Java Programming/Print version

Wikibooks.org

March 24, 2011.

Contents

1 About This Book **1**
- 1.1 Who Should Read This Book . . . 1
- 1.2 How this book is evolving . . . 2

2 History **3**
- 2.1 The Green team . . . 4
- 2.2 Reshaping thought . . . 5
- 2.3 The demise of an idea, birth of another . . . 7
- 2.4 Recent history . . . 8
- 2.5 Versions . . . 8
- 2.6 Citations . . . 11

3 The Java Platform **13**
- 3.1 Java technologies . . . 13
- 3.2 Java Runtime Environment . . . 14
- 3.3 Languages . . . 15
- 3.4 Similar Platforms . . . 15

4 Java Programming Environment **17**
- 4.1 The Java Compiler . . . 17
- 4.2 The Java Runtime Environment . . . 19
- 4.3 Other JDK tools . . . 20

5 Installation **25**
- 5.1 Windows . . . 25
- 5.2 Ubuntu Linux . . . 26
- 5.3 External Links . . . 26

6 Compilation **27**
- 6.1 Compiling to bytecode . . . 27
- 6.2 Automatic Compilation of Dependent Classes . . . 27
- 6.3 Packages, Subdirectories, and Resources . . . 28
- 6.4 Filename Case . . . 29
- 6.5 Compiler Options . . . 29
- 6.6 Additional Tools . . . 29
- 6.7 JBuilder . . . 29
- 6.8 JCreator . . . 30
- 6.9 Eclipse . . . 30
- 6.10 NetBeans . . . 31
- 6.11 BlueJ . . . 31
- 6.12 Kawa . . . 31

6.13 Ant . . . 31
6.14 The JIT compiler . . . 32

7 Execution **33**
7.1 JSE code execution . . . 33
7.2 J2EE code execution . . . 36
7.3 Jini . . . 41

8 First Java Program **43**
8.1 Hello World . . . 43
8.2 Modifying the Program . . . 44
8.3 Common Problems . . . 45
8.4 The Next Step . . . 45

9 Understanding a Java Program **47**
9.1 The Distance Class: Intent, Source, and Use . . . 47
9.2 Detailed Program Structure and Overview . . . 48
9.3 Comments in Java programs . . . 56

10 Syntax **57**
10.1 Unicode . . . 58
10.2 Literals . . . 59
10.3 Blocks . . . 60
10.4 Whitespaces . . . 60
10.5 Required Whitespace . . . 61
10.6 Indentation . . . 61

11 Statements **63**
11.1 What exactly are statements? . . . 63
11.2 Where do you find statements . . . 63
11.3 Variables . . . 64
11.4 Data types . . . 64
11.5 Whole numbers and floating point numbers . . . 64
11.6 Assignment statements . . . 65
11.7 Program Control Flow . . . 67
11.8 Statement Blocks . . . 67
11.9 Branching Statements . . . 67
11.10 Iteration Statements . . . 71
11.11 The continue and break statements . . . 74

12 Classes, Objects and Types **75**
12.1 Objects and Classes . . . 75
12.2 Instantiation and Constructors . . . 75
12.3 Type . . . 76
12.4 Multiple classes in a Java file . . . 77
12.5 External links . . . 78

13 Packages **79**
13.1 Java Package / Name Space . . . 79
13.2 Wildcard imports . . . 80

13.3 Importing packages from .jar files . . . 80
13.4 Class Loading / Name Space . . . 81

14 Nested Classes **83**
14.1 Nest a class inside a class . . . 83
14.2 Nest a class inside a method . . . 84
14.3 Anonymous Classes . . . 84

15 Access Modifiers **87**
15.1 Access modifiers . . . 87

16 Methods **89**
16.1 Method Definition . . . 89
16.2 Method Overloading . . . 89
16.3 Method Overriding . . . 91
16.4 Parameter Passing . . . 92
16.5 Functions . . . 93
16.6 Return Parameter . . . 93
16.7 Special method, the Constructor . . . 95
16.8 Static Method . . . 96
16.9 External links . . . 97

17 Primitive Types **99**

18 Types **101**
18.1 Data Types in Java . . . 101
18.2 About Java Types . . . 102
18.3 Examples of Types . . . 102
18.4 Array Types . . . 104
18.5 Primitive Data Types . . . 104
18.6 Data Conversion (Casting) . . . 105
18.7 Autoboxing/unboxing . . . 106

19 java.lang.String **107**
19.1 java.lang.String . . . 107
19.2 Using StringBuffer/StringBuilder to concatenate strings . . . 108
19.3 Comparing Strings . . . 109
19.4 Splitting a String . . . 110
19.5 Creating substrings . . . 111
19.6 Modifying String cases . . . 112
19.7 See also . . . 112

20 Arrays **113**
20.1 Intro to Arrays . . . 113
20.2 Array Fundamentals . . . 113
20.3 Two-Dimensional Arrays . . . 113
20.4 Multidimensional Array . . . 114

21 Data and Variables **115**
21.1 Strong Typing . . . 116

21.2 Case Conventions . . . 116
21.3 Scope . . . 116

22 Generics **119**
22.1 What are Generics? . . . 119
22.2 Introduction . . . 120
22.3 Note for C++ programmers . . . 121
22.4 Class<T> . . . 122
22.5 Variable Argument . . . 123
22.6 Wildcard Types . . . 124

23 Defining Classes **127**
23.1 Fundamentals . . . 127

24 Creating Objects **131**
24.1 Introduction . . . 131
24.2 Creating object with the **new** keyword . . . 131
24.3 Creating object by cloning an object . . . 132
24.4 Creating object receiving from a remote source . . . 134

25 Interfaces **137**
25.1 Interfaces . . . 137
25.2 External links . . . 138

26 Using Static Members **139**
26.1 What does static mean? . . . 139
26.2 What can it be used for? . . . 139
26.3 Danger of static variables . . . 140
26.4 External links . . . 140

27 Destroying Objects **141**
27.1 finalize() . . . 141

28 Overloading Methods and Constructors **143**

29 Arrays **145**
29.1 Intro to Arrays . . . 145
29.2 Array Fundamentals . . . 145
29.3 Two-Dimensional Arrays . . . 145
29.4 Multidimensional Array . . . 146

30 Collection Classes **147**
30.1 Introduction to Collections . . . 147
30.2 Generics . . . 148
30.3 Collection or Map . . . 149
30.4 Set or List or Queue . . . 152
30.5 Map Classes . . . 159
30.6 Thread Safe Collections . . . 161
30.7 Classes Diagram (UML) . . . 161
30.8 External links . . . 163

31 Throwing and Catching Exceptions **165**
31.1 Catching Matching rules . . . 165
31.2 Example of handling exceptions . . . 169
31.3 Application Exceptions . . . 170
31.4 Runtime Exceptions . . . 171
31.5 Main Exception Classes . . . 172
31.6 See Also . . . 174
31.7 Minimize the use of the keyword 'null' in assignment statements . . . 174
31.8 Minimize the use of the new Type[int] syntax for creating arrays of objects . . . 175
31.9 Check all references obtained from 'untrusted' methods . . . 175
31.10 Comparing string variable with a string literal . . . 175
31.11 See also . . . 176

32 Links **177**
32.1 External References . . . 177
32.2 External links . . . 177

33 License **181**

34 GNU Free Documentation License **183**

35 Authors **185**

List of Figures **191**

1 About This Book

Java has gained a considerable foothold in the world of programming since its inception in 1995. Since then, the way people code has been gradually evolving into a more standardized manner, rendering Java programming as a pivotal first step into the realm of software code that is OBJECT ORIENTED PROGRAMMING[1].

In an effort to enable software enthusiasts to program in Java as their first language, this book finds itself a mission: to deliver as much information as is possible using Java as a primary programming language. Because of that this book can be considered as a reference book of Java and its related technologies. Hosting the book on WIKIBOOKS[2] means that this book will constantly be evolving into a more comprehensive text as time goes by.

1.1 Who Should Read This Book

This book is for programmers who wish to learn how to program with Java.

This book does not teach general programming constructs: we assume you know what variables are, what assignment is, etc. We also assume you know the basics of Object Oriented programming.

If you need to brush up on these topics, a suggested place to start is SUBJECT:COMPUTER PROGRAMMING CONCEPTS[3] which has a book on OBJECT ORIENTED PROGRAMMING[4].

Also there are many good commercial books that can be used to learn Java (see the external links). This book does not intend to replace them. Rather this book can be used to supplement them, during the learning of Java. It always helps the learning process to hear the same information in different ways, and this book represents one of the ways. And when the Java language is mastered, this book can be used as a reference. When you want to look something up quickly, instead of searching in a printed book, you can look it up in this book.

This book can also be used by advanced Java programmers either by contributing or using this book as a reference. Also there is an Advanced Topics section for advanced Java programmers.

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/OBJECT%20ORIENTED%20PROGRAMMING
2 HTTP://EN.WIKIBOOKS.ORG/WIKI/MAIN%20PAGE
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/SUBJECT%3ACOMPUTER%20PROGRAMMING%20CONCEPTS
4 HTTP://EN.WIKIBOOKS.ORG/WIKI/OBJECT%20ORIENTED%20PROGRAMMING

1.2 How this book is evolving

Any Wikibooks user can edit or modify this book. With readily available information on the internet, this book is expected to include all aspects of the Java Programming language albeit it is taken into consideration that not everything is crammed into the book.

2 History

On 23 MAY[1] 1995[2], JOHN GAGE[3], the director of the Science Office of the SUN MICROSYSTEMS[4] along with MARC ANDREESEN[5], co-founder and executive vice president at NETSCAPE[6] announced to an audience of SunWorldTM that Java technology wasn't a myth and that it was a reality and that it was going to be incorporated into NETSCAPE NAVIGATOR[7].[8]

At the time the total number of people working on Java were less than 30 people.<ref name="CITEREFEarlyYearsSun1"/> This team would shape the future in the next decade and no one had any idea as to what was in store. From being the mind of an unmanned vehicle on MARS[9] to the operating environment on most of the consumer electronics, e.g., cable set-top boxes,VCR's, toasters, and also for PERSONAL DATA ASSISTANTS[10] (PDAs).[11] Java has come a long way from its inception. Let's see how it all began.

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/23%20MAY
2 HTTP://EN.WIKIPEDIA.ORG/WIKI/1995
3 HTTP://EN.WIKIPEDIA.ORG/WIKI/JOHN%20GAGE
4 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUN%20MICROSYSTEMS
5 HTTP://EN.WIKIPEDIA.ORG/WIKI/MARC%20ANDREESEN
6 HTTP://EN.WIKIPEDIA.ORG/WIKI/NETSCAPE
7 HTTP://EN.WIKIPEDIA.ORG/WIKI/NETSCAPE%20NAVIGATOR
8 UNKNOWN TEMPLATE "cite web"
9 HTTP://EN.WIKIPEDIA.ORG/WIKI/MARS
10 HTTP://EN.WIKIPEDIA.ORG/WIKI/PERSONAL%20DATA%20ASSISTANTS
11 UNKNOWN TEMPLATE "cite web"

2.1 The Green team

Figure 1: James Gosling, architect and designer of the compiler for the Java technology

Behind closed doors, a project was initiated in December of 1990[12], whose aim was to create a programming tool that could render obsolete the C and C++ programming languages. Engineer Patrick Naughton had become extremely frustrated with the state of Sun's C++ and C APIs (application programming interfaces) and tools. While he was considering to move towards NeXT[13], he was offered a chance to work on new technology and the "Stealth Project" was started, a secret nobody but he knew.

This Stealth Project was later named the "Green Project" when James Gosling[14] and Mike Sheridan[15] joined Patrick.<ref name="CITEREFEarlyYearsSun1"/> Over the period of time that the Green Project teethed, the prospects of the project started becoming clearer to the engineers working on it. No longer was it's aim to create a new language far superior to the present ones, but it aimed to target the language to devices other than the computer.

Staffed at 13 people, they began work in a small office on Sand Hill Road[16] in Menlo Park, California[17]. This team would be called "Green Team" henceforth in time. The project they underwent was chartered by Sun Microsystems to anticipate and plan for the "next-wave" in computing. For the team, this meant at least one significant trend, that of the convergence of digitally controlled consumer devices and computers.<ref name="CITEREFEarlyYearsSun1"/>

12 http://en.wikipedia.org/wiki/1990
13 http://en.wikipedia.org/wiki/NeXT
14 http://en.wikipedia.org/wiki/James%20Gosling
15 http://en.wikipedia.org/wiki/Mike%20Sheridan
16 http://en.wikipedia.org/wiki/Sand%20Hill%20Road
17 http://en.wikipedia.org/wiki/Menlo%20Park%2C%20California

2.2 Reshaping thought

The team started thinking of replacing C++ with a better version, a faster version, a responsive version. But the one thing they hadn't thought of, as of yet, was that the language they were aiming for, had to be developed for an EMBEDDED SYSTEM[18] with limited resources. An **embedded system** is a computer system scaled to a minimalistic interface demanding only a few functions from its design. For such a system, C++ or any successor would seem too large as all the languages at the time demanded a larger footprint than what was desired. And, other than this, the language lacked some other important features as well. The team thus had to think in a different way to go about solving all these problems.

Co-founder of Sun Microsystems, Bill Joy, envisioned a language combining the power of MESA[19] and C in a paper he wrote for the engineers at Sun named *Further*. Gathering ideas, Gosling began work on enhancing C++ and named it "C++ ++ --", a pun on the evolutionary structure of the language's name. The ++ and -- meant, *putting in* and *taking out stuff*. He soon abandoned the name and called it **Oak**<ref name="CITEREFEarlyYearsSun1"/> after the tree that stood outside his office.

Table 1: Who's who of the Java technology<ref name="CITEREFEarlyYearsSun1"/> Has worked for GT (Green Team), FP (FirstPerson) and JP (Java Products Group) **Name**	**GT**	**FP**	**JP**	**Details**
Lisa Friendly		✓Yes	✓Yes	FirstPerson employee and member of the Java Products Group
John Gage				Science Office (Director), Sun Microsystems
JAMES GOSLING[20]	✓Yes	✓Yes	✓Yes	Lead engineer and key architect of the Java technology

18 HTTP://EN.WIKIPEDIA.ORG/WIKI/EMBEDDED%20SYSTEM
19 HTTP://EN.WIKIPEDIA.ORG/WIKI/MESA%20%28PROGRAMMING%20LANGUAGE%29
20 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAMES%20GOSLING

Table 1: Who's who of the Java technology<ref name="CITEREFEarlyYearsSun1"/> Has worked for GT (Green Team), FP (FirstPerson) and JP (Java Products Group) **Name**	**GT**	**FP**	**JP**	**Details**
Bill Joy				Co-founder and VP, Sun Microsystems; Principal designer of the UC Berkeley, version of the UNIX® OS
Jonni Kanerva		✓Yes		Java Products Group employee, author of The Java FAQ1
Tim Lindholm		✓Yes	✓Yes	FirstPerson employee and member Java Products Group
Scott McNealy				Chairman, President, and CEO of Sun Microsystems
Patrick Naughton	✓Yes	✓Yes		Green Team member, FirstPerson co-founder
George Paolini				Corporate Marketing (Director), Sun's Java Software Division
Kim Polese		✓Yes		FirstPerson product marketing

Table 1: Who's who of the Java technology<ref name="CITEREFEarlyYearsSun1"/> Has worked for GT (Green Team), FP (FirstPerson) and JP (Java Products Group) **Name**	**GT**	**FP**	**JP**	**Details**
Lisa Poulson				Original director of public relations for Java technology (Burson-Marsteller)
Wayne Rosing		✓Yes		FirstPerson President
Eric Schmidt				Former Sun Microsystems Chief Technology Officer
Mike Sheridan	✓Yes			Green Team member

2.3 The demise of an idea, birth of another

By now, the work on Oak had been significant but come the year 1993[21], people saw the demise of set-top boxes, interactive TV and the PDAs. A failure that completely ushered the inventors' thoughts to be reinvented. Only a miracle could make the project a success now. And such a miracle awaited anticipation.

NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS[22] (NCSA) had just unveiled its new commercial web browser for the internet the previous year. The focus of the team, now diverted towards where they thought the "next-wave" of computing would be – the internet. The team then divulged into the realms of creating the same embeddable technology to be used in the web browser space calling it **AN APPLET**[23] – *a small application.* The team now needed a proper identity and they decided on naming the new technology they created Java ushering a new generation of products for the internet boom. A by-product of the project was a cartoon named "DUKE[24]" created by Joe Parlang which became its identity then.

21 HTTP://EN.WIKIPEDIA.ORG/WIKI/1993
22 HTTP://EN.WIKIPEDIA.ORG/WIKI/NATIONAL%20CENTER%20FOR%20SUPERCOMPUTING%20APPLICATIONS
23 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLET
24 HTTP://EN.WIKIPEDIA.ORG/WIKI/DUKE%20%28MASCOT%29

Finally at the SunWorldTM conference, Andreesen unveiled the new technology to the masses. Riding along with the explosion of interest and publicity in the Internet, Java quickly received widespread recognition and expectations grew for it to become the dominant software for browser and consumer applications.<ref name="CITEREFClarkLindseyHistory1"/>

2.4 Recent history

Initially Java was owned by Sun Microsystems, but later it was released to open source; the term Java was a trademark of Sun Microsystems. Sun released the source code for its HotSpot Virtual Machine and compiler in November 2006, and the most of the source code of the class library in May 2007. Some parts were missing because they were owned by third parties, not by Sun Microsystems. The released parts were published under the terms of the GNU GENERAL PUBLIC LICENSE[25], a free software license.

2.5 Versions

Unlike C and C++, Java's growth is pretty recent. Here, we'd quickly go through the development paths that Java took with age.

Figure 2: Development of Java over the years. From version 1.0 to version 1.7, Java has displayed a steady growth.

2.5.1 Initial Release (versions 1.0 and 1.1)

Introduced in 1996 for the SOLARIS[26], WINDOWS[27], MAC OS[28] Classic and LINUX[29], Java was initially released as the Java Development Kit 1.0 (JDK 1.0). This included the Java runtime (the virtual machine and the class libraries), and the development tools (e.g., the Javac compiler). Later, Sun also provided a runtime-only package, called the Java Runtime Environment (JRE). The first name stuck, however, so usually people refer to a particular version of Java by its JDK version (e.g., JDK 1.0).

25 HTTP://EN.WIKIPEDIA.ORG/WIKI/GNU%20GENERAL%20PUBLIC%20LICENSE
26 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOLARIS%20OPERATING%20ENVIRONMENT
27 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT%20WINDOWS
28 HTTP://EN.WIKIPEDIA.ORG/WIKI/MAC%20OS
29 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINUX

2.5.2 Java 2 (version 1.2)

Introduced in 1998 as a quick fix to the former versions, version 1.2 was the start of a new beginning for Java. The JDKs of version 1.2 and later versions are often called *Java 2* as well. For example, the official name of JDK 1.4 is *The Java(TM) 2 Platform, Standard Edition version 1.4*.

