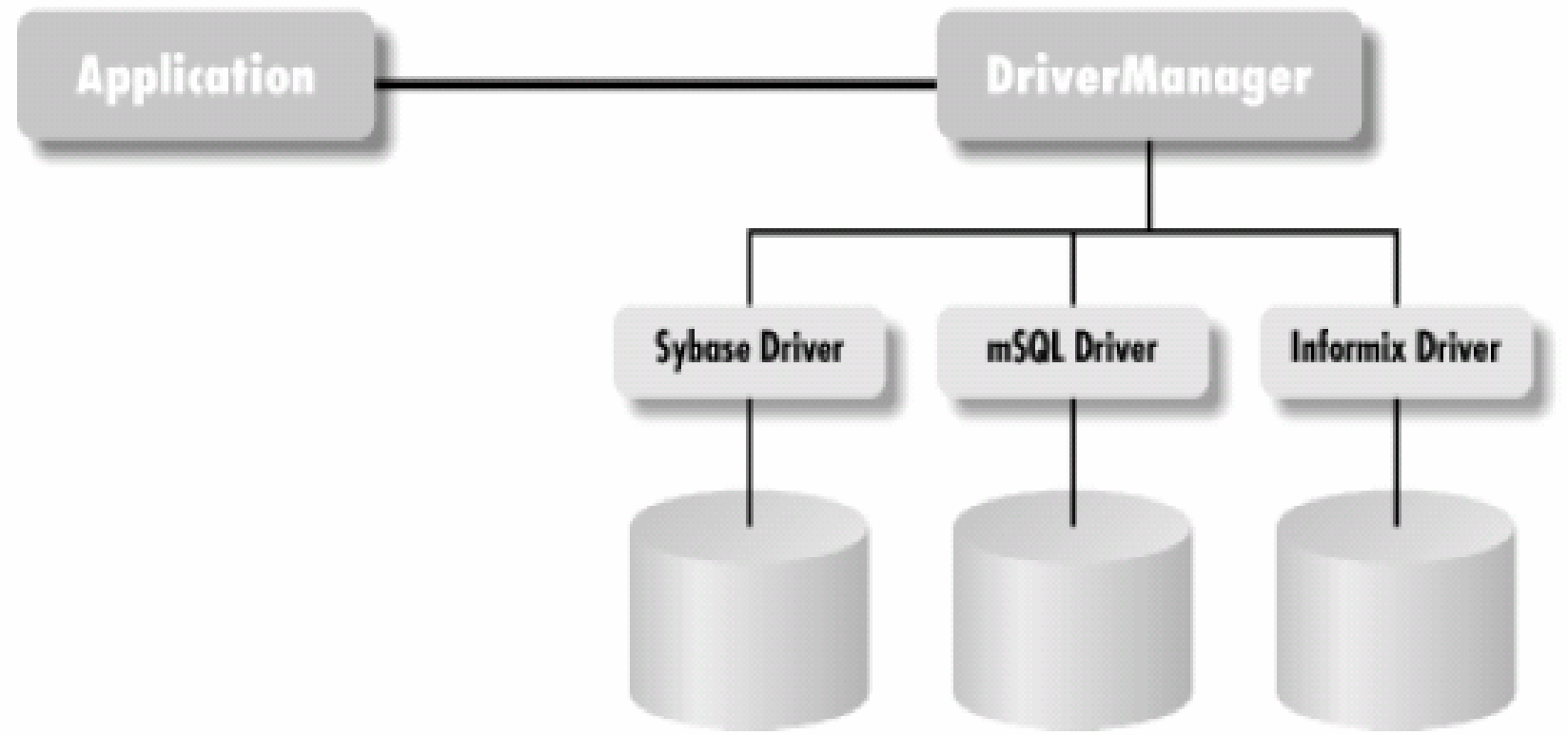


JDBC Class Diagram



Connecting to Database

JDBC shields an application from the specifics of individual database implementation.



Connection Troubles

The JDBC Connection process is the most difficult part of JDBC to get right.

There are generally two basic connection problems:

1) Connection fails with the message "Class not found“
   Solution: Set your JDBC driver in your `CLASSPATH`

2) Connection fails with the message "Driver not found"
   Solution: register the JDBC driver with the `DriverManager` class

Connection Process 1

When you write a Java database applet or application, the only driver-specific information JDBC requires from you is the database URL.

You can even have your application derive the URL at runtime—based on user input or applet parameters.

What happens when the URL and whatever properties the JDBC driver requires (generally a user ID and password) is passed?

1) the application will first request a `java.sql.Connection` implementation from the `DriverManager`

2) the `DriverManager` in turn will search through all of the known `java.sql.Driver` implementations for the one that connects with the URL you provided

Connection Process 2

3) if it exhausts all the implementations without finding a match, it throws an exception back to the application

4) once a `Driver` recognizes the URL, it creates a database connection using the properties specified

5) it then provides the `DriverManager` with a `java.sql.Connection` implementation representing that database connection

6) the `DriverManager` then passes that `Connection` object back to the application

7) the entire database connection process is handled by these two lines

```
Connection con = null;
con = DriverManager.getConnection(url, uid, password);
```

Connection Process 3

How does the JDBC `DriverManager` learn about a new driver implementation?

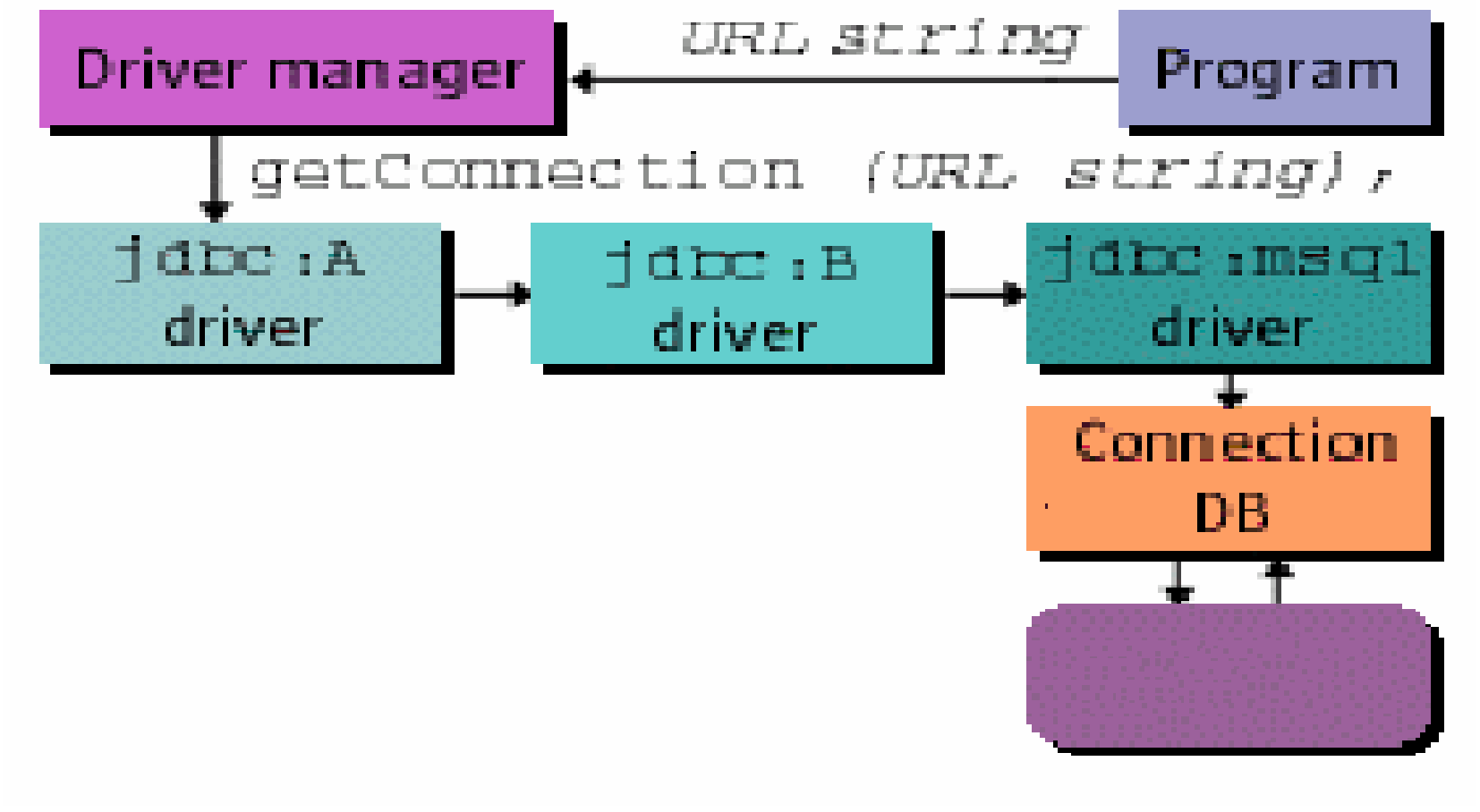
1) the `DriverManager` actually keeps a list of classes that implement the `java.sql.Driver` interface

2) `Driver` implementations has to be registered for any potential database drivers it might require with the `DriverManager`

3) The act of instantiating a `Driver` class thus enters it in the `DriverManager`'s list

4) The process is called Driver Loading

Connection Process 4



Loading JDBC Drivers

There are three basic ways of loading the drivers:

1) explicitly call new to load your driver's implementation of `Driver`

2) use the `jdbc.drivers` property

```
>java -Djdbc.drivers=jdbc.odbc.JdbcOdbcDriver queryDB
```

3) load the class using `Class.forName`

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Class for Creating a Connection 1

A class and two Interfaces are used for creating a connection to a database:

1) `java.sql.Driver`
   a) unless you are writing your own JDBC implementation, you should never have to deal with this class from your application

   b) a launching point for database connectivity by responding to `DriverManager` connection requests and providing information about the implementation in question

Class for Creating a Connection 2

2) `java.sql.DriverManager`
   a) Its main responsibility is to maintain a list of `Driver` implementations and present an application with one that matches a requested URL.
   b) has two methods `registerDriver()` and `deregisterDriver()`
   c) the methods allow `Driver` implementation to register and unregister itself with the `DriverManager`
   d) You can get an enumeration of registered drivers through the `getDrivers()` method

3) `java.sql.Connection`
   a) The Connection class represents a single logical database connection.

Example: Simple Connection 1

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class SimpleConnection {
 static public void main(String args[]) {
  Connection connection = null;
  // Process the command line
  if( args.length != 4 ) {
   System.out.print("Syntax: java SimpleConnection “);
   System.out.println("DRIVER URL UID PASSWORD");
   return;
  }
```

Example: Simple Connection 2

```
try { // load the driver
 Class.forName(args[0]).newInstance( );
}catch( Exception e ) {
 e.printStackTrace( );
 return;
}
try {
 connection = DriverManager.getConnection(args[1],
                                  args[2], args[3]);
 System.out.println("Connection successful!");
 // Do whatever queries or updates you want here!!!
}catch( SQLException e ) {
 e.printStackTrace( );
}
```

Example: Simple Connection 3

```
`finally {
   if( connection != null ) {
    try { connection.close( );
    }catch( SQLException e ) {
     e.printStackTrace( );
    }
   }
  }
 }
}
```

Lab Work: Creating Connection

1) Open `LicenseApp.java` file stored on the server
2) The program creates an interface for you to enter your license information into the database you created. Study the code and understand what it does.
3) Import appropriate packages into the program. Locate `connectToDB()`
4) Write a code to connect to the database you have created in mySQL using the following parameters

```
url = "jdbc:mysql://localhost/emacao"
username = root, Password = ""
Driver = "com.mysql.jdbc.Driver"
```

Note: set your `classpath` to the jar files provided along with the code. Print out the `Connection` object.

Database Access

The most basic kind of database access involves writing

1) updates- `INSERT`, `UPDATE`, or `DELETE`
2) queries – `SELECT`

With these you know ahead of time the type of statements you are sending to the database.

Database Access Steps

Accessing database involves:

1) creating a `Connection` object

2) generating implementation of `java.sql.Statement` tied to the database

3) use the statement to rollback or commit the statement object associated with that Connection

4) with the `Statement` object you can execute updates and queries

5) The result of executing queries and update is `java.sql.ResultSet`

6) `ResultSet` provides you with access to the data retrieved by a query.

Basic JDBC Classes

JDBC's most fundamental classes are :

1) `java.sql.Connection`
2) `java.sql.Statement`
3) `java.sql.ResultSet`

We have discussed (1), we now consider (2) and (3)

Statement

`Statement` class represents SQL statements.

It has three generic forms of statement execution methods:

1) `executeQuery(String query)`

Usage: for any SQL calls that expect to return data from database

2) `executeUpdate(String query)`

Usage: when SQL calls are not expected to return data from database
It returns the number of row affected by `query`

3) `execute()`

Usage: when you cannot determine whether SQL is an update or query
return `true` if row is returned, use `getResultset()` to get the row
otherwise returns `false`

Submitting a Query 1

Submitting a query involves

1) create a `Statement` object

```
try {
    Statement stmt = con.createStatement ();
} catch (SQLException e) {
    System.out.println (e.getMessage());
}
```

SQL exceptions occur when there is a database access error.

Errors are detected when a connection is broken or the database server goes down.

Submitting a Query 2

2) use the one the statement query method to submit the SQL statement to the database depending on the type of the SQL.

JDBC does not attempt to interpret queries.

Example:

```
ResultSet rs = null;
rs = stmt.execteQuery("select * from license");
```

Example: Statement 1

```
import java.sql.*;
public class Update {
 public static void main(String args[]) {
  Connection connection = null;
  if( args.length != 2 ) {
   System.out.print("Syntax: <java Update [number]”);
   System.out.println(“[string]>");
   return;
  }
  try {
   String driver = "com.mysql.jdbc.Driver";
   Class.forName(driver).newInstance( );
   String url =“jdbc:mysql://localhost/emacao";
   con = DriverManager.getConnection(url, “root", "");
```

Example: Statement 2

```
 Statement s = con.createStatement( );
 String test_id = args[0];
 String test_val = args[1];
 int update_count =
 s.executeUpdate("INSERT INTO test (test_id, test_val)
  " + "VALUES(" + test_id + ", '" + test_val + "')");
 System.out.println(update_count + " rows inserted.");
 s.close( );
}catch( Exception e ) {
 e.printStackTrace( );
}
```

Example: Statement 3

```
  finally {
   if( con != null ) {
    try {
      con.close( );
    }catch( SQLException e ) {
     e.printStackTrace( );
    }
   }
  }
 }
}
```

Lab Work: Statement

1) Using the `LicenseApp.java` program stored on the server insert records into `license` table in `emacao` database

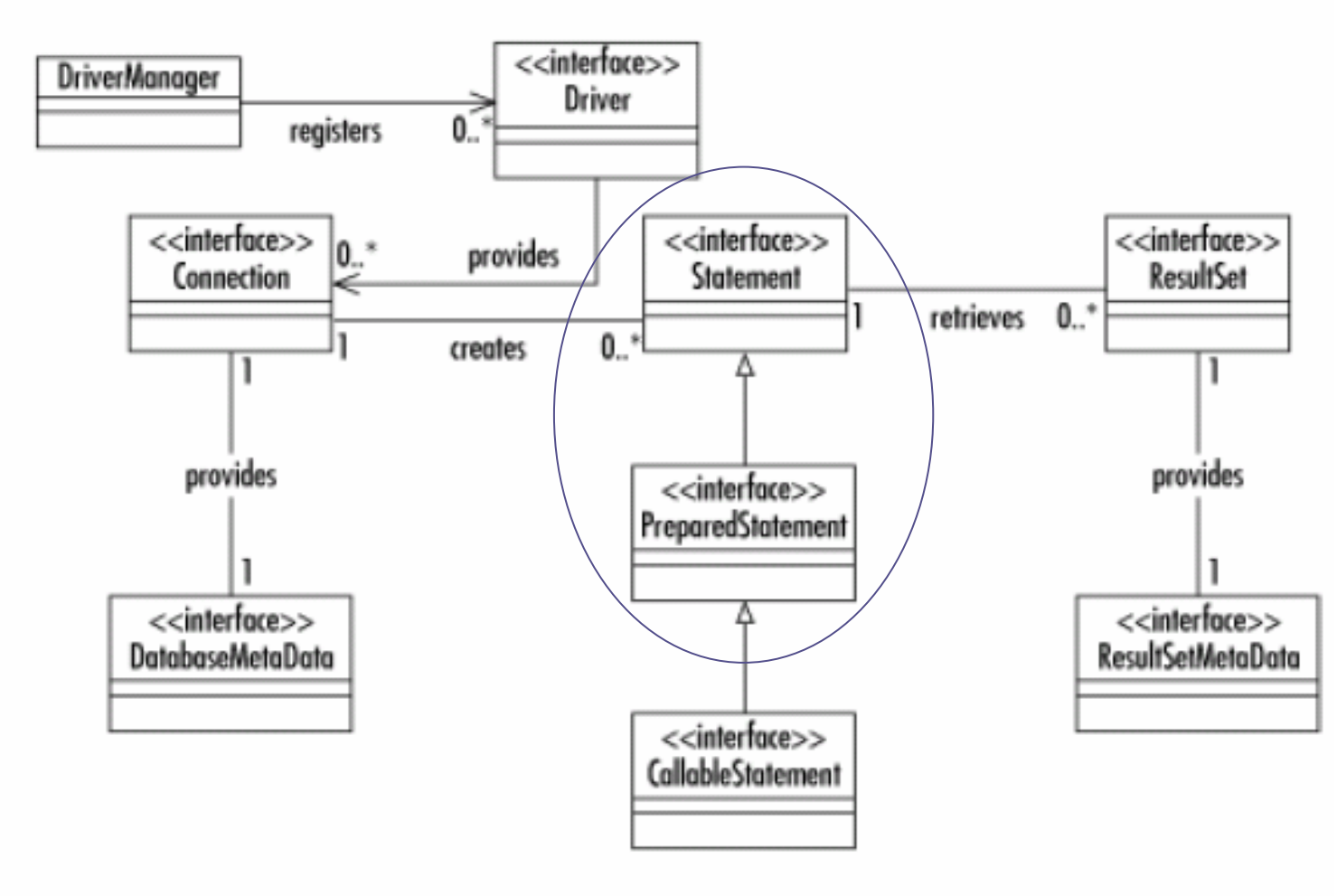
PreparedStatement

Prepared Statement is a precompiled SQL statement.

It is more efficient than calling the same SQL statement over and over.

The `PreparedStatement` class extends the `Statement` class by adding the capability of setting parameters inside of a statement.

PreparedStatement Inheritance



setXXX Methods 1

The `PreparedStatement` class extends the `Statement` class by adding the capability of setting parameters inside of a statement.

The `setXXX` methods are used to set SQL IN parameters values.

Must specify the types that are compatible with the defined SQL input type parameters.

For example, if the IN parameter has SQL type Integer, then you should use the `setInt` method

setXXX Methods 2

Method	SQL Types
setArrayLocator	Locator(<array>)
setASCIIStream	Uses an American Standards Code for Information Exchange (ASCII) stream to produce a LONGVARCHAR
setBigDecimal	NUMERIC
setBinaryStream	Uses a binary stream to produce a LONGVARBINARY
setBlobLocator	LOCATOR(BLOB)
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY or LONGVARBINARY (depending upon the size relative to the limits on VARBINARY)
setCharacterStream	Uses Java.io.Reader to produce a LONGVARCHAR
setClobLocater	LOCATOR(CLOB)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	The given Java technology object ("Javaobject") is converted to the target SQL Type before sent
setShort	SMALLINT
set String	VARCHAR OR LONGVARCHAR (depending upon the size relative to the driver's limits on VARCHAR)
setStructLocator	LOCATOR(<structure_type>)
setTime	TIME
setTimeStamp	TIMESTAMP
setUnicodeStream	Uses a Unicode stream to produce a LONGVARCHAR

Example: PreparedStatement

```
public boolean prepStatement(String name, String sex){
   String query = null;
   PreparedStatement prepStmnt = null;
   query = "update license set name = ?, sex = ? where
                                              id= 126";
   prepStmnt = con.preparedStatement (query);
   prepStmnt.setFloat(1, name);
  prepStmnt.setString(2, sex);
  Int rowsUpdate = prepStmnt.executeUpdate();
  return (rowUpdate > 0);
}
```

CallableStatement

`CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

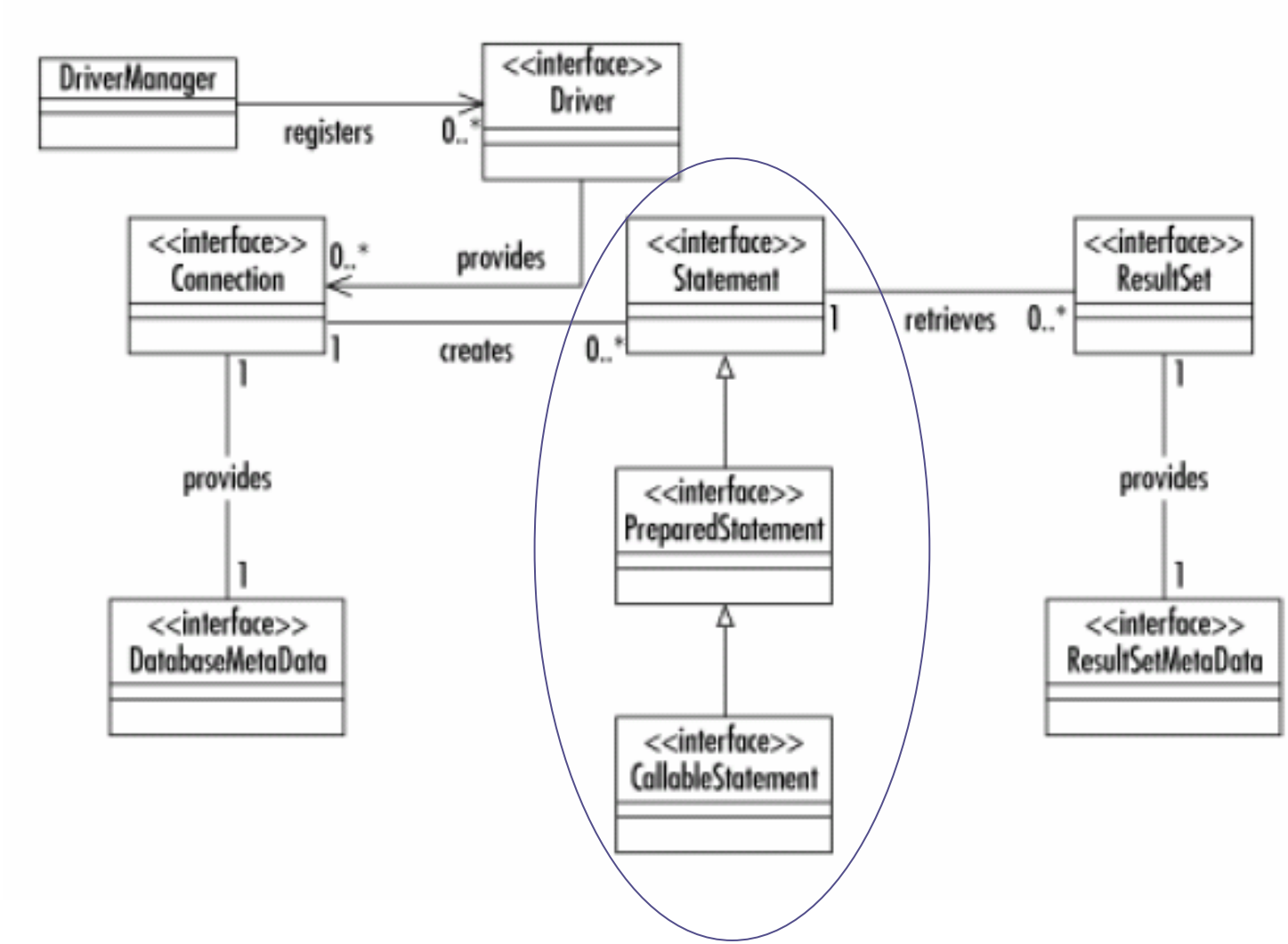
`CallableStatement` class extends the `PreparedStatement` class, which provides the methods for setting IN parameters.

Methods for retrieving multiple results with a stored Procedure are supported with the `Statement.getMoreResults()` method.

Example: CallableStatement

```
 int id= 126;
CallableStatement callStm = null;
String storProcName="{?=call return_license(?)}"
querySales = con.prepareCall(storProcName);
try {
     callStm.registerOutParameter(1, Type.VARCHAR);
     callStm.setInt(2, id);
     callStm.execute();
     String license = callStm.getString(1);
} catch (SQLException e) {
     e.printStackTrace();
}
```

CallableStatement Inheritance



Transaction Management

A transaction is a set of one or more statements that are executed together as a unit.

Either all of the statements are executed, or none of the statements is executed.

There are times when you do not want one statement to take effect unless another one also succeeds.

This is achieved through the `setAutoCommit()` method of `Connection` object.

Transaction Management

A transaction is a set of one or more statements that are executed together as a unit.

Either all of the statements are executed, or none of the statements is executed.

There are times when you do not want one statement to take effect unless another one also succeeds.

This is achieved through the `setAutoCommit()` method of `Connection` object.

The method takes a `boolean` value as a parameter.

Disabling Auto-commit Mode

When a connection is created, it is in auto-commit mode.

Each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode.

Example:

```
con.setAutoCommit(false);
```

Committing a Transaction

Once auto-commit mode is disabled, no SQL statements will be committed until you call the method commit explicitly.

This is achieved through the `commit()` method of connection objects.

All statements executed after the previous call to the `commit()` method will be included in the current transaction and will be committed together as a unit.

If you are trying to execute one or more statements in a transaction and get an `SQLException` , you should call the `rollback()` method to abort the transaction and start the transaction all over again.

Example: Transaction Commit

```
con.setAutoCommit(false);
PreparedStatement updateName =null;
String query = null;
Query="UPDATE license SET name = ? WHERE id = 126"
updateName= con.prepareStatement(query);
updateName.setString(1, name);
updateName.executeUpdate();
PreparedStatement updateSex = null;
query = "UPDATE test SET test_value =?”
updateSex = con.prepareStatement(query);
updateSex.setString(1, "Male");
updateSex.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

ResultSet

A `ResultSet` is one or more rows of data returned by a database query.