Major changes include:

- Rewrite the event handling (Add Event Listeners)
- Change Thread synchronizations
- Introduction of the JIT-Just in time compilers

2.5.3 Kestrel (Java 1.3)

2.5.4 Merlin (Java 1.4)

Java 1.4 has improved programmer productivity by expanding language features and available APIs

- Assertion
- Regular Expression
- XML processing
- Cryptography and Secure Socket Layer (SSL)
- Non-blocking I/O(NIO)
- Logging

2.5.5 Tiger (version 1.5.0; Java SE 5)

Released in September 2004

Major changes include:

- GENERICS[30] - Provides compile-time type safety for collections :and eliminates the drudgery of casting.
- AUTOBOXING/UNBOXING[31] - Eliminates the drudgery of manual conversion between primitive types (such as int) and wrapper types (such as Integer).
- Enhanced for - Shorten the for loop with Collections use.
- Static imports - Lets you import all the static part of a class.
- ANNOTATION[32]/Metadata - Enabling tools to generate code and deployment descriptors from annotations in the source code. This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it. Annotations can be inspected through source parsing or by using the additional reflection APIs added in Java 5.

30 Chapter 22 on page 119

31 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FPRIMITIVE%20TYPES%23AUTOBOXING%2FUNBOXING

32 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FANNOTATION

- JVM Improvements - Most of the run time library is now mapped into memory as a memory image, as opposed to being loaded from a series of class files. Large portion of the runtime libraries will now be shared among multiple JVM instances.

(from HTTP://JAVA.SUN.COM/FEATURES/2003/05/BLOCH_QA.HTML[33])

2.5.6 Mustang (version 1.6.0; Java SE 6)

Released on 11 December 2006.[34]

What's New in Java SE 6:

- Web Services - First-class support for writing XML web service client applications.
- Scripting - You can now mix in JavaScript technology source code, useful for prototyping. Also useful when you have teams with a variety of skill sets. More advanced developers can plug in their own scripting engines and mix their favorite scripting language in with Java code as they see fit.
- Database - No more need to find and configure your own JDBC database when developing a database application! Developers will also get the updated JDBC 4.0, a well-used API with many important improvements, such as special support for XML as an SQL datatype and better integration of Binary Large OBjects (BLOBs) and Character Large OBjects (CLOBs) into the APIs.
- More Desktop APIs - GUI developers get a large number of new tricks to play like the ever popular yet newly incorporated SwingWorker utility to help you with threading in GUI apps, JTable sorting and filtering, and a new facility for quick splash screens to quiet impatient users.
- Monitoring and Management - The really big deal here is that you don't need do anything special to the startup to be able to attach on demand with any of the monitoring and management tools in the Java SE platform.
- Compiler Access - Really aimed at people who create tools for Java development and for frameworks like JavaServer Pages (JSP) or Personal Home Page construction kit (PHP) engines that need to generate a bunch of classes on demand, the compiler API opens up programmatic access to javac for in-process compilation of dynamically generated Java code. The compiler API is not directly intended for the everyday developer, but for those of you deafened by your screaming inner geek, roll up your sleeves and give it a try. And the rest of us will happily benefit from the tools and the improved Java frameworks that use this.
- Pluggable Annotations allows programmer to write annotation processor so that it can analyse your code semantically before javac compiles. For example, you could write an annotation processor that verifies whether your program obeys naming conventions.

33 HTTP://JAVA.SUN.COM/FEATURES/2003/05/BLOCH_QA.HTML

34 UNKNOWN TEMPLATE "cite web"

- Desktop Deployment - At long last, Java SE 6 unifies the Java Plug-in technology and Java Web-Start engines, which just makes sense. Installation of the Java WebStart application got a much needed makeover.
- Security - Java SE 6 has simplified the job of its security administrators by providing various new ways to access platform-native security services, such as native Public Key Infrastructure (PKI) and cryptographic services on Microsoft Windows for secure authentication and communication, Java Generic Security Services (Java GSS) and Kerberos services for authentication, and access to LDAP servers for authenticating users.
- The -lities: Quality, Compatibility, Stability - Bug fixes ...

2.5.7 Dolphin (version 1.7.0; Java SE 7)

Anticipated for 2010

2.6 Citations

CATEGORY:JAVA PROGRAMMING[35]

35 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

3 The Java Platform

Points to ponder:
The newer versions of Java, i.e., version 6.0 and 7.0 are being developed and are known by their code-names MUSTANG[1] and DOLPHIN[2] respectively.
You can test-ride their nightly builds, available online via the given links.

The **Java platform** is the name for a computing environment, or PLATFORM[3], from SUN MICROSYSTEMS[4] which can run applications developed using the JAVA PROGRAMMING LANGUAGE[5] and set of development tools. In this case, the platform is not a specific hardware or operating system, but rather an execution engine called a VIRTUAL MACHINE[6], and a set of standard libraries which provide common functionality.

The platform is properly called the **Java 2 Platform** (although the "2" is to be dropped HTTP://JAVA.SUN.COM/DEVELOPER/TECHNICALARTICLES/JAVAONE2005/NAMING.HTML[7]), and includes a Standard Edition or J2SE[8] (now Java SE), an ENTERPRISE[9] Edition or J2EE[10] (now Java EE), and a Micro Edition or J2ME[11] (now Java ME). The current version of the Java 2 platform is alternatively specified as version 1.6 or version 6 (both refer to the same version). A good overview of the myriad of technologies that makes up the Java 2 Platform can be found on the JDK DOCUMENTATION PAGE[12].

3.1 Java technologies

The Java platform consists of a wide array of technologies, each of which provides a distinct portion of the overall development or runtime environment. For example, end-users typically interface with the JAVA VIRTUAL MACHINE[13] and the standard set of class libraries. In addition, there are numerous ways for Java applications to be deployed, including being embedded into a web page. Lastly, developers who are creating applications for the platform use a set of development tools called the JAVA DEVELOPMENT KIT[14].

3 HTTP://EN.WIKIPEDIA.ORG/WIKI/PLATFORM%20%28COMPUTING%29
4 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUN%20MICROSYSTEMS
5 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20PROGRAMMING%20LANGUAGE
6 HTTP://EN.WIKIPEDIA.ORG/WIKI/VIRTUAL%20MACHINE
7 HTTP://JAVA.SUN.COM/DEVELOPER/TECHNICALARTICLES/JAVAONE2005/NAMING.HTML
8 HTTP://EN.WIKIPEDIA.ORG/WIKI/J2SE
9 HTTP://EN.WIKIPEDIA.ORG/WIKI/ENTERPRISE
10 HTTP://EN.WIKIPEDIA.ORG/WIKI/J2EE
11 HTTP://EN.WIKIPEDIA.ORG/WIKI/J2ME
12 HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/INDEX.HTML
13 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20VIRTUAL%20MACHINE
14 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20DEVELOPMENT%20KIT

3.2 Java Runtime Environment

A program targeting the Java platform needs two components to be present on its host: a Java virtual machine, and a set of class libraries providing any services on which it depends. Sun's distribution of their JVM and their implementation of the standard classes is known as the Java Runtime Environment (JRE).

3.2.1 Java Virtual Machine

The heart of the Java platform is the concept of a common "virtual" processor that executes JAVA BYTECODE[15] programs. This bytecode is the same no matter what hardware or operating system the program is running under. The Java platform provides an INTERPRETER[16] called the JAVA VIRTUAL MACHINE[17] (JVM), which translates the Java bytecode into native processor instructions at run-time. This permits the same application to be run on any platform that has a virtual machine available.

Since JRE version 1.2, Sun's JVM implementation has also included a JUST-IN-TIME COMPILER[18]. Instead of interpreting the bytecode one instruction at a time, this converts the bytecode for a program into equivalent native machine code as the program is loaded into the virtual machine, allowing it to execute much faster at the cost of a small delay whenever new bytecode is loaded. This allows the JIT compiler to target a specific host platform and hardware, even potentially optimizing the output code in different ways based on observations of the program's behaviour.

This is not to say that one can truly compile Java code to its fullest extent (in order to reap the benefits of speedy native machine code). Yes, there are "compilers" available that will attempt this feat, but not all Java libraries have a machine code equivalent. For instance, the "reflect" library, which allows Java programmers to delve into instructions only available at runtime, is not well represented (if at all) by machine code.

Java was not the first virtual-machine-based platform, though it is by far the most successful and well-known. Previous uses for virtual machine technology primarily involved EMULATORS[19] to aid development for not-yet-developed hardware or operating systems, but the JVM was designed to be implemented entirely in software, while making it easy to efficiently port an implementation to hardware of all kinds.

3.2.2 Class libraries

In most modern operating systems, a large body of reusable code is provided to simplify the programmer's job. This code is typically provided as a set of DYNAMICALLY LOADABLE LIBRARIES[20]

15 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20BYTECODE
16 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERPRETER%20%28COMPUTING%29
17 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20VIRTUAL%20MACHINE
18 HTTP://EN.WIKIPEDIA.ORG/WIKI/JUST-IN-TIME%20COMPILATION
19 HTTP://EN.WIKIPEDIA.ORG/WIKI/EMULATORS
20 HTTP://EN.WIKIPEDIA.ORG/WIKI/LIBRARY%20%28COMPUTER%20SCIENCE%29%23DYNAMIC%20LINKING

that applications can call at runtime. Because the Java platform is not dependent on any specific operating system, applications cannot rely on any of the existing libraries. Instead, the Java platform provides a comprehensive set of standard class libraries, containing much of the same reusable functions commonly found in modern operating systems.

The Java class libraries serve three purposes within the Java platform. Like other standard code libraries, they provide the programmer with a well-known set of functions to perform common tasks, such as maintaining lists of items or performing complex string parsing. In addition, the class libraries provide an abstract interface to tasks that would normally depend heavily on the hardware and operating system. Tasks such as network access and file access are often heavily dependent on the native capabilities of the platform. The Java java.net and java.io libraries implement the required native code internally, then provide a standard interface for the Java applications to perform those tasks. Finally, some underlying platforms may not support all of the features a Java application expects. In these cases, the class libraries can either emulate those features using whatever is available, or provide a consistent way to check for the presence of a specific feature.

3.3 Languages

The word Java, by itself, usually refers to the Java programming language which was designed for use with the Java platform. Programming languages are typically outside of the scope of the phrase "platform". However, Sun does not encourage the use of any other languages with the platform, and lists the Java programming language as a core part of the Java 2 platform. The language and runtime are therefore commonly considered a single unit.

Nevertheless, third parties have produced a number of compilers which target the JVM. Some of these are for existing languages, while others are for extensions to the Java language itself. These include:

- GROOVY[21]
- PIZZA[22]
- GJ[23] (Generic Java), which was incorporated into official Java as of Sun's version 1.5.
- NETREXX[24]

Another option is to use a more interface approach like the JYTHON[25].

3.4 Similar Platforms

The success of Java and its WRITE ONCE, RUN ANYWHERE[26] concept has also led to other similar efforts. The most notable of these is the MICROSOFT .NET[27] platform, which borrows many of

21 HTTP://EN.WIKIPEDIA.ORG/WIKI/GROOVY%20PROGRAMMING%20LANGUAGE
22 HTTP://EN.WIKIPEDIA.ORG/WIKI/PIZZA%20PROGRAMMING%20LANGUAGE
23 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENERIC%20JAVA%20%28PROGRAMMING%20LANGUAGE%29
24 HTTP://EN.WIKIPEDIA.ORG/WIKI/NETREXX
25 HTTP://EN.WIKIPEDIA.ORG/WIKI/JYTHON
26 HTTP://EN.WIKIPEDIA.ORG/WIKI/WRITE%20ONCE%2C%20RUN%20ANYWHERE
27 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT%20.NET

the concepts and innovations of Java; in fact, it has an implementation of a Java-like language called VISUAL J#[28] (formerly known as J++[29]). (It is *Java-like* in that J# is not the Java language. Instead, J# contains non-standard extensions of the language.)

Later Microsoft stopped, withdrew its J# support, and created a new language called C#. C# is very similar to J# and Java, but not compatible with them. The differences between Java and C# can be read at COMPARISON OF C SHARP AND JAVA[30].

CATEGORY:JAVA PROGRAMMING[31]

28 HTTP://EN.WIKIPEDIA.ORG/WIKI/J%20SHARP%20PROGRAMMING%20LANGUAGE
29 HTTP://EN.WIKIPEDIA.ORG/WIKI/J%20PLUS%20PLUS
30 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPARISON%20OF%20C%20SHARP%20AND%20JAVA
31 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

4 Java Programming Environment

To compile Java programs, you will need to download and install the Java Development Kit (JDK). This is available from SUN'S WEBSITE[1]. Other hardware and operating system vendors also supply Java Development Kits for their platforms, although they may change the name of the kit. Sun produces JDKs for Windows, Linux, and Solaris.

There are numerous environments you can use to develop your Java programs. You can choose to write your programs in a text editor and then compile them using the command line, or you can use an integrated development environment (IDE). IDEs like NETBEANS[2] and ECLIPSE[3] provide many useful functions such as syntax error checking, code completion, automatic compilation and debugging, which you may find useful, especially if this is your first foray into programming.

4.1 The Java Compiler

The JDK consists of a set of tools necessary to construct Java programs. The most notable tool in the JDK is the Java compiler, also known as `javac`.

`javac` compiles Java source files into executable Java class files. Source files are text files with a `.java` file name extension. You can create such files with a text editor, like NOTEPAD[4], or an IDE. `javac` then compiles these files into loadable and executable class files, using the `.class` extension. For example, if you create a Java class `org/wikibooks/util/PrintDate.java`

```
package org.wikibooks.util;
import java.util.Date;
public class PrintDate
{
    public static void main(String[] args)
    {
        System.out.println(new Date());
    }
}
```

you can compile it by entering the following command in a command shell:

1 HTTP://JAVA.SUN.COM/
2 HTTP://WWW.NETBEANS.ORG
3 HTTP://WWW.ECLIPSE.ORG
4 HTTP://EN.WIKIPEDIA.ORG/WIKI/NOTEPAD

```
javac org/wikibooks/util/PrintDate.java
```

You would normally invoke this in a shell whose working directory is the root directory containing all of your source files resides. (The Java package statement, `package org.wikibooks.util`, corresponds to the directory structure org/wikibooks/util see JAVA PACKAGES[5].)

`javac` will create the file `PrintDate.class` in the same directory where the source file is located. If there are syntax errors, javac will print those to the shell. The `PrintDate.class` file contains the byte code, that will run under all hardware where the Java runtime is installed. Even if the `PrintDate.class` file was created in Windows operating system, you can copy this file to unix and it will be executed fine.

Usually there is more than one class file created and those files are packaged to a `application_name.jar` file to distribute to run in any hardware.

IDEs may manage this process automatically. For example, Eclipse contains its own Java compiler and thus does not use `javac` directly. It automatically compiles the Java source files when you save them. The use of IDE's such as Eclipse are beyond the scope of this module, however, so consult the IDE's tutorials and help to see how they provide a Java programming environment.

4.1.1 The bytecode

It should be clear from the above paragraph that the Java compiler compiles source code text files with the extension `.java` into executable code usually confined into a class file with the `.class` extension. Such code is called BYTECODE[6].

In many languages prior to Java, the source code would generally compile into the machine-code for the particular machine the program was compiled upon. Therefore, if a program was compiled on an X86 MACHINE[7], it would run only on an X86 machine and no other. Java, on the other hand, produces a *bytecode* - an intermediate binary form of code that is a portable representation of the Java class. Any Java Virtual Machine on any hardware/operating system platform can then execute this same bytecode. There are some restrictions to this portability. For example, a Java ME system cannot execute all programs compiled for the Java SE environment because Java ME is pared down to small devices. But in general, a Java SE program can run unmodified on any Java SE virtual machine.

4.1.2 The JIT compiler

Being compiled halfway through, it is the job of the Java Virtual Machine to compile the rest of the program to native code at the time of its execution making Java code follow the "Write Once, Run Anywhere" (WORA) policy. The compiler used to compile bytecode into machine-code at runtime is called the *Just-In-Time* or *JIT compiler*. Once a piece of code is compiled by the JVM to execution code, the code is used and re-used again and again, to speed up execution.

5 Chapter 13 on page 79
6 HTTP://EN.WIKIPEDIA.ORG/WIKI/BYTECODE
7 HTTP://EN.WIKIPEDIA.ORG/WIKI/X86%20ARCHITECTURE

4.2 The Java Runtime Environment

The Java Runtime Environment, or JRE, is responsible for the execution of Java programs. The Sun JDK also includes the JRE. The JRE however can also be installed and used without installing the JDK which is useful if you wish to execute Java programs but not build them. The JRE helps load Java programs into the memory and executes them.

4.2.1 Main entry point

In Java programming, CLASSES[8] are used to define objects and entities that hold particular data. To execute a Java program, a special class is required to assist in loading the program into the computer's memory. Such a class contains a METHOD[9] with the following signature.

```
public static void main(String[] args) {...}
```

The class is hence said to have a *main entry point* defined and is usually called a *Java program*. The method described above as the main entry point is usually nicknamed the `main` *method*. Java classes without `main` methods are simply classes, although they may be part of a program.

Each Java class can have a *main* method, as opposed to C, and C++ where there can only be one, for the whole program. This feature is good in a sense that the *main* method can be implemented to do some testing on the class. The program must define one class's main method as the entry point for the program.

4.2.2 Executing a command-line Java program

If a class which you compile with `javac` has a `main` entry point, you can execute the class by specifying the class name as an argument to the `java` program.

```
java org.wikibooks.util.PrintDate
```

This will run the `main` method in the `PrintDate` class in the `org.wikibooks.util` package. (Packages provide a convenient way to provide NAMESPACES[10] and organization of Java classes. We'll use `org.wikibooks.util` as a parent package in this module. MORE ON PACKAGES[11].)

Your program will begin execution in the shell window. Inputs and outputs will be gathered from and to your shell window. On a Windows platform, you can use a DOS[12] command window as

8 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FJAVA%20FOUNDATIONS%23OBJECTS%20AND%20CLASSES

9 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FJAVA%20FOUNDATIONS%23METHODS

10 Chapter 13 on page 79

11 Chapter 13 on page 79

12 HTTP://EN.WIKIPEDIA.ORG/WIKI/MSDOS

the command shell program for the execution of Java programs. The JRE is normally called `java` because the `java` program is the most widely used program to execute Java programs.

4.2.3 Executing a Graphical User Interface Java program

On the Windows platform, there is an alternate JRE executable called `javaw.exe` or `javaw` which runs Java programs as a Windows native application - that is, with no console for standard input or output.

```
javaw org.wikibooks.util.ViewDate
```

Rather than executing in the console, the Java program would be executed in a separate Windows native process. This is typically done for Java applications which create their own graphical user interface (GUI) windows. The above-mentioned `org.wikibooks.util.PrintDate` program, which prints output to the standard output stream, is not appropriate for use with `javaw` as there is no console output. Instead, `org.wikibooks.util.ViewDate` would be a program which creates its own windows to display the date.

As with `java`, IDE's also manage the execution of Java programs in slightly different ways. They may provide shortcuts for running programs and windows for capturing the output.

On UNIX/Linux this does not matter. If the program is launched graphically (by file association in a file manager) a console is not shown. GUI programs will have the titlebars following the look and feel of your desktop (KDE, GNOME, Fluxbox, XFCE) theme.

4.3 Other JDK tools

Apart from the tools specified above in detail, the JDK has matured over the years and has included in itself several other tools. Where some of these tools are no longer used, others offer a far greater deal of capability to the Java Development Kit. Below is a list of some of the tools available for the JDK.

4.3.1 The *apt* tool

In Java 1.5 (alias Java 5.0) Sun added a mechanism called *annotations*. Annotations allow the addition of meta-data to Java source code, and even provide mechanisms to carry that meta-data forth into a compiled class files.

Also starting with Java 1.5 Sun added the *apt* tool to the JDK. *apt* works on Java source code. It is an *annotation processing tool* which digs through source code, finds annotation statements in the source code and executes actions if it finds known annotations. The most common task is to generate some particular source code.

The actions *apt* performs when finding annotations in the source code are not hard-coded into *apt*. Instead, one has to code particular annotation handlers (in Java). These handlers are called *annotation processors*.

The most difficult thing with *apt* is that Sun decided to use a whole set of new terminology. *apt* can simply be seen as a source code preprocessor framework, and *annotation processors* are typically just code generators.

See also: GETTING STARTED WITH THE ANNOTATION PROCESSING TOOL (APT)[13]

4.3.2 The *appletviewer* tool

Java applets require a particular environment to execute. Typically, this environment is provided by a browser with a Java plug-in, and a web server serving the applet. However, during development and testing of an applet it might be more convenient to start an applet without the need to fiddle with a browser and a web server. In such a case, Sun's *appletviewer* from the JDK can be used to run an applet.

4.3.3 The *javah* tool

A Java class can call *native*, or non-Java, code that has been prepared to be called from Java. The details and procedures are specified in the JNI (*Java Native Interface*). Commonly, native code is written in C (or C++). The JDK tool *javah* helps to write the necessary C code, by generating C header files and C stub code.

4.3.4 The *extcheck* tool

extcheck also appeared first with Java 1.5. It can be used prior to the installation of a Java extension into the JDK or JRE environment. It checks if a particular Jar file conflicts with an already installed extension.

4.3.5 Security Tools

The JDK comes with a large number of tools related to the security features of Java. Usage of these tools first requires study of the particular security mechanisms.

The tools are:

keytool: To manage keys and certificates

jarsigner: To generate and verify digital signatures of JARs (Java ARchives)

policytool: To edit policy files

kinit: To obtain Kerberos v5 tickets

klist: To manage Kerberos credential cache and key table

ktab: To manage entries in a key table

13 HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/GUIDE/APT/GETTINGSTARTED.HTML

4.3.6 The *native2ascii* tool

native2ascii is an important, though underappreciated, tool for writing *properties files* -- files containing configuration data -- or *resource bundles* -- files containing language translations of text.

Such files can contain only ASCII and Latin-1 characters, but international programmers need a full range of character sets. Text using these characters can appear in properties files and resource bundles only if the non-ASCII and non-Latin-ˆ1 characters are converted into Unicode escape sequences (\uXXXX notation).

The task of writing such escape sequences is handled by *native2ascii*. You can write the international text in an editor using the appropriate character encoding, then use *native2ascii* to generate the necessary ASCII text with embedded Unicode escape sequences. Despite the name, *native2ascii* can also convert from ASCII to native, so it is useful for converting an existing properties file or resource bundle back to some other encoding.

native2ascii makes most sense when integrated into a build system to automate the conversion.

4.3.7 RMI Tools

4.3.8 Java IDL and RMI-IIOP Tools

4.3.9 Deployment & Web Start Tools

4.3.10 Browser Plug-In Tools

4.3.11 Monitoring and Management Tools / Troubleshooting Tools

With Java 1.5 a set of *monitoring and management tools* have been added to the JDK, in addition to a set of troubleshooting tools.

The monitoring and management tools are intended for monitoring and managing the virtual machine and the execution environment. They allow, for example, monitoring memory usage during the execution of a Java program.

The troubleshooting tools provide rather esoteric insight into aspects of the virtual machine. (Interestingly, the Java debugger is not categorized as a troubleshooting tool.)

All the monitoring and management and troubleshooting tools are currently marked as "experimental" (which does not affect jdb). So they might disappear in future JDKs.

4.3.12 The Jar tool

Jar is short for *Java archive*. It is a tool for creating Java archives or `jar` *files* - a file with `.jar` as the extension. A Java archive is a collection of compiled Java classes and other resources which

those classes may require (such as text files, configuration files, images) at runtime. Internally, a `jar` file is really a `.zip` FILE[14].

4.3.13 The jdb tool

Jdb is short for *Java debugger*. The Java debugger is a command-line console that provides a DEBUGGING[15] environment for Java programs. Although you can use this command line console, IDE's normally provide easier to use debugging environments. <!--To learn more about debugging in Java, *refer* to DEBUGGING IN JAVA[16].-->

4.3.14 The Javadoc tool

As programs grow large and complex, programmers need ways to track changes and to understand the code better at each step of its evolution. For decades, programmer have been employing the use of special programming constructs called COMMENTS[17] - regions that help declare user definitions for a code snippet within the source code. But comments are prone to be verbose and incomprehensible, let alone be difficult to read in applications having hundreds of lines of code.

Java provides the user with a way to easily publish documentation about the code using a special commenting system and the `javadoc` tool. The `javadoc` tool generates documentation about the APPLICATION PROGRAMMING INTERFACE[18] (API) of a set of user-created Java classes. `javadoc` reads source file comments from the `.java` source files and generates HTML documents that are easier to read and understand without looking at the code itself.

4.3.15 The javap tool

Where Javadoc provide a detailed view into the API and documentation of a Java class, the `javap` tool prints information regarding members (constructors, methods and variables) in a class. In other words, it lists the class' API and/or the compiled instructions of the class. `javap` is a formatting disassembler for Java bytecode.

See OTHER TOOLS[19] for all the tools that are bundled with Sun's JDK.

CATEGORY:JAVA PROGRAMMING[20]

14 HTTP://EN.WIKIPEDIA.ORG/WIKI/ZIP%20%28FILE%20FORMAT%29
15 HTTP://EN.WIKIPEDIA.ORG/WIKI/DEBUGGING
16 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3AJAVA%3ADEBUGGING%20IN%20JAVA
17 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3AJAVA_COMMENTS
18 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20PROGRAMMING%20INTERFACE
19 HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/TOOLDOCS/INDEX.HTML
20 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

5 Installation

To formally begin programming in Java, you would need to obtain the Java software necessary and install it on your system. This section will help you set up any system with the latest Java software. There are two things you need to have on your computer to develop Java programs: a way to make the programs, and a way to write those programs. The following sections will describe both.

5.1 Windows

5.1.1 Download/Install

Downloading the program involves navigating to the website HTTP://JAVA.SUN.COM/JAVASE/DOWNLOADS/INDEX.JSP[1]. Here you are presented by a wide array of choices. Don't fret. For the moment, all you need is the JDK, so click on the red button with "Download JDK" inside. If you want to add something like Netbeans to your system, it's easy enough to do that at a later period.

Once downloaded, go through the install process, reading and agreeing to the terms and conditions. I recommend you keep everything at their default settings to keep things simple.

5.1.2 Editor

With your system set up you'll want a method of writing your java files. Later on, you might want to think about getting an IDE such as Netbeans. IDE's make it convenient if you work as part of a large team in an industrial environment. They do a lot of work for you, but if you don't fully understand the Java code itself, an IDE can do more harm than good.

So it's recommended for the starting chapter that you use a text editor. You can use notepad, which is default on all Windows operating systems. However, there is a free and open source program called Notepad++, which you can download from HTTP://NOTEPAD-PLUS.SOURCEFORGE.NET[2]. This program is designed to be used as a programmer's notepad, with what's called **syntax highlighting**, which is a way of colouring text based on various keywords. This makes the code more readable, and it can be easier to spot errors in code.

1 HTTP://JAVA.SUN.COM/JAVASE/DOWNLOADS/INDEX.JSP
2 HTTP://NOTEPAD-PLUS.SOURCEFORGE.NET

5.2 Ubuntu Linux

5.2.1 Download/Install

Downloading and installing the Java Development Kit on Ubuntu (and most other linux distributions) is incredibly simple, open up terminal (press ALT+F2 and type xterm or gnome-terminal) and type or copy the following (to copy, select and press middle button on the terminal window):

```
sudo apt-get install openjdk-6-jdk openjdk-6-jre openjdk-6-doc
```

And everything is set.

5.2.2 Editor

Gedit is the default program provided when Ubuntu is installed. Think of it like the windows notepad, but much better. Because Linux is made by hackers and programmers, the default notepad provided has all the things you'd expect a programmers writer to have.

The Notepad++ equivalent for Linux is Scite, if you feel like installing it you can use the command:

```
sudo apt-get install scite
```

5.3 External Links

- SUN'S WEBSITE[3]
- JAVA.NET JDK REPOSITORY[4]

CATEGORY:JAVA PROGRAMMING[5]

3 HTTP://JAVA.SUN.COM
4 HTTPS://JDK.DEV.JAVA.NET
5 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

6 Compilation

We have already discussed COMPILATION BASICS [1]. Here's a recap of the concepts we'd seen earlier and some additional details.

6.1 Compiling to bytecode

In Java, programs are not compiled into executable files; they are compiled into BYTECODE[2] (as discussed EARLIER[3]), which the JVM then executes at runtime. Java source code is compiled into bytecode when we use the `javac` compiler. The bytecode gets saved on the disk with the file extension `.class`. When the program is to be run, the bytecode is converted, using the JUST-IN-TIME(JIT) COMPILER[4]. The result is machine code which is then fed to the memory and is executed.

So Java has two step compilation:

- Step one to create byte-code
- Step two to create machine level code

The Java classes/Byte Codes are compiled to machine code and loaded into memory by the JVM when needed the first time. This is different than other languages like C/C++ where the whole program had to be compiled to machine code and linked to create an executable file, before the program could start.

JIT compilers compile byte-code once and the compiled machine code are re-used again and again, to speed up execution. Early Java compilers compiled the byte-code to machine code each time it was used, but more modern compilers cache this machine code for reuse on the machine. Even then, java's JIT compiling was still faster than an "interpreter-language", where code is compiled from **high level language**, instead of from byte-code each time it was used.

6.2 Automatic Compilation of Dependent Classes

In Java, if you have used any reference to any other java object, then the class for that object will be automatically compiled, if that was not compiled already. These automatic compilations are nested, and this continues until all classes are compiled that are needed to run the program. It is usually enough to compile only the high level class, since all the dependent classes will be

1 Chapter 4 on page 17
2 HTTP://EN.WIKIPEDIA.ORG/WIKI/BYTECODE
3 Chapter 4.1.1 on page 18
4 Chapter 4.1.2 on page 18

automatically compiled.

```
javac ... MainClass.java
```

However, you can't rely on this feature if your program is using reflection to create objects, or you are compiling for servlets or a "jar" package. In these cases you should list these classes for explicit compilation.

```
javac ... MainClass.java, ServletOne.java, ...
```

The best way is to use a build tool to build your application. The build tool would check all the needed dependencies and compile only the needed class for the build. The ANT[5] tool is the best and the most popular build tool currently available. Using ANT[6] you would build your application from the command line by typing:

```
ant build.xml
```

The xml file contains all the information needed to build the application.

Note: In rare cases, your code may appear to compile correctly but the program behaves as if you were using an old copy of the source code (or otherwise reports errors during runtime.) When this occurrs, you may need to clean your compilation folder by either deleting the class files or using the Clean command from the IDE.

The next most popular way to build applications are using an IDE. IDE stands for *Integrated Development Environment*, examples of which are listed below.

6.3 Packages, Subdirectories, and Resources

Each Java top level class belongs to a package (covered in the chapter about PACKAGES[7]). This may be declared in a `package` statement at the beginning of the file; if that is missing, the class belongs to the unnamed package.

For compilation, the file must be in the right directory structure. A file containing a class in the unnamed package must be in the current/root directory; if the class belongs to a package, it must be in a directory with the same name as the package.

The convention is that package names and directory names corresponding to the package consist of only lower case letters.

5 Chapter 6.13 on page 31
6 Chapter 6.13 on page 31
7 Chapter 13 on page 79

Example:
Top level package. A class with this package declaration `package example;` has to be in a directory named `example`

Example:
Subpackages. A class with this package declaration `package org.wikibooks.en;` has to be in the `org/wikibooks/en` directory.

Java programs often contain non-code files such as images and properties files. These are referred to generally as 'resources' and stored in directories local to the classes in which they're used. For example, if the class `com.example.ExampleApp` uses the `icon.png` file, this file could be stored as `/com/example/resources/icon.png`. These resources present a problem when a program is complied, because `javac` does not copy them to wherever the `.class` files are being complied to (see above); it is up to the programmer to move the resource files and directories. *See also the section on how to automate this using* `ant`[8]*, below.*

6.4 Filename Case

The Java source file name must be the same as the public class name, the file contains. There can be only one public class defined per file. The Java class name is case sensitive, as is the source file name.

The naming convention for the class name is for it to start with a capital letter.

6.5 Compiler Options

6.5.1 Debugging and Symbolic Information

6.6 Additional Tools

6.6.1 IDEs

This section contains a little about the different IDEs available and their strengths and weaknesses.

6.7 JBuilder

JBuilder is a IDE with proprietary source code, sold by BORLAND[9]. One of the advantages in integration with together, a modeling tool.

8 Chapter 6.13 on page 31
9 HTTP://EN.WIKIBOOKS.ORG/WIKI/BORLAND

6.8 JCreator

There's info at: http://www.apcomputerscience.com/ide/jcreator/index.htm

6.9 Eclipse

Eclipse is a free IDE, plus a developer tool framework that can be extended for a particular development need. IBM was behind this free software development and it replaced IBM Visual Age tool. The idea was to create a standard look and feel that can be extended. The extendibility has distinguished Eclipse from other IDE tools. Eclipse also meant to compete with Microsoft Visual Studio tools. Microsoft tools give a standard way of developing code in the Microsoft world. Eclipse gives a similar standard way of developing code in the Java world, with big success so far. With the online error checking only, coding can be speed up by at least 50%(coding does not include programming).

The goal for Eclipse are twofold:

- Give a standard IDE for developing code
- Give a starting point, and the same look and feel for all other more sophisticated tools built on Eclipse

IBM's WSAD, and later IBM Rational Software Development Platform are built on Eclipse.

Standard Eclipse features:

- Standard window management (perspectives, views, browsers, explorers, ...)
- As you type error checking (immediate error indications, ...)
- As you type help window (type ., or <ctrl> space, ...)
- Automatic build (changed source code automatically compiled, ...)
- Built in debugger (full featured GUI debugger)
- Source code generation (getters and setters, ...)
- Searches (for implementation, for references, ...)
- Code refactoring (global reference update, ...)
- Plug-in-based architecture (be able to build tools that integrate seamlessly with the environment and other tools)
- ...

For more information see;

- ECLIPSE [10].
- PLUGINCENTRAL[11]

10 HTTP://WWW.ECLIPSE.ORG/
11 HTTP://WWW.ECLIPSEPLUGINCENTRAL.COM/

6.10 NetBeans

The NetBeans IDE is a free, open-source Integrated Development Environment for software developers. The IDE runs on many platforms including Windows, Linux, Solaris, and the MacOS. It is easy to install and use straight out of the box. The NetBeans IDE provides developers with all the tools they need to create professional cross-platform desktop, enterprise, web and mobile applications.

More info can be found at http://www.netbeans.org/products/ide/

6.11 BlueJ

BlueJ is an IDE that includes templates and will compile and run the applications for you. BlueJ is often used by classes because it is not necessary to set classpaths. BlueJ has it's own sets of Library's and you can add your own under preferences. That sets the classpath for all compilations that come out of it to include those you have added and the BlueJ libraries.

BlueJ offers an interesting GUI for the creation of packages and programs. Classes are represented as boxes with arrows running between them to represent inheritance/implementation or if one is constructed in another. BlueJ adds all those classes (the project) into the classpath at compile time.

BLUEJ HOMESITE[12]

6.12 Kawa

Kawa was developed by Tek-Tools. It is basically a Java editor which does not include wizards, and GUI tools. It is best suited to experienced Java programmers in small and midsized development teams.

The latest version is 4.0, you can DOWNLOAD IT.[13] For more info. see KAWA FROM TEK-TOOLS[14]. See a JAVAWORLD ARTICLE[15]

It looks that there is no new development for Kawa.

6.13 Ant

For comprehensive information about all aspects of Ant, please see the ANT WIKIBOOK[16].

12 HTTP://WWW.BLUEJ.ORG
13 HTTP://WWW.OLABS.COM/KAWA/
14 HTTP://WWW.TEK-TOOLS.COM/KAWA/
15 HTTP://WWW.JAVAWORLD.COM/JAVAWORLD/JW-06-2000/JW-0602-IW-KAWA.HTML
16 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3AAPACHE%20ANT

Ant is a build management tool designed to replace `make` as the tool for automated builds of large Java applications. Like Java, and unlike `make`, Ant is designed to be platform independent.

Building a Java application requires certain tasks to be performed. Those tasks may include not only compiling the code, but also copying files, packaging the program into a `jar` file, running tests and so on. Some of these tasks may depend upon others having been done previously (not creating a `jar` unless the program has been complied, for instance). It might also be a good idea to not execute all tasks every time the program is complied -- e.g. to only compile changed source files. **Ant makes all of these things easy.**

The tasks and their dependencies are defined in a `build.xml` file, generally kept in the root directory of the java project. Ant parses this file and executes the tasks therein. Below we give an example `build.xml` file.

Ant tool is written in Java and is open source, so it can be extended if there is a task you'd like to be done during the build that is not in the pre-defined tasks list. It is very easy to hook your ant task code to the other tasks: your code only needs to be in the classpath, and the Ant tool will load it at runtime. For more information about writing your own Ant tasks, please see the project website at http://ant.apache.org/.

UNKNOWN TEMPLATE "code"

6.14 The JIT compiler

The standard JIT compiler runs *on demand*. When a method is called repeatedly, the JIT compiler analyzes the bytecode and produces highly efficient machine code, which runs very fast. The JIT compiler is smart enough to recognize when the code has already been compiled, so as the application runs, compilation happens only as needed. As Java applications run, they tend to become faster and faster, because the JIT can perform runtime profiling and optimization to the code to meet the execution environment. Methods or code blocks which do not run often receive less optimization; those which run often (so called *hotspots*) receive more profiling and optimization.

7 Execution

There are various ways in which Java code can be executed. A complex Java application usually uses third party APIs or services. In this section we list the most popular ways a piece of Java code may be packed together and/or executed.