The class simply provides a series of methods for retrieving columns from the results of a database query

General form:

```
type gettype(int | String)
```

in which the argument represents either the column number or column name desired

can store values in the database as one type and retrieve them as a completely different type

ResultSet getXXX() Methods

Method	Java Type Returned
getArrayLocator	LOCATOR(<array>)
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBlobLocator	LOCATOR(**BLOB**)
getBoolean	boolean
getByte	byte
getBytes	byte[]
getCharacterStream	java.io.Reader
getClobLocator	LOCATOR(**CLOB**)
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getStructLocator	LOCATOR(**<structure-type>**)
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream or Unicode characters

SQL and Java Type Mapping

SQL TYPE	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Getting the Next Record

`ResultSet` class handles only a single row from the database at any given time.

The class provides the `next( )` method for making it reference the next row of a result set.

If `next()` returns `true`, you have another row to process and any subsequent calls you make to the `ResultSet` object will be in reference to that next row.

If there are no rows left, it returns `false`.

Example: ResultSet

```
String query = “select * from license”;
Statement stm = null;
stm = con.getStatement();
ResultSet rs = stm.executeQuery(query);
while(rs.next( )) {
    int a;
    String str;
    a = rs.getInt("id");
    if( rs.wasNull( ) ) {
      a = -1;
    }
    str = rs.getString(“name");
    if( rs.wasNull( ) ) {
     str = null;
    }
}
```

SQL Null Versus Java null

SQL and Java have a serious mismatch in handling null values.

Java `ResultSet` has no way of representing a SQL NULL value for any numeric SQL column.

After retrieving a value from a `ResultSet`, it is therefore necessary to ask the `ResultSet` if the retrieved value represents a SQL NULL.

To avoid running into database oddities, however, it is recommended that you always check for SQL NULL.

Checking for SQL NULL involves a single call to the `wasNull( )` method in your `ResultSet` after you retrieve a value.

Example: wasNull()

```
rs.afterLast( );
while(rs.previous( )) {
    int a;
    String str;
    a = rs.getInt("test_id");
    if( rs.wasNull( ) ) {
      a = -1;
    }
    str = rs.getString("test_val");
    if( rs.wasNull( ) ) {
     str = null;
    }
}
```

Scrollable ResultSet 1

The single most visible addition to the JDBC API in its 2.0 specification is support for scrollable result sets.

Using scrollable result sets starts with the way in which you create statements.

The `Connection` class actually has two versions of `createStatement()`

1) the zero parameter version

Example:

```
Statement stm = con.createStatement();
```

Scrollable ResultSet 2

2) a two parameter version that supports the creation of `Statement` instances that generate scrollable `ResultSet` objects.

```
createStatement(int rsType,int rSConcurrency)
```

Parameters:

`rsType` - a result set type; one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

`rsConcurrency` - a concurrency type; one of `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

ResultSet Constants

JDBC defines three types of result sets:

1) TYPE_FORWARD_ONLY
2) TYPE_SCROLL_SENSITIVE
3) TYPE_SCROLL_INSENSITIVE

Out of these three `TYPE_FORWARD_ONLY` is the only type that is not scrollable.

The other two types are distinguished by how they reflect changes made to them.

`TYPE_SCROLL_INSENSITIVE` `ResultSet` is unaware of in-place edits made to modifiable instances.

`TYPE_SCROLL_SENSITIVE`, on the other hand, means that you can see changes made to the results if you scroll back to the modified row at a later time.

Result Set Navigation 1

When `ResultSet` is first created, it is considered to be positioned before the first row.

Positioning methods such as `next()` point a `ResultSet` to actual rows.

Your first call to `next( )`, for example, positions the cursor on the first row.

Subsequent calls to `next()` move the `ResultSet` ahead one row at a time.

With a scrollable ResultSet, however, a call to `next()` is not the only way to position a result set.

Result Set Navigation 2

The method `previous( )` works in an almost identical fashion to next( ).

While `next()` moves one row forward, `previous()` moves one row backward.

If it moves back beyond the first row, it returns `false`. Otherwise, it returns `true`.

Because a `ResultSet` is initially positioned before the first row, you need to move the `ResultSet` using some other method before you can call `previous()`.

Example: Result Set Navigation 1

```
import java.sql.*;
import java.util.*;
public class ReverseSelect {
 public static void main(String argv[]) {
  Connection con = null;
  try {
   String url = "jdbc:mysql://localhost/emacao";
   String driver = "com.mysql.jdbc.Driver";
   Statement stmt;
   ResultSet rs;
   Class.forName(driver).newInstance( );
   con = DriverManager.getConnection(url, "root", "");
   stmt =con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                  ResultSet.CONCUR_READ_ONLY);
   rs = stmt.executeQuery("SELECT * from license ORDER BY id");

   System.out.println("Got results:");
```

Example: Result Set Navigation 2

```
 rs.afterLast( );
 while(rs.previous( )) {
  int a;
  String str;
  a = rs.getInt("id");
  a = rs.wasNull( ) ? -1 : a;
  str = rs.getString("name");
  str = rs.wasNull( ) ? null : str;
  System.out.print("\tid= " + a);
  System.out.println("/str= '" + str + "'");
 }
 System.out.println("Done.");
}catch( Exception e ) {
 e.printStackTrace( );
}
```

Example: Result Set Navigation 3

```
 finally {
   if( con != null ) {
    try {
     con.close( );
    }catch( SQLException e ) {
     e.printStackTrace( );
    }
   }
  }
 }
}
```

Other Navigation Methods

JDBC 2.0 provides new methods to navigate around rows in result sets:

1) `beforeFirst( )`
2) `first( )`
3) `last( )`
4) `isBeforeFirst( )`
5) `isFirst( )`
6) `isLast( )`
7) `isAfterLast()`
8) `getRow( )`
9) `relative( )`
10) `absolute( )`

Except for `absolute()` and `relative()`, the names of the methods say exactly what they do. Each take integer arguments.

absolute() 1

For `absolute( )`, the argument specifies a row to navigate to.

Example:
A call to `absolute(5)` moves the ResultSet to row 5 unless there are four or fewer rows in the `ResultSet`.

A call to `absolute()` with a row number beyond the last row is therefore identical to a call to `afterLast()`

absolute() 2

You can also pass negative numbers to `absolute()`.

A negative number specifies absolute navigation backwards from the last row

Example:
`absolute(1)` is identical to `first()`, `absolute(-1)` is identical to `last()`

Similarly, `absolute(-3)` is the third to last row in the `ResultSet`. If there are fewer than three rows in the `ResultSet`.

relative()

The `relative()` method handles relative navigation through a `ResultSet`.

In other words, it tells the `ResultSet` how many rows to move forward or backward.

Example:

A value of 1 behaves just like `next()` and a value of -1 just like `previous()`.

Clean Up

The `Connection`, `Statement`, and `ResultSet` classes all have `close( )`.

It is always a good idea to close any instance of these objects when you are done with them.

It is useful to remember that closing a `Connection` implicitly closes all `Statement` instances associated with the Connection.

Similarly, closing a `Statement` implicitly closes `ResultSet` instances associated with it.

Example: Clean Up

```
try{
   // Connection, Statements here
}catch(SQLException ex){
   ex.printStarkTrace();
}finally {
    if( con != null ) {
     try {
       con.close( );
     }catch( SQLException e ) {
      e.printStackTrace( );
     }
    }
   }
}
```

Lab Work: ResultSet

1) Using the `LicenseApp.java`, before you save, check if the data you are saving exists, if it is update with the new values else insert a new record.

Exercise: JDBC

Message-Orientation

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

JavaMail

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) message-orientation
    a) javamail
    b) jms
6) distributed objects
    a) rmi
    b) corba
    c) Javaidl
7) summary

Overview

Email was the Internet's first killer application and still generates more Internet traffic than any protocol except HTTP.

One of the most frequently asked questions about Java is how to send email from a Java applet or application or how to send asynchronous messages between a Java application and homo-sapiens?

We shall be considering:

1) Introduction to JavaMail API
2) Protocols – SMTP, POP, IMAP MIME
3) Installation and configuration
4) Core Classes – Session, Message, Address, Authenticator, Transport, Store and Folder
5) Usage – sending and receiving email, processing HTML messages etc

What Is the JavaMail API?

The JavaMail API is a standard extension to Java that provides a class library for email clients.

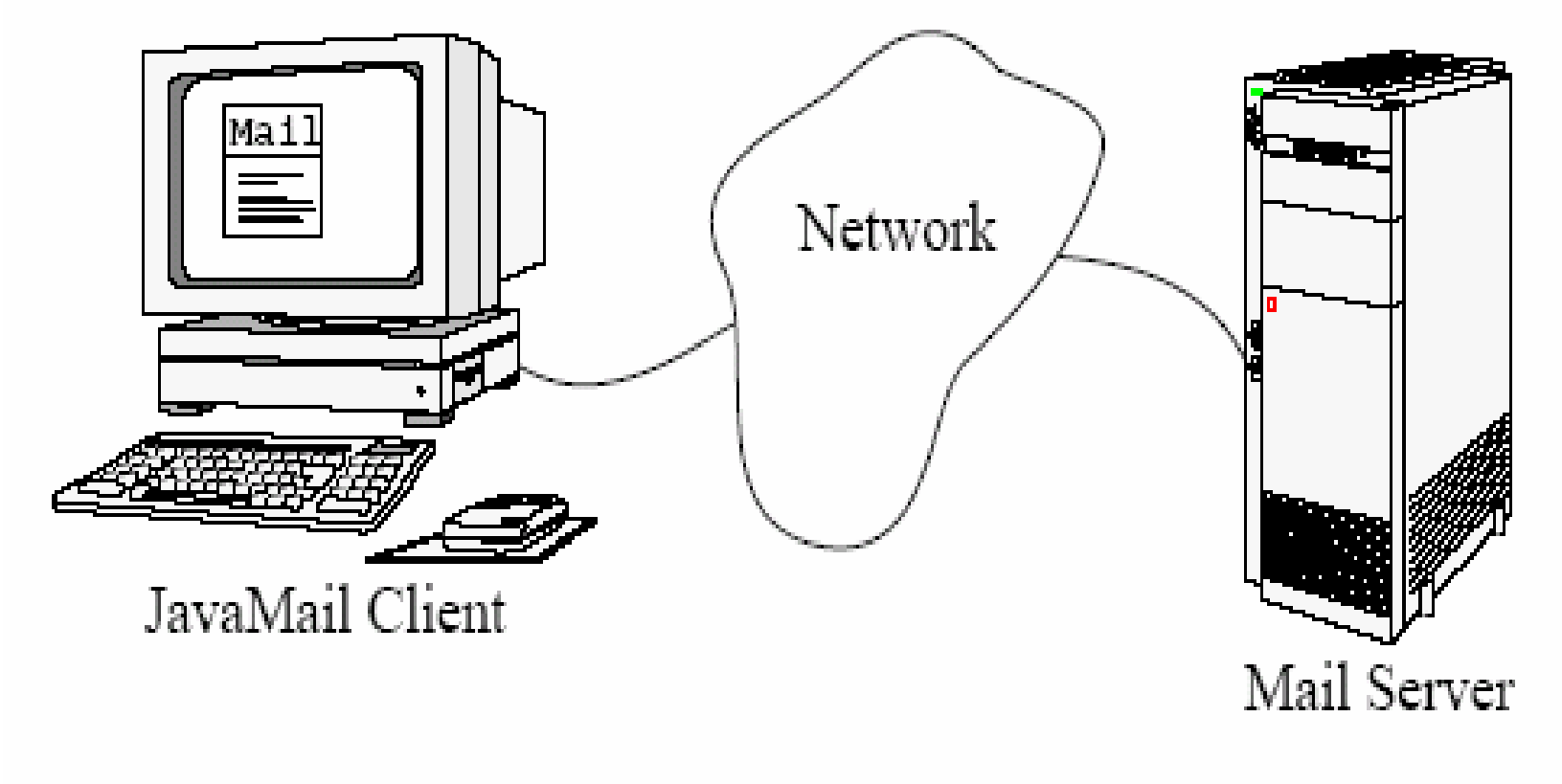
Is an optional package (standard extension) for reading, composing, and sending electronic messages.

You use the package to create Mail User Agent (MUA) type programs, similar to Eudora, Pine, and Microsoft Outlook.

Purpose:

1) transporting
2) delivering and
3) Forwarding messages like sendmail or other Mail Transfer Agents

Mail Client and Server



Why Mail?

There are situation in which an application may need to send an email

1) an error situation occurs

2) when the next step in some workflow must be started

3) or in response to some events that has occurred

JavaMail Applications

There are several areas in which JavaMail is useful.

Some are discussed below:

1) A server-monitoring application such as Whistle Blower can periodically load pages from a web server running on a different host and email the webmaster if the web server has crashed.

2) An applet can use email to send data to any process or person on the Internet that has an email address, in essence using the web server's SMTP server as a simple proxy to bypass the usual security restrictions about whom an applet is allowed to talk to. In reverse, an applet can talk to an IMAP server on the applet host to receive data from many hosts around the Net.

3) A newsreader could be implemented as a custom service provider that treats NNTP as just one more means of exchanging messages.

Related Protocols 1

There are four protocols are commonly used with the API:

1) Simple Mail Transfer Protocol (SMTP)

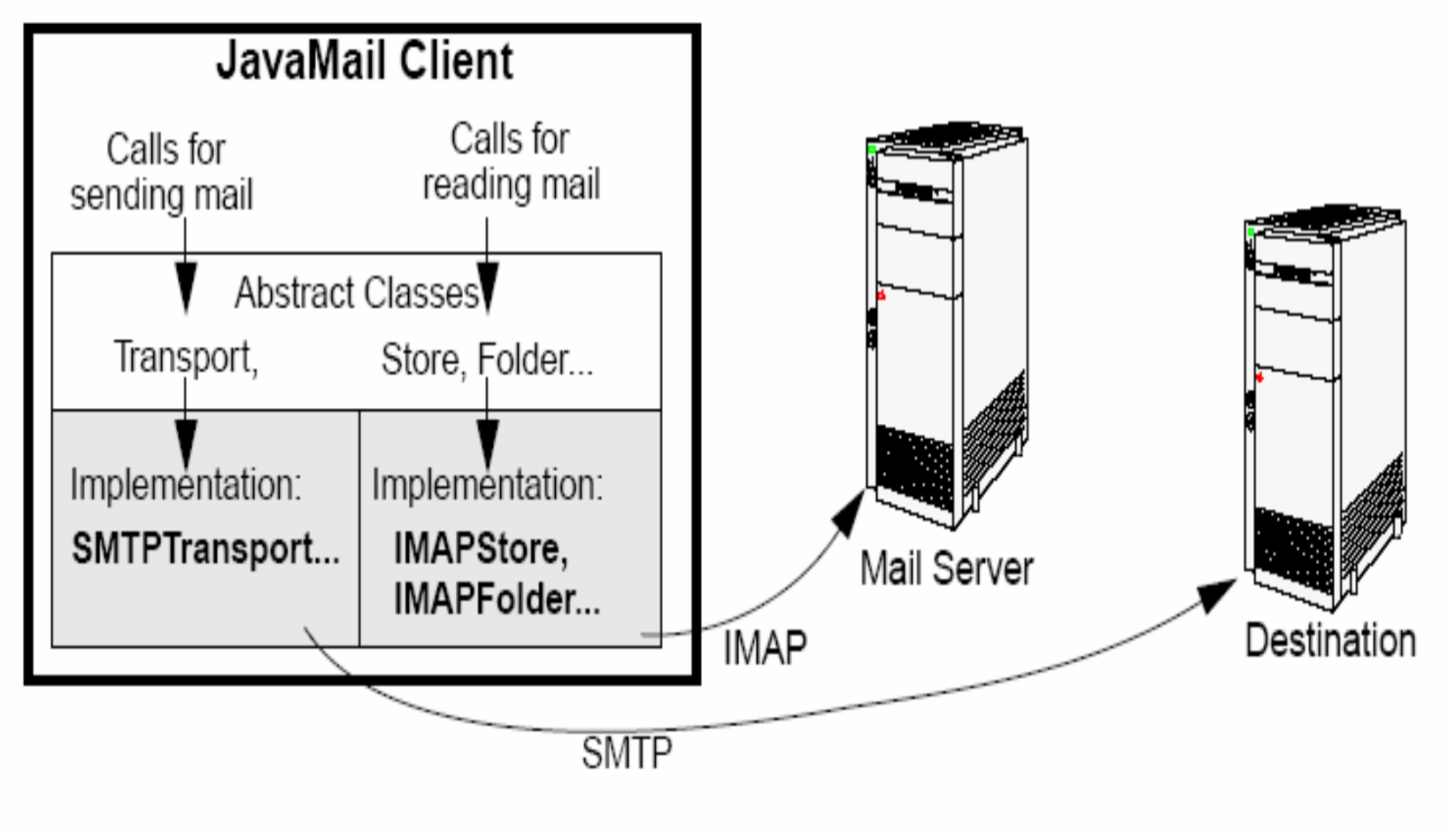
2) Post Office Protocol (POP)

3) Internet Message Access Protocol (IMAP)

4) Multipurpose Internet Mail Extensions (MIME)

Each will be considered.

Related Protocols 2



SMTP

The Simple Mail Transfer Protocol (SMTP) is the mechanism for delivery of email.

In the context of JavaMail,

- JavaMail-based program will communicate with company or Internet Service Provider's (ISP's) SMTP server.

2) The SMTP server will relay the message on to the SMTP server of the recipient to be acquired eventually by the user through POP or IMAP

POP

Post Office Protocol (POP) is the mechanism most people on the Internet use to get their mail.

It defines support for a single mailbox for each user.

Currently in version 3, also known as POP3

The ability to see how many new mail messages you have, are not supported by POP at all.

These capabilities are built into programs like Eudora or Microsoft Outlook, which remember things like the last mail received and calculate how many are new for you. So, when using the JavaMail API, if you want this type of information, you have to calculate it yourself.

IMAP

Internet Message Access Protocol (IMAP) more advanced protocol for receiving messages.

Currently in version 4, also known as IMAP4

Your mail server must support the protocol before you can use it.

You can't just change your program to use IMAP instead of POP and expect everything in IMAP to be supported.

Assuming your mail server supports IMAP, your JavaMail-based program can take advantage of users having multiple folders on the server and these folders can be shared by multiple users.

IMAP Drawbacks

It places a much heavier burden on the mail server requiring the server to receive the new messages, deliver them to users when requested, *and* maintain them in multiple folders for each user.

While this does centralize backups, as users' long-term mail folders get larger and larger, everyone suffers when disk space is exhausted.

But with POP, saved messages get offloaded from the mail.

MIME

MIME stands for Multipurpose Internet Mail Extensions

It is not a mail transfer protocol.

Instead, it defines the content of what is transferred.

For example:

1) format of the messages
2) attachments, and
3) etc

Installation

There are three versions of the JavaMail API commonly used today:

1) version 1.1.3
2) version 1.2
3) version 1.3.2

Version 1.3.2 is the latest.

The version of the JavaMail API you want to use affects what you download and install.

Installing JavaMail 1.3.2

1) Download `javamail-1_3_2.zip` from
   `http://java.sun.com/products/javamail`
2) Extract the zip file into a folder
3) set it in the `CLASSPATH` environment variable
4) Include the following archive files in the `CLASSPATH`
   a) `imap.jar`
   b) `mailapi.jar`
   c) `pop3.jar`
   d) `smtp.jar`

JavaMail needs a framework in order to complete its functions.

This framework is known as JavaBeans Activation Framework (JAF).

JAF

JavaBeans Activation Framework (JAF) is a standard extension that enables developers who use Java technology to take advantage of standard services:

1) to determine the type of an arbitrary piece of data,

2) encapsulate access to it,

3) discover the operations available on it,

4) and to instantiate the appropriate bean to perform the said operation(s).

It is the basic MIME-type support found in many browsers and mail tools.

Example: JAF

If a browser obtained a JPEG image JAF:

1) enables the browser to identify that stream of data as a JPEG image

2) and from that type, the browser could locate and instantiate an object that could manipulate, or view that image

3) discover the operations available on it,

4) and to instantiate the appropriate bean to perform the said operation(s).

Installing JAF

1) Download `jaf-1_0_2-upd.zip` from
`http://java.sun.com/products/javabeans/glasgow/jaf.html`

2) extract the zip file into a folder

3) set it in the `CLASSPATH` environment variable

4) include `activation.jar` in the `CLASSPATH`

Installing JavaMail Using J2EE

JavaMail is bundled with J2EE

There is nothing special you have to do to use the basic JavaMail API.

Just make sure the `j2ee.jar` file is in your `CLASSPATH` and you are set.

Note: This will be deferred to J2EE courses!

Other Referencing Options

If you don't want to change the `CLASSPATH` environment variable:

1) copy the JAR files to your lib/ext directory under the Java Runtime environment (JRE) directory

2) for instance, `%JAVA_HOME%\lib\ext` on a Windows platform

Exercise

1) Download the latest version of the JavaMail API implementation.
2) Download the latest version of the JavaBeans Activation Framework.
3) Extract the zip files to a folder
4) Install the archive files.

Core Classes

There are seven core classes that make JavaMail API:

1) `Session`
2) `Message`
3) `Address`
4) `Authenticator`
5) `Transport`
6) `Store`
7) `Folder`

Each will be considered.

Session

It defines a basic mail session.

It is through this session that everything else works.

The `Session` object takes advantage of a `java.util.Properties` object to get information like mail server, username, password, and other information that can be shared across your entire application.

`Session` class is singleton

Session: Singleton 1

The constructors for the class are private.

An instance of the class can be created in four ways by calling the following methods of the class:

1) `getDefaultInstance(Properties props)`

2) `getDefaultInstance(Properties props,`
`Authenticator authenticator)`

3) `getInstance(Properties props)`

4) `getInstance(Properties props,`
`Authenticator authenticator)`

Session: Singleton 2

Each method returns either a default or new `Session` object.

The (1) and (2) methods get the default instance if one exists and if not a new session object is created.

The (3) and (4) create a new instance.

`props` is the `Properties` object that holds relevant properties

`authenticator` is `Authenticator` object used to call back to the application when a user name and password is needed.

Session: Usage

1) Get a default instance

```
Properties props = new Properties();
// fill props with any information
Session session = Session.getDefaultInstance(props,
                                             null);
```

2) Create a unique session

```
Properties props = new Properties();
// fill props with any information
Session session = Session.getInstance(props, null);
```

In both cases here the `null` argument is an `Authenticator` object.

Message 1

This class models an email message. It is an abstract class.

Subclasses provide actual implementations.

Characteristics:

- Message implements the `Part` interface.

- Direct subclass is `MimeMessage`

2) Message contains a set of attributes and a "content".

3) Messages within a folder also have a set of flags that describe its state within the folder.

Message 2

Message defines some new attributes in addition to those defined in the Part interface.

These attributes specify meta-data for the message - i.e., addressing and descriptive information about the message.

Message objects are obtained either from a Folder or by constructing a new Message object of the appropriate subclass.

Messages that have been received are normally retrieved from a folder named "INBOX".

Message is an abstract class, you cannot work with it. Use the subclasses.

MimeMessage

`MimeMessage` is the direct subclass of `Message`

It is an email message that understands MIME types and headers.

Message headers are restricted to US-ASCII characters only, though non-ASCII characters can be encoded in certain header fields.

Once you have your `Session` object, then you can create the message to send.