7.1 JSE code execution

Java language first edition came out in the client-server era. Thick clients were developed with rich GUI interfaces. Java first edition, JSE (Java Standard Edition) had/has the following in its belt:

- GUI capabilities (AWT, Swing)
- Network computing capabilities (RMI[1])
- Multi-tasking capabilities (Threads)

With JSE the following Java code executions are possible:

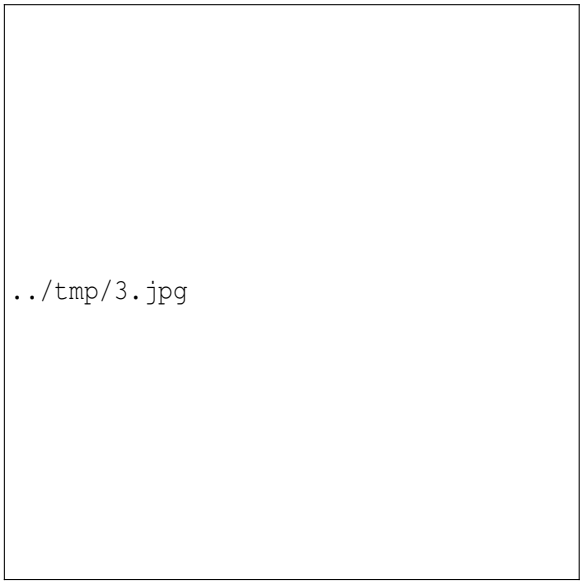


Figure 3: Figure 1: Stand alone execution

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20REMOTE%20METHOD%20INVOCATION

Stand alone Java application : (Figure 1) Stand alone application refers to a Java program where both the user interface and business modules are running on the same computer. The application may or may not use a database to persist data. The user interface could be either AWT or Swing.

The application would start with a `main()` method of a Class. The application stops when the `main()` method exits, or if an exception is thrown from the application to the JVM. Classes are loaded to memory and compiled as needed, either from the file system or from a *.jar file, by the JVM.
Invocation of Java programs distributed in this manner requires usage of the command line. Once the user has all the class files, he needs to launch the application by the following command line (where Main is the name of the class containing the main() method.)

```
java Main
```

Java 'jar' class libraries : Utility classes, framework classes, and/or third party classes are usually packaged and distributed in Java ' *.jar' files. These 'jar' files need to be put in the CLASSPATH of the java program from which these classes are going to be used.

If a jar file is executable, it can be run from the command line:

```
java -jar Application.jar
```

../tmp/4.jpg

Figure 4: Figure 2: Applet Execution

Java Applet code : (Figure 2) Java Applets are Java code referenced from HTML[2] pages, by the <APPLET> tag. The Java code is downloaded from a server and runs in the client browser JVM. Java has built-in support to render applets in the browser window.

Sophisticated GUI clients were found hard to develop, mostly because of the download time, incompatibilities between browser's JVM, and its communication requirements back to the server. Applets are rarely used today, and are most commonly used as small, separate graphic-like animation applets. The popularity of Java declined when Microsoft withdrew its Java support from INTERNET EXPLORER[3] default configuration, however, the plugin is still available as a free download from JAVA.COM[4].

More information can be found about applets at the APPLET CHAPTER[5], in this book. Also, Wikipedia has an article about JAVA APPLETS[6].

Client Server applications : The client server applications consist of a front-end, and a back-end part, both running on a separate computer. The idea is that the business logic would be on the back-end part of the program, which would be reused by all the clients. Here the challenge is to achieve a separation between front-end user interface code, and the back-end business logic code.

The communication between the front-end and the back-end can be achieved by two ways.

- One way is to define a data communication PROTOCOL[7] between the two tiers. The back-end part would listen for an incoming request. Based on the PROTOCOL[8] it interprets the request and sends back the result in data form.
- The other way is to use JAVA REMOTE INVOCATION[9] (RMI). With the use of RMI, a remote object can be created and used by the client. In this case Java objects are transmitted across the network.

More information can be found about client-server programming, with sample code, at the CLIENT SERVER CHAPTER[10] in this book.

Web Applications : For applications needed by lots of client installations, the client-server model did not work. Maintaining and upgrading the hundreds or thousands of clients caused a problem. It was not practical. The solution to this problem was to create a unified, standard client, for all applications, and that is the BROWSER[11].

2 HTTP://EN.WIKIPEDIA.ORG/WIKI/HTML
3 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERNET%20EXPLORER
4 HTTP://JAVA.COM/
5 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPPLETS
6 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20APPLET
7 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROTOCOL%20%28COMPUTING%29
8 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROTOCOL%20%28COMPUTING%29
9 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20REMOTE%20METHOD%20INVOCATION
10 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FCLIENT%20SERVER
11 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEB%20BROWSER

Having a standard client, it makes sense to create a unified, standard back-end service as well, and that is the APPLICATION SERVER[12].
Web Application is an application that is running in the APPLICATION SERVER[13], and it can be accessed and used by the BROWSER[14] client.
There are three main area of interest in Web Applications, those are:

- The WEB BROWSER[15]. This is the container of rendering HTML text, and running client scripts
- The HTTP[16] PROTOCOL[17]. Text data are sent back and forth between Browser and the Server
- The WEB SERVER[18] to serve static content, APPLICATION SERVER[19] to serve dynamic content and host EJB[20]s.

Wikipedia also has an article about WEB APPLICATION[21].

7.2 J2EE code execution

As the focus was shifting from reaching GUI clients to thin client applications, with Java version 2, Sun introduced J2EE (Java 2 Extended Edition). J2EE added :

- COMPONENTS BASE ARCHITECTURE[22], (Servlet, JSP, EJB Containers)

With J2EE the following Java component executions are possible:

12 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20SERVER
13 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20SERVER
14 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEB%20BROWSER
15 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEB%20BROWSER
16 HTTP://EN.WIKIPEDIA.ORG/WIKI/HYPERTEXT%20TRANSFER%20PROTOCOL
17 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROTOCOL%20%28COMPUTING%29
18 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEB%20SERVER
19 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20SERVER
20 HTTP://EN.WIKIPEDIA.ORG/WIKI/ENTERPRISE%20JAVABEAN
21 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEB%20APPLICATION
22 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOFTWARE%20COMPONENTRY

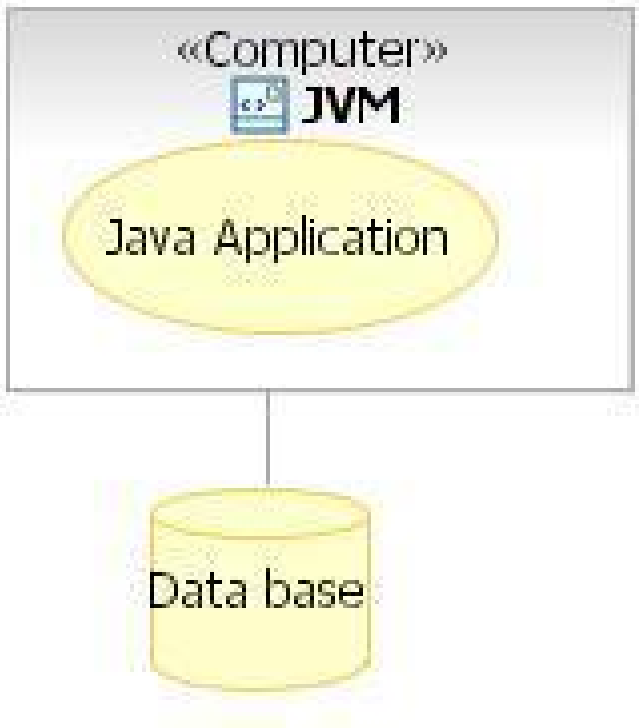


Figure 5: Figure 3: Servlet Execution

Java Servlet code : (Figure 3) Java got its popularity with server side programming, more specifically with J2EE[23] servlets. Servlets are running in a simple J2EE framework to handle client HTTP[24] requests. They are meant to replace CGI PROGRAMMING[25] for web pages rendering dynamic content.

The servlet is running in a so called SERVLET-CONTAINER/WEB CONTAINER[26]. The servlet's responsibility is to:

- Handle the request by doing the business logic computation,
- Connecting to a database if needed,
- Create HTML to present to the user through the browser

The HTML output represents both the presention logic and the results of the business computations. This represents a huge problem, and there is no real application relying only on servlets to handle the presention part of the responsibility. There are two main solutions to this:

- Use a template tool (Store the presentation part in an HTML file, marking the areas that need to be replaced after business logic computations).

23 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20EE
24 HTTP://EN.WIKIPEDIA.ORG/WIKI/HYPERTEXT%20TRANSFER%20PROTOCOL
25 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMMON%20GATEWAY%20INTERFACE
26 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEB%20CONTAINER

- Use JSP (See next section)

Wikipedia also has an article about SERVLETS[27].

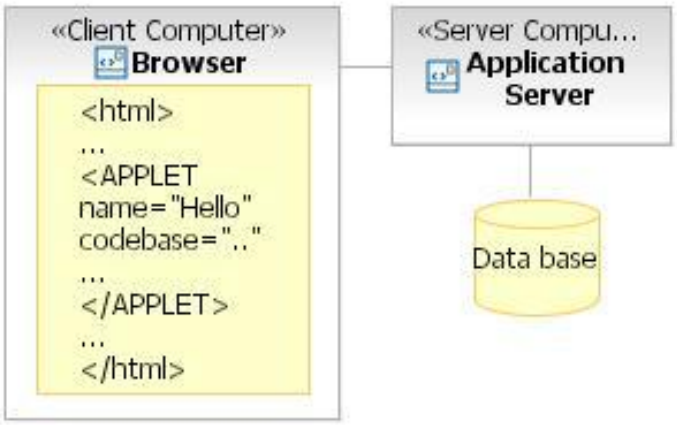


Figure 6: Figure 4: Jsp Execution

Java Server Pages (JSP) code : (Figure 4) JSP is an HTML file with embedded Java code inside. The first time the JSP is accessed, the JSP is converted to a Java Servlet. This servlet outputs HTML which has inside the result of the business logic computation. There are special JSP tags that helps to add data dynamically to the HTML. Also JSP technology allows to create custom tags.

Using the JSP technology correctly, business logic computations should not be in the embedded Java part of the JSP. JSP should be used to render the presentation of the static and dynamic data. Depending on the complexity of the data, 100% separation is not easy to achieve. Using custom tags, however may help to get closer to 100%. This is advocated also in MVC[28] architecture (see below).

27 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20SERVLET
28 HTTP://EN.WIKIPEDIA.ORG/WIKI/MODEL-VIEW-CONTROLLER

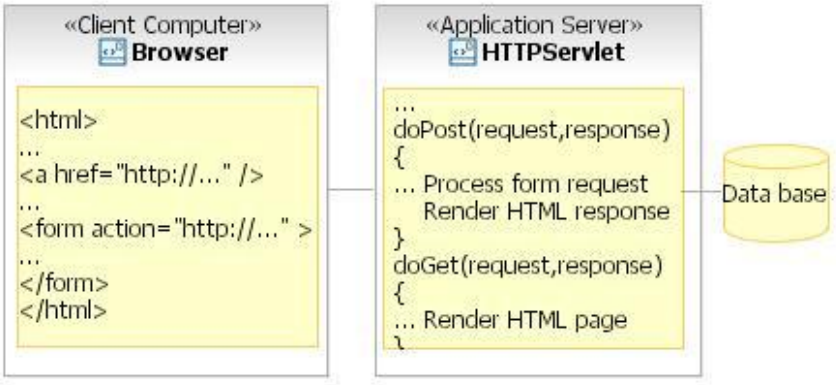


Figure 7: Figure 5: EJB Execution

EJB code : (Figure 5) In the 1990s, with the client server computing, a trend started, that is to move away from Mainfraim computing. That resulted in many small separate applications in a Company/Enterprise. Many times the same data was used in different applications. A new philosophy, "Enterprise Computing", was created to address these issues. The idea was to create components that can be reused throughout the Enterprise. The Enterprise Java Beans (EJBs) were supposed to address this.

An **EJB** is an application component that runs in an EJB container. The client accesses the EJB modules through the container, never directly. The container manages the life cycle of the EJB modules, and handles all the issues that arise from network/enterpise computing. Some of those are SECURITY/ACCESS CONTROL[29], OBJECT POOLING[30], TRANSACTION MANAGEMENT[31], ...
EJBs have the same problems as any reusable code: they need to be generic enough to be able to be reused and the changes or maintenance of EJBs can affect existing clients. Many times EJBs are used unnecessarily when they are not really needed. An EJB should be designed as a separate application in the enterprise, fulfilling one function.

29 HTTP://EN.WIKIPEDIA.ORG/WIKI/ACCESS%20CONTROL
30 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT%20POOL
31 HTTP://EN.WIKIPEDIA.ORG/WIKI/TRANSACTION%20PROCESSING

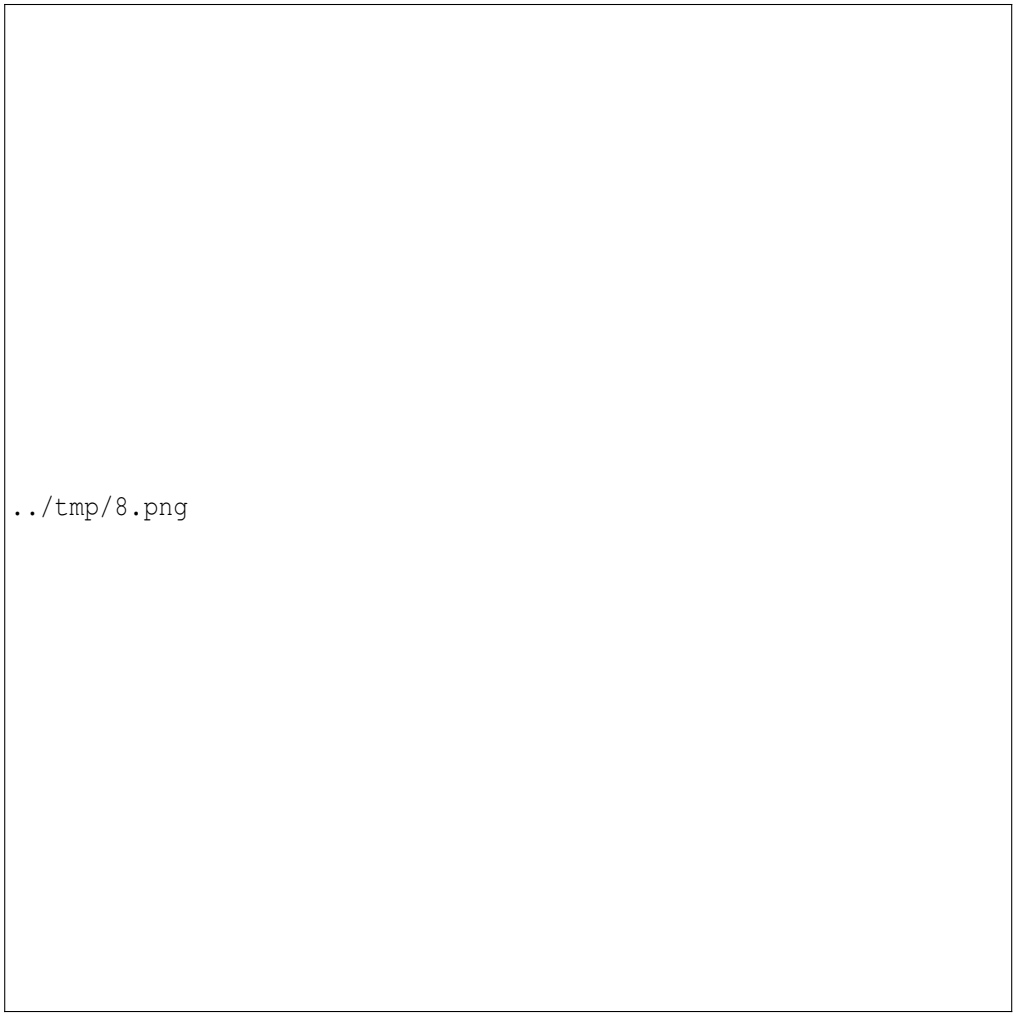


Figure 8: Figure 6: MVC Execution

Combine J2EE components to create an MVC architecture : This leads us to the three layers/tiers as shown in (Figure 6).

In modern web applications, with lots of static data and nice graphics, how the data is presented to the user became very important and usually needs the help of a graphic artist.
To help programmers and graphic artists to work together, the separation between data, code, and how it is presented became crucial.

- The **view** (User Interface Logic) contains the logic that is necessary to construct the presentation. This could be handled by JSP technology.
- The servlet acts as the **controller** and contains the logic that is necessary to process user events and to select an appropriate response.

- The business logic (**model**) actually accomplishes the goal of the interaction. This might be a query or an update to a database. This could be handled by EJB technology.

For more information about MVC, please see MVC[32].

7.3 Jini

After J2EE Sun had a vision about the next step of network computing. That is JINI[33]. The main idea is that in a network environment, there would be many independent services and consumers. Jini would allow these services/consumers to interact dynamically with each other in a robust way. The basic features of Jini are:

- **No user intervention** is needed when services are brought on or offline. (In contrast to EJBs where the client program has to know the server and port number where the EJB is deployed, in Jini the client is *supposed to find*, to discover, the service in the network.)
- **Self healing** by adapting when services (consumers of services) come and go. (Services periodically need to renew a lease to indicate that they are still available.)
- Consumers of JINI services do not need prior knowledge of the service's implementation. The **implementation is downloaded dynamically** and run on the consumer JVM, without configuration and user intervention. (For example, the end user may be presented with a slightly different user interface depending upon which service is being used at the time. The implementation of the user interface code would be provided by the service being used.)

A minimal Jini network environment consists of:

- One or more **services**
- A **lookup-service** keeping a list of registered services
- One or more **consumers**

Jini is not widely used at the current writing (2006). There are two possible reasons for it. One is Jini a bit complicated to understand and to set it up. The other reason is that Microsoft pulled out from Java, which caused the industry to turn to the use of proprietary solutions.

CATEGORY:JAVA PROGRAMMING[34]

32 HTTP://EN.WIKIPEDIA.ORG/WIKI/MODEL-VIEW-CONTROLLER
33 HTTP://EN.WIKIPEDIA.ORG/WIKI/JINI
34 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

8 First Java Program

8.1 Hello World

Generally when you first start programming in any language, you'll start with the traditional *Hello World* example. That said, let's start building your first Java program. You guessed it, it's *Hello World*! Before starting this exercise, make sure you know how to COMPILE[1] and RUN[2] Java programs.