Creating a Message

1) pass along the `Session` object to the `MimeMessage` constructor.

```
MimeMessage message = new MimeMessage(session);
```

2) set its parts, as `Message` implements the `Part` interface (with `MimeMessage` implementing `MimePart`).

```
message.setContent("Hello", "text/plain");
```

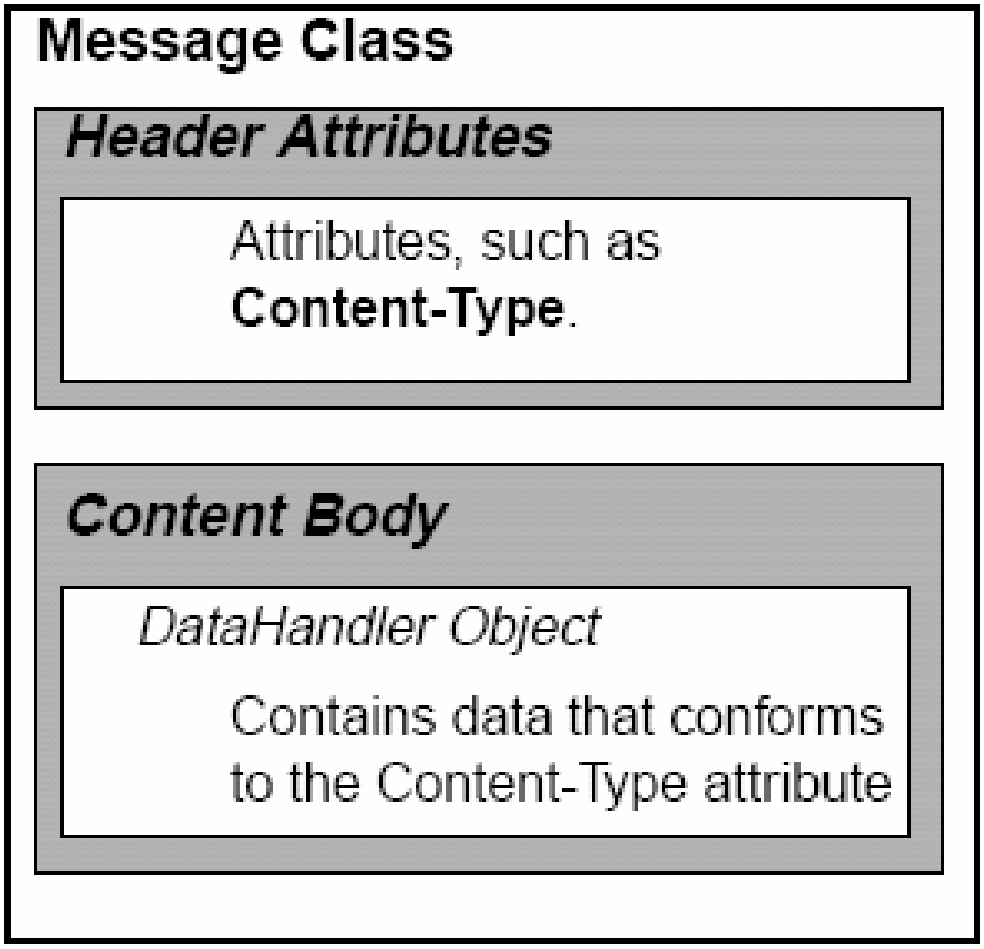
3) If, however, you know you are working with a `MimeMessage` and your message is plain text, then use `setText()` method

```
message.setText("Hello");
```

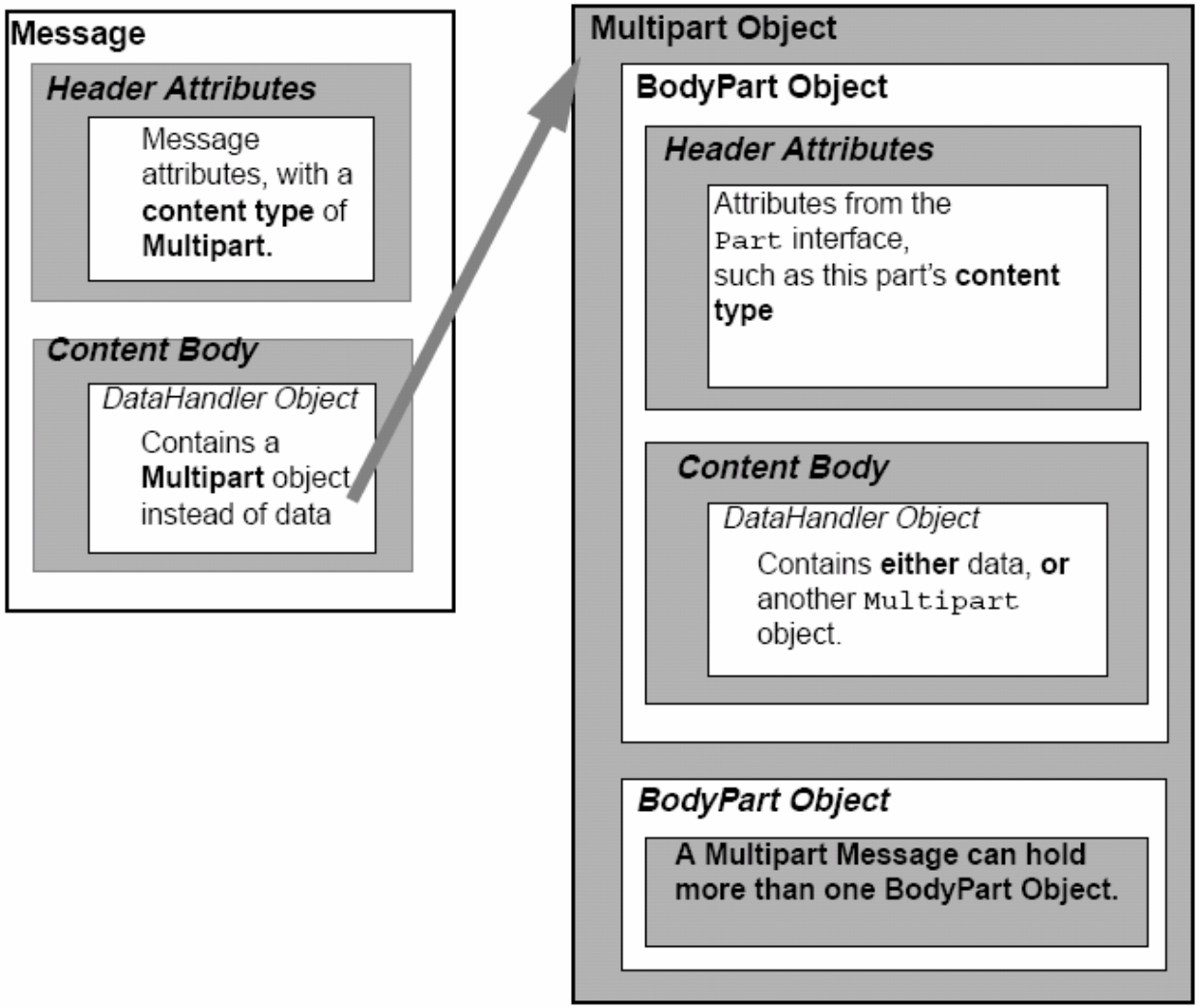
4) set the subject using the `setSubject()` method

```
message.setSubject("First");
```

Simple Message



Multipart Message



Address

Once you've created the `Session` and the `Message`, as well as filled the message with content, it is time to address your letter with an `Address`.

This is done using `Address Class`.

Characteristics:

1) like `Message`, `Address` is an abstract class, hence use the subclass

2) you use the `javax.mail.internet.InternetAddress` class

Creating an Address 1

1) To create an address with just the email address, pass the email address to the constructor

```
Address address = new InternetAddress("xx@server.com");
```

2) If you want a name to appear next to the email address

```
Address address = new InternetAddress(xx@server.com,
                                      "Mr. Gabriel");
```

Creating an Address 2

Once you have created the Addresses you connect them to a message in one of two ways:

1) For identifying the sender, you use the `setFrom()` and `setReplyTo()` methods.

```
message.setFrom(address);
```

or If your message needs to show multiple from addresses, use the `addFrom()` method

```
Address address[] = ...;
message.addFrom(address);
```

Creating an Address 3

2) For identifying the message recipients, you use the `addRecipient()` method.

This requires a `Message.RecipientType` besides the address.

The three predefined types of address are:

a) `Message.RecipientType.TO`
b) `Message.RecipientType.CC`
c) `Message.RecipientType.BCC`

Creating an Address 4

```
…

Address toAddress = new
                InternetAddress("president@server.com");

Address ccAddress = new
                InternetAddress("first.lady@server.com");

message.addRecipient(Message.RecipientType.TO,
                                          toAddress);
message.addRecipient(Message.RecipientType.CC,
                                          ccAddress);

…
```

Authenticator

`Authenticator` Class provide access to protected resources (mail server) via a username and password

To use the `Authenticator`, you subclass the abstract class and return a `PasswordAuthentication` instance from the `getPasswordAuthentication()` method.

Example:

```
Properties props = new Properties();
// fill props with any information
Authenticator auth = new MyAuthenticator();
Session session = Session.getDefaultInstance(props,
                                             auth);
```

Transport 1

The final part of sending a message is to use the `Transport` class.

This class speaks the protocol-specific language for sending the message (usually SMTP).

It's an abstract class and works something like `Session`.

There are two ways of sending a message:

1) You can use the default version of the class by just calling the static `send()` method:

```
Transport.send(message);
```

Transport 2

2) You can get a specific instance from the session for your protocol, pass along the username and password (blank if unnecessary), send the message, and close the connection:

```
message.saveChanges(); // implicit with send()
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message,message.getAllRecipients(
                                                    ));
transport.close();
```

Transport 3

This latter way is better when you need to send multiple messages.

It will keep the connection with the mail server active between messages.

The basic `send()` mechanism makes a separate connection to the server for each method call.

Store and Folder 1

Getting messages starts similarly to sending messages:

1) Get a Session Object

2) You connect to a `Store`, quite possibly with a username and password or Authenticator.

3) Like Transport, you tell the `Store` what protocol to use

```
//Store store = session.getStore("imap");
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

Store and Folder 2

4) Get a `Folder`, which must be opened before you can read messages from it:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
```

5) Get its content with `getContent()` or write its content to a stream with `writeTo()`. The `getContent()` method only gets the message content, while `writeTo()` output includes headers.

```
System.out.println(((MimeMessage)message).getContent());
```

Store and Folder 3

6) Once you're done reading mail, close the connection to the folder and store.

```
folder.close(aBoolean);
store.close();
```

The boolean passed to the `close()` method of folder states whether or not to update the folder by removing deleted messages.

Using JavaMail API

We are going to demonstrate the usage of the API with the following:

1) sending messages
2) fetching messages
3) deleting Messages and Flags
4) authenticating Yourself
5) replying to Messages
6) forwarding Messages
7) working with attachments – sending and getting
8) processing HTML Messages – sending and including images

Sending Messages

This involves three steps:

1) getting a session
2) creating and filling a message
3) Send the message using the static `Transport.send()` method

You can specify your SMTP server by setting the `mail.smtp.host` property for the `Properties` object passed when getting the Session

Example: Sending Messages 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
…
String host = pop3.iist.unu.edu;
String from = gab@iist.unu.edu;
String to = milton@iist.unu.edu;

// Get system properties
Properties props = System.getProperties();

// Setup mail server
props.put("mail.smtp.host", host);
```

Example: Sending Messages 2

```
// Get session
Session session = Session.getDefaultInstance(props,
                                                null);
// Define message
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                                new InternetAddress(to));
message.setSubject("Hello JavaMail");
message.setText("Welcome to JavaMail");

// Send message
Transport.send(message);
```

Lab Work: Sending Messages

1) Starting with the skeleton code, get the system Properties.
2) Add the name of your SMTP server to the properties for the `mail.smtp.host` key.
3) Get a Session object based on the Properties.
4) Create a MimeMessage from the session.
5) Set the from field of the message.
6) Set the to field of the message.
7) Set the subject of the message.
8) Set the content of the message.
9) Use a Transport to send the message.
10) Compile and run the program, passing your SMTP server, from address, and to address on the command line.

Lab Work : Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class MailExample {
public static void main (String args[]) throws

   Exception {
   String host = args[0];
   String from = args[1];
   String to = args[2];
   // Get system properties
   // Setup mail server
   // Get session
   // Define message
   // Set the from address
```

Lab Work : Skeleton Code 2

```
            // Set the to address
            // Set the subject
            // Set the content
            // Send message
    }

}
```

Fetching Messages

Reading messages involves five steps:

1) getting a session
2) get and connect to an appropriate store for your mailbox
3) open the appropriate folder
4) get your message(s)
5) and close the connection when done.

Example: Fetching Messages 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.Internet.*;
…
String host = ...;
String username = ...;
String password = ...;

// Create empty properties
Properties props = new Properties();

// Get session
Session session = Session.getDefaultInstance(props,
                                             null);
```

Example: Fetching Messages 2

```
// Get the store
Store store = session.getStore("pop3");
store.connect(host, username, password);
// Get folder
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);

// Get directory
Message message[] = folder.getMessages();

for (int i=0, n=message.length; i<n; i++) {
  System.out.print(i + ": " + message[i].getFrom()[0]);
  System.out.println("\t" + message[i].getSubject());
}
```

Example: Fetching Messages 3

```
// Close connection
folder.close(false);
store.close();
```

This code snippet displays the subjects of the messages.

To display the whole message:
1) you can prompt the user after seeing the from and subject fields,

2) and then call the message's `writeTo()` method if they want to see it

Example: Displaying Content 1

```
BufferedReader reader = new BufferedReader (
                    new InputStreamReader(System.in));

// Get directory
Message message[] = folder.getMessages();
for (int i=0, n=message.length; i<n; i++) {
  System.out.print(i + ": " + message[i].getFrom()[0]);
  System.out.println("\t" + message[i].getSubject());

  System.out.print("Do you want to read message? “);
  System.out.println("[YES to read/QUIT to end]");
  String line = reader.readLine();
}
```

Example: Displaying Content 2

```
  if ("YES".equals(line)) {
    message[i].writeTo(System.out);
  } else if ("QUIT".equals(line)) {
    break;
  }
}
```

Lab Work: Fetching Messages 1

1) Starting with the skeleton code, get or create a Properties object.
2) Get a Session object based on the Properties.
3) Get a Store for your email protocol, either pop3 or imap.
4) Connect to your mail host's store with the appropriate username and password.
5) Get the folder you want to read. More than likely, this will be the INBOX.
6) Open the folder read-only.
7) Get a directory of the messages in the folder. Save the message list in an array variable named message.
8) For each message, display the from field and the subject.
9) Display the message content when prompted.

Lab Work: Fetching Messages 2

10) Close the connection to the folder and store.

11) Compile and run the program, passing your mail server, username, and password on the command line. Answer YES to the messages you want to read. Just hit ENTER if you don't. If you want to stop reading your mail before making your way through all the messages, enter QUIT.

Lab Work : Skeleton Code 1

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class GetMessageExample {
  public static void main (String args[]) throws
                        Exception{
   String host = args[0];
   String username = args[1];
   String password = args[2];
   // Create empty properties
   // Get session
```

Lab Work : Skeleton Code 2

```
 // Get the store
 // Connect to store
 // Get folder
 // Open read-only
BufferedReader reader = new BufferedReader ( new

    InputStreamReader(System.in));
// Get directory
for (int i=0, n=message.length; i<n; i++) {
   // Display from field and subject
   System.out.print("Do you want to read message?”);
          System.out.println(“[YES to read/QUIT to
end]");
   String line = reader.readLine();
```

Lab Work : Skeleton Code 3

```
      if ("YES".equals(line)) {
         // Display message content
       } else if ("QUIT".equals(line)) {
           break;
       }
    } // Close connection
  }
}
```

Flags

The `Flags` class represents the set of flags on a `Message`. `Flags` are composed of predefined system flags, and user defined flags.

A System flag is represented by the `Flags.Flag` inner class.

1) `Flags.Flag.ANSWERED`
2) `Flags.Flag.DELETED`
3) `Flags.Flag.DRAFT`
4) `Flags.Flag.FLAGGED`
5) `Flags.Flag.RECENT`
6) `Flags.Flag.SEEN`
7) `Flags.Flag.USER`

Use the `getPermanentFlags()` method of `Folder` class to find out what flags are supported

A User defined flag is represented as a `String`.

Deleting Messages

To delete messages, you set the message's DELETED flag:

```
message.setFlag(Flags.Flag.DELETED, true);
```

Open up the folder in READ_WRITE mode first though:

```
folder.open(Folder.READ_WRITE);
```

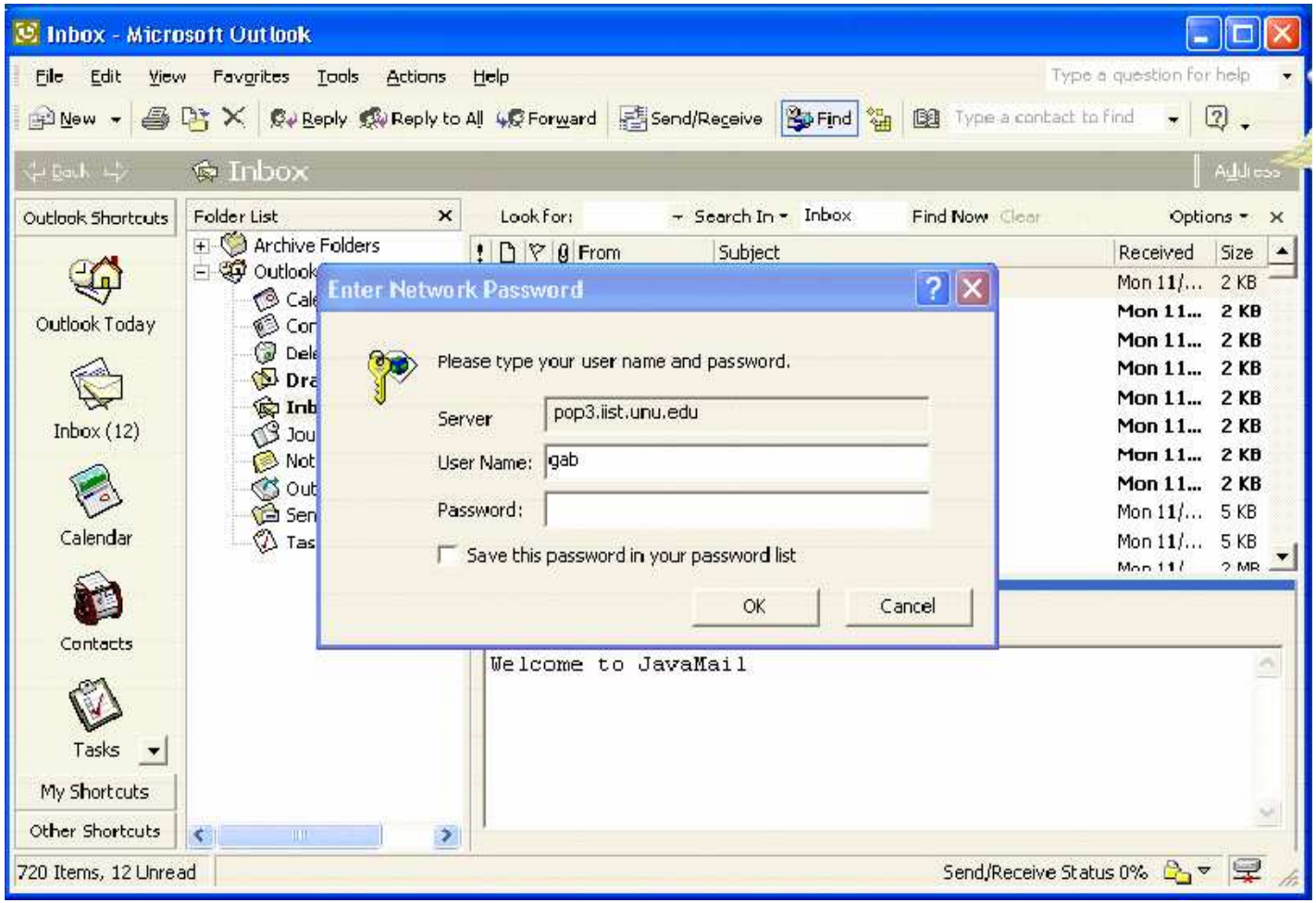
Then, when you are done processing all messages, close the folder, passing in a `true` value to expunge the deleted messages.

To unset a flag, just pass `false` to the `setFlag()` method.

To see if a flag is set, check with `isSet()`.

Authentication 1

How do you achieve something like this using JavaMail?



Authentication 2

Use an `Authenticator` to prompt for username and password when needed.

Instead of connecting to the `Store` with the host, username, and password, you configure the `Properties` to have the host, and tell the `Session` about your custom `Authenticator` instance.

Example:

```
Properties props = System.getProperties();
props.put("mail.pop3.host", host);

// Setup authentication, get session
Authenticator auth = new PopupAuthenticator();
Session session = Session.getDefaultInstance(props,
                                             auth);
```

Authentication 3

```
// Get the store
Store store = session.getStore("pop3");
store.connect();
```

PopupAuthenticator 1

```
import javax.mail.*;
import javax.swing.*;
import java.util.*;

public class PopupAuthenticator extends Authenticator {

  public PasswordAuthentication
                        getPasswordAuthentication(){
    String username, password;
    String result = JOptionPane.showInputDialog("Enter
                              'username,password'");
    StringTokenizer st = new StringTokenizer(result,
                                              ",");
```

PopupAuthenticator 2

```
        username = st.nextToken();
        password = st.nextToken();
        return new PasswordAuthentication(username,
                                          password);
    }
}
```

Replying to Messages

The `Message` class includes a `reply()` method to configure a new message with the proper recipient and subject, adding "Re: " if not already there.

This does not add any content to the message, only copying the from or reply-to header to the new recipient.

The method takes a `boolean` parameter indicating whether to reply to only the sender (`false`) or reply to all (`true`).

Example:

```
MimeMessage reply = (MimeMessage)message.reply(false);
reply.setFrom(new InternetAddress("xxx@server.com"));
reply.setText("Thanks");
Transport.send(reply);
```

Lab Work: Replying to Messages

1) The skeleton code already includes the code to get the list of messages from the folder and prompt you to create a reply.
2) When answered affirmatively, create a new MimeMessage from the original message.
3) Set the from field to your email address.
4) Create the text for the reply. Include a canned message to start. When the original message is plain text, add each line of the original message, prefix each line with the "> " characters.
5) Set the message's content, once the message content is fully determined. Send the message.
6) Compile and run the program, passing your mail server, SMTP server, username, password, and from address on the command line. Answer YES to the messages you want to send replies. Just hit ENTER if you don't. If you want to stop going through your mail before making your way through all the messages, enter QUIT.

Lab Work: Skeleton Code 1

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class ReplyExample {
   public static void main (String args[]) throws
                                              Exception {
      String host = args[0];
   String sendHost = args[1];
      String username = args[2];
      String password = args[3];
      String from = args[4];
```

Lab Work: Skeleton Code 2

```
// Create empty properties
Properties props = System.getProperties();
props.put("mail.smtp.host", sendHost);
// Get session
Session session = Session.getDefaultInstance
                                (props, null);
// Get the store
Store store = session.getStore("pop3");
store.connect(host, username, password);
// Get folder
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
```

Lab Work: Skeleton Code 3

```
BufferedReader reader = new BufferedReader
      ( new InputStreamReader(System.in));
// Get directory
Message message[] = folder.getMessages();
for (int i=0, n=message.length; i<n; i++) {
      System.out.println(i + ": ") +
            message[i].getFrom()[0] + "\t" +
                  message[i].getSubject());
      System.out.println("Do you want to reply to
            the message? [YES to reply/QUIT to
                                    end]");
      String line = reader.readLine();
```

Lab Work: Skeleton Code 4

```
            if ("YES".equals(line)) {
                  // Create a reply message
                  // Set the from field
                  // Create the reply content, copying
                  //over the original if text
                  // Set the content
                  // Send the message
            }else if ("QUIT".equals(line)) {
                  break;
            }
      }
```

Lab Work: Skeleton Code 5

```
        // Close connection
        folder.close(false);
        store.close();
    }
}
```

Message Parts

A mail message can be made up of multiple parts

Each part is a `BodyPart`, or more specifically, a `MimeBodyPart` when working with MIME messages.

The different body parts get combined into a container called `Multipart` or, again, more specifically a `MimeMultipart`.

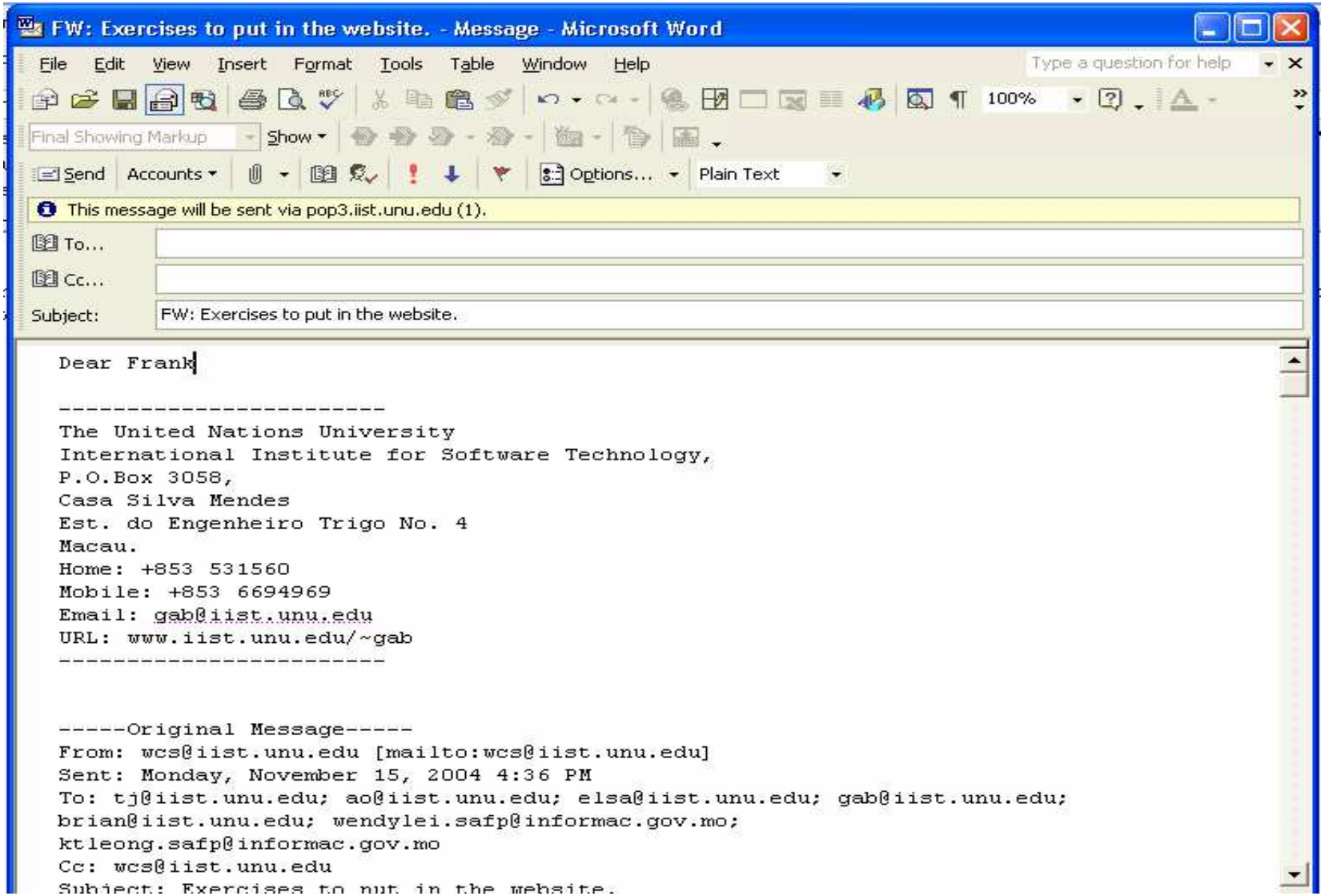
Forwarding Message 1

To forward a message:

1) you create one part for the text of your message
2) and a second part with the message to forward,
3) and combine the two into a multipart.
4) Then you add the multipart to a properly addressed message and send it.

To copy the content from one message to another, just copy over its `DataHandler`, a class from the JavaBeans Activation Framework.

Forwarding Message 2



Example: Forwarding Message 1

```
// Create the message to forward
Message forward = new MimeMessage(session);

// Fill in header
forward.setSubject("Fwd: " + message.getSubject());
forward.setFrom(new InternetAddress(from));
forward.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Here you go with the original
                                    message:\n\n");
```

Example: Forwarding Message 2

```
// Create a multi-part to combine the parts
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);

// Create and fill part for the forwarded content
messageBodyPart = new MimeBodyPart();
messageBodyPart.setDataHandler(message.getDataHandler());
// Add part to multi part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
forward.setContent(multipart);
// Send message
Transport.send(forward);
```

Working with Attachments

Attachments are resources associated with a mail message, usually kept outside of the message like a text file, spreadsheet, or image.

With JavaMail you can:

1) attach resources to your mail message with the JavaMail API

2) and get those attachments when you receive the message

Sending Attachments

To send an attachment with your mail

1) Create a new `MimeBodyPart`

2) Create a `DataSource` object. A `DataSource` object is part of JAF defined in `javax.activation` package.

3) Wrap the `DataSource` object in a `DataHandler`. This will allow us to pass the `DataHandler` to the body part object.

Example: Sending Attachments 1

```
// Define message
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                                new InternetAddress(to));
message.setSubject("Hello JavaMail Attachment");

// Create the message part
BodyPart messageBodyPart = new MimeBodyPart();

// Fill the message
messageBodyPart.setText("Pardon Ideas");

Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
```

Example: Sending Attachments 2

```
// Part two is attachment
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new
                                DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Put parts in message
message.setContent(multipart);

// Send the message
Transport.send(message);
```

Lab Work: Sending Attachment 1

1) The skeleton code already includes the code to get the initial mail session.
2) From the session, get a Message and set its header fields: to, from, and subject.
3) Create a BodyPart for the main message cotent and fill its content with the text of the message.
4) Create a Multipart to combine the main content with the attachment. Add the main content to the multipart.
5) Create a second BodyPart for the attachment.
6) Get the attachment as a DataSource.
7) Set the DataHandler for the message part to the data source. Carry the original filename along.
8) Add the second part of the message to the multipart.

Lab Work: Sending Attachment 2

9) Set the content of the message to the multipart.

10) Compile and run the program, passing your SMTP server, from address, to address, and filename on the command line. This will send the file as an attachment.

Lab Work: Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class AttachExample {
   public static void main (String args[]) throws
  Exception {
     String host = args[0];
     String from = args[1];
     String to = args[2];
     String filename = args[3];
     // Get system properties
     Properties props = System.getProperties();
```

Lab Work: Skeleton Code 2

```
// Setup mail server
   props.put("mail.smtp.host", host);
// Get session
   Session session = Session.getInstance(props,
                                                null);
   // Define message
   // Create the message part
   // Fill the message
   // Create a Multipart
   // Add part one //
   // Part two is attachment //
   // Create second body part
```

Exercise : Skeleton Code 3

```
            // Get the attachment
        // Set the data handler to the attachment
        // Set the filename
        // Add part two
            // Put parts in message
        // Send the message
    }
}
```

Getting Attachments

The content of your message is a `Multipart` object when it has attachments.

You then need to process each `Part`, to get the main content and the attachment(s).

Parts marked with a disposition of `Part.ATTACHMENT` from `part.getDisposition()` are clearly attachments.

However, attachments can also come across with no disposition (and a non-text MIME type) or a disposition of `Part.INLINE`.

Just get the original filename with `getFileName()` and the input stream with `getInputStream()`.

Example: Getting Attachments

```
Multipart mp = (Multipart)message.getContent();

for (int i=0, n=multipart.getCount(); i<n; i++) {
  Part part = multipart.getBodyPart(i));
  String disposition = part.getDisposition();

  if ((disposition != null) &&

  ((disposition.equals(Part.ATTACHMENT) ||

     (disposition.equals(Part.INLINE))) {
    saveFile(part.getFileName(),
  part.getInputStream());
  }
}
```

Writing Attachments

The `saveFile()` method just creates a File from the filename, reads the bytes from the input stream, and writes them off to the file.

In case the file already exists, a number is added to the end of the filename until one is found that doesn't exist.

```
// from saveFile()
File file = new File(filename);
for (int i=0; file.exists(); i++) {
   file = new File(filename+i);
}
```

Attachment: General Case

The code above covers the simplest case where message parts are flagged appropriately.

To cover all cases, handle when the disposition is `null` and get the MIME type of the part to handle accordingly.

```
if (disposition == null) {
   // Check if plain
   MimeBodyPart mbp = (MimeBodyPart)part;
   if (mbp.isMimeType("text/plain")) {
            // Handle plain
   } else {
            // Special non-attachment cases here of
            // image/gif, text/html, ...
   }... }
```

Sending HTML Messages

To send a HTML file as the message and let the mail reader worry about fetching any embedded images or related pieces

1) use the `setContent()` method of Message
2) passing along the content as a `String` and setting the content type to `text/html`.

Example:

```
String htmlText = "<H1>Hello</H1>" +
   "<imgsrc=\"http://www.jguru.com/images/logo.gif\">";
message.setContent(htmlText, "text/html"));
```

Including Images in HTML

if you want your HTML content message to be complete, with embedded images included as part of the message:

1) you must treat the image as an attachment

2) and reference the image with a special cid URL, where the cid is a reference to the Content-ID header of the image attachment.

3) tell the `MimeMultipart` that the parts are related by setting its subtype in the constructor (or with `setSubType()`)

4) and set the Content-ID header for the image to a random string which is used as the src for the image in the `<img>` tag.

Example: Including Images 1

```
String file = ...;

// Create the message
Message message = new MimeMessage(session);

// Fill its headers
message.setSubject("Embedded Image");
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));
// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
String htmlText = "<H1>Hello</H1>" + "<img
                                src=\"cid:memememe\">";
```

Example: Including Images 2

```
messageBodyPart.setContent(htmlText, "text/html");

// Create a related multi-part to combine the parts
MimeMultipart multipart = new MimeMultipart("related");
multipart.addBodyPart(messageBodyPart);

// Create part for the image
messageBodyPart = new MimeBodyPart();

// Fetch the image and associate to part
DataSource fds = new FileDataSource(file);
messageBodyPart.setDataHandler(new DataHandler(fds));
messageBodyPart.setHeader("Content-ID","<memememe>");
```

Example: Including Images 3

```
// Add part to multi-part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
message.setContent(multipart);
```

Lab Work: Sending HTML

1) The skeleton code already includes the code to get the initial mail session, create the main message, and fill its headers (to, from, subject).
2) Create a BodyPart for the HTML message content.
3) Create a text string of the HTML content. Include a reference in the HTML to an image (<img src="…">) that is local to the mail message.
4) Set the content of the message part. Be sure to specify the MIME type is text/html.
5) Create a Multipart to combine the main content with the attachment. Be sure to specify that the parts are related. Add the main content to the multipart.
6) Create a second BodyPart for the attachment.
7) Get the attachment as a DataSource, and set the DataHandler for the message part to the data source.

Lab Work:

8) Set the Content-ID header for the part to match the image reference specified in the HTML.
9) Add the second part of the message to the multipart, and set the content of the message to the multipart.
10) Send the message.
11) Compile and run the program, passing your SMTP server, from address, to address, and filename on the command line. This will send the images as an inline image within the HTML text.

Lab Work: Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class HtmlImageExample {
   public static void main (String args[]) throws
                                        Exception {
      String host = args[0];
      String from = args[1];
      String to = args[2];
      String file = args[3];
```

Lab Work: Skeleton Code 2

```
        // Get system properties
Properties props = System.getProperties();
        // Setup mail server
props.put("mail.smtp.host", host);
// Get session
Session session = ession.getDefaultInstance(props,


null);
// Create the message
Message message = new MimeMessage(session);
// Fill its headers
message.setSubject("Embedded Image");
message.setFrom(new InternetAddress(from));
```

Lab Work: Skeleton Code 3

```
    message.addRecipient(Message.RecipientType.TO,

    new InternetAddress(to));
    // Create your new message part
    // Set the HTML content, be sure it references
     //the attachment
    // Set the content of the body part
    // Create a related multi-part to combine the