Open your IDE and write the following text. Pay close attention to capitalization, as Java is case sensitive.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(string);
    }
}
```

Save it as `HelloWorld.java`. Again, make sure that the filename is the same case as the class name. Compile and run it:

```
javac HelloWorld.java
java HelloWorld
```

Your computer should display

```
Hello, world!
```

8.1.1 Line-by-line Analysis

The first line of the class,

```
public class HelloWorld {
```

declares a Java class named `HelloWorld`. This class is declared **public** - it is available to any other class. The next line,

1 Chapter 6 on page 27
2 Chapter 7 on page 33

```
public static void main(String[] args) {
```

begins a Java method named `main`. The `main` method is where Java will start executing the program. **args** is a method parameter which will contain the command line arguments used to run the program. The method must be both **public** and **static** for the program to run correctly. For more information on modifiers such as **public** and **static**, see ACCESS MODIFIERS[3], though at this level, you don't need to know a whole lot about them.

The

```
System.out.println(string);
```

statement sends the text `Hello, world!` to the console (with a line terminator). The final two braces mark the end of the `main` method and the end of the class.

8.2 Modifying the Program

Now, we will modify this program to print the first command line argument, if there is one, along with the greeting. For example, if you invoke the program as

```
java HelloWorld wikibooks
```

it will print

```
Hello, wikibooks!
```

Go back to the program, and modify it to read

```
public class HelloWorld {
    public static void main(String[] args) {
        String who;
        if (args.length > 0) {
            who = args[0];
        } else {
            who = string;
        }
        System.out.println(string + who + string);
    }
}
```

Run it again. It should display

3 Chapter 15 on page 87

```
Hello, wikibooks!
```

or, if you do not pass a command line parameter, it will simply print

```
Hello, World!
```

Wikipedia has a hello world sample for each type of java code, for more information see JAVA (PROGRAMMING LANGUAGE)#HELLO WORLD[4]

8.3 Common Problems

If the program does not work as you expect, check the following common errors.

- Are you sure all words are spelled correctly and with the exact case as shown?
- Are there semicolons and brackets in the appropriate spot?
- Are you missing a quote? Usually, modern IDEs would try coloring the entire source as a quote in this case.
- Are you launching the javac or java correctly? Javac requires the full filename with the .java extension, while java requires the class name itself.

8.4 The Next Step

Now that you have seen the classic *Hello, World* program in Java, let's move on to a more realistic example which highlights the object oriented nature of Java. Visit UNDERSTANDING A JAVA PROGRAM[5] which presents key Java language features along with a more complete explanation of the syntax and structure of a basic Java program.

CS:KURZ PROGRAMOVÁNÍ V JAVĚ/HELLO WORLD[6] PT:JAVA/CRIANDO E EXECUTANDO O PRIMEIRO PROGRAMA[7]

4 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20%28PROGRAMMING%20LANGUAGE%29%23HELLO%20WORLD

5 Chapter 9 on page 47

6 HTTP://CS.WIKIBOOKS.ORG/WIKI/KURZ%20PROGRAMOV%E1N%ED%20V%20JAV%01%1B%2FHELLO%20WORLD

7 HTTP://PT.WIKIBOOKS.ORG/WIKI/JAVA%2FCRIANDO%20E%20EXECUTANDO%20O%20PRIMEIRO%20PROGRAMA

9 Understanding a Java Program

This article presents a small Java program which can be run from the console. It computes the distance between two points on a plane. You need not understand the structure and meaning of the program just yet; we will get to that soon. Also, because the program is intended as a simple introduction, it has some room for improvement, and later in the module we will show some of these improvements. But let's not get too far ahead of ourselves!

9.1 The Distance Class: Intent, Source, and Use

This class is named *Distance*, so using your favorite editor or Java IDE, first create a file named `Distance.java`, then copy the source below and paste it into the file and save the file.

```
public class Distance
{
  private java.awt.Point point0, point1;

  public Distance(int x0, int y0, int x1, int y1)
  {
    point0 = new java.awt.Point(x0, y0);
    point1 = new java.awt.Point(x1, y1);
  }

  public void printDistance()
  {
    System.out.println("Distance between " + point0 + " and " +
point1
                    + " is " + point0.distance(point1));
  }

  public static void main(String[] args)
  {
    Distance dist = new Distance(
              intValue(args[0]), intValue(args[1]),
              intValue(args[2]), intValue(args[3]));
    dist.printDistance();
  }

  private static int intValue(String data)
  {
    return Integer.parseInt(data);
  }
}
```

At this point, you may wish to review the source to see how much you might be able to understand. While perhaps not being the most literate of programming languages, someone with understanding of other procedural languages such as C, or other OO languages such as C++ or C#, will be able to understand most if not all of the sample program.

Once you save the file, COMPILE[1] the program:

```
javac Distance.java
```

(If the `javac` command fails, review the JAVA INSTALLATION INSTRUCTIONS[2].)

To run the program, you supply it with the x and y coordinates of two points on a plane. (For this version of Distance, only integer points are supported.) The command sequence is

```
java Distance x0 y0 x1 y1
```

to compute the distance between the points (x_0, y_0) and (x_1, y_1)

For example, the command

```
java Distance 0 3 4 0
```

will compute the distance between the points (0,3) and (4,0) and print the following:

```
Distance between java.awt.Point[x=0,y=3] and java.awt.Point[x=4,y=0]
is 5.0
```

The command

```
java Distance -4 5 11 19
```

will compute the distance between the points (-4,5) and (11,19):

```
Distance between java.awt.Point[x=-4,y=5] and
java.awt.Point[x=11,y=19] is 20.518284528683193
```

We'll explain this strange looking output, and also show how to improve it, later.

9.2 Detailed Program Structure and Overview

As promised, we will now provide a detailed description of this Java program. We will discuss the syntax and structure of the program and the meaning of that structure.

1 Chapter 6 on page 27
2 Chapter 5 on page 25

9.2.1 Introduction to Java Syntax

The *syntax* of a Java class is the characters and symbols and their structure used to code the class using Unicode characters. A fuller treatment of the syntax elements of Java may be found at SYNTAX[3]. We will provide here only enough description of the syntax to grasp the above program.

Java programs consist of a sequence of tokens. There are different kinds of tokens. For example, there are word tokens such as `class` and `public` which represent KEYWORDS[4] - special words with reserved meaning in Java. Other words (non keywords such as `Distance`, `point0`, `x1`, and `printDistance`) are *identifiers*. Identifiers have many different uses in Java but primarily they are used as names. Java also has tokens to represent numbers, such as `1` and `3`; these are known as LITERALS[5]. STRING LITERALS[6], such as `"Distance between "`, consist of zero or more characters embedded in double quotes, and OPERATORS[7] such as `+` and `=` are used to express basic computation such as addition or String concatenation or assignment. There are also left and right braces (`{` and `}`) which enclose BLOCKS[8]. The body of a class is one such block. Some tokens are punctuation, such as periods `.` and commas `,` and semicolons `;`. You use WHITESPACE[9] such as spaces, tabs, and newlines, to separate tokens. For example, whitespace is required between keywords and identifiers: `publicstatic` is a single identifier with twelve characters, not two Java keywords.

9.2.2 Declarations and Definitions

Sequences of tokens are used to construct the next building blocks of Java classes: declarations and definitions. A class declaration provides the name and visibility of a class. For our example,

```
public class Distance
```

is the class declaration. It consists (in this case) of two keywords, `PUBLIC`[10] and `CLASS`[11] followed by the identifier `Distance`.

This means that we are defining a class named `Distance`. Other classes, or in our case, the command line, can refer to the class by this name. The `public` keyword is an ACCESS MODIFIER[12] which declares that this class and its members may be accessed from other classes. The `class` keyword, obviously, identifies this declaration as a class. Java also allows declarations of INTERFACES[13] and (as of Java 5) ANNOTATIONS[14].

3 Chapter 10 on page 57
4 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS
5 Chapter 10.2 on page 59
6 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FSTRING%20LITERALS
7 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FOPERATORS
8 Chapter 10.3 on page 60
9 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FWHITESPACE
10 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FPUBLIC
11 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FCLASS
12 Chapter 15 on page 87
13 Chapter 25 on page 137
14 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FANNOTATIONS

The class declaration is then followed by a block (surrounded by curly braces) which provides the class's definition. The definition is the implementation of the class - the declaration and definitions of the class's members. This class contains exactly six members, which we will explain in turn.

1. Two field declarations, named `point0` and `point1`
2. A constructor declaration
3. Three method declarations

Example: Instance Fields

The declaration

```
private java.awt.Point point0, point1;
```

declares two INSTANCE FIELDS[15]. Instance fields represent named values that are allocated whenever an instance of the class is constructed. When a Java program creates a `Distance` instance, that instance will contain space for `point0` and `point1`. When another `Distance` object is created, it will contain space for its *own* `point0` and `point1` values. The value of `point0` in the first `Distance` object can vary independently of the value of `point0` in the second `Distance` object.

This declaration consists of:

1. The `PRIVATE`[16] access modifier,
   which means these instance fields are not visible to other classes.
2. The type of the instance fields. In this case, the type is `java.awt.Point`.
   This is the class `Point` in the `java.awt` package.
3. The names of the instance fields in a comma separated list.

These two fields could also have been declared with two separate but more verbose declarations,

```
private java.awt.Point point0;
private java.awt.Point point1;
```

Since the types of these fields is a reference type (i.e. a field that *refers to* or can hold a *reference to* an object value), Java will implicitly initialize the values of `point0` and `point1` to null when a `Distance` instance is created. The null value means that a reference value does not refer to an object. The special Java literal, `null` is used to represent the null value in a program. While you can explicitly assign null values in a declaration, as in

```
private java.awt.Point point0 = null;
private java.awt.Point point1 = null;
```

it is not necessary and most programmers omit such default assignments.

15 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FINSTANCE%20FIELDS
16 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FPRIVATE

Example: Constructor

A CONSTRUCTOR[17] is a special method in a class which is used to construct an instance of the class. The constructor can perform initialization for the object, beyond that which the Java VM does automatically. For example, Java will automatically initialize the fields `point0` and `point1` to null.

Below is the constructor for this class. It consists of five parts:

1. The optional ACCESS MODIFIER(S)[18].
   In this case, the constructor is declared `public`
2. The constructor name, which must match the class name exactly: `Distance` in this case.
3. The constructor PARAMETERS[19].
   The parameter list is required. Even if a constructor does not have any parameters, you must specify the empty list `()`. The parameter list declares the type and name of each of the method's parameters.
4. An optional **throws** clause which declares the exceptions that the constructor may throw. This constructor does not declare any exceptions.
5. The constructor body, which is a Java BLOCK[20] (enclosed in `{}`). This constructor's body contains two statements.

`public Distance(int x0, int y0, int x1, int y1) { point0 = new java.awt.Point(x0, y0); point1 = new java.awt.Point(x1, y1); }` This constructor accepts four parameters, named `x0`, `y0`, `x1` and `y1`. Each parameter requires a parameter type declaration, which in this example is **int** for all four parameters. Java integer values are signed, 32 bit twos complement integers. The parameters in the parameter list are separated by commas.

The two assignments in this constructor use Java's **new** *operator* to allocate two `java.awt.Point` objects. The first allocates an object representing the first point, `(x0, y0)`, and assigns it to the `point0` instance variable (replacing the null value that the instance variable was initialized to). The second statement allocates a second `java.awt.Point` instance with `(x1, y1)` and assigns it to the `point1` instance variable.

This is the constructor for the Distance class. Distance implicitly extends from `java.lang.Object`. Java inserts a call to the super constructor as the first executable statement of the constructor if there is not one explicitly coded. The above constructor body is equivalent to the following body with the explicit super constructor call:

```
{
   super();
   point0 = new java.awt.Point(x0, y0);
   point1 = new java.awt.Point(x1, y1);
}
```

17 Chapter 16.7 on page 95
18 Chapter 15 on page 87
19 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FPARAMETERS
20 Chapter 10.3 on page 60

While it is true that this class could be implemented in other ways, such as simply storing the coordinates of the two points and computing the distance as $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$, this class instead uses the existing `java.awt.Point` class. This choice matches the abstract definition of this class: to print the distance between two points on the plane. We take advantage of existing behavior already implemented in the Java platform rather than implementing it again. We will see later how to make the program more flexible without adding much complexity, because we choose to use object abstractions here. However, the key point is that this class uses information hiding. That is, *how* the class stores its state or how it computes the distance is hidden. We can change this implementation without altering how clients use and invoke the class.

Example: Methods

Methods are the third and most important type of class member. This class contains three *methods* in which the behavior of the Distance class is defined: `printDistance()`, `main()`, and `intValue()`

The printDistance() method

The `printDistance()` method prints the distance between the two points to the standard output (normally the console).

```
   public void printDistance()
   {
      System.out.println("Distance between " + point0
                         + " and " + point1
                         + " is " + point0.distance(point1));   }
   }
```

This INSTANCE METHOD[21] executes within the context of an implicit `Distance` object. The instance field references, `point0` and `point1`, refer to instance fields of that implicit object. You can also use the special variable `this` to explicitly reference the current object. Within an instance method, Java binds the name `this` to the object on which the method is executing, and the type of `this` is that of the current class. The body of the `printDistance` method could also be coded as

```
      System.out.println("Distance between " + this.point0
                         + " and " + this.point1
                         + " is " +
this.point0.distance(this.point1));   }
```

to make the instance field references more explicit.

This method both computes the distance and prints it in one statement. The distance is computed with `point0.distance(point1);` `distance()` is an instance method of the `java.awt.Point` class (of which `point0` and `point1` are instances. The method operates on `point0` (binding `this` to the object that `point0` refers to during the execution of the method)

21 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FINSTANCE%20METHODS

and accepting another Point as a parameter. (Actually, it is slightly more complicated than that, but we'll explain later.) The result of the `distance()` method is a double precision floating point number.

This method uses the syntax

```
"Distance between " + this.point0
+ " and " + this.point1
+ " is " + this.point0.distance(this.point1)
```

to construct a String to pass to the `System.out.println()`. This expression is a series of STRING CONCATENATION[22] methods which concatenates Strings or the String representation of primitive types (such as doubles) or objects, and returns a long string. For example, the result of this expression for the points (0,3) and (4,0) is the String

```
"Distance between java.awt.Point[x=0,y=3] and java.awt.Point[x=4,y=0]
is 5.0"
```

which the method then prints to `System.out`.

In order to print, we invoke the `println()`. This is an instance method from `java.io.PrintStream`, which is the type of the static field `out` in the class `java.lang.System`. The Java VM binds `System.out` to the standard output stream when it starts a program.

The main() method

The `main()` method is the main entry point which Java invokes when you start a Java program from the command line. The command

```
java Distance 0 3 4 0
```

instructs Java to locate the Distance class, put the four command line arguments into an array of String values, then pass those arguments the `public static main(String[])` method of the class. (We will introduce arrays shortly.) Any Java class that you want to invoke from the command line or desktop shortcut must have a main method with this signature.

```
public static void main(String[] args)
{
   Distance dist = new Distance(
         intValue(args[0]), intValue(args[1]),
         intValue(args[2]), intValue(args[3]));
   dist.printDistance();
}
```

22 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FSTRING%20CONCATENATION

The `main()` method invokes the final method, `intValue()`, four times. The `intValue()` takes a single string parameter and returns the integer value represented in the string. For example, `intValue("3")` will return the integer 3.

The `intValue()` method

The `intValue()` method delegates its job to the `Integer.parseInt()` method. The main method could have called `Integer.parseInt()` directly; the `intValue()` method simply makes the `main()` method slightly more readable.

```
private static int intValue(String data)
{
   return Integer.parseInt(data);
}
```

This method is PRIVATE[23] since, like the fields `point0` and `point1`, it is part of the internal implementation of the class and is not part of the external programming interface of the `Distance` class.

Static vs. Instance Methods

Both the `main()` and `intValue()` methods are STATIC METHODS[24]. The static keyword tells the compiler to create a single memory space associated with the class. Each individual object instantiated has its own private state variables and methods but use the same **static** methods and members common to the single class object created by the compiler when the first class object is instantiated or created. This means that the method executes in a static or non-object context - there is no implicit separate instance available when the static methods run from various objects, and the special variable `this` is not available. As such, static methods cannot access instance methods or instance fields (such as `printDistance()`) or `point0`) directly. The `main()` method can only invoke the instance method `printDistance()` method via an instance reference such as `dist`.

9.2.3 Data Types

Most declarations have a data type. Java has several categories of data types: reference types, primitive types, array types, and a special type, void.

Reference Types

A *reference type* is a Java data type which is defined by a Java class or interface. Reference types derive this name because such values *refer to* an object or contain a *reference to* an object. The idea is similar to pointers in other languages like C.

23 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FPRIVATE
24 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FSTATIC%20METHODS

Java represents sequences of character data, or STRING[25], with the reference type `java.lang.String` which is most commonly referred to as `String`. *String literals*, such as `"Distance between "` are constants whose type is String.

This program uses three separate reference types:

1. java.lang.String (or simply String)
2. Distance
3. java.awt.Point

For more information see chapter : JAVA PROGRAMMING/CLASSES, OBJECTS AND TYPES[26].

Primitive Types

In addition to object or reference types, Java supports PRIMITIVE TYPES[27]. The primitive types are used to represent Boolean, character, and numeric values. This program uses only one primitive type explicitly, **int**, which represents 32 bit signed integer values. The program also implicitly uses **double**, which is the return type of the `distance()` method of `java.awt.Point`. `double` values are 64 bit IEEE floating point values. The `main()` method uses integer values 0, 1, 2, and 3 to access elements of the command line arguments. The `Distance()` constructor's four parameters also have the type `int`. Also, the `intValue()` method has a return type of `int`. This means a call to that method, such as `intValue(args[0])`, is an expression of type `int`. This helps explain why the main method cannot call

```
new Distance(args[0], args[1], args[2], args[3]) // this is an error
```

Since the type of the `args` array element is String, and our constructor's parameters must be `int`, such a call would result in an error because Java cannot automatically convert values of type String into `int` values.

Java's primitive types are **boolean**, **byte**, **char**, **short**, **int**, **long**, **float** and **double**, each of which are also Java language keywords.

Array Types

Java supports ARRAYS[28], which are aggregate types which have a fixed element type (which can be any Java type) and an integral size. This program uses only one array, `String[] args`. This indicates that `args` has an array type and that the element type is `String`. The Java VM constructs and initializes the array that is passed to the `main` method. See ARRAYS[29] for more details on how to create arrays and access their size.

The elements of arrays are accessed with integer indices. The first element of an array is always element 0. This program accesses the first four elements of the `args` array explicitly with the

25 Chapter 19 on page 107
26 Chapter 12 on page 75
27 Chapter 17 on page 99
28 Chapter 29 on page 145
29 Chapter 29 on page 145

indices 0, 1, 2, and 3. (This program does *not* perform any input validation, such as verifying that the user passed at least four arguments to the program. We will fix that later.)

void

VOID[30] is not a type in Java; it represents the absence of a type. Methods which do not return values are declared as `void` methods.

This class defines two `void` methods:

```
public static void main(String[] args) { ... }
public void printDistance()  { ... }
```

9.3 Comments in Java programs

See HERE[31] for more information on that important topic.

30 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FVOID

31 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA_PROGRAMMING%2FUNDERSTANDING_A_JAVA_PROGRAM%2FJAVADOC_AND_OTHER_COMMENTS

10 Syntax

Java derives much of its *syntax* from the C[1] programming language: basic assignment statement syntax, expressions, control flow statements and blocks, etc. will be very familiar to C programmers.

Unicode : Java source code are built by Unicode characters.

Tokens : Java programs consist of a sequence of different kinds of tokens. For example, there are word tokens such as **class** and **public** which are KEYWORDS[2].

KEYWORDS[3] : Those are special words with reserved meaning in Java. Those words can not be used by the programers to name identifiers.

Identifiers : Other words (non keywords) are identifiers. Identifiers have many different uses in Java but primarily they are used as names, class names, method names, and variable names... .

LITERALS[4] : Java also has tokens to represent numbers, such as `1` and `3`; these are known as LITERALS[5].

String LITERALS[6], such as `"http://en.wikibooks.org/Java_Programming"`, consist of zero or more characters embedded in double quotes.

OPERATORS[7] : And OPERATORS[8] such as + and = are used to express basic computation such as addition or String concatenation or assignment.

BLOCKS[9] : There are also left and right braces ({ and }) which enclose BLOCKS[10]. The body of a class is one such block.

STATEMENTS[11] : A Block contains one or more Java STATEMENT(S)[12], separated by semicolons. A statement is the smallest building block of Java.

Separators : Some tokens are punctuation, such as periods `.` and commas `,` and semicolons `;`.

---

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3AC
2 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS
4 Chapter 10.2 on page 59
5 Chapter 10.2 on page 59
6 Chapter 10.2 on page 59
7 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FOPERATORS
8 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FOPERATORS
9 Chapter 10.3 on page 60
10 Chapter 10.3 on page 60
11 Chapter 11 on page 63
12 Chapter 11 on page 63

WHITESPACE[13] : You use WHITESPACE[14] such as spaces, tabs, and newlines, to separate tokens. For example, whitespace is required between keywords and identifiers: `publicstatic` is a single identifier with twelve characters, not two Java keywords.

COMMENTS[15] : Comments are not part of the executing code. Comments are used to document the code.

10.1 Unicode

Most Java program text consists of ASCII[16] characters, but any Unicode character can be used as part of identifier names, in comments, and in character and string literals. UNICODE ESCAPE SEQUENCES[17] may also be used to express a Unicode character.

For example, π (which is the Greek Lowercase Letter **pi**) is a valid Java identifier.

```
double π = Math.PI;
```

and in a string literal

```
String pi = "\u03C0";
```

may also be represented in Java as the *Unicode escape sequence* `\u03C0`. Thus, the following is a valid, but not very readable, declaration and assignment:

```
double \u03C0 = Math.PI;
```

The following demonstrate the use of Unicode escape sequences in other Java syntax:

```
// Declare Strings pi and quote which contain \u03C0 and \u0027
// respectively:

String pi = "\u03C0";
String quote = "\u0027";
```

Note that a Unicode escape sequence functions just like any other character in the source code. E.g., `\u0022` (double quote, ") needs to be quoted in a string just like \".

13 Chapter 10.4 on page 60
14 Chapter 10.4 on page 60
15 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FSYNTAX%2FCOMMENTS
16 HTTP://EN.WIKIPEDIA.ORG/WIKI/ASCII
17 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FSYNTAX%2FUNICODE%20ESCAPE%20SEQUENCES

```
Declare Strings doubleQuote1 and doubleQuote2 which both contain "
 (double quote):


String doubleQuote1 = string;
String doubleQuote2 = string;

"\u0022" doesn't work since """ doesn't work.
```

See UNICODE ESCAPE SEQUENCES[18] for full details.

CATEGORY:JAVA PROGRAMMING[19]

10.2 Literals

Java Literals are syntactic representations of boolean, character, numeric, or string data. Literals provide a means of expressing specific values in your program. For example, in the following statement, an integer variable named `count` is declared and assigned an integer value. The literal **0** represents, natually enough, the value zero.

```
int count = 0;
```

The following method call passes a String literal **string** the boolean literal **true** and the special null value **null** to the method `parse()`:

```
List items = parse(string, true, null);
```

- BOOLEAN LITERALS[20]
- NUMERIC LITERALS[21]
  - CHARACTER LITERALS[22]
  - INTEGER LITERALS[23]
  - FLOATING POINT LITERALS[24]

18 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FSYNTAX%2FUNICODE%20ESCAPE%20SEQUENCES

19 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

20 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FBOOLEAN%20LITERALS

21 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FNUMERIC%20LITERALS

22 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FNUMERIC%20LITERALS%2FCHARACTER%20LITERALS

23 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FNUMERIC%20LITERALS%2FINTEGER%20LITERALS

24 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FNUMERIC%20LITERALS%2FFLOATING%20POINT%20LITERALS

- STRING LITERALS[25]
- **null**[26]

10.3 Blocks

Java has a concept called block that is enclosed between the { and } characters, called curly braces. A block executed as a single statetement, and can be used where a single statetement is accepted.

After a block is executed all local variables defined inside the block is discarded, go out of scope.

```
{
...
 // -- This is a block ---
}
```

Blocks can be nested:

```
{
...
  {
     // -- This is a nested block ---
  }
}
```

CATEGORY:JAVA PROGRAMMING[27] lllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllll

10.4 Whitespaces

Whitespace in Java is used to separate the tokens in a Java source file. Whitespace is required in some places, such as between ACCESS MODIFIERS[28], TYPE NAMES[29] and Identifiers, and is used to improve readability elsewhere.

Wherever whitespace is required in Java, one or more whitespace characters may be used. Wherever whitespace is optional in Java, zero or more whitespace characters may be used.

Java whitespace consists of the

- space character ' ' (0x20),
- the tab character (hex 0x09),
- the form feed character (hex 0x0c),
- the line separators characters newline (hex 0x0a) or carriage return (hex 0x0d) characters.

25 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FSTRING%20LITERALS
26 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FLITERALS%2FNULL
27 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING
28 Chapter 15 on page 87
29 Chapter 18 on page 101

Line separators are special whitespace characters in that they also terminate line comments, whereas normal whitespace does not.

Other Unicode space characters, including vertical tab, are not allowed as whitespace in Java.

10.5 Required Whitespace

Below is the declaration of an **abstract** method taken from a Java class

```
public abstract Distance distanceTo(Destination dest);
```

Whitespace is required between **public** and **abstract**, between **abstract** and `Distance`, between `Distance` and `distanceTo`, and between `Destination` and `dest`.

However, the following is not legal:

```
publicabstractDistance distanceTo(Destination dest);
```

because whitespace is required between keywords and identifiers. The following is lexically valid

```
publicabstractDistance distanceTo(Destination dest);
```

but means something completely different: it declares a method which has the return type `publicabstractDistance` It is unlikely that this type exists, so the above would result in a semantic error.

10.6 Indentation

Java ignores all whitespace in front of a statement. As this, these two code snippets are identical for the compiler:

```
public static void main(String[] args) {
    printMessage();
}

void printMessage() {
    System.out.println("Hello World!");
}
```

```
public static void main(String[] args) {
printMessage();
}

void printMessage() {
System.out.println("Hello World!");
}
```

However, the first one's style (with whitespace) is preferred, as the readability is higher. (The method body is easier to distinguish from the head, even at a higher reading speed.)

CATEGORY:JAVA PROGRAMMING[30]

CATEGORY:JAVA PROGRAMMING[31]

30 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING
31 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

11 Statements

Now, that we have the Java platform on our systems and have run the first program successfully, we are geared towards understanding how programs are actually made. As we have already discussed. A program is a set of instructions – very simple tasks provided to a computer. These instructions are called **statements** in Java. Statements can be anything from a single line of code to a complex mathematical equation. This section helps you understand what statements are and how they work.

11.1 What exactly are statements?

Statements are a single instruction in a program – a single unit of code. Consider the following line:

Listing 1.1: A simple assignment statement.

```
int age = 24;
```

This line is a simple instruction that tells the system to initialize a variable and set its value as 24. Within that simple definition, we talked about initialization of a variable and setting its value. This all might sound a bit too technical, but it will make sense as you read ahead.

11.2 Where do you find statements

Java, in the same style as C and C++, places statements within functions (or methods). The function in turn is placed within a class declaration. If the above statement was the only one in the program, it would look similar to this:

```
public class MyProgram
{
    public static void main (String[] args)
    {
        int age = 24;
    }
}
```

The class declaration and function declaration will be described in the upcoming chapters.

11.3 Variables

Christmases are usually exciting, birthdays too. And the one thing that makes them exciting are: you get presents. Think of a present you've ever gotten. A tiny (or big, if you're lucky) box with your name on it. Now think of variables as something similar. They are tiny little boxes in the computer's memory that save something within themselves. This something within them is called a value.

Note:
A **variable** is an identifier to a value in the system's memory.

11.4 Data types

Take a look at the code in Listing 1.1. Here the variable we have just created is `age`. The word `int` tells us what is inside the `age` variable. `int` actually stands for integer – a number. On the right to this variable is the value of the variable which is the number `24`. Just like `int`, we can use `byte`, `short`, `long`, `double`, `float`, `boolean`, `char` or `String`. All these tell us what type of data is within a variable. These are hence called **data types**.

We explored the statement in Listing 1.1, where the variable held an integer within in. Let's put another type of data within it. Let's say, the number `10.5`. The code would look something like this:

Listing 1.2: Putting a number with decimal point inside an integer variable.

```
int age = 10.5;
```

This is actually wrong. By definition (if you have been awake throughout your mathematics lectures) you'd know that integers are whole numbers: 0, 1, 2, all the way up to infinity. Anything with a decimal point is not an integer, hence the statement by virtue of it is wrong.

What would make it right is when you assign a certain type to the variable that would accept numbers with decimal points – numbers with decimal points are called **floating points**.

11.5 Whole numbers and floating point numbers

The data types that one can use for whole numbers are `byte`, `short`, `int` and `long` but when it comes to floating point numbers, we use `float` or `double`. Now that we know that, we can modify the code in Listing 1.2 as:

Listing 1.3: The correct way to assign a type to floating point variables

```
double age = 10.5;
```

Why not `float`, you say? Well, there are several reasons why not. `10.5` can be anything – a `float` or a `double` but by a certain rule, it is given a certain type. This can be explained further by looking at the table below.

Data type	Values accepted	Declaration
`byte`	Any number between `-128` and `127`.	byte b = 123;
`short`	Any number between `-32,768` and `32,767`.	short s = 12345;
`int`	Any number between `-2,147,483,648` and `2,147,483,647`.	int i = 1234567;
`long`	Any number between `-9,223,372,036,854,775,808` and `9,223,372,036,854,775,807`.	long l = 1234567890L;
`float`	Extremely large numbers beyond the scope of discussion here.	float f = 123.4567f;
`double`	Extremely large numbers beyond the scope of discussion here. The only difference between `double` and `float` is the addition of an `f` as a suffix after the `float` value.	double d = 1234.56789;

The above table only list the number data types. We will look at the others as we go on. So, did you notice why we used a `double` in listing 1.3, and not a `float`? The answer is pretty simple. If we'd used a `float`, we would have to append the number with a `f` as a suffix, so `10.5` should be `10.5f` as in:

Listing 1.4: The correct way to define floating point numbers of type `float`.

```
float age = 10.5f;
```

11.6 Assignment statements

Up until now, we've assumed the creation of variables as a single statement. In essence, we assign a value to those variables, and that's just what it is called. When you assign a value to a variable in a statement, that statement is called an **assignment statement**. Did you notice one more thing? The semicolon (;). It's at the end of each statement. A clear indicator that a line of code is a statement is its termination with an ending semicolon. If one was to write multiple statements, it is usually done on each separate line ending with a semicolon. Consider the example below:

Listing 1.5: Multiple assignment statements.

```
int a = 10;
int b = 20;
int c = 30;
```

You do not necessarily have to use a new line to write each statement. Just like English, you can begin writing the next statement where you ended the first one as depicted below:

Listing 1.6: Multiple assignment statement on the same line.

```
int a = 10; int b = 20; int c = 30;
```

However, the only problem with writing such code is, it's very difficult to read it back. It doesn't look that intimidating at first, but once you've got a significant amount of code, it's usually better to organize it in a way that makes sense. It would look more complex and incomprehensible written as it is in Listing 1.6.

Now that we have looked into the anatomy of a simple assignment statement, we can look back at what we've achieved. We know that...

- A statement is a unit of code in programming.
- If we are assigning a variable a value, the statement is called an assignment statement.
- An assignment statement include three parts: a data type, variable name (also called an identifier) and the value of a variable. We will look more into the nature of identifiers and values in the section titled IDENTIFIERS, LITERALS AND EXPRESSIONS[1] later.

Now, before we more on to the next topic, you need to try and understand what the code below does.

Listing 1.7: Multiple assignment statements with expressions

```
int firstNumber = 10;
int secondNumber = 20;
int result = firstNumber + secondNumber;
```

The first two statements are pretty much similar to those in Listing 1.5 but with different variable names. The third however is a bit interesting. We've already talked of variables as being similar to gift boxes. Think of your computer's memory as a shelf where you put all those boxes. Whenever you need a box (or variable), you call its identifier (that's the name of the variable). So calling the variable identifier `firstNumber` gives you the number `10`, calling `secondNumber` would give you `20` hence when you add the two up, the answer should be `30`. That's what the value of the last variable `result` would be. The part of the third statement where you add the numbers, i.e., `firstNumber + secondNumber` is called an **expression** and the expression is what decides what the value is to be. If it's just a plain value, nothing fancy, then it's called a **literal**.

With the information you have just attained, you can actually write a decent Java program that can sum up values. To learn more, continue reading.

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FIDENTIFIERS%2C%20LITERALS%20AND%20EXPRESSIONS

<!--

11.7 Program Control Flow

Statements are evaluated in the order as they occur. The execution of flow begins at the top most statement and proceed downwards till the last statement is encountered. A statement can be substituted by a statement block. There are special statements that can redirect the execution flow based on a condition, those statements are called *branching* statements, described in detail in a later section.

11.8 Statement Blocks

A bunch of statements can be placed in braces to be executed as a single block. Such a block of statement can be named or be provided a condition for execution. Below is how you'd place a series of statements in a block.

```
{
    int a = 10;
    int b = 20;
    int result = a + b;
}
```

11.9 Branching Statements

Program flow can be affected using function/method calls, loops and iterations. Of various types of branching constructs, we can easily pick out two generic branching methods.

- Unconditional Branching
- Conditional Branching

11.9.1 Unconditional Branching Statements

If you'd closely look at a method, you'll see that a method is a named statement block that is executed by calling that particular name. An unconditional branch is created either by invoking the method or by calling `break`, `continue`, `return` or `throw`, all of which are described in below.

When a name of a method is encountered in a flow, it stops execution in the current method and branches to the newly called method. After returning a value from the called method, execution picks up at the original method on the line below the method call.

```
public class UnconditionalBranching
{
    public static void main(String[] args)
    {
```

```
        System.out.println("Inside main method! Invoking aMethod!");
        aMethod();
        System.out.println("Back in main method!");
    }
    public static void aMethod()
    {
        System.out.println("Inside aMethod!");
    }
}
```

Running the above code would provide us with this screen of information.

```
Inside main method! Invoking aMethod!
Inside aMethod!
Back in main method!
```

The program flow begins in the `main` method. Just as `aMethod` is invoked, the flow travels to the called method. At this very point, the flow branches to the other method. Once the method is completed, the flow is returned to the point it left off and resumes at the next statement after the call to the method.

11.9.2 Conditional Branching Statements

Conditional branching is attained with the help of the `if...else` and `switch` statements. A conditional branch occurs only if a certain condition expression evaluates to true.

Conditional Statements

Also referred to as **if** *statements*, these allow a program to perform a test and then take action based on the result of that test.

The form of the **if** statement:

```
if (condition) {
   do statements here if condition is true
} else {
   do statements here if condition is false
}
```

The *condition* is a boolean expression which can be either **true** or **false**. The actions performed will depend on the value of the condition.

Example:

```
if ( i > 0 ){
   System.out.println("value stored in i is greater than zero");
}
else {
   System.out.println("value stored is not greater than zero");
}
```

If statements can also be made more complex using the else if combination:

```
if (condition 1) {
    do statements here if condition 1 is true
}
else if (condition 2) {
    do statements here if condition 1 is false and condition 2 is true
} else {
   do statements here if neither condition 1 nor condition 2 is true
}
```

Example:

```
if ( i > 0 ){
   System.out.println("value stored in i is greater than zero");
}
else if (i < 0){
   System.out.println("value stored in i is less than zero");
}
else {
   System.out.println("value stored is equal to 0");
}
```

If there is only one statement to be executed after the condition, as in the above example, it is possible to omit the curly braces, however Sun's JAVA CODE CONVENTIONS[2] explicitly state that the braces should always be used.

There is no looping involved in an if statement so once the condition has been evaluated the program will continue with the next instruction after the statement.

If...else statements

The **if** ... **else** statement is used to conditionally execute one of two blocks of statements, depending on the result of a boolean condition.

Example:

```
if (list == null) {

  this block of statements executes if the condition is true



}
else {

  this block of statements executes if the condition is false
```

2 HTTP://JAVA.SUN.COM/DOCS/CODECONV/HTML/CODECONVENTIONS.DOC6.HTML#449

```
}
```

Sun's JAVA CODE CONVENTIONS[3] recommend that the braces should always be used.

An **if** statement has two forms:

```
if (boolean-condition)
   statement1
```

and

```
if (boolean-condition)
   statement1
else
   statement2
```

Use the second form if you have different statements to execute if the boolean-condition is true or if it is false. Use the first if you only wish to execute $statement_1$ if the condition is true and you do not wish to execute alternate statements if the condition is false.

The following example calls two **int** methods, `f()` and `y()`, stores the results, then uses an **if** statement to test if `x` is less than y and if it is, the $statement_1$ body will swap the values. The end result is x always contains the larger result and y always contains the smaller result.

```
int x = f();
int y = y();
if ( x < y ) {
  int z = x;
  x = y;
  y = z;
}
```

if...**else** statements also allow for the use of another statement, **else if**. This statement is used to provide another **if** statement to the conditional that can only be executed if the others are not true. For example:

```
if (x == 2)
 x = 4;
else if (x == 3)
 x = 6;
else
 x = -1;
```

The **else if** statement is useful in this case because if one of the conditionals is true, the other must be false. Keep in mind that if one is true, the other *will not* execute. For example, if the statement contained in the first conditional, `if(x == 2)`, were changed to `x = 3;`, the second conditional, the **else if**, would still not execute. However, when dealing with primitive types in

3 HTTP://JAVA.SUN.COM/DOCS/CODECONV/HTML/CODECONVENTIONS.DOC6.HTML#449

conditional statements, it is more desirable to use SWITCH STATEMENTS[4] rather than multiple **else if** statements.

Switch statements

The **switch** conditional statement is basically a shorthand version of writing many **if**...**else** statements. The syntax for **switch** statements is as follows:

```
switch(<variable>)
{
  case <result>: <statements>; break;
  case <result>: <statements>; break;
  default: <statements>; break;
}
```

This means that if the variable included equals one of the case results, the statements following that case, until the word `break` will run. The `default` case executes if none of the others are true. **Note:** the only types that can be analysed through **switch** statements are **char**, **byte**, **short**, or **int** primitive types. This means that **String** variables **can not** by analyzed through **switch** statements.