parts
    // Add body part to multipart
    // Create part for the image
    // Fetch the image and associate to part
    // Add a header to connect to the HTML
    // Add part to multi-part
    // Associate multi-part with message
    // Send message  } }
```

Exercise: JavaMail

Java Message Service

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Overview

1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

Introduction 1

Information systems are increasingly based on distributed architectures

Needs for integrating existing stand-alone systems are increasing

Middleware is an attempt to ease distributed system development, and try to embedded complexity of communication between programs such as:

a) Different data representations & encodings
b) Different transport protocols
c) Different programming languages, ...

Introduction 2

Types of middleware

1) Procedure-oriented

a) Client/Server e.g. RPC

2) Object-oriented

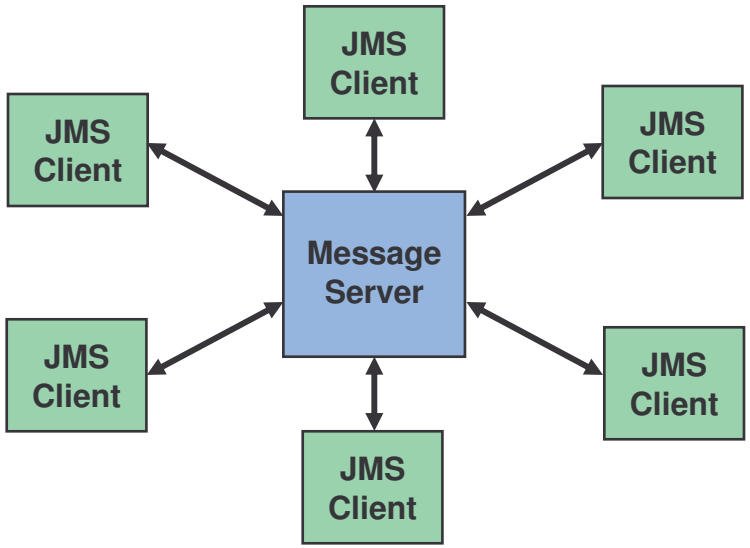
a) Distributed Objects e.g. CORBA, RMI

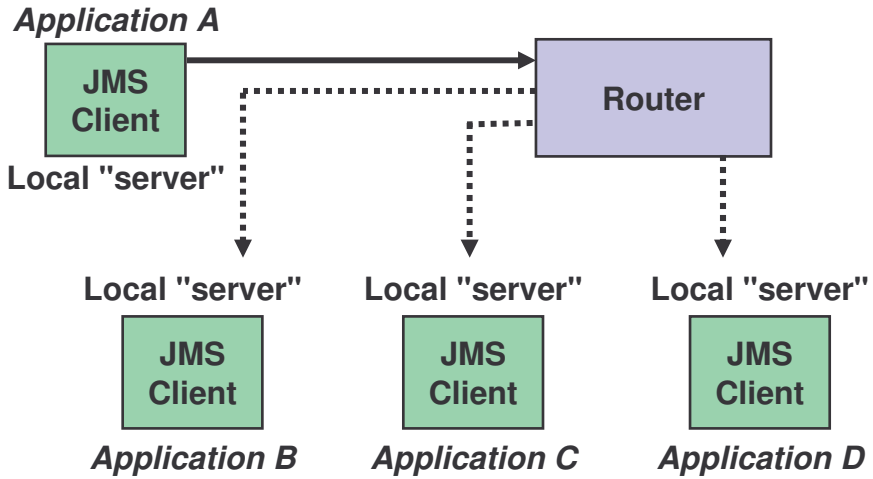
3) Message Oriented Middlewares(MOMs)

a) Asynchronous messaging e.g. JMS

What is Messaging ?

1) A method of peer-to-peer communication between software components or applications.
2) Enables distributed communication that is loosely coupled; differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.



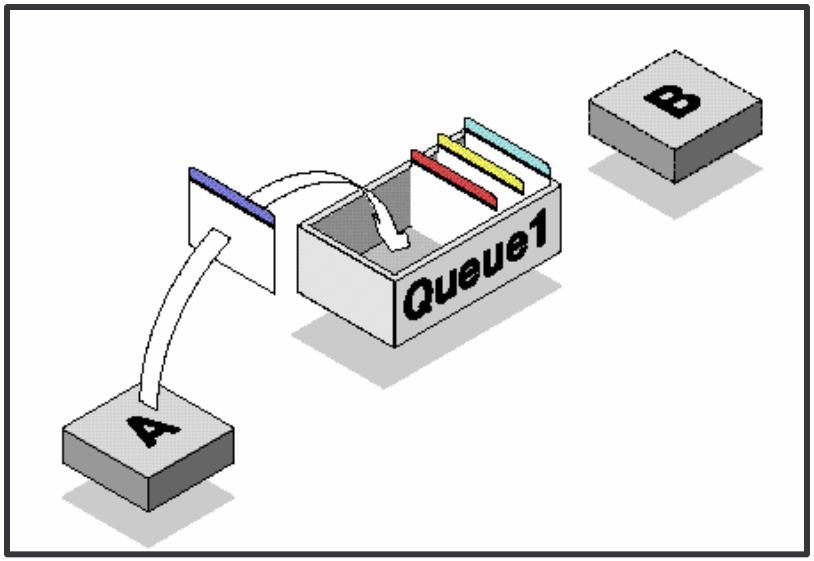


Reliable Messaging With Queues

MOMs provide asynchronous messaging

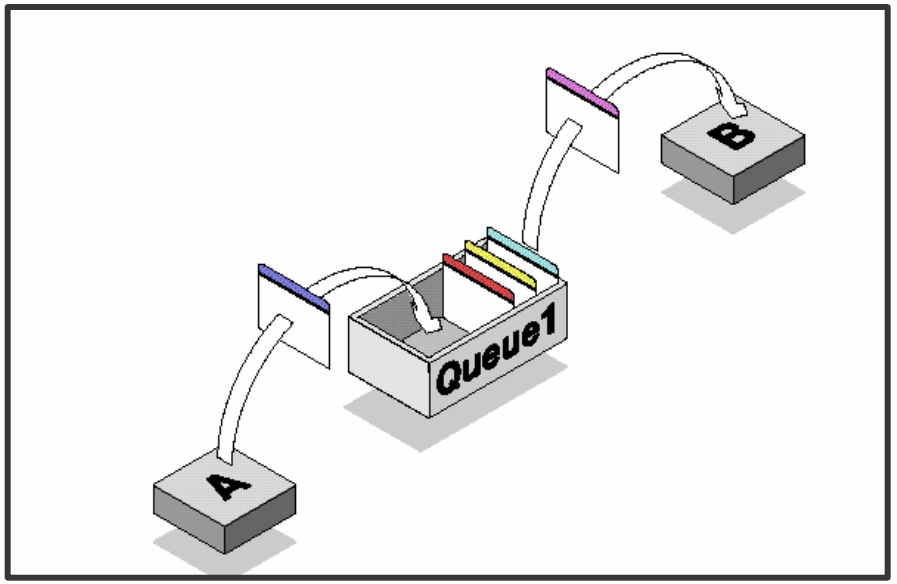
If one party is unavailable, messaging subsystem is still available and functional

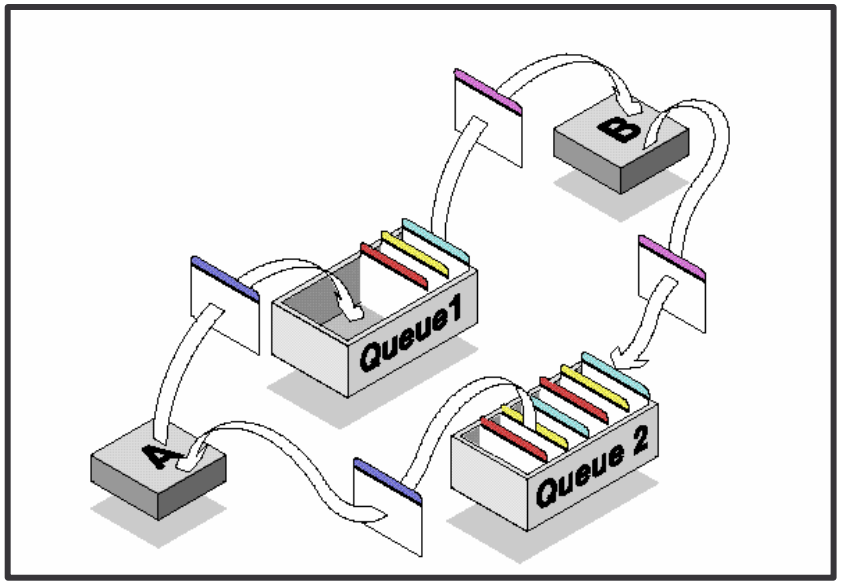
Queues exist independent from the applications



Queuing Basics 1

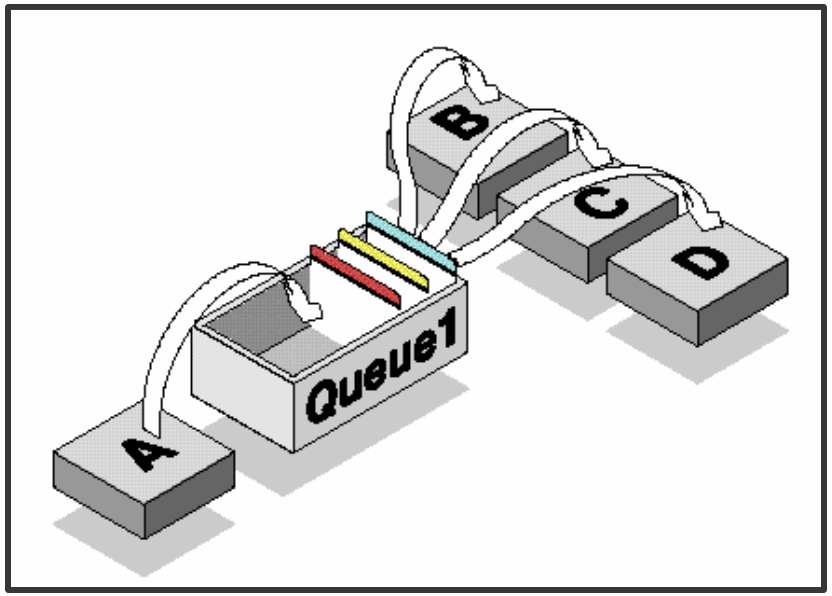
Queues are uni-directional, but multiple queues may be used to provide bi-directional messaging

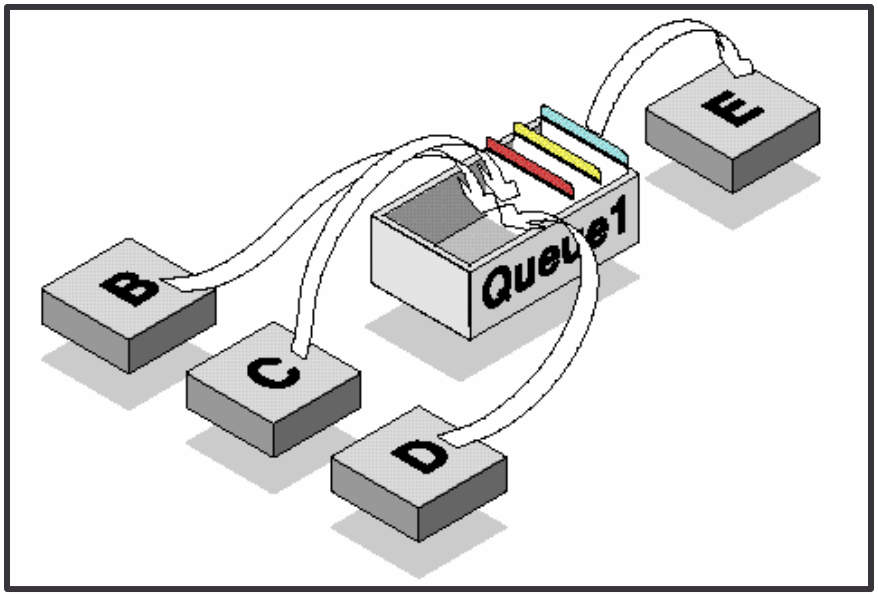




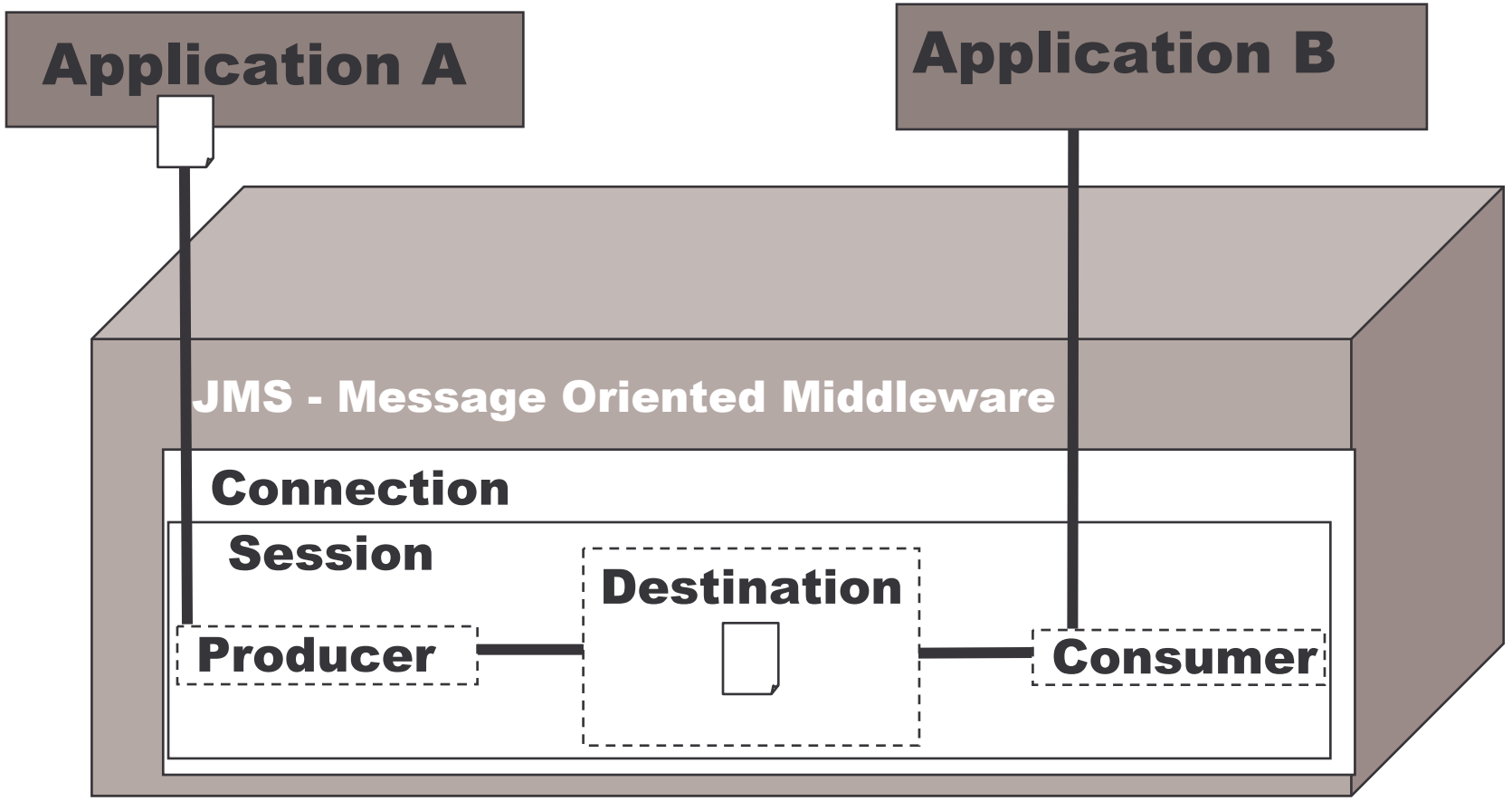
Queuing Basics 2

Queues can work in different models and make one to many and many to one relations possible





Producer and Consumer



The Java Messaging Service 1

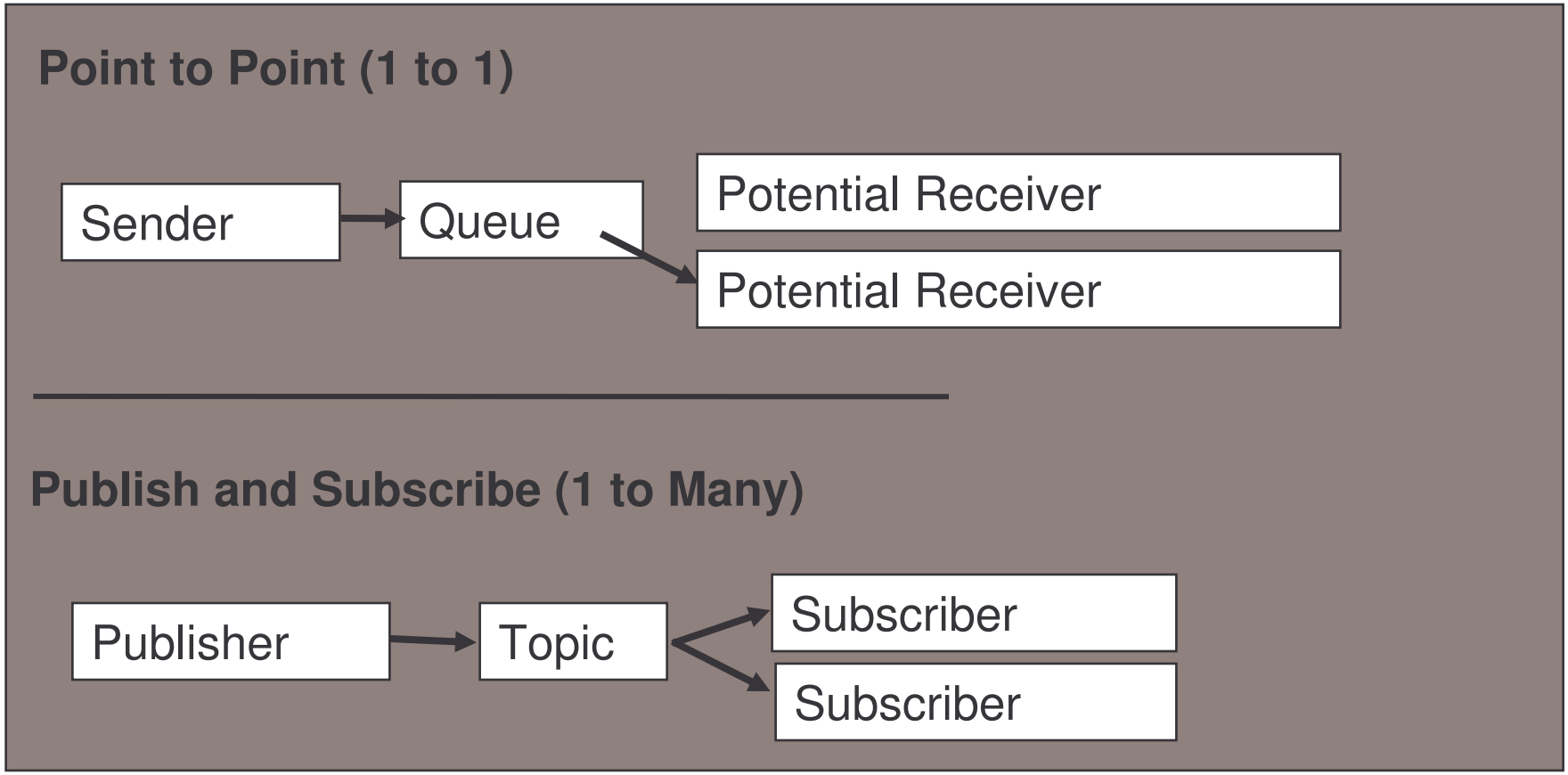
A J2EE API to access MOM products from Java

Vendor-neutral API for higher-interoperability

Has two models:

1) Publish and Subscribe
    a)0 or more recipients
    b)Messages passed between publishers and subscribers via topics
    c)Message can be subscribed to in a durable manner
    d)Message are consumed at least once
2) Point-to-Point
    a)One recipient only
    b)Messages are consumed at most once and only once

The Java Messaging Service 2



The Promises of JMS

1) “Messaging for the masses”
   a) Could have similar impact that SQL had on databases
   b) Similar to JDBC (which all vendors now support)
2) First enterprise messaging API to achieve wide industry support (standard)
3) Simplifies development of enterprise applications (ease of use)
4) Leverages existing enterprise-proven messaging systems (implementation)
5) Easy to write portable messaging based business applications (write once, run anywhere)

The Promises of JMS

1) “Messaging for the masses”
   a) Could have similar impact that SQL had on databases

2) First enterprise messaging API to achieve wide industry support

3) Simplifies development of enterprise applications

4) Leverages existing enterprise-proven messaging systems

5) Easy to write portable messaging based business applications

Limitations of the JMS

JMS does not address

a) Security

b) Load Balancing

c) Fault Tolerance

d) Error Notification (apart from Exceptions)

e) Administration API

f) Transport protocol for messaging

Overview

1) introduction
2) JMS messaging model
3) JMS programming model and implementation
4) advance configuration
5) summary

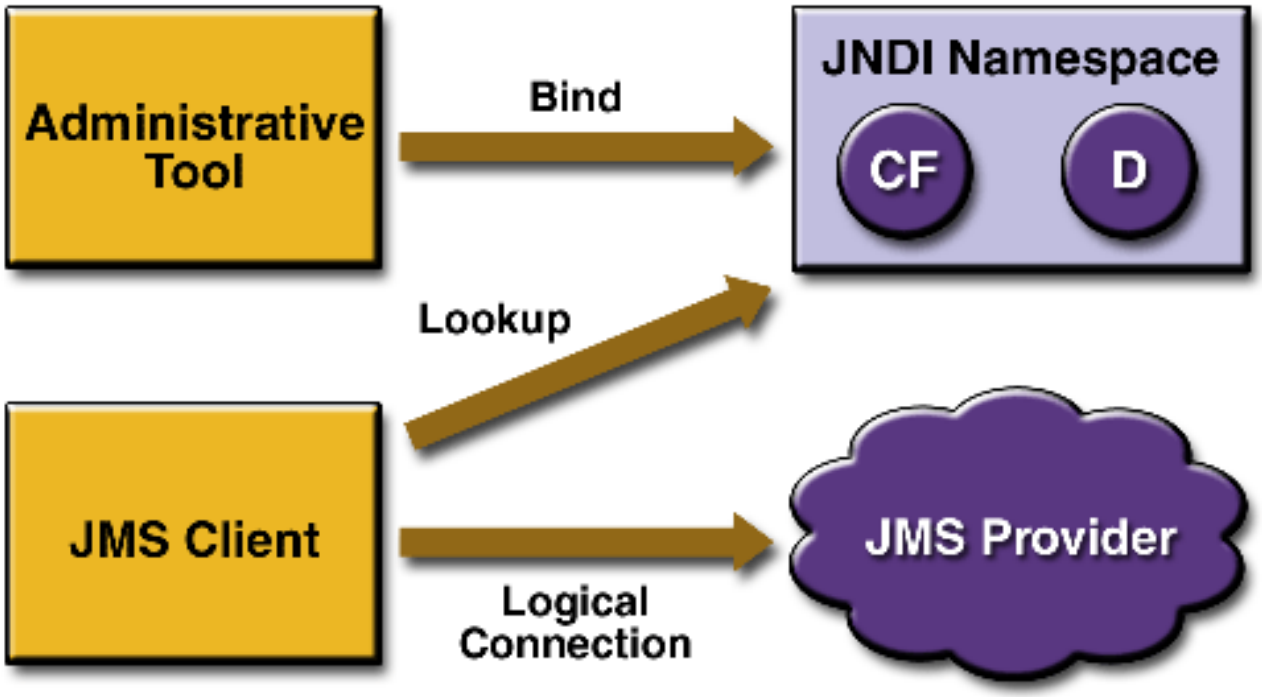
JMS API Concepts 1

A JMS application is composed of the following parts:

1) JMS Provider
    a) messaging system that implements JMS and administrative functionality, e.g. IBM's MQSeries and JBossMQ message server.
2) JMS Clients
    a) Java programs that send/receive messages
3) Messages
    a) Items of information sent between JMS clients.
4) Administered Objects
    a) preconfigured JMS objects created by an admin for the use of clients
    b) `ConnectionFactory`, Destination (queue or topic)

JMS API Concepts 2

Interaction between different parts of JMS:



JMS Messaging Domains

Point-to-Point (PTP)

a) built around the concept of message queues
b) each message has only one consumer

Publish-Subscribe systems

a) uses a “topic” to send and receive messages
b) each message has multiple consumers

Point-to-Point Messaging 1

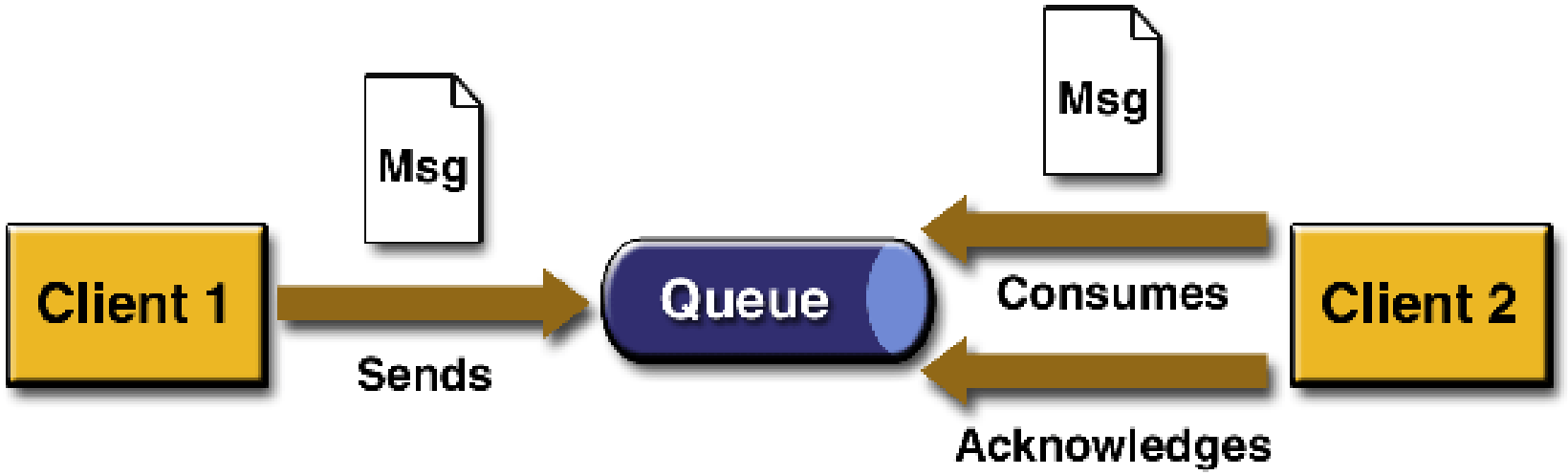
Each message is addressed to a specific queue

Receiving clients extract messages from the queue(s)

Queues retain all messages sent to them until:

a) the messages are consumed

b) the messages expire



Point-to-Point Messaging 2

Characteristics of Point-to-Point Messaging :

a) Each message has only one consumer.

b) A sender and a receiver of a message have no timing dependencies.

c) The receiver acknowledges the successful processing of a message.

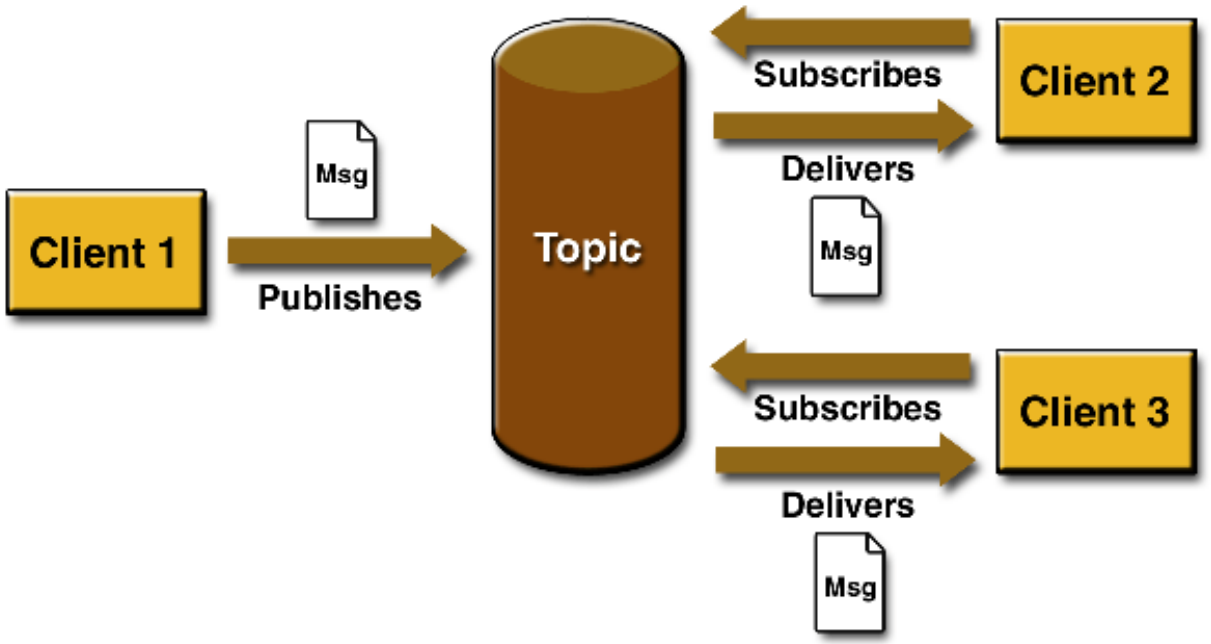
d) Should be used when every message send must be processed successfully by one consumer.

Publish/Subscribe Messaging 1

Clients address messages to a topic.

The system takes care of distributing the messages.

Topics retain messages only as long as it takes to distribute them to current subscribers.



Publish/Subscribe Messaging 2

Characteristics of Pub/Sub Messaging :

a) Each message may have multiple consumers.

b) A client that subscribes to a topic can consume only messages published after the client has created a subscription.

c) The subscriber must continue to be active in order for it to consume messages.

d) Exception for time dependency is allowed for durable subscription. (Will be discussed later)

JMS Message

A JMS message has three parts:

1) Message Header
   a) used for identifying and routing messages
   b) contains vendor-specified values, but could also contain application-specific data
   c) typically name/value pairs
2) Message Properties (optional)
   a) act like additional headers
3) Message Body(optional)
   a) contains the data
   b) five different message body types in the JMS specification

JMS Header

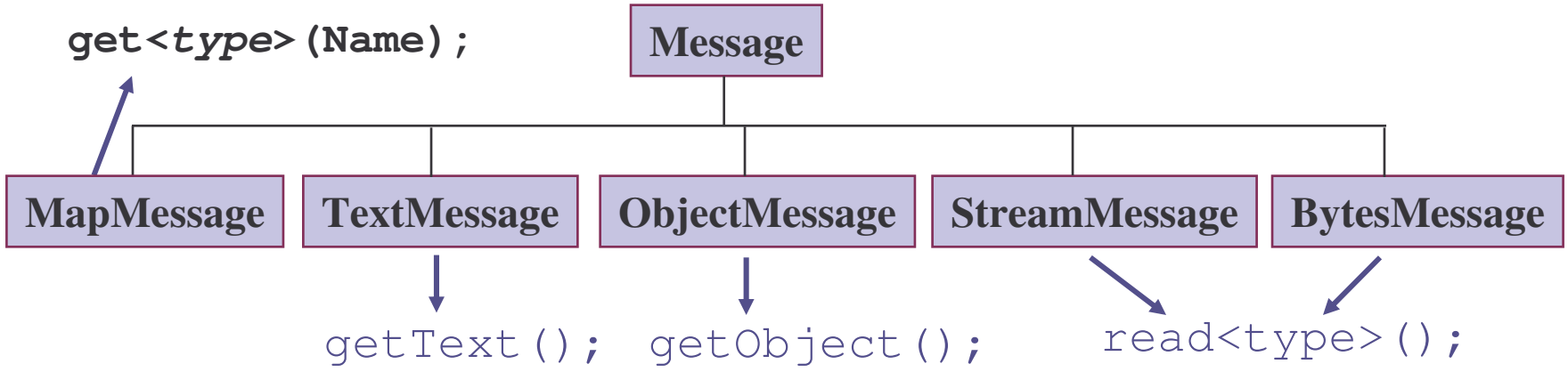
Automatically assigned headers
JMSDestination
JMSDeliveryMode
JMSMessageID
JMSTimestamp
JMSExpiration
JMSRedelivered
JMSPriority

Developer-assigned headers
JMSReplyTo
JMSCorrelationID
JMSType

JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble, getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes, writeString, readString
StreamMessage	stream of primitive values	writeString,writeDouble,writeLong, readString
ObjectMessage	serialize object	setObject,getObject

Accessing JMS Message



Messages Consumption

In the JMS Specification, messages can be consumed in either of two ways:

Synchronously

a) A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
b) The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.

Asynchronously

a) A client can register a *message listener* with a consumer.
b) Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

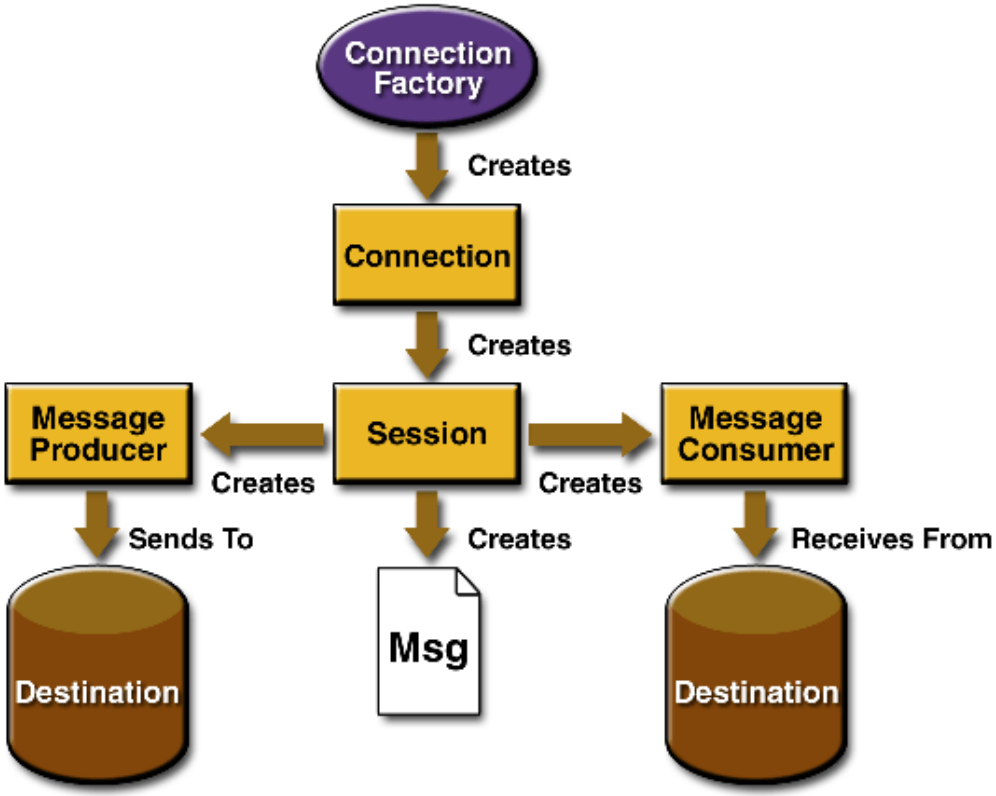
Overview

1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

JMS API Programming Model

The basic building blocks of a JMS application:

1) Administered objects
2) Sessions
3) Message producers
4) Message consumers
5) Messages



JMS Client Setup Procedure

A typical pub/sub JMS client executes the following setup procedure:

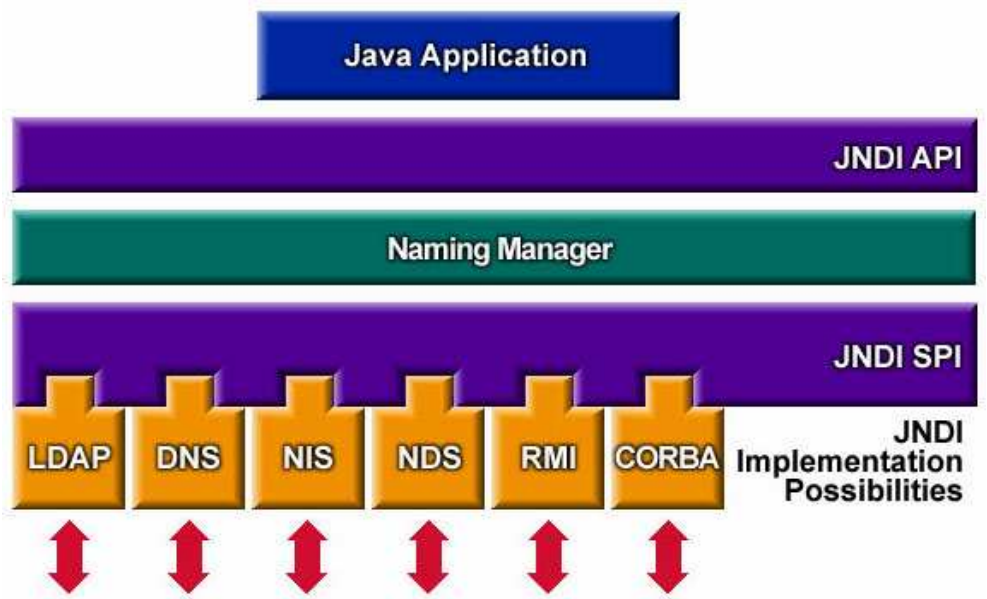
1) Use `JNDI` to find a `ConnectionFactory` object
2) Use `JNDI` to find one or more Destination objects
3) Use the `ConnectionFactory` to create a JMS Connection
4) Use the Connection to create one or more JMS Sessions
5) Use a Session and the Destinations to create the `TopicPublisher` and `TopicSubscriber` needed
6) Enable the Connection to start delivering messages to `TopicSubscriber`

What is JNDI

Java Naming and Directory Interface (JNDI) is an integral component of J2EE technology

JNDI is an application programming interface (API) that provides directory and naming services to Java applications.

JNDI is defined to be independent of any specific naming or directory service implementation. A variety of services can be accessed in a common way.



JNDI package

Following are the `JNDI` packages:

1) `javax.naming`
2) `javax.naming.directory`
3) `javax.naming.event`
4) `javax.naming.ldap`
5) `javax.naming.spi`

Obtain JNDI Connection 1

1) Instantiate an Properties object:

```
Properties env = new Properties();
```

2) Specify the JNDI properties specific to the vendor:

```
env.put("java.naming.factory.initial",
        "org.jnp.interfaces.NamingContextFactory");
env.put("java.naming.provider.url",
        "jnp://localhost:1099");
env.put("java.naming.factory.url.pkgs",
        "org.jboss.naming:org.jnp.interfaces");
```

3) Obtain JNDI Connection

```
Context jndi=new InitialContext(env);
```

Obtain JNDI Connection 2

If a file named `jndi.properties` is in the classpath of the client program, you can use the following setting:

```
Context jndi = new
          InitialContext(System.getProperties());
```

This can remove the vendor specific code from the client program.

Setup Using JNDI

1) Use `JNDI` to find a `ConnectionFactory` object :

```
TopicConnectionFactory conFactory =
    (TopicConnectionFactory)jndi.lookup
        ("ConnectionFactory");
```

2) Use `JNDI` to find one or more Destination objects :

```
Topic myTopic =
    (Topic)jndi.lookup(topicName);
```

remark: In JBoss, the topic name can be found in the file :

`<Jboss_Home>\server\default\deploy\jms\jbossmq-destinations-service.xml`

Setup Connection and Session

1) Use `ConnectionFactory` to create a JMS Connection

```
TopicConnection connection =
        conFactory.createTopicConnection();
```

2) Use the Connection to create one or more JMS Sessions

```
TopicSession pubSession =
        connection.createTopicSession(false,
          Session.AUTO_ACKNOWLEDGE);
TopicSession subSession =
        connection.createTopicSession(false,
          Session.AUTO_ACKNOWLEDGE);
```

Message Publisher

1) Creating producer

```
MessageProducer producer=
   pubSession.createProducer(myTopic);
```

2) Send a message

```
TextMessage m=pubSession.createTextMessage();
m.setText("just another message");
publisher.publish(m);
```

3) Closing the connection

```
connection.close();
```

Message Subscriber

1) Creating subscriber

```
TopicSubscriber subscriber =
                    subSession.createSubscriber(myTopic);
```

2) Set a JMS message listener

```
subscriber.setMessageListener(<Message Listener>);
```

Message Listener 1

A Message Listener is a class implements interface `javax.jms.MessageListener` and has to implement the `onMessage(javax.jms.Message message)` method

Example of `onMessage` method:

```
public void onMessage(Message message) {
    TextMessage msg = null;
    try {
        if (message instanceof TextMessage) {
            msg = (TextMessage) message;
            System.out.println("Reading message: " +
              msg.getText());
        }
```

Message Listener 2

```
else {
        System.out.println("Message of wrong type: "
    +  message.getClass().getName());
      }
} catch (JMSException e) {
        System.out.println("JMSException in
        onMessage(): " + e.toString());
   } catch (Throwable t) {
        System.out.println("Exception in onMessage():"
        + t.getMessage());
   }
}
```

Start and Close the Connection

Enable the Connection to start delivering messages to `TopicSubscriber`