```
int n = 2, x;
switch(n)
{
 case 1: x = 2; break;
 case 2: x = 4; break;
 case 3: x = 6; break;
 case 4: x = 8; break;
}
return x;
```

In this example, since the integer variable `n` is equal to 2, `case 2` will execute, make `x` equal to 4. Thus, 4 is returned by the method.

11.10 Iteration Statements

Iteration Statements are statements that used to iterate a block of statements. Such statements are often referred to as loops. Java offers four kinds of iterative statements.

- The `while` loop
- The `do...while` loop
- The `for` loop
- The `foreach` loop

4 Chapter 11.9.2 on page 71

11.10.1 The while loop

Main Page: JAVA PROGRAMMING/KEYWORDS/WHILE[5] The `while` loop iterates a block of code while the condition it specifies is `true`.

The syntax for the loop is:

```
while (condition) {
   statement;
 }
```

Here the condition is an expression. An expression as discussed earlier is any statement that returns a value. `While` condition statements evaluate to a boolean value, that is, either `true` or `false`. As long as the condition is `true`, the loop will iterate the block of code over and over and again. Once the condition evaluates to `false`, the loop exits to the next statement outside the loop.

11.10.2 The do...while loop

The do-while loop is functionally similar to the while loop, except the condition is evaluated AFTER the statement executes

```
do {
   statement;
 } while (condition);
```

11.10.3 The for loop

Main Page: JAVA PROGRAMMING/KEYWORDS/FOR[6] The for loop is a specialized while loop whose syntax is designed for easy iteration through a sequence of numbers Example:

```
 for (int i = 0; i < 100; i++) {
    System.out.println(i + "\t" + i * i);
 }
```

If you compile and run the statement above, the program will print the numbers 0 to 99 and their squares

```
0       0
1       1
2       4
3       9
...
99      9801
```

5 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FWHILE
6 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FFOR

The same statement in a while loop:

```
int i = 0;
while (i < 100){
   System.out.println(i + "\t" + i * i);
   i++;
}
```

11.10.4 The foreach loop

The foreach statement allows you to iterate through all the items in a collection, examining each item in turn while still preserving its type. The syntax for the foreach statement is:

```
for (type item : collection) statement;
```

For an example, we'll take an array of `String`s denoting days in a week and traverse through the collection, examining one item at a time.

```
String[] days = {"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"};

for (String day : days) {
    System.out.println(day);
}
```

The output of this program would be:

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Notice that the loop automatically exits after the last item in the collection has been examined in the statement block.

Although the enhanced for loop can make code much clearer, it can't be used in some common situations.

- **Only access.** Elements can not be assigned to, eg, not to increment each element in a collection.
- **Only single structure.** It's not possible to traverse two structures at once, eg, to compare two arrays.
- **Only single element.** Use only for single element access, eg, not to compare successive elements.
- **Only forward.** It's possible to iterate only forward by single steps.
- **At least Java 5.** Don't use it if you need compatibility with versions before Java 5.

11.11 The continue and break statements

At times, you would like to re-iterate a loop without executing the remaining statement within the loop. The **continue** statement causes the loop to re-iterate and start over from the top most statement inside the loop.

Where there is an ability to re-iterate the loop, there is an ability to exit the loop when required. At any given moment, if you'd like to exit a loop and end all further work within the loop, the **break** ought to be used.

The **continue** and **break** statements can be used with a label like follows:

```
String s = "A test string for the switch!\nLine two of test
string...";
outer: for (int i=0;i<s.length();i++) {
    switch (s.charAt(i)) {
        case '\n': break outer;
        case ' ': break;
        default: System.out.print(s.charAt(i));
    }
}
```

Compiling and running this statement will produce

```
Ateststringfortheswitch!
```

CATEGORY:JAVA PROGRAMMING[7]

7 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

12 Classes, Objects and Types

12.1 Objects and Classes

An **object** is composed of **members** and **methods**. The members, also called *data members*, *characteristics*, *attributes*, or *properties*, describe the object. The methods generally describe the actions associated with a particular object. Think of an object as a noun, its members as adjectives describing that noun, and its methods as the verbs that can be performed by or on that noun.

For example, a sports car is an object. Some of its members might be its height, weight, acceleration, and speed. An object's members just hold data about that object. Some of the methods of the sports car could be "drive", "park", "race", etc. The methods really don't mean much unless associated with the sports car, and the same goes for the members.

The blueprint that lets us build our sports car object is called a **class**. A class doesn't tell us how fast our sports car goes, or what color it is, but it does tell us that our sports car will have a member representing speed and color, and that they will be say, a number and a word (or hex color code), respectively. The class also lays out the methods for us, telling the car how to park and drive, but these methods can't take any action with just the blueprint - they need an object to have an effect.

In Java, a class is located in a file similar to its own name. If you want to have a class called SportsCar, its source file needs to be `SportsCar.java`. The class is created by placing the following in the source file:

```
public class SportsCar
{
   /* Insert your code here */
}
```

The class doesn't do anything yet, as you will need to add methods and member variables first.

12.2 Instantiation and Constructors

In order to get from class to object, we "build" our object by **instantiation**. Instantiation simply means to create an **instance** of a class. Instance and object are very similar terms and are sometimes interchangeable, but remember that an instance refers to a *specific object*, which was created from a class.

This instantiation is brought about by one of the class's methods, called a **constructor**. As its name implies, a constructor builds the object based on the blueprint. Behind the scenes, this means that computer memory is being allocated for the instance, and values are being assigned to the data members.

In general there are four constructor types: default, non-default, copy, and cloning.

A **default constructor** will build the most basic instance. Generally, this means assigning all the members values like null, zero, or an empty string. Nothing would stop you, however, from your default sports car color from being red, but this is generally bad programming style. Another programmer would be confused if your basic car came out red instead of say, colorless.

A **non-default constructor** is designed to create an object instance with prescribed values for most, if not all, of the object's members. The car is red, goes from 0-60 in 12 seconds, tops out at 190mph, etc.

A **copy constructor** is not included in the Java language, however one can easily create a constructor that do the same as a copy constructor. It's important to understand what it is. As the name implies, a copy constructor creates a new instance to be a duplicate of an already existing one. In Java, this can be also accomplished by creating the instance with the default constructor, and then using the assignment operator to equivocate them. This is not possible in all languages though, so just keep the terminology under your belt.

Java has the concepts of **cloning object**, and the end results are similar to copy constructor. Cloning an object is faster than creation with the new keyword, because all the object memory is copied at once to destination cloned object. This is possible by implementing the Cloneable interface, which allows the method Object.clone() to perform a field-by-field copy.

12.3 Type

When an object is created, a reference to the object is also created. The object can not be accessed directly in Java, only through this object reference. This object reference has a **type** assigned to it. We need this type when passing the object reference to a method as a parameter. Java does strong type checking.

Type is basically a list of features/operations, that can be performed through that object reference. The object reference type basically is a contract that guarantees that those operations will be there at run time.

When a car is created, it comes with a list of features/operations listed in the user manual that guarantees that those will be there when the car is used.

When you create an object from a class by default its type is the same as its class. It means that all the features/operations the class defined are there and available, and can be used. See below:

```
(new ClassName()).operations();
```

You can assign this to a variable having the same type as the class:

```
ClassName objRefVariable = new ClassName();
objRefVariable.operations();
```

You can assign the created object reference to the class super class, or to an interface the class implements:

```
SuperClass objectRef = new ClassName();   // features/operations list
are defined by the SuperClass class
..
Interface inter = new ClassName();  // features/operations list are
defined by the interface
```

In the car analogy, the created car may have different **Type** of drivers. We create separate user manuals for them, Average user manual, Power user manual, Child user manual, or Handicapped user manual. Each type of user manual describes only those features/operations appropriate for the type of driver. The Power driver may have additional gears to switch to higher speeds, that are not available to other type of users...

When the car key is passed from an adult to a child we replacing the user manuals, that is called **Type Casting**.

In Java, casts can occur in three ways:

- up casting: going up in the inheritance tree, until we reach the **Object**
- up casting: to an interface the class implements
- down casting: until we reach the class the object was created from

Type and **Type Casting** will be covered in more details later at 'JAVA PROGRAMMING/TYPES[1]' module.

12.4 Multiple classes in a Java file

Normally, a Java file can have one and only one **public** Java class. However, a given file can contain additional non-public classes.

```
public class OuterClass
{
    ...
}
class AdditionalClass
{
    ...
}
```

Because they have the "package (default)" access specifier, the 'AdditionalClass' can be accessed only in the same package.

These "additional" classes compile to separate ".class" bytecode files when compiled, just as if they were in separate source files. However, including multiple classes in one file may increase the difficulty in examining the structure of a given application.

1 Chapter 18 on page 101

12.5 External links

- CONSTRUCTORS, INTERACTIVE JAVA LESSON[2]

CATEGORY:JAVA PROGRAMMING[3]

2 HTTP://JAVALESSONS.COM/CGI-BIN/FUN/JAVA-TUTORIALS-MAIN.CGI?SES=AO789&CODE=CTR&SUB=FUN

3 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

13 Packages

13.1 Java Package / Name Space

Usually a Java application is built by many developers and it is common that third party modules/-classes are integrated. The end product can easily contain hundreds of classes. Class name collision is likely to happen. To avoid this a Java class can be put in a "name space". This "name space" in Java is called the **package**.

The Java package needs to be unique across Vendors to avoid name collisions. For that reason Vendors usually use their domain name in reverse order. That is guaranteed to be unique. For example a company called 'Your Company Inc.', would use a package name something like this: `com.yourcompany.yourapplicationname.yourmodule.YourClass`.

To put a class in a package, the `package` keyword is used at the top of each class file. For Example,

```
package com.yourcompany.yourapplication.yourmodule;
```

When we want to reference a Java class that is defined outside of the current package name space, we have to specify which package name space that class is in. So we could reference that class with something like *com.yourcompany.yourapplication.yourmodule.YourClass* . To avoid having to type in the package name each time when we want to reference an outside class, we can declare which package the class belongs to by using the **import** Java keyword at the top of the file. For Example,

```
import com.yourcompany.yourapplication.yourmodule.YourClass;
```

Then we can refer to that class by just using the class name *YourClass* .

In rare cases it can happen that you need to reference two classes having the same name in different packages. In those cases, you can not use the import keyword for both classes. One of them needs to be referenced by typing in the whole package name. For Example,

```
package com.mycompany.myapplication.mymodule;
...
import com.yourcompany.yourapplication.youmodule.SameClassName;
...
SameClassName yourObjectRef = new SameClassName();
com.hercompany.herapplication.hermodule.SameClassName herObjectRef =
   new com.hercompany.herapplication.hermodule.SameClassName();
```

The Java package has one more interesting characteristic; the package name corresponds where the actual file is stored on the file system. And that is actually how the compiler and the class loader find the Java files on the file system. For example, the class *com.yourcompany.yourapplication.yourmodule.YourClass*, is stored on the file system in the corresponding directory : *com/yourcompany/yourapplication/yourmodule/YourClass.* Because of this, package names should be lowercase, since in some operating systems the directory names are not case sensitive.

13.2 Wildcard imports

It is possible to import an entire package, using an asterisk:

```
import javax.swing.*;
```

While it may seem convenient, it may cause problems if you make a typographical error. For example, if you use the above import to use JFrame, but then type `JFraim frame=new JFraim();`, the Java compiler will report an error similar to "Cannot find symbol: JFraim". Even though it seems as if it was imported, the compiler is giving the error report at the first mention of JFraim, which is half-way through your code, instead of the point where you imported JFrame along with everything else in javax.swing.

If you change this to `import javax.swing.JFraim;` the error will be at the import instead of within your code.

Furthermore, if you `import javax.swing.*;` and `import java.util.*;`, and javax.swing.Queue is later added in a future version of Java, your code that uses Queue (java.util) will fail to compile. This particular example is fairly unlikely, but if you are working with non-Sun libraries, it may be more likely to happen.

13.3 Importing packages from .jar files

If you are importing library packages or classes that reside in a `.jar` file, you must ensure that the file is in the current classpath (both at compile- and execution-time). Apart from this requirement, importing these packages and classes is the same as if they were in their full, expanded, directory structure.

Example:
To compile and run a class from a project's top directory (that contains the two directories `/source` and `/libraries`) you could use the following command:

```
javac -classpath libraries/lib.jar source/MainClass.java
```

And then to run it, similarly:

```
java -classpath libraries/lib.jar source/MainClass
```

(The above is simplified, and demands that MainClass be in the default package, or a package called 'source', which isn't very desirable.)

13.4 Class Loading / Name Space

A fully qualified class name : consist of the package name plus the class name.

For example, the fully qualified class name of `HashMap` is `java.util.HashMap`.
Sometime is can happen that two class has the same name, but it can not have on the same package, otherwise it would be the same class.
It can be said that the two class with the same name is in different name space. In the above example, the `HashMap` class is in the `java.util` name space.

Let be two `Customer` class with different name space (in different package).

- `com.bluecompany.Customer`
- `com.redcompany.Customer`

When we need to use both class in the same program file, we can use the **import** keyword only for one of the class. For the other we need to use the fully qualified name.

The runtime identity of a class in Java 2 : is defined by the fully qualified class name and its defining class loader. This means that the same class, loaded by two different class loaders, is seen by the Virtual Machine as two completely different types.

14 Nested Classes

In Java you can define a class inside an other class.

A class can be nested:

- inside another class,
- or inside a method

14.1 Nest a class inside a class

When a class is declared inside another class, the nested class' access modifier can be **public**, **private** or `package(default)`.

```
public class OuterClass
{
   private String outerInstanceVar;

   public class InnerClass
   {
      public void printVars()
      {
         System.out.println( "Print Outer Class Instance Var.:" + outerInstanceVar);
      }
   }
}
```

The inner class has access to the enclosing class instance's variables and methods, even private ones, as seen above. This makes it very different from the nested class in C++, which are equivalent to the "static" inner classes, see below.

An inner object has a reference to the outer object. The nested object can only be created with a reference to the 'outer' object. See below.

```
public void testInner()
{
    ...
    OuterClass outer = new OuterClass();
    OuterClass.InnerClass inner = outer.new InnerClass();
    ...
}
```

(When in a non-static method of the outer class, you can directly use `new InnerClass()`, since the class instance is implied to be `this`.)

You can directly access the reference to the outer object from within an inner class with the syntax `OuterClass.this`; although this is usually unnecessary because you already have access to its fields and methods.

Inner classes compile to separate ".class" bytecode files, usually with the name of the enclosing class, followed by a "$", followed by the name of the inner class. So for example, the above inner class would typically be compiled to a file named "OuterClass$InnerClass.class".

14.1.1 Static inner class

An inner class can be declared *static*. A static inner class has no enclosing instance, and therefore cannot access instance variables and methods of the outer class. You do not specify an instance when creating a static inner class. This is equivalent to the inner classes in C++.

14.2 Nest a class inside a method

These inner classes, also called *local classes*, cannot have access modifiers, like local variables, since the class is 'private' to the method. The inner class can be only **abstract** or **final**.

```
public class OuterClass
{
    public void method()
    {
        class InnerClass
        {

        }
    }
}
```

In addition to instance variables of the enclosing class, local classes can also access local variables of the enclosing method, but only ones that are declared `final`. This is because the local class instance might outlive the invocation of the method, and so needs its own copy of the variable. To avoid problems with having two different copies of a mutable variable with the same name in the same scope, it is required to be *final*, so it cannot be changed.

14.3 Anonymous Classes

In Java a class definition and its instantiation can be combined into a single step. By doing that the class does not require a name. Those classes are called anonymous classes. An anonymous class can be defined and instantiated in contexts where a reference can be used, and it is a nested class to an existing class. Anonymous class is a special case of the local class to a method, above; and hence they also can use final local variables of the enclosing method.

Anonymous classes are most useful to subclass and upcast to an 'Adapter Class' or to an interface.

```
public interface ActionListener
{
    public void click();
}
...
ActionListener clk = new ActionListener()
    {
        public void click()
        {
            // --- implementation of the click event ---
            ...
            return;
        }
    };
```

In the above example the class that implements the `ActionListener` is **anonymous**. The class is defined where it is instantiated.

The above code is harder to read than if the class explicitly defined, so why use it? If many implementations are needed for an interface and those classes are used only in one particular place, using **anonymous** class makes sense.

The following example uses anonymous class to implement an action listener.

```
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;
class MyApp implements Serializable
{
   BigObjectThatShouldNotBeSerializedWithAButton bigOne;
   Button aButton = new Button();
   MyApp()
   {
       aButton.addActionListener( new ActionListener()
           {
               public void actionPerformed(ActionEvent e)
               {
                   System.out.println("Hello There");
               }
           }
       );
   }
}
```

The following example does the same thing, but it names the class that implements the action listener.

```
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;
class MyApp implements Serializable
{
   BigObjectThatShouldNotBeSerializedWithAButton bigOne;
   Button aButton = new Button();
   // --- Nested class to implement the action listener ---
   class MyActionListener implements ActionListener
   {
       public void actionPerformed(ActionEvent e)
       {
```

```
            System.out.println("Hello There");
        }
    }
    MyApp()
    {
        aButton.addActionListener( new MyActionListener() );
    }
}
```

Using **anonymous** classes is especially preferable when you intend to use many different classes that each implement the same Interface.

CATEGORY:JAVA PROGRAMMING[1]

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

15 Access Modifiers

15.1 Access modifiers

You surely would have noticed by now, the words **public**, **protected** and **private** at the beginning of class's method declarations used in this book. These keywords are called the **access modifiers** in the Java language syntax, and can be defined as...

Quote:
.. keywords that help set the visibility and accessibility of a class, its member variables, and methods.

The following table shows what Access Modifiers are appropriate for classes, nested classes, member variables, and methods:

	Class	**Nested class**	**Method, or Member variable**	**Interface**	**Interface method signature**
public	visible from anywhere	same as its class	same as its class	visible from anywhere	visible from anywhere
protected	N/A	its class and its subclass	its class and its subclass, and from its package	N/A	N/A
package (default)	only from its package	only from its package	only from its package	only from its package	N/A, default is public
private	N/A	only from its class	only from its class	N/A	N/A

Points to ponder:
Note that Interface method visibility is `PUBLIC`[1] by default. You do not need to specify the access modifier it will default to `PUBLIC`[2]. For clarity it is considered a good practice to put the `PUBLIC`[3] keyword.
The same way all member variables defined in the Interface by default will become `STATIC`[4] `FINAL`[5] once inherited in a class.

If a class has public visibility, the class can be referenced by anywhere in the program. If a class has package visibility, the class can be referenced only in the package where the class is defined. If a class has private visibility, (it can happen only if the class is defined nested in an other class) the class can be accessed only in the outer class.

If a variable is defined in a public class and it has public visibility, the variable can be reference anywhere in the application through the class it is defined in. If a variable has package visibility, the variable can be referenced only in the same package through the class it is defined in. If a variable has private visibility, the variable can be accessed only in the class it is defined in.

If a method is defined in a public class and it has public visibility, the method can be called anywhere in the application through the class it is defined in. If a method has package visibility, the method can be called only in the same package through the class it is defined in. If a method has private visibility, the method can be called only in the class it is defined in.

6

6 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

16 Methods

16.1 Method Definition

A method is an operation on a particular object. An object is an instance of a class. When we define a class we define its member variables and its methods. For each method we need to give a name, we need to define its input parameters and we need to define its return type. We also need to set its visibility(private, package, or public). If the method throws an Exception, that needs to be declared as well. The syntax of method definition is:

```
class MyClass
{
 ...
    public ReturnType methodName( ParamOneType param1, ParamTwoType param2 ) throws
ExceptionName
    {
      ReturnType retType;
      ...
      return retType;
    }
 ...
}
```

We can declare that the method does not return anything using the **void** java keyword. For example:

```
private void methodName( String param1, String param2 )
{
 ...
  return;
}
```

When the method returns nothing, the **return** keyword at the end of the method is optional. The **return** keyword can be used anywhere in the method, when the executation flow reach the **return** keyword, the method execution is stopped and the execution flow returns to the caller method.

16.2 Method Overloading

For the same class we can define two methods with the same name. However the parameter types and/or the number of parameters must be different for those two methods. In the java terminology, this is called **method overloading**. It is useful to use method overloading when we need to do something different based on a parameter type. For example we may have the operation : runAroundThe. We can define two methods with the same name, but different input parameter type:

```
public void runAroundThe( Building block )
{
  ...
}
public void runAroundThe( Park park )
{
  ...
}
```

Related terminology is the **method signature**. In java the method signature contains method name and the input parameter types. The java compiler takes the signature for each method and makes sure that each method signature is unique for a class. For example the following two method definitions are valid:

```
public void logIt( String param, Error err )
{
  ...
}
public void logIt( Error err, String param )
{
  ...
}
```

Because the type order is different. If both input parameters were type String, that would be a problem since the compiler would not be able to distinguish between the two:

```
public void logIt( String param, String err )
{
  ...
}
public void logIt( String err, String param )
{
  ...
}
```

The compiler would give an error for the following method definitions as well:

```
public void logIt( String param )
{
  ...
}
public String logIt( String param )
{
  String retType;
  ...
  return retValue;
}
```

Note, the return type is not part of the unique signature. Why not? The reason is that a method can be called without assigning its return value to a variable. This feature came from C and C++. So for the call:

```
{
  logIt( msg );
}
```

the compiler would not know which method to call.

16.3 Method Overriding

Obviously a method signature has to be unique inside a class. The same method signature can be defined in different classes. If we define a method that exist in the super class then we override the super class method. The terminology for this is **method overriding**. This is different from method overloading. Method overloading happens with methods with the same name different signature. Method overriding happens with same name, same signature between inherited classes.

The return type can cause the same problem we saw above. When we override a super class method the return type also must be the same. In fact if that is not the same, the compiler will give you an error.

Method overriding is related dynamic linking, or runtime binding. In order for the Method Overriding to work, the method call that is going to be called can not be determined at compilation time. It will be decided at runtime, and will be looked up in a table.