```
connection.start();
```

Stop the Connection before ending the client program.

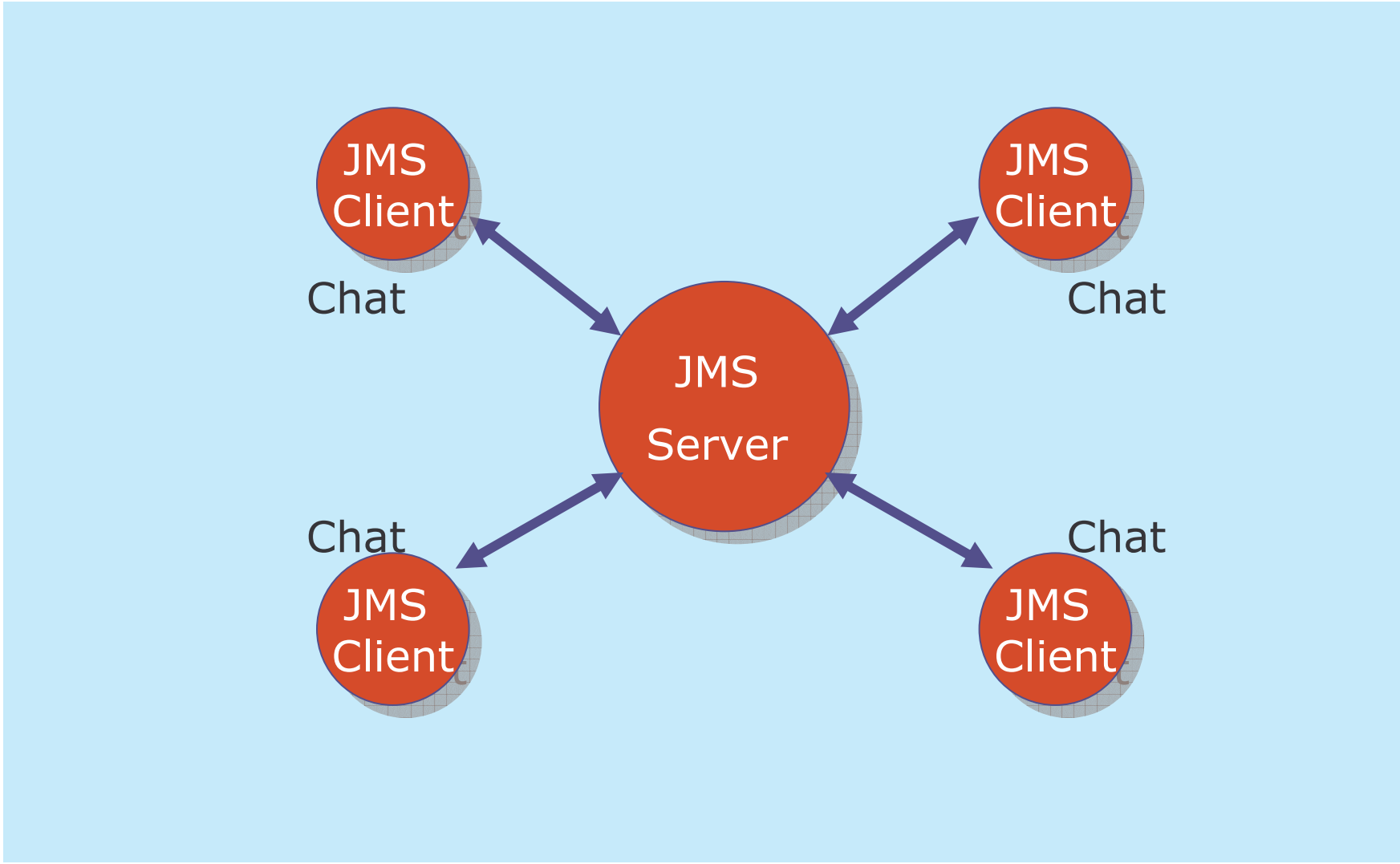
```
connection.close();
```

Both methods throws `javax.jms.JMSException`

Lab Work: A JMS Chat Client 1

1) According to the procedure we discussed, write a pub/sub chatting program using JMS.
   a) Use JBoss as the JMS server.
   b) Create a topic "emacao" in JBoss. You can modify the file `<JBoss Home>\server\default\deploy\jms\jbossmq-destinations-service.xml` for creating a topic.
   c) Execute the program from the command line:
      1. `Java Chat topic/emacao username`
      2. Note: for JBoss, the default JNDI name for a topic is `topic/<topic name>`
      3. and is `queue/<queue name>` for a queue.

Lab Work: A JMS Chat Client 2



Point-to-Point Messaging 1

Point-to-Point (PTP) application is built around the concept of message queues, sender and receivers.

a) Each message is addressed to a specific queue and the receiving clients extract messages from the queues established to hold their messages.

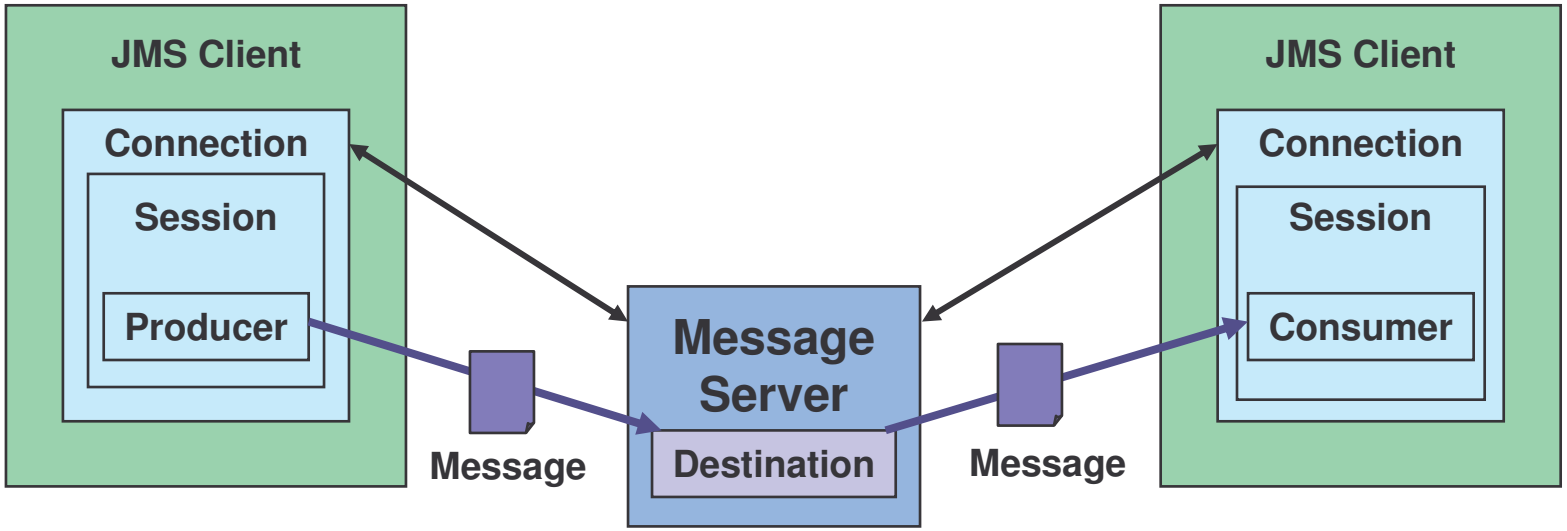
b) Each message has only one consumer.

c) A sender and receiver have no time dependencies.

d) The receiver acknowledges the successful processing of a message.

e) Use PTP when every message you send must be processed successfully by one consumer

Point-to-Point Messaging 2



Message Queue Sender

1) Performs a Java Naming and Directory Interface (`JNDI`) API lookup of the `QueueConnectionFactory` and `Queue.`
2) Creates a `QueueConnection` and a `QueueSession.`
3) Creating Queue Sender

```
javax.jms.QueueSender
        sender=session.createSender(<queue name>);
```

4) Send a message

```
Message m=session.createTextMessage();
m.setText("just another message");
sender.send(m);
```

5) Closing the connection

```
connection.close();
```

Message Queue Receiver

1) Performs a Java Naming and Directory Interface (`JNDI`) API lookup of the `QueueConnectionFactory` and `Queue.`
2) Creates a `QueueConnection` and a `QueueSession.`
3) Creating Queue Receiver

```
javax.jms.QueueReceiver
  queueReceiver=session.createReceiver(<queue name>);
```

4) Starts the connection, causing message delivery to begin
5) Receives the messages sent to the queue until the end-of-message-stream
   - `Message m = queueReceiver.receive();`
   - `Message m = queueReceiver.receive(0);`
6) Closing the connection

```
connection.close();
```

Timed Synchronous Receive

If you do not want your program to consume system resources unnecessarily, do one of the following:

1) Call the receive method with a timeout argument greater than 0:

```
Message m = queueReceiver.receive(1); // 1 ms
```

2) Call the receiveNoWait method, which receives a message only if one is available:

```
Message m = queueReceiver.receiveNoWait();
```

3) The `receive()` method is also available for the `TopicSubscriber` and will negate the use of the `onMessage()` callback.

Basic Reliability Mechanisms

1) Specifying message persistence.
   a) You can specify that messages are persistent, meaning that they must not be lost in the event of a provider failure.

2) Controlling message acknowledgment.
   a) You can specify various levels of control over message acknowledgment.

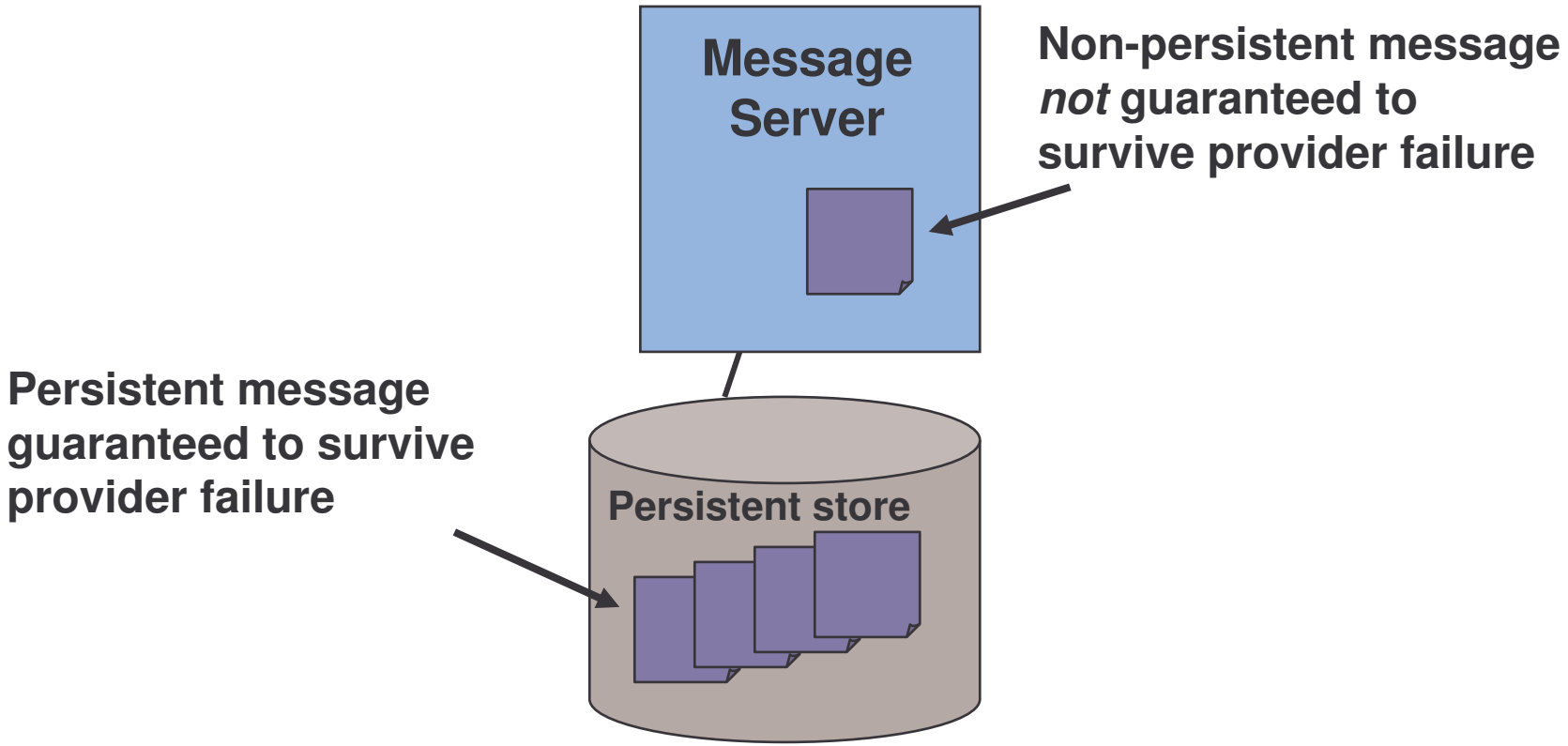
3) Setting message priority levels.

4) Allowing messages to expire.
   a) You can specify an expiration time for messages

Persistent Messages

Messages can be marked as persistent.

The implementation of the storage mechanism is up to the JMS provider.



Exercise: Message Queue

1) Create Point-to-Point messaging program
   a) Create a queue in JBoss with a name qex.
   b) According to our discussion, please create a JMS client for sending message to the qex queue.
   c) Create a Queue Receiver for the qex queue.
   d) Try to stop the receiver and use the sender to send some message. Restart the receiver and check if it receive the message.

Overview

1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

Temporary Topics 1

Is a topic that is dynamically created by the JMS provider, using the `createTemporaryTopic()` of the `TopicSession` object.

Is a topic associated with the connection that belongs to the `TopicSesson` that created it.

It lasts only as long as its associated client connection is active.

Topic identity is transferred using the `JMSReplyTo` header.

Temporary Topics 2

Procedure to create a temporary topic:

1) After a session,`mySession`, is created, the client can create a dynamic topic:

```
javax.jmx.Topic tempTopic =
        mySession.createTemporaryTopic();
```

2) Create a message for the subscriber to reply to:

```
javax.jmx.TextMessage message =
        mySession.createTextMessage();
```

3) Set up the `JMSReplyTo` destination

```
message.setJMSReplyTo(tempTopic);
```

Temporary Topics 3

When a client needs to respond to the message, it can use the `JMSReplyTo` Desination:

```
public void onMessage(javax.jms.Message amessage){
 ...
   TextMessage message = (TextMessage)amessage;
   javax.jms.Topic tempTopic =
     (javax.jms.Topic)message.getJMSReplyTo();
```

Durable Subscriptions

By default a subscriber gets only messages published on a topic while a subscriber is alive

Durable subscription retains messages until they are received by a subscriber or expire

You can use the `createDurableScubsriber` method of the `java.jms.TopicSession` to create a durable subscription:

```
...
javax.jms.TopicSubscriber subscriber =
    session.createDurableSubscriber(tempTopic,"subsc
    ription name");
...
```

Unsubscribing

In order to explicitly unsubscribe a subscription, you can use the follow methods:

For nondurable subscription:

```
...
        subcriber.close();
```

For durable subscription:

```
...
        session.unsubscribe
            (“<subscription name>”);
```

Lab Work: Temporary Topic 1

1) Write two JMS clients to simulate the following scenario :

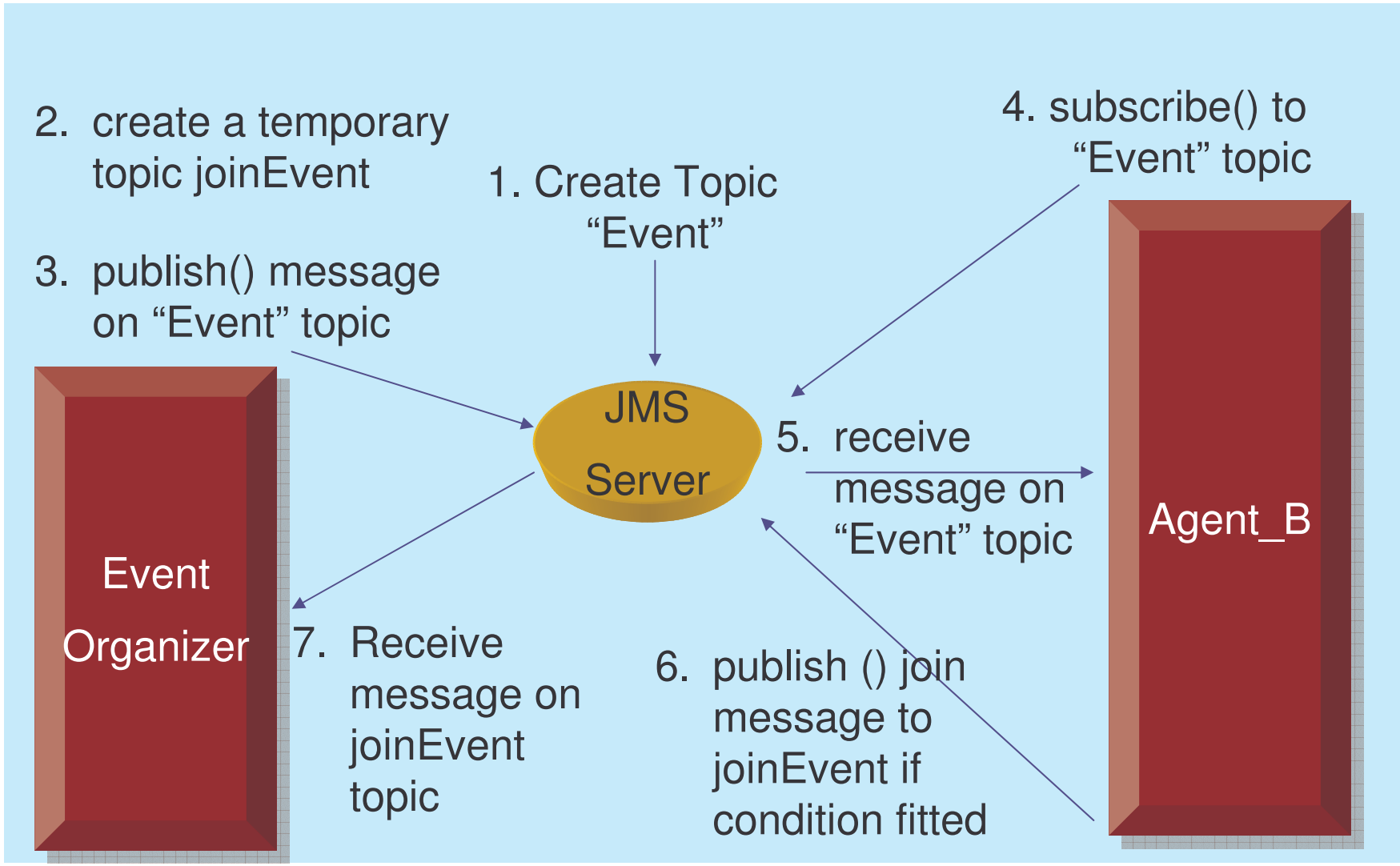
   a) An event organizer is constantly promoting events for its agents. It will publish the event message to and deliver to all the subscribed agents.

   b) After received the message, the Agents' program will evaluate the event according to certain criteria and decide to either joining the event or not. In the exercise, you can make up your own criteria such as cost or date.

   c) If the agent decided to join the event, it's program will automatically send a message back to the organizer.

   d) Your organizer's program required to create a temporary topic and attached it as the destination for the agents to reply to.

Lab Work: Temporary Topic 2



Overview

1) introduction
2) JMS Messaging Model
3) JMS programming model and implementation
4) advance configuration
5) summary

Summary

In this session, we cover the followings:

1) Concepts of Message Oriented Middleware
2) Concepts of Messaging
3) Design model of Java Message Service
4) Programming Model of Java Message Service
5) Programming publisher/subscriber JMS application
6) Programming Point-to-Point JMS application

Distributed Objects

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Overview

What options do I have for distributed application development?

Developers who program using the Java programming language can choose several solutions for creating distributed applications programs.

1) Java RMI technology

2) Java IDL technology (for CORBA programmers)

3) Enterprise JavaBeans technology

In this section we shall be talking about Java RMI and IDL technologies.

Java RMI

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

Introduction 1

Remote Method Invocation (RMI) technology was first introduced in JDK1.1.

RMI allows programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs.

RMI is based on a similar, earlier technology for procedural programming called remote procedure call (RPC)

Introduction 2

Disadvantages of RPC

a) RPC supports a limited set of data types

Therefore it is not suitable for passing and returning Java Objects

b) RPC requires the programmer to learn a special interface definition language (IDL) to describe the functions that can be invoked remotely

Introduction 3

The RMI architecture defines

a) How objects behave.

b) How and when exceptions can occur.

c) How memory is managed.

d) How parameters are passed to, and returned from, remote methods. The remote object model for Enterprise JavaBeans (EJB) is RMI-based.

Introduction 4

RMI is designed for Java-to-Java distributed applications.

RMI is simpler and easier to maintain than using socket.

Other options for creating Java-to-non-Java distributed applications are:

a) Java Interface Definition Language (IDL)

b) Remote Method Invocation (RMI) over Internet Inter-ORB Protocol (IIOP) -- RMI-IIOP.

Overview

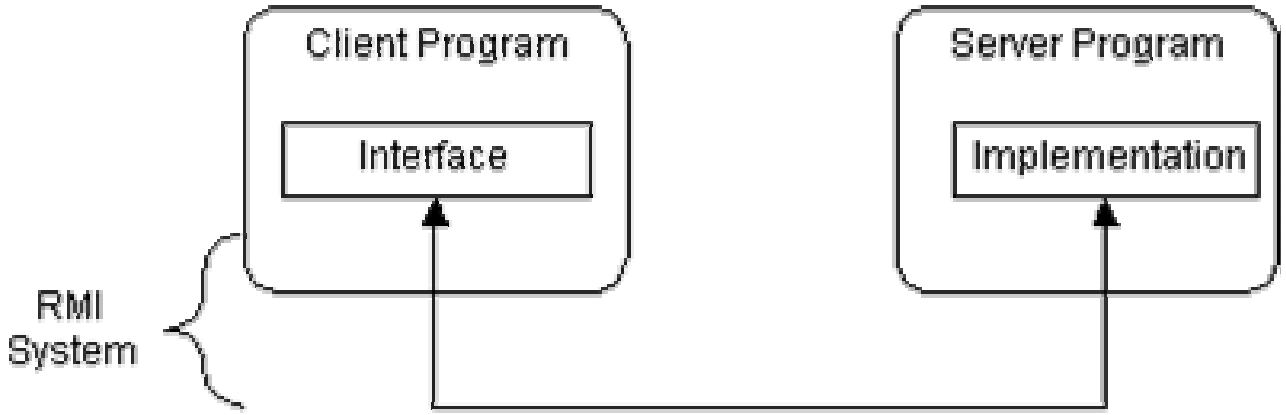
1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

Architecture 1

RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

At client side, RMI uses interfaces to define behavior.

At server side, RMI uses classes to define implementation.

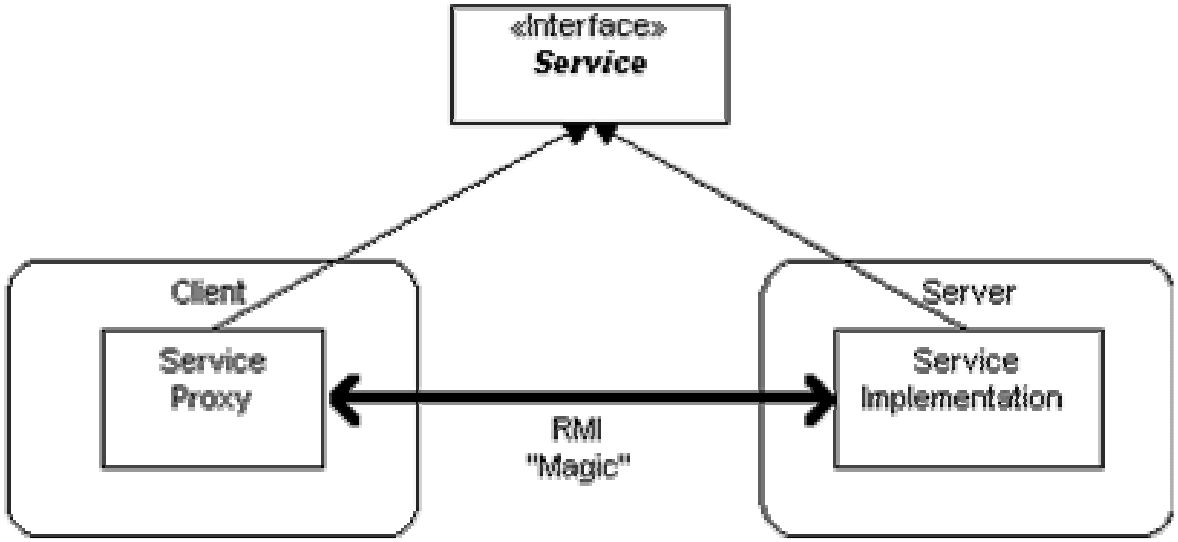


Architecture 2

The service interface is implements by two classes.

a) The first one is at the server side which implements the behavior.

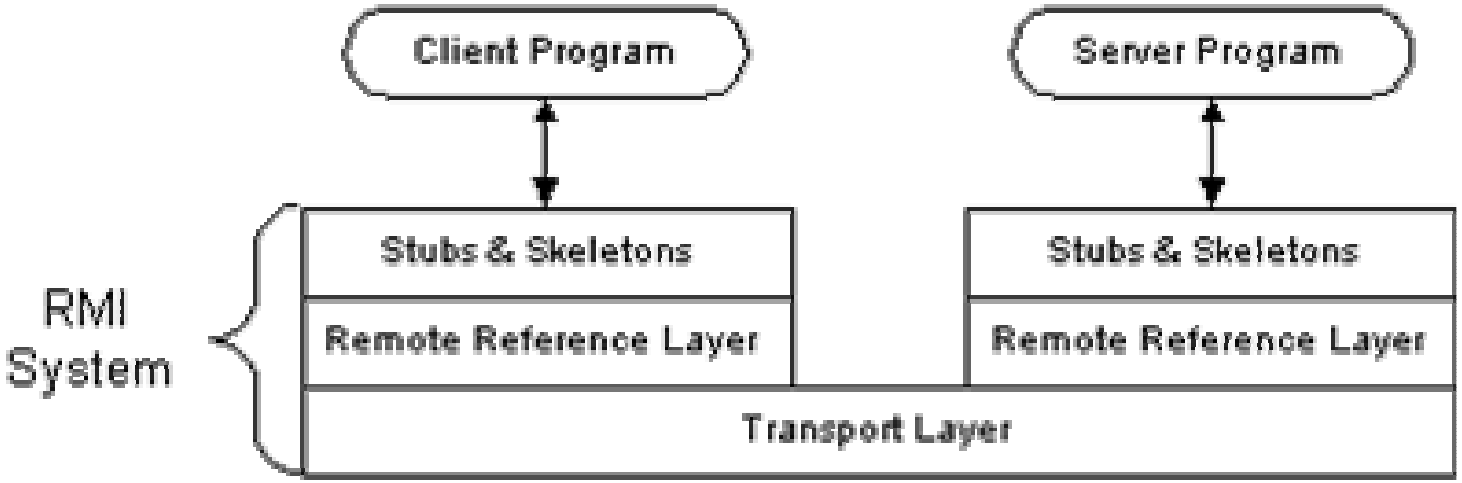
b) The second one is at the client side which acts as a proxy.



Layers

The RMI implementation is built from three abstraction layers.

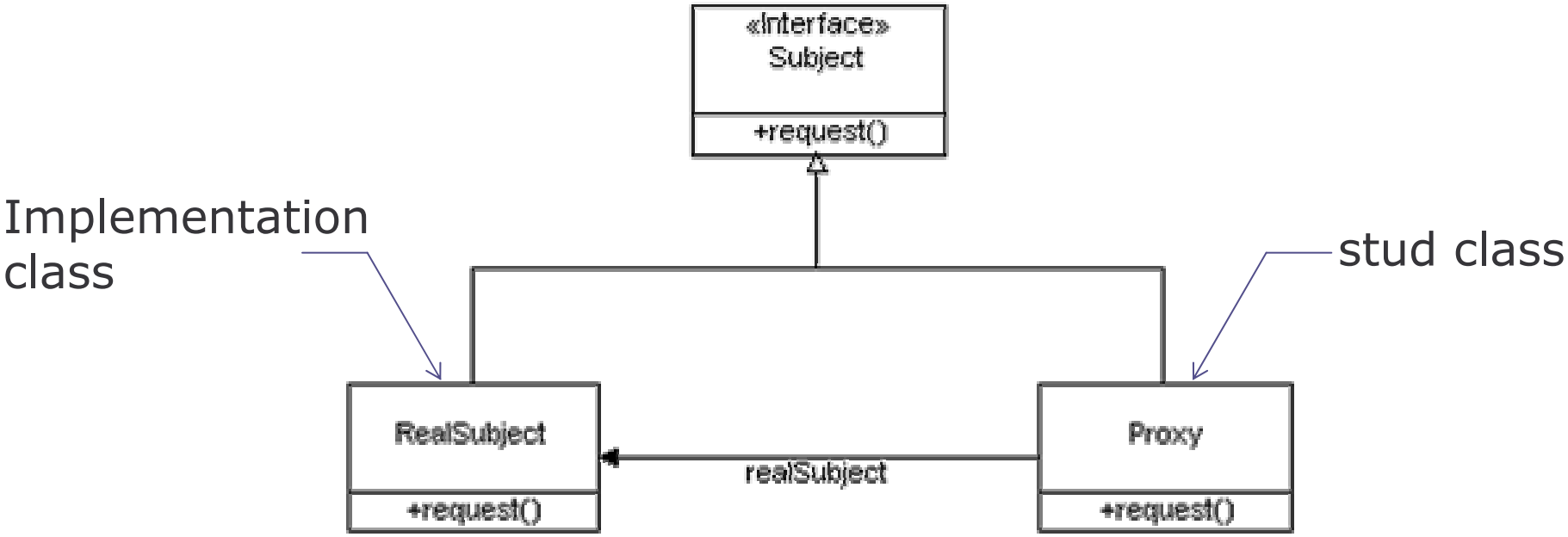
a) The Stub and Skeleton layer
b) The Remote Reference Layer
c) The transport layer



Stub and Skeleton Layer 1

The first layer lies beneath the view of the developer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

RMI uses the Proxy design pattern in this layer.



Stub and Skeleton Layer 2

A skeleton is a helper class that is generated for to communicate with the stub across the RMI link

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete.

You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

Remote Reference Layer

This layer provides a RemoteRef object that represents the link to the remote service implementation object.

In JDK 1.1, only unicast, point-to-point connection is supported. Before a client can use a remote service, the remote service must be instantiated on the server and ran all the time.

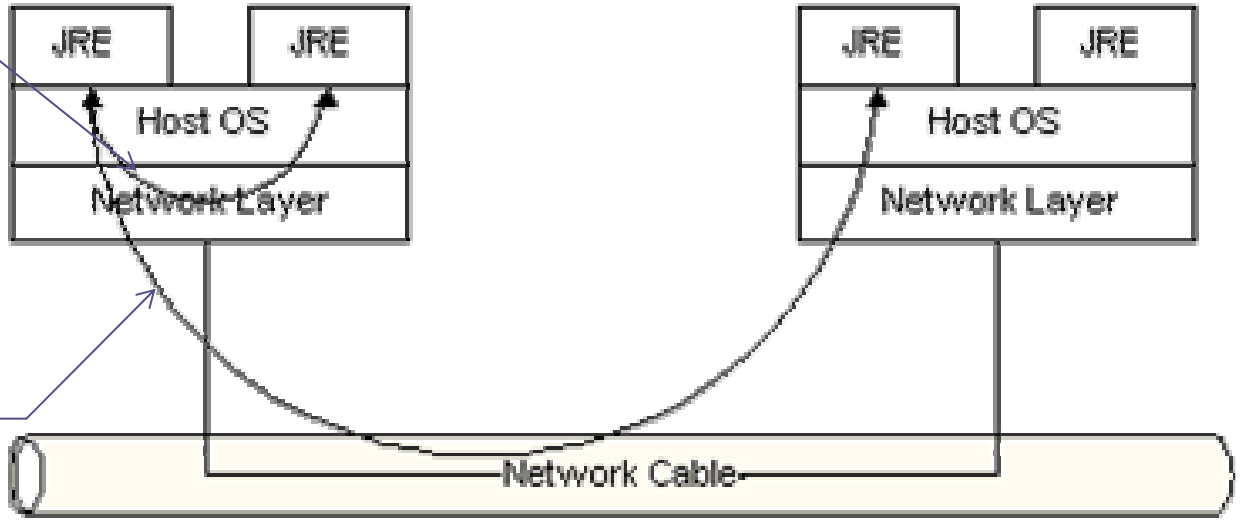
In Java 2 SDK, client-server connection is added and activatable remote objects is supported . With the introduction of the RMI daemon, rmid, remote objects can be created and execute "on demand," rather than running all the time.

Transport Layer 1

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

Communication between the same host using TCP/IP

Communication between different host using TCP/IP



Transport Layer 2

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP).

JRMP is a proprietary, stream-based protocol.

Sun and IBM have jointly worked on another version of RMI, called RMI-IIOP(Remote Method Invocation over Internet Inter-ORB Protocol), which combines RMI-style ease of use with CORBA cross-language interoperability.

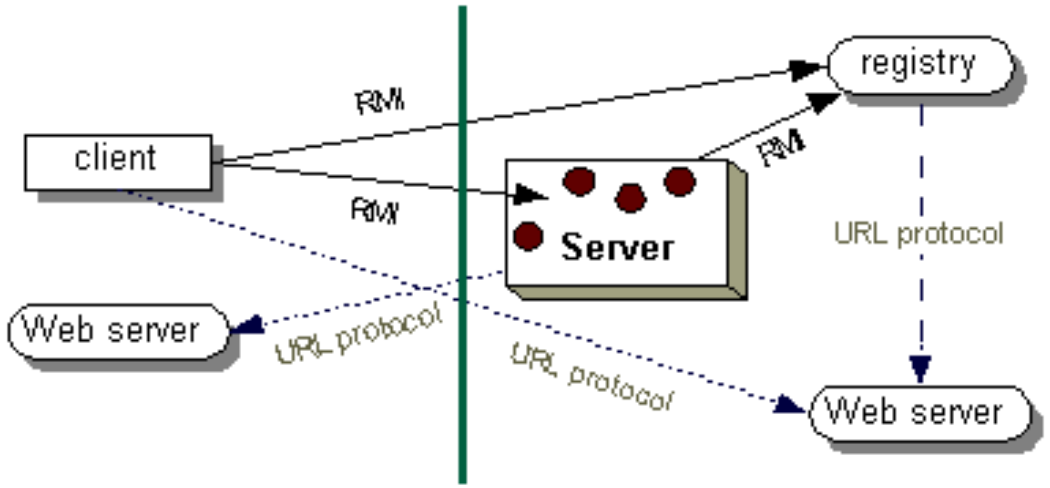
The remote object model for Enterprise Java Beans(EJBs) is RMI-based.

Naming Remote Objects

In RMI, clients find remote services by using a naming or directory service.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI).

RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.



Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

Example : build a RMI system

In this example, we shall build a simple remote calculator service and use it from a client program.

A working RMI system is composed of several parts.

a) Interface definitions for the remote services
b) Implementations of the remote services
c) Stub and Skeleton files
d) A server to host the remote services
e) An RMI Naming service that allows clients to find the remote services
f) A class file provider (an `HTTP` or `FTP` server)
g) A client program that needs the remote services

Interface 1

The first step is to write and compile the Java code for the service interface.

All the interface has to extend the `java.rmi.Remote` interface and all the methods has to declare that it may throw a `RemoteException` object.

Interface 2

The interface may look like the following:

```
public interface Calculator extends
    java.rmi.Remote
{ public long add(long a, long b) throws
    java.rmi.RemoteException;
  public long sub(long a, long b) throws
    java.rmi.RemoteException;
  public long mul(long a, long b) throws
    java.rmi.RemoteException;
  public long div(long a, long b) throws
    java.rmi.RemoteException;
 }
```

Implement 1

The second step is to write the implementation for the remote service.

The implementation class may extend from the `java.rmi.server.UnicastRemoteObject` to link into the RMI system.

It must also provide an explicit default constructor throwing `RemoteException`. When this constructor calls super(), it activates code in `UnicastRemoteObject` that performs the RMI linking and remote object initialization.

Implement 2

```
public class CalculatorImpl extends
   java.rmi.server.UnicastRemoteObject implements
   Calculator {
        // Implementations must have an
        // explicit default constructor
        //  in order to declare the
        //   RemoteException exception
    public CalculatorImpl() throws
        java.rmi.RemoteException
    { super(); }
   public long add(long a, long b) throws
        java.rmi.RemoteException
    { return a + b; }
    ...
```

Lab Work: Implementation

1) Please write the rest of implementation for the Calculator interface.

Note: If your must extend some other classes other than extending from `UnicastRemoteObject`, the implementation may use the static `exportObject()` method of the `UnicastRemoteObject` instead.

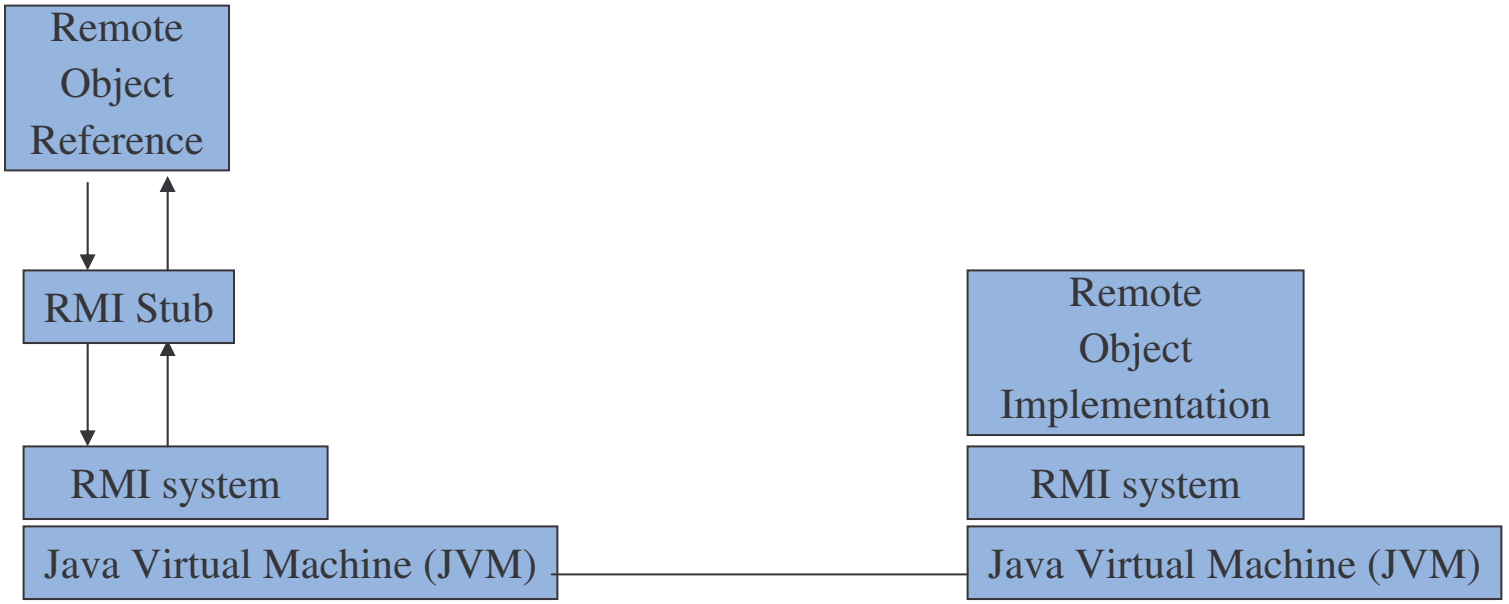
Be careful that you may need to synchronize some portions of your remotely available method. But it is not necessary for this example.

Stubs and Skeletons

To generate the Stub and Skeleton files, use the RMI complier, rmic as the following:

```
>rmic CalculatorImpl
```

The default option will create stubs/skeletons compatible with both JDK 1.1 and Java 2.



Host Server 1

Remote RMI services must be hosted in a server process. The following code is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;
public class CalculatorServer {
   public CalculatorServer() {
     try {
       Calculator c = new CalculatorImpl();

     Naming.rebind("rmi://localhost:1099/Calculator
     Service", c);
     } catch (Exception e) {
       System.out.println("Trouble: " + e);
     }
   }
```

Host Server 2

```
    public static void main(String args[]) {
      new CalculatorServer();
    }
}
```

Client 1

1) In the client's code, all you need to do is to lookup the object and use it's methods as local methods.
2) The client's code may look like the following:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {
    public static void main(String[] args) {
     try {
         Calculator c = (Calculator)Naming.lookup (
             "rmi://localhost/CalculatorService");
```

Client 2

```
            System.out.println( c.sub(4, 3) );
            System.out.println( c.add(4, 5) );
            System.out.println( c.mul(3, 6) );
            System.out.println( c.div(9, 3) );
        }
         catch (MalformedURLException murle) {}
         catch (RemoteException re){}
         catch (NotBoundException nbe){}
         catch (java.lang.ArithmeticException ae)
        {}
    }
}
```

Running the RMI System

1) Start up three consoles, one for the server, one for the client, and one for the RMIRegistry.
2) Type `rmiregistry` in the directory that contains the classes you have written.
3) In the server's console, type `java CalculatorServer` to start the server.
4) In the client's console, type java `CalculatorClient` to start the cleint program.
5) The output should look like:

```
1
9
18
3
```

Lab Work: RMI System

1) Please follow what we have discussed to develop a RMI server which hosts a service for calculating the square root of a number.

2) Compile your RMI server and generate the corresponding stub class.

3) Create a client to test the RMI service.

Passing Parameters

All parameters passed from an RMI client to an RMI server must either be serializable or be a remote object.

For serializable:

a) Data is extracted from the local object and sent across the network to the remote server.

b) Object is then reconstructed in the remote server.

c) Any changes to the object in the RMIServer will not be reflected in the object held in the RMI client and vice versa.

For a remote object:

a) Stub information, not a copy of data, is actually sent over RMI.

b) Any call made to the parameter object become a remote calls back to the actual object.

c) Changes made in one JVM are reflected in the original JVM.

Conditions for serializability

If an object is to be serialized:

a) The class must be declared as public

b) The class must implement Serializable

c) The class must have a default (no-argument) constructor

d) All fields of the class must be serializable: either primitive types or serializable objects

Remote interfaces and class

A Remote class has two parts:

a) The interface (used by both client and server):

1. Must be public
2. Must extend the interface `java.rmi.Remote`
3. Every method in the interface must declare that it throws `java.rmi.RemoteException` (other exceptions may also be thrown)

b) The class itself (used only by the server):

1. Must implement a `Remote` interface
2. Should extend `java.rmi.server.UnicastRemoteObject`
3. May have locally accessible methods that are not in its `Remote` interface

Security

Your program should guarantee that the classes that get loaded do not perform operations that they are not allowed to perform.

A more conservative security manager than the default should be installed. The following code should be added to the main method of the server and client program:

```
if (System.getSecurityManager() == null{
  System.setSecurityManager(new
      RMISecurityManager());
}
```

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) Implementing activatable RMI server
5) summary

Activatable Server

Enable server programs to wake up and start to run when they are needed.

Java RMI Activation System Daemon (`rmid`) is introduced to handle this task.

When a client requests a reference to the server from the `rmiregistry`, the `rmid` program, which holds the servers details, will be requested to start up the server and return the reference to the client. After that, the `rmiregistry` will be able to provide the reference of the server directly.

Activatable Server:Implementation

1) Subcalss the `java.rmi.activation.Activatable` class and implement the remote interface.
2) Implement the following constructor:

```
public Server(ActivationID id, MarshalledObject
     data) throws RemoteException{
      super(id,0);//register activatable object and
                  // export on anonymous port
      }
```

3) Create an activation description used by the `rmid` program.
4) Register the activation description with the `rmid` program.
5) Compile the activatable server with `javac` and `rmic` compiler.
6) The client program needs no modification.

Activatable Server:Setup 1

Before we can use the activatable server, you need to generate the activation description used by the `rmid` and register the description with the `rmid` program. We will group these processes into a utility program for illustration.

The structure of the utility program may look like this:

```
//1. Make the appropriate imports
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
public class SetupServer{
     public static void main(String args[]){
     try{
```

Activatable Server:Setup 2

```
//2. Declare for a security policy file
System.setSecurityManager(new RMISecurityManager());
Properties props =
    (Properties)System.getProperties();
props.put("java.security.policy",<location of
    security policy file>);
//3.Create an activation group description even there
//  is only one server
ActivationGroupDesc agd = new ActivationGroupDesc
    (props, null);
//4. Create a new activation group
ActivationGroupID agid =
    ActivationGroup.getSystem().registerGroup(agd);
```

Activatable Server:Setup 3

```
//5. Create the actual activation description
//   Don't miss the trailing slash (/)
String codebase = "file:/<location of server
     implementation file>/";
ActivationDesc desc = new ActivationDesc(agid,
"<name of the server>",codebase, null);
//6. Register the activation description to the rmid
//   program. Suppose the remote interface of the
//   server is RemoteInterface, the code will look
     like this:
RemoteInterface ref =
     (RemoteInterface)Activatable.register(desc);
```

Activatable Server:Setup 4

```
        //7. Bind the server in rmiregistry
        Naming.rebind("Server", ref);
        //8. Exit the setup program
        System.exit(0);
        }catch (Exception e){}
    }
}
```

Compile and Run 1

1) Compile all the classes use `javac`.
2) Run `rmic` on the implementation class
3) Start rmi registry use `rmiregistry`.
   a) make sure that the shell or window in which you will run the registry, either has no `CLASSPATH` set or has a `CLASSPATH` that does not include the path to any classes that you want downloaded to your client, including the stubs for your remote object implementation classes.
   b) If you start the `rmiregistry`, and it can find your stub classes in its `CLASSPATH`, it will ignore the server's `java.rmi.server.codebase` property, and as a result, your client(s) will not be able to download the stub code for your remote object.

Compile and Run 2

4) Start the activation daemon, `rmid`. Use `-J` option for a runtime flag.

```
rmid -J-Djava.security.policy=rmid.policy
```

The policy file may look like this:

```
grant{
permission com.sun.rmi.rmid.ExecOptionPermission
    "-Djava.*";
permission com.sun.rmi.rmid.ExecOptionPermission
    "-Dsun.*";
permission com.sun.rmi.rmid.ExecOptionPermission
    "-Dfile.*";
```

Compile and Run (3)

```
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dpath.separator=*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Duser.*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dos.*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dline.separator=*";
permission com.sun.rmi.rmid.ExecOptionPermission "-
    Dawt.*";
```

Compile and Run (4)

5) Running the setup program.

```
java
-Djava.security.policy
    =<full path of the policy file>
-Djava.rmi.server.codebase
    =file:/<location of the implementation stubs>/
<class name of the setup program>
```

Compile and Run (5)

6) Running the client program.

```
java
-Djava.security.policy
    =<full path of the policy file>
<client name>
```

For testing purpose, use the following `security.policy`:

```
grant {
    permission java.security.AllPermission "", "";
};
```

Exercise: Activatable RMI

1) Write a remote interface called `HelloInterface`.
2) Define a method `getMessage (String s)` in it. This method has a return type as a String. Don't forget to throw the proper exception.
3) Create a class named Server which has to be a subclass of the `java.rmi.activation.Activatalbe` class.
4) Implement the `getMessage` method which will append "Hello" the argument and return it as a `String`.
5) Create the Setup program for the server.
6) Create a client program which should look up the activatable server and use the `getMessage` method of it.
7) Compile and generate the corresponding files.
8) Run the client and check the result.

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) summary

Summary

In this session, we cover the followings:

1) Architecture of RMI
2) Building RMI system including both client and server
3) Implementation for activatable RMI server

Lab Work: Activatable RMI 1

1) Build an activatable RMIServer for a chatting system.

   As our focus will be on building an activatable RMIServer, all the codes for the client program and interfaces will be provided. You only need to implement the activatable RMIServer.

   The interface for the server is provided as `chat.interface.ChatServer.java`. You need to write an implementation class for it naming `chat.server.ChatServerImpl.java`.

Lab Work: Activatable RMI 2

2) Write a setup program for the server.

3) Write a class `chat.Message.java` to represent the message sending between the server and client. It is required to keep the information of the sender and the message content.

Lab Work: Activatable RMI 3

4) Test the program
   a) For testing the dynamic class downloading, please download a basic HTTP server form the following address:

```
java.sun.com/products/jdk/rmi/class-server.zip
```

   b) Extract the files and compile them using the following command:

```
javac -d . *.java
```

   c) After starting the `rmid`, you can start the HTTP server using the following command:

```
java examples.classServer.ClassFileServer <port number> <path for server's download directory>
```

Lab Work: Activatable RMI 4

d) The directory structure and .class files of the exercise should be copied to the server's download directory.

e) Start the setup program of your server. And for the codebase option, you should use http protocol instead of file this time.

```
-Djava.rmi.server.codebase =
        http://localhost:port/
```

f) You can use the security policy file in the example for testing purpose.

6) Examine the files in chat.client.packages. These are the classes for the client program. You will notice that the client needs no modification for dealing with the activatable RMIServer.

Lab Work: Activatable RMI 5

ChatServer interface:

```
package interfaces;
import java.rmi.*;
import chat.Message;

public interface ChatServer extends Remote {
   // register new ChatClient with ChatServer
   // In implementation, you may need to choose a
   // collection type for storing the connected client
   public void registerClient (ChatClient
   client)throws RemoteException;
   public void unregisterClient(ChatClient
   client)throws RemoteException;
```

Lab Work: Activatable RMI 6

```
    // post new message to ChatServer
    public void postMessage(Message message) throws
    RemoteException;
    // sending message to the clients that the server
    // is stopping
    public void stopServer()throws RemoteException;
}
```

Lab Work: Activatable RMI 7

ChatClient interface:

```
package interfaces;
import java.rmi.*;
import chat.Message;
public interface ChatClient extends Remote{
   //call back method allows the server to send
  //message to client
   public void deliverMessage(Message message)throws
   RemoteException;

   // method called when server shuting down
   public void serverStopping()throws
   RemoteException ;
}
```

Lab Work: Activatable RMI 8

MessageManager interface:

```
package interfaces;
public interface MessageManager {
   //connect to the server. Check the code in
   //RMIMessageManager for implementation
   public void connect (MessageListener listener)
   throws Exception;

   //disconnect to the server. Check the code in
   //RMIMessageManager for implementation
   public void disconnect (MessageListener
   listener)throws Exception;
```

Lab Work: Activatable RMI 9

```
  //send message to the server. Check the code in
  //RMIMessageManager for implementation
  public void sendMessage(String from, String
                message)throws Exception;


  //Registers a DisconnectListener to be notified
  //when the ChatServer disconnects the client.
  //Each ClientGUI will do the registration when
  //connection is made to the server.
  public void setDisconnectListener
  (DisconnectListener Listener);
}
```

Lab Work: Activatable RMI 10

MessageListenr interface:

```
package interfaces;

public interface MessageListener {
   //The inner class MyMessageListener defined inside
   //the ClientGUI implements this interface for
   //handling the message received.
   public void messageReceived (String from, String
   message);

}
```

Lab Work: Activatable RMI 11

DisconnectHandler interface:

```
package interfaces;
//An inner class DisconnectHandler actually
//implements this interface.
//The DisconnectHandler will update GUI in thread
//safe manner after received disconnect notifcation.
public interface DisconnectListener {

   public void serverDisconnected(String message);
}
```

Overview

1) introduction
2) RMI architecture
3) implementing and running RMI system
4) summary

Summary

In this session, we cover the followings:

1) Architecture of RMI
2) Building RMI system including both client and server
3) Implementation and environment set up procedure for activatable RMI server

CORBA

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

The OMG

1) Object Management Group
    a) Founded in 1989
    b) Not-for-Profit organization
    c) Vendor neutral
    d) ~800 member companies
2) Key Specifications
    a) UML
    b) CORBA

What is CORBA?

Defines a framework for object-oriented distributed applications.

Defined by a consortium of vendors under the direction of OMG.

Allows distributed programs in different languages and different platforms to interact as though they were in a single programming language on one computer.

Brings advantages of OO to distributed systems.

Allows you design a distributed application as a set of cooperating objects and to reuse existing objects.

Key CORBA Features

1) Application Development Transparencies
   a) Hardware/Language neutral
   b) Vendor neutral
   c) Object oriented paradigm

2) CORBA Interface Definition Language (IDL)

3) CORBAServices
   a) Naming
   b) Event
   c) Transaction
   d) Security

4) Interoperability

Object Request Broker (ORB)

1) A software component that mediates transfer of messages from a program to an object located on a remote host.

2) Hides underlying network communications from a programmer.

3) ORB allows you to create software objects whose member functions can be invoked by client programs located anywhere.

4) A server program contains instances of CORBA objects.

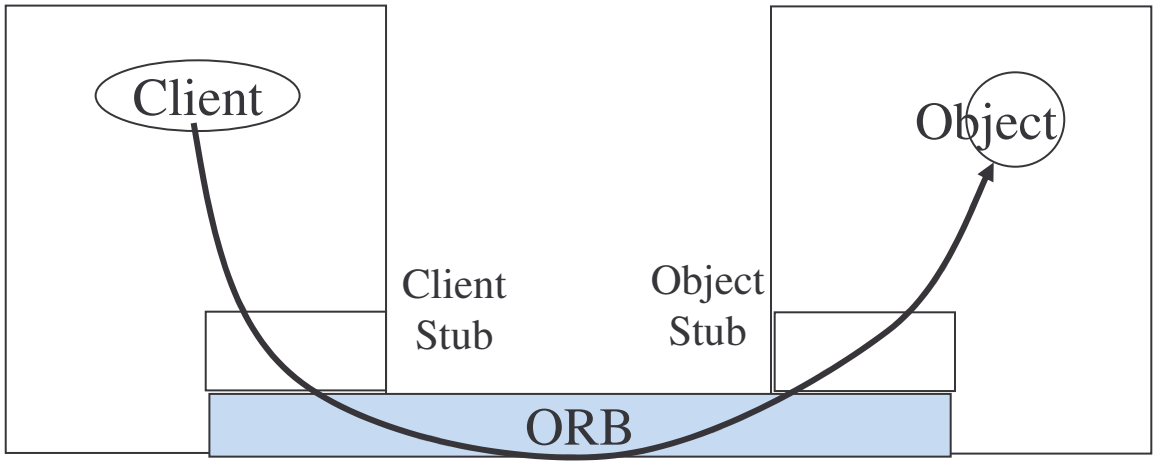
ORB: Conceptual View

1) When a client invokes a member function on a CORBA object, the ORB intercepts the function call.

2) ORB directs the function call across the network to the target object.

3) The ORB then collects the results from the function call returns these to the function call.

Implementation Details

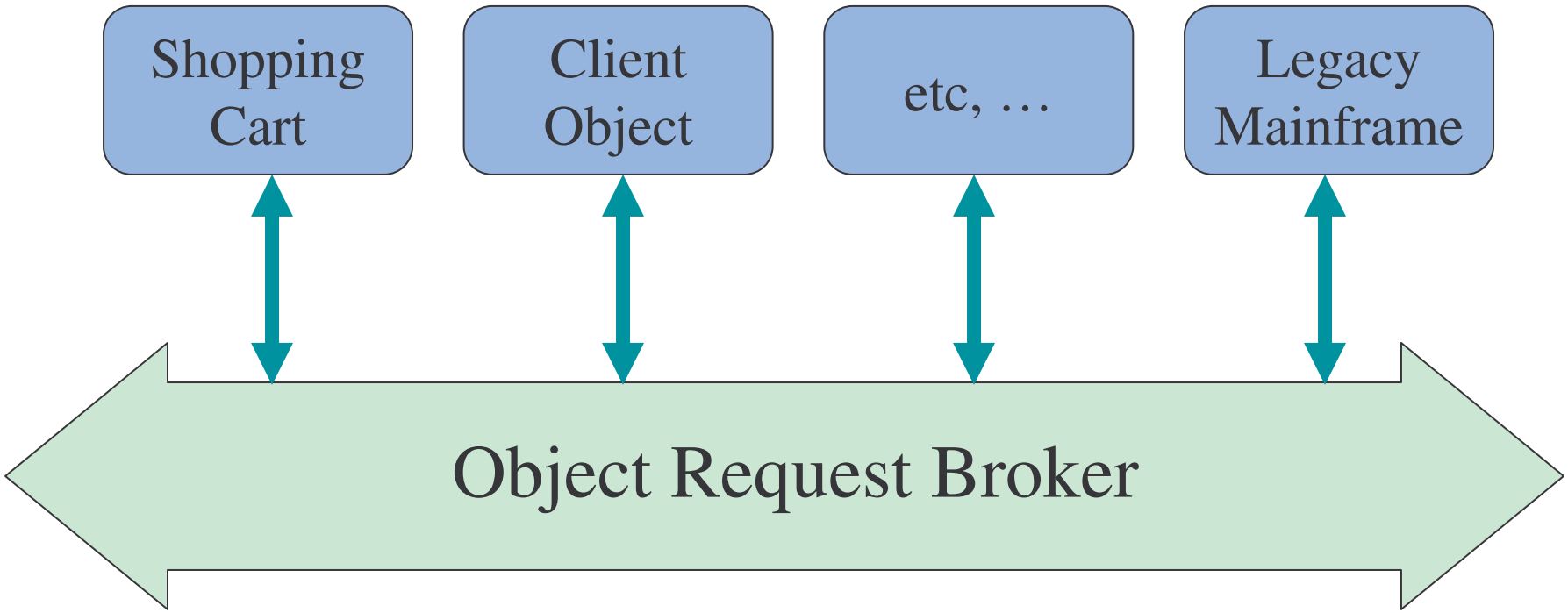


Access to the services provided by an Object

ORB : (Object-oriented middleware) Object Request Broker
ORB mediates transfer between client program and server object.

CORBA: A “Software Bus”

All CORBA objects connect to each other via ORB.



CORBA IDL

1) Interface Definition Language
   a) used to generate application code (stubs/skeletons)
   b) language neutral (Ada, C++, Java, …)

2) IDL is NOT a programming language
   a) lacks control structures
   b) provides no implementation details
   c) a specification

CORBA Objects and IDL

1) These are standard software objects implemented in any supported language including Java, C++ and Smalltalk.

2) Each CORBA object has a clearly defined interface specified in CORBA `interface definition language` (IDL).

3) The interface definition specifies the member functions available to the client without any assumption about the implementation of the object.

Client and IDL

1) To call a member function on a CORBA object the client needs only the object's IDL.

2) Client need not know the object's implementation, location or operating system on which the object runs.

Interface and Implementation

1) Interface and implementation can be in two different languages.

2) Interface abstracts and protects details (trade secrets) from client

3) Interface offers a means of expressing design without worrying about implementation.

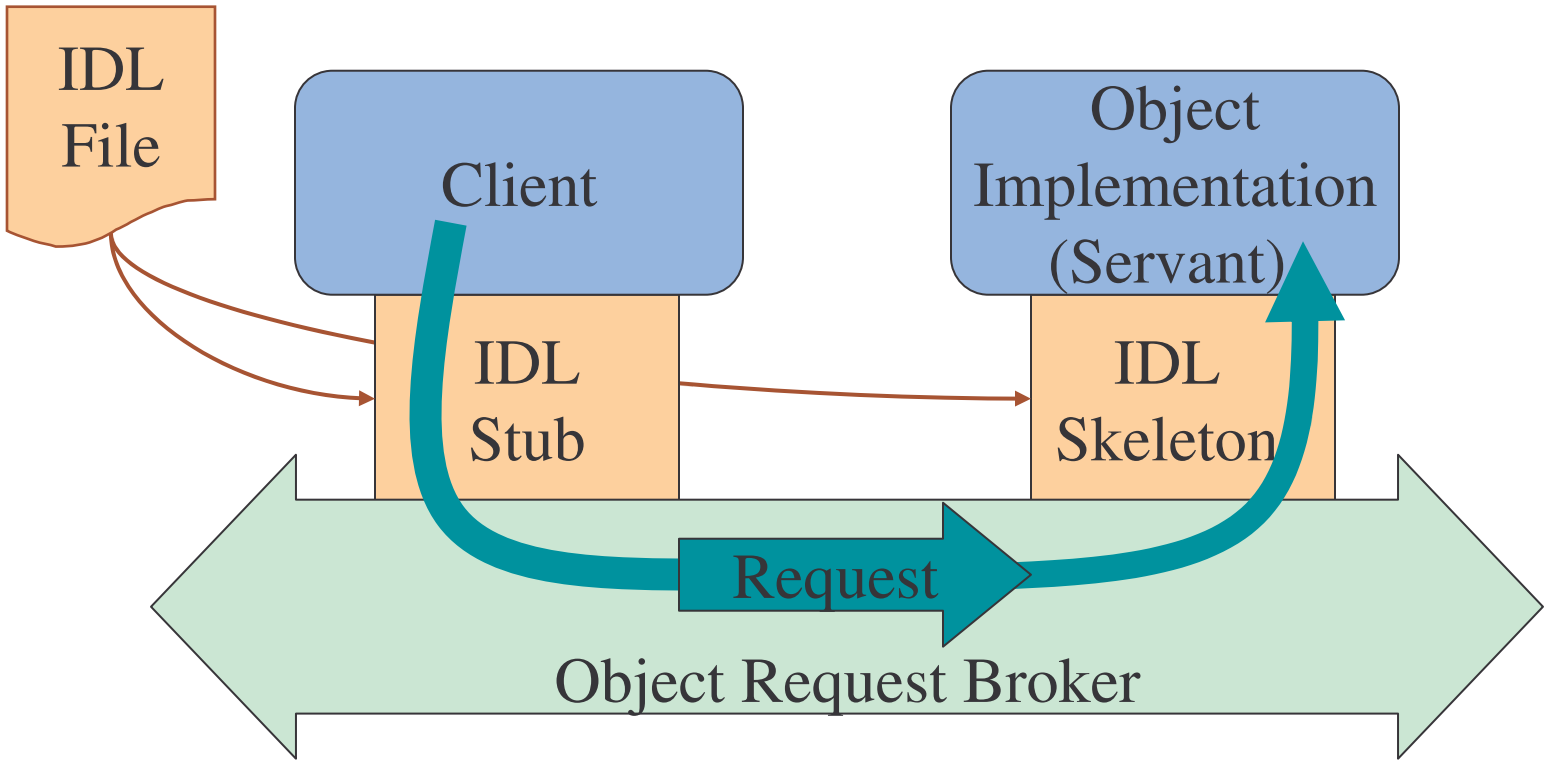
4) Interface is separated from implementation

Example: CORBA IDL

```
module BankExample {
   interface Account {
      exception BadCheck {
         float fee;
        };
      float deposit(in float amount);
      float writeCheck(in float amount)
         raises (BadCheck);
    };
   interface AccountManager {
      Account openAccount(in string name);
    };
};
```

CORBA Application Diagram

Objects are identified by Interoperable Object References (IORs)



CORBA Development Steps

1) Design the Application
2) IDL Specification
3) IDL Compilation (Code Generation)
4) Write the Client & Server implementation specific code
5) Compile the source code
6) Run the application

JavaIDL

Course Outline

1) introduction
2) streams
3) networking
4) database connectivity
5) architectures
    a) message-orientation
        1) javamail
        2) jms
    b) distributed objects
        1) rmi
        2) corba
        3) JavaIDL
6) summary

Modules and Interfaces

- IDL modules

```
module MyStuff
{
 …
};
```

  - Provide a namespace to group a set of interfaces. Names are scoped using the "::" operator.

- IDL interface

```
interface Foo { };
```

- Java packages

```
package MyStuff;
…
```

  - Provide Internet-wide namespaces. Scoped using the "." operator.

- Java interface

```
public interface Foo
  {…};
```

IDL to Java: Parameters

1) Java uses pass-by-value for parameters (including parameters that are references)

2) IDL has `in`, `out` and `inout` types of parameters

3) The `in` parameter type maps to a normal Java parameter since it does not need to be changed

4) `out` and `inout` parameter types are passed via instances of Java `Holder` classes

Holder Classes

1) `Holder` classes encapsulate the real value of a parameter which can then be reassigned to

   a) a member "value"

```
// generated java

package Example;
public interface Modes {
  int operation (IntHolder outArg,
                 IntHolder inoutArg);
}
```

```
// user code

// select a target object
Example.Modes target = …;
// prepare to receive out
IntHolder outHolder = new IntHolder();
// set up the in side of the inout
IntHolder inoutHolder = new IntHolder (131);
// make the invocation
int result = target.operation (
                    outHolder, inoutHolder);
// use the values of holders
…outHolder.value…
…inoutHoulder.value…
```

Helper Classes

1) all user-defined IDL types have a Helper Java class
2) insert and extract Any
3) get CORBA::TypeCode of the type
4) narrow (for interfaces only)

IDL to Java: Attributes

- IDL attributes

```
attribute long assignable;
readonly attribute long
                    fetchable;
```

- Java "`get`" and "`set`" methods

```
public int assignable();
public void assignable(int
                       val);
public int fetchable();
```

Basic Types

IDL Type	Java Type	Exception
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
octet	byte	
string	java.lang.String	CORBA::MARSHAL…
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

IDL to Java: Basic Types

- IDL char

```
const char MyChar = 'A';
```

- Java char

```
final public class MyChar
{
final public static char
value = (char)'A';
}
```

IDL to Java: Basic Types

- IDL octet
  ```
  void foo(in octet x);
  ```
- Java byte
  ```
  public void foo(byte x);
  ```

IDL to Java: Basic Types

- IDL boolean

```
const boolean truth =
TRUE;
```

- IDL constants `TRUE` and `FALSE`

- Java boolean

```
final public class truth
{
 final public static
boolean value = true;
}
```

- Java constants `true` and `false`

IDL to Java: Basic Types

- IDL string

```
const string MyString =
"Hello World";
```

- Java java.lang.String

```
final public class MyString
{
 final public static String
    value = "Hello World";
}
```

IDL to Java: Basic Types

- IDL integers
  - (unsigned) short
  - (unsigned) long
  - (unsigned) long long?

```
const unsigned short
MyUnsignedShort = 1580;
```

- Java integers
  - short
  - int
  - long

```
final public class
MyUnsignedShort
{
    final public static
short value =
(short)1580;
}
```

IDL to Java: Basic Types

- IDL floating-point float, double

```
const double MyDouble =
1.23456789;
```

- Java floating-point *float, double*

```
final public class
MyDouble
{
    final public static
    double value =
       (double)1.23456789;
}
```

IDL to Java: Constructed Types

- IDL Enum

```
enum MyEnum
{none,first,second};
```

- Java class

```
final public class MyEnum
{
 final public static int
none = 0;
 final public static int
first = 1;
 final public static int
second = 2;
 final public static int
narrow(int i) throws
CORBA.BAD_PARAM {..};
}
```

- the narrow method is for checking enum values

IDL to Java: Constructed Types

- IDL *struct*

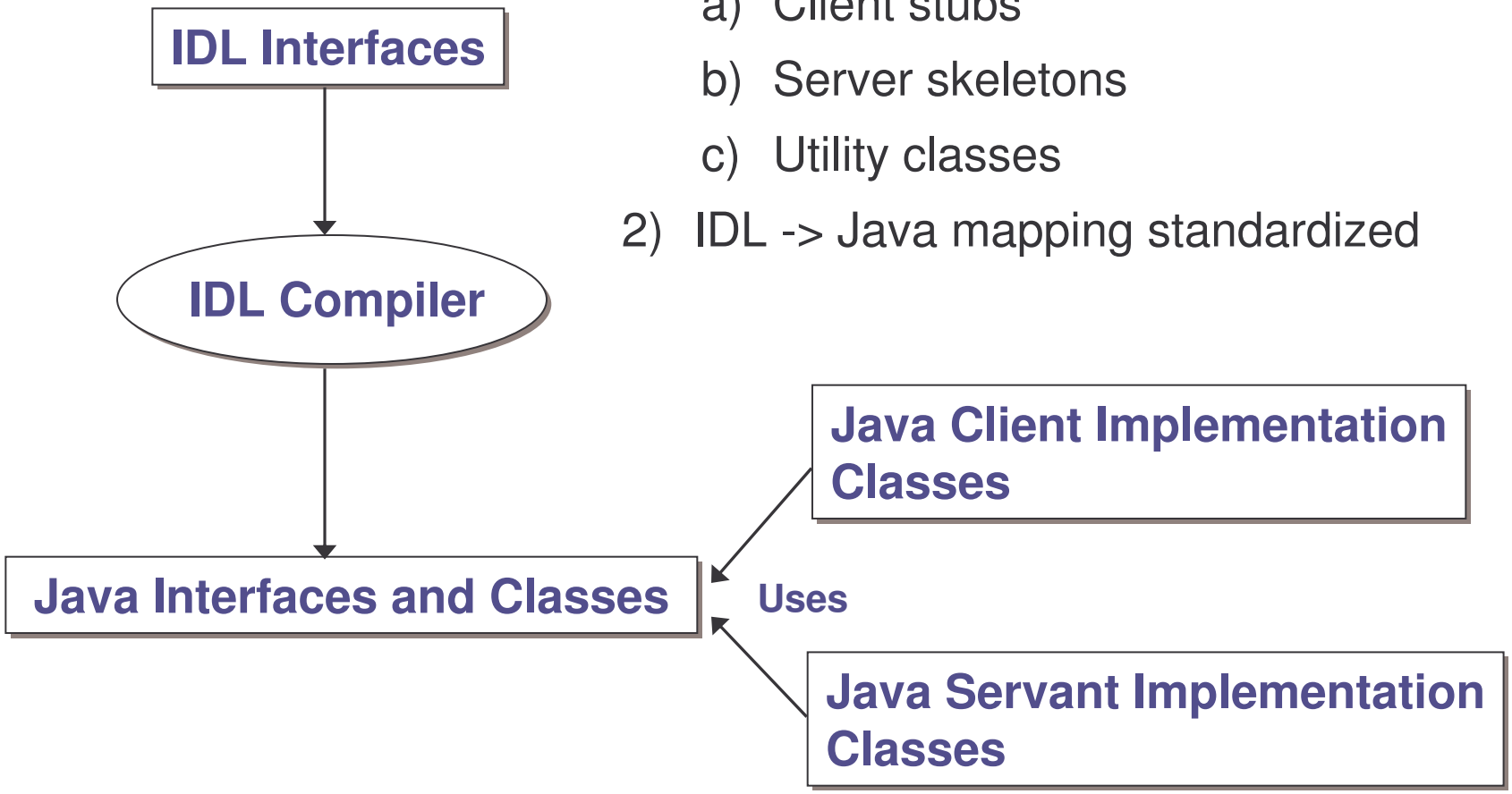
```
struct MyStruct
{
 long mylong;
 string mystring;
};
```

- Java *class*

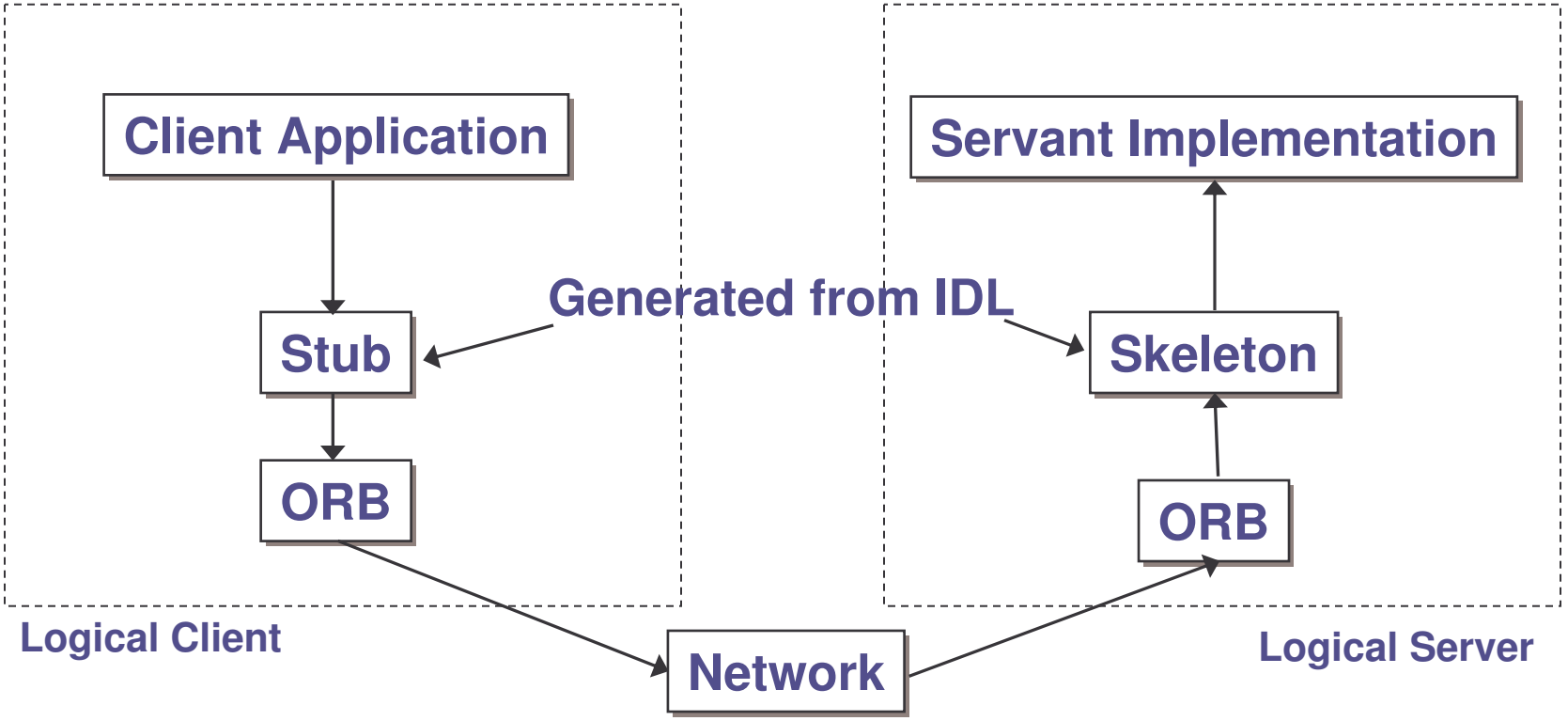
```
final public class
MyStruct
{
 public MyStruct(int
_mylong, String
_mystring) {…};
 public MyStruct() {…};

 public int mylong;
 public String mystring;
}
```

The Big Picture

1) Compiler outputs:
   a) Client stubs
   b) Server skeletons
   c) Utility classes
2) IDL -> Java mapping standardized



Big Picture (Invocation)



Example: File Transfer

This presents a file download CORBA application

The client request for a file and the server in turn sends the file to the client which then saves it on the local machine.

There are a number of steps involved:

1) Define an interface in IDL
2) Map the IDL interface to Java (done automatically)
3) Implement the interface
4) Develop the server
5) Develop a client
6) Run the naming service, the server, and the client.

Step 1: Define the IDL Interface 1

The first thing to do is to determine the operation that the server will support.

In this application, the client will invoke a method to download a file.

Here is the code.

```
interface FileInterface {
   typedef sequence<octet> Data;
   Data downloadFile(in string fileName);
};
```

Save this file as `FileInterface.idl`

Step 1: Define the IDL Interface 2

`Data` is a new type introduced using the `typedef` keyword.

A `sequence` in IDL is similar to an array except that a sequence does not have a fixed size

An `octet` is an 8-bit quantity that is equivalent to the Java type `byte`

The `downloadFile` method takes one parameter of type `string` that is declared `in`.

IDL defines three parameter-passing modes: `in` (for input from client to server), `out` (for output from server to client), and `inout` (used for both input and output).

Step 2: Map IDL to Java

Once you finish defining the IDL interface, you are ready to map the IDL interface to Java.

Java comes with the `idlj` compiler, which is used to map IDL definitions into Java declarations and statements.

The `idlj` compiler accepts options that allow you to specify if you wish to generate client stubs, server skeletons, or both.

let's compile the `FileInterface.idl` and generate both client and server-side files.

Step 3: Compile the IDL Interface

1) Compile the IDL Interface using:

```
prompt> idlj -oldImplBase -fall FileInterface.idl
```

2) IDL compilation produces many java constructs (interfaces and classes).

3) Each one is placed with a `<filename>.java`

Files Generated by IDL Compiler

1) Each file generated contains a Java interface or class scoped within a package.
2) This package is physically located in a directory of the same name according to Java conventions.

Client Side Files

1) `FileInterface.java` - an interface to provide a client a view of the methods in the IDL.

2) `_FileInterfaceStub.java` - a Java class that implements the methods defined in interface Grid. Provides functionality that allows client method invocations to be forwarded to a server.

Server Side Files

1) `_FileInterfaceImplBase.java` - an abstract Java class that allows server-side developers to implement the `FileInterface` interface.

2) `Other files: FileInterfaceHelper.java,`
`FileInterfaceHolder.java,`
`FileInterfaceOperations.java,`

Step 4: Implement the Interface 1

Provide an implementation to the `downloadFile()` method. This implementation is known as a servant.

```
import java.io.*;
public class FileServant extends _FileInterfaceImplBase
                                                        {
   public byte[] downloadFile(String fileName){
      File file = new File(fileName);
      byte buffer[] = new byte[(int)file.length()];
      try {
         BufferedInputStream input = new
         BufferedInputStream(new
                              FileInputStream(fileName));
         input.read(buffer,0,buffer.length);
         input.close();
```

Step 4: Implement the Interface 2

```
        } catch(Exception e) {
            System.out.println("FileServant Error:
                                        "+e.getMessage());
            e.printStackTrace();
        }
        return(buffer);
    }
}
```

Step 5: Develop the Server 1

The next step is developing the CORBA server.

Write `FileServer` class that implements a CORBA server that does the following:

1) Initializes the ORB
2) Creates a `FileServant` object
3) Registers the object in the CORBA Naming Service (COS Naming)
4) Prints a status message
5) Waits for incoming client requests

Step 5: Develop the Server 2

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class FileServer {
   public static void main(String args[]) {
      try{
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);
         // create the servant and register it with ORB
         FileServant fileRef = new FileServant();
         orb.connect(fileRef);
```

Step 5: Develop the Server 3

```
// get the root naming context
org.omg.CORBA.Object objRef =
 orb.resolve_initial_references("NameService");
NamingContext ncRef =
          NamingContextHelper.narrow(objRef);
// Bind the object reference in naming
NameComponent nc = new
          NameComponent("FileTransfer", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, fileRef);
System.out.println("Server started....");
```

Step 5: Develop the Server 4

```
            // Wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized(sync){
               sync.wait();
            }
        } catch(Exception e) {
            System.err.println("ERROR: " + e.getMessage());
            e.printStackTrace(System.out);
        }
    }
}
```

Step 6: Develop the Client 1

The next step is developing the CORBA client.

Write `FileClient` class that implements a CORBA client that does the following:

1) Initializes the ORB
2) Retrieve the `FileTransfer` service from the naming server
3) Call the `downloadFile` method.

Step 6: Develop the Client 2

```
import java.io.*;
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class FileClient {
   public static void main(String argv[]) {
      try {
         // create and initialize the ORB
         ORB orb = ORB.init(argv, null);
         // get the root naming context
          org.omg.CORBA.Object objRef =
         orb.resolve_initial_references("NameService");
```

Step 6: Develop the Client 2

```
NamingContext ncRef =
          NamingContextHelper.narrow(objRef);
NameComponent nc = new
          NameComponent("FileTransfer", " ");
// Resolve the object reference in naming
NameComponent path[] = {nc};
FileInterfaceOperations fileRef =
     FileInterfaceHelper.narrow(ncRef.resolve(path));


if(argv.length < 1) {
    System.out.println("Usage: java FileClient
                                        filename");
}
```

Step 6: Develop the Client 2

```
    // save the file
    File file = new File(argv[0]);
    byte data[] = fileRef.downloadFile(argv[0]);
    BufferedOutputStream output = new
    BufferedOutputStream(new FileOutputStream(argv[0]));
    output.write(data, 0, data.length);
    output.flush();
    output.close();
  }catch(Exception e) {
      System.out.println("FileClient Error: " +
                                          e.getMessage());
      e.printStackTrace();
   }
}}
```

Step 7: Run the Application

1) Running the the CORBA naming service.

```
prompt> tnameserv -ORBInitialPort 2500
```

2) Start the server

```
prompt> java FileServer -ORBInitialPort 2500
```

3) Run the client

```
prompt> java FileClient c:\hello.txt -ORBInitialHost
          mycomputerName  -ORBInitialPort  2500
```

Summary

1) We introduced general operation of CORBA.
2) Also details of specifying a client, server application, compiling them and registering and running.
3) You will have to configure your system before you try to do these steps.

Project Exercise

1) Implement the controller of your project as a distributed object by using either RMI or JavaIDL.
2) Device an approach through which the controller would locate a requested business object to handle a particular request and also invoke the appropriate operation requested for.
3) Persist your data using mySQL database engine.
4) See how you can make use of Java Message Service in your project.