```
{
1  MyClass obj = new SubOfMyClass();
2
3  MyClass obj = new MyClass();
4
5  obj.myMethod(); // -- During compilation, it is not known what
reference the 'obj' has, MyClass or SubOfMyClass
}
```

In the above example 'obj' reference has the type MyClass on both line 1 and line 3. However the 'obj' reference points two different objects. On line 1 it references SubOfMyClass object, on line 3 it references MyClass object. So on line 5 which method will be called, method define in MyClass, or the method that defined in its subclasses. Because the 'obj' reference can point to object and all its sub object, and that will be known only at runtime, a table needs to be kept with all the possible method address to be called.

Also another rule is that when you do an override, the visibility of the new method that overrides the super class method can not be reduced. The visibility can be increased, however. So if the super class method visibility is public, the override method can not be package, or private.

In the case of the exception the override method may throw can be the same as the super class or it can be one of that exception inherited class. So the common rule is that the override method must throw the same exception or it is any of its subclasses.

NOTE: A common mistake to think that if we can override methods, we could also override member variables. This is not the case, as member variables are not overriden.

```
{
1  MyClass obj = new SubOfMyClass();
2
3  MyClass obj = new MyClass();
4
5  String var = obj.myMemberVar;  // -- The myMemberVar is defined in
the MyClass object
}
```

In the above example, it does not count what object the 'obj' reference points to, because it was declared MyClass type on both line 1 and line 3, the variable in the MyClass object will be referenced. In real examples we rarely use public variables, but if you do keep in mind that Java does not support variable overriding.

16.4 Parameter Passing

We can pass in all the primitive data types or any object references to a method. An object cannot be passed to a method, only its references. All parameters (those are primitive types and object references) are passed by value. In other words if you change the passed in parameter values inside the method, that will have no effect on the original variable that was passed in. When you pass in an object reference to a method and then you change that inside the method, that will have no effect on the original object reference. However if you modify the object itself, that will stay after the method returns. Think about the object reference as a pointer to an object. If you change the object the reference points at, that will be permanent. For example:

```
1 {
2   int var1 = 10;
3   int var2 = 20;
4   ...
5   myMethod( var1, var2 );
6   ...
7   System.out.println( "var1="+var1 +"var2="+var2 );  // -- The
variable values did not change
8 }
9 ...
10 void myMethod( int var1, int var2 )
11 {
12    ...
13    var1 = 0;
14    var2 = 0;
15    ...
16 }
```

On line 7 the value of var1 is 10 and the value of var2 is 20. When the variables were passed in to the methods their values were copied. This is called passing the parameter by value. In java we do not represent an object directly, we represent an object throught an **object reference**. You can think of an object reference as a variable having the address of the object. So the object reference passed in by value, but the object itself is not. For example:

```
1  {
2    MyObjOne obj = new MyObjOne();
```

```
3    obj.setName("Christin");
4    ...
5    myMethod( obj );
6    String name = obj.getName();  // --- The name attribute was
changed to 'Susan' inside the method
7  }
8  void myMethod( MyObjOne obj )
9  {
10   obj.setName("Susan");
11   ...
12   obj = new MyObjOne();
13   obj.setName("Sonya");
14   ...
15 }
```

On line 2, we created an object, on line 3 we set its name property to 'Christin'. On line 5 we called the `myMethod( obj )`. Inside the method, we changed the name to 'Susan' through the passed in object reference. So that change will stay. Note however that after we reassigned the obj reference to a new object, that is no effect whatsoever on the passed in object.

16.5 Functions

In java, functions (methods really) are just like in C++ except that they must be declared inside a class and objects are passed by reference automatically. You cannot create pointers to a function but Java has events which really are function pointers under the hood for when you need that type of functionality.

```
int a_function(double d)
{
    return (int)d;
}
```

16.6 Return Parameter

So as we can see, a method may or may not return a value. If the method does not return a value we use the **void** java keyword. Same as the parameter passing, the method can return a primitive type or an object reference. So a method can return only one value. What if you want to return more than one value from a method. You can always pass in an object reference to the method, and let the method modify the object properties. The modified values can be considered as an output value from the method. However better option, and cleaner if you create an Object array inside the method, assign the return values and return the array to the caller. You could have a problem however, if you want to mix primitive data types and object references as the output values from the method. There is a better approach. Defines special return object with the needed return values. Create that object inside the method, assign the values and return the reference to this object. This special object is "bound" to this method and used only for returning values, so do not use a public class. The best way is to use a nested class, see example below:

```
public class MyObject
{
  ...

  /** Nested object is for return values from 'getPersonInfoById'
method */
  public static class  ReturnObj
  {
      private int     age;
      private String name;

      public void setAge( int val )
      {
          this.age = val;
      }
      public int getAge()
      {
          return age;
      }

      public void setName( String val )
      {
          name = val;
      }
      public String getName()
      {
          return name;
      }
  } // --- End of nested class defination ---

  /** Method using the nested class to return values */
  public ReturnObj getPersonInfoById( int ID )
  {
     int    age;
     String name;
   ...
     // --- Get the name and age based on the ID from the database
---
   ...
     ReturnObj ret = new ReturnObj();
     ret.setAge( age );
     ret.setName( name );

   return ret;
  }
}
```

In the above example the 'getPersonInfoById' method returns an object reference that contains both values the name and age. See below how you may use that object:

```
{
 ...
   MyObject obj = new MyObject();
   MyObject.ReturnObj person = obj.getPersonInfoById( 102 );

   System.out.println( "Person Name=" + person.getName() );
   System.out.println( "Person Age =" + person.getAge() );
 ...
}
```

16.7 Special method, the Constructor

There is a special method for each class that will be executed each time an object is created from that class. That is the **Constructor**. Constructor does not have a return value and its name is the same as the class name. The Constructor can be overloaded; you can define more than one constructor with different parameters. For example:

```
public class MyClass
{
  private String memberField;

  /**
   * MyClass Constructor, there is no input parameter
   */
  public MyClass()
  {
     ...
  }

  /**
   * MyClass Constructor, there is one input parameter
   */
  public MyClass( String param1 )
  {
     memberField = param1;
     ...
  }
}
```

In the above example we defined two constructors, one with no input parameter, and one with one input parameter. You may ask which constructor will be called. Its depends how the object is created with the **new** keyword. See below:

```
{
   ...
   MyClass obj1 = new MyClass();              // The constructor with
no input parameter will be called
   MyClass obj2 = new MyClass("Init Value");  // The constructor with
one input param. will be called
   ...
 }
```

In the above example we created two objects from the same class, or we can also say that obj1 and obj2 both have the same type. The difference between the two is that in the first one the `memberVar` field is not initialized, in the second one that is initialized to 'Init Value'. obj1, and obj2 contains the reference to the object. Each class must have a constructor. If we do not define one, the compiler will create a default so called empty constructor automatically.

```
public class MyClass
{
  /**
   * MyClass Empty Constructor
   */
  public MyClass()
  {
```

```
    }
}
```

The Constructor is called automatically when an object is created with the **new** keyword. A constructor may also be called from an other constructor, see below:

```
public class MyClass
{
  private String memberField;

  /**
   * MyClass Constructor, there is no input parameter
   */
  public MyClass()
  {
     MyClass("Default Value");
  }

  /**
   * MyClass Constructor, there is one input parameter
   */
  public MyClass( String param1 )
  {
     memberField = param1;
     ...
  }
}
```

In the above example, the constructor with no input parameter calls the other constructor with the default initial value. This gives an option to the user, to create the object with the default value or create the object with a specified value.

16.8 Static Method

We defined method above as an operation on an object. Static Methods are defined inside a class, but they are not an operation on an object. No object needs to be created to execute a Static Method, they are simply global functions, with input parameters and a return value.

```
public class MyObject
{
   static public String myStaticMethod()
   {
      ...
     return("I am a Static Method");
   }

 }
```

Static Methods can be referenced anywhere prefixed by the class name. See below:

```
{
...
```

```
    // --- Call myStaticMethod  ---
    System.out.println( "Output from the myStaticMethod:" + MyObject.myStaticMethod() );
...
}
```

You can write a non object oriented program by using only Static Methods in java. Because java evolved from the C programming language, Static Method is a left over from a non object oriented language.

You write Static Method the same way as normal method, the only difference is that you can not reference any member variables and any object methods. Static Methods can reference only Static variables and call only other Static Methods. However you can create an object an use it inside a Static Method.

```
public class MyObject
{
   public String memberVar;

   static private String memberStaticVar;

   static public String myStaticMethod()
   {
     memberVar = "Value"; -->  ERROR Cannot reference member var.

     memberStaticVar = "Value";  // --- This is okay, static vars. can be used

     // --- Create an object ---
     MyObject obj = new MyObject();
     obj.memberVar = "Value";  // --- This is okay since an object is created --
      ...
     return("I am a Static Method");
   }
 }
```

16.9 External links

- ABSTRACT METHODS, INTERACTIVE JAVA LESSON[1]

1 HTTP://JAVALESSONS.COM/CGI-BIN/FUN/JAVA-TUTORIALS-MAIN.CGI?SES=AO789&CODE=ABS&SUB=FUN

17 Primitive Types

1. redirect JAVA PROGRAMMING/TYPES[1]

1 Chapter 18 on page 101

18 Types

Data Types (or simply **Types**) in Java are a way of telling what certain data *is.* It is seen as a way of declaring allowed values for certain data, the structure of such data and operations associated with it. Any data, be it numeric or a sentence of words, can have a different data type. For instance, just to define different types of numbers in Java, there are about six simple types available to programmers – some define whole numbers (integers numbers) and others define numbers with a decimal values (floating point numbers). Java also gives freedom to programmers to create complex and customizable data types. We will deal with complex data types in later chapters.

18.1 Data Types in Java

Java is considered a **strongly typed** programming language in that it is obligatory for all data, expressions and declarations within the code to have a certain type associated with it. This is either declared or inferred and the Java language only allows programs to run if they adhere to type constraints.

As we have discussed above, you can have types that define a number, or types that define textual content within your program. If you present a numeric type with data that is not numeric, say textual content, then such declarations would violate Java’s type system. This gives Java the unique ability of **type safety**. Java checks if an expression or data is encountered with an incorrect type or none at all. It then automatically flags this occurrence as an error at compile time. Most type-related errors are caught by the Java compiler, hence making a program more secure and safe once compiled completely and successfully.

In the Java language, there are three broad categories of data types:

- PRIMITIVES[1]
- OBJECT REFERENCE TYPES[2]
- ARRAYS[3]

In the following section, we will discuss these three categories in detail.

18.1.1 Primitives

Primitives are the most basic data types available within the Java language. These types serve as the building blocks of data manipulation in Java. Such types serve only one purpose – containing pure, simple values of a kind. Because these data types are defined into the Java type system by

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23PRIMITIVES
2 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23OBJECT%20REFERENCE%20TYPES
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23ARRAYS

default, they come with a number of operations predefined. You can not define a new operation for such primitive types. In the Java type system, there are three further categories of primitives:

- NUMERIC PRIMITIVES[4] These primitive data types hold only numeric data. Operations associated with such data types are those of simple arithmetic (addition, subtration, etc.) or of comparions (is greater than, is equal to, etc.)
- TEXTUAL PRIMITIVES[5] These primitive data types hold characters (which can be alphabets or even numbers), but unlike numbers, they do not have operations that serve arithmetic purposes. Rather, operations associated with such types are those of textual manipulation (comparing two words, joining characters to make words, etc.)
- BOOLEAN AND NULL PRIMITIVES[6]

18.1.2 Object Reference Types

18.1.3 Arrays

18.2 About Java Types

some type errors can still occur at runtime because Java supports a CAST[7] operation which is a way of changing the type of one expression to another. However, Java performs run time type checking when doing such casts, so an incorrect type cast will cause a runtime exception rather than succeeding silently and allowing data corruption.

Java is also known as a **hybrid language**. While supporting object oriented (OO) programming, Java is not a pure OO language like SMALLTALK[8] or RUBY[9]. Instead, Java offers both object types and PRIMITIVE TYPES[10]. Primitive types are used for boolean, character, and numeric values and operations. This allows relatively good performance when manipulating numeric data, at the expense of flexibility. For example, you cannot subclass the primitive types and add new operations to them.

18.3 Examples of Types

Below are two examples of Java types and a brief description of the allowed values and operations for these types. Additional details on each are available in other modules.

4 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FNUMERIC%20PRIMITIVES
5 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FTEXTUAL%20PRIMITIVES
6 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FOTHER%20PRIMITIVES
7 HTTP://EN.WIKIBOOKS.ORG/WIKI/%2FJAVA%20PROGRAMMING%2FCASTS
8 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3ASMALLTALK
9 HTTP://EN.WIKIBOOKS.ORG/WIKI/RUBY%20PROGRAMMING
10 Chapter 17 on page 99

18.3.1 Example: int

The PRIMITIVE TYPE[11] **int** represents a signed 32 bit integer value. The allowed data values for `int` are the integers between -2147483648 to 2147483647 inclusive.

The set of operations that may be performed on `int` values includes integer arithmetic such as `+`, `-`, `*`, `/`, `%`, comparison operations (`==`, `!=`, `<`, `>`, `<=`, `>=`), assignments (`=`, `++`, `--`, `+=`, `-=`), bit-wise operations such as logical and, logical or, logical xor, negation (`&`, `|`, `^`, `~`), bit shift operations (`<<`, `>>`, `>>>`), CONVERSIONS[12] to other numeric types and PROMOTION[13] to other integer types.

For example, to declare a **private** integer instance field named `length`, one would use the declaration

```
private int length;
```

18.3.2 Example: String

You use **class** and **interface** definition in Java to define new types. Class and interface types are associated with object references also sometime refered to as *Reference types.* An object reference has two main attributes:

- Its **type** associated with a class or an interface
- A java object it references, that is created by instantiating a class

The **String** class is one such example. String values are a sequence of 0 or more Unicode characters. The **null** reference is another valid value for a String expression.

The operations on a String reference variable are those available for all reference types, such as comparison operations `==`, `!=` and assignment `=`.

The allowed operations on String object, however are the set of methods in the `java.lang.String` class, which includes `length()`, `toString()`, `toLowerCase()`, `toUpperCase()`, `compareTo(String anotherString)` and more... .

In addition, String objects also inherit the set of operations from the base class that String extends from, which is `java.lang.Object`. These operations include methods such as `equals()`, `hashCode()`, `wait()`, `notifyAll()`, and `getClass()`.

```
private String name = "Marry Brown";
```

In the above example the *name* object reference's attributes are:

- **Type is** : **String**
- The **referenced object** is also : **String**

11 Chapter 17 on page 99
12 Chapter 17 on page 99
13 Chapter 17 on page 99

Both the `java.lang.String` class methods and `java.lang.Object` class methods are available for the object reference *name*.

```
private Object name = "Marry Brown";
```

In the above example the *name* object reference's attributes are:

- **Type is** : **Object**
- The **referenced object** is : **String**

Only the `java.lang.Object` class methods are available for the object reference *name*.

18.4 Array Types

Arrays in Java are represented as a built-in Array object. As with object types, they behave as a reference but have a few differences allowing them to allow easy access to sub elements.

When one declares an array, the data type is changed to include square brackets. (While you can instead place these square brackets next to the variable name, this is not recommended.) To create an array, you will need to use the new operator to have it create the Array with the specified number of elements:

```
/* Declares an array named data, and has it assigned to an array with
25 elements. */
private int[] data = new int[25];
```

Arrays are described in more detail in the Arrays section.

18.5 Primitive Data Types

The Java primitive data types contain pure values, no operations. It is basically data types similar to what other non-object-oriented languages have.

There are arrays of primitive types in Java; but because they are not objects, primitive values can not be put in a collection.

For this reason object wrappers are defined in JDK 'java.lang.*' package for all the primitive types.

float	32	flo 65
double	64	dou 65.
Other **boolean**	1	boo tru
void	--	--

The types **short**, **int**, **long**, **float**, and **double** are usually used in arithmetic operations; **byte** and **char** can also be used, but less commonly.

The character type **char** is used for text processing. The type **byte** is commonly used in binary file input output operations.

String objects representing literal character strings in Java, in the java.lang.* package. java.lang.String is not a primitive type, but instead is a special class built into the Java language. For further info, see **String**.

18.6 Data Conversion (Casting)

Data conversion (casting) can happen between two primitive types. There are two kinds:

- Implicit : casting operation is not required; the magnitude of the numeric value is always preserved. However, *precision* may be lost when converting from integer to floating point types
- Explicit : casting operation required; the magnitude of the numeric value may not be preserved

Example for implicit casting:

```
int  i = 65;
long l = i;  // --- int is converted to long, casting is not needed
```

Example for explicit casting:

```
long l = 656666L;
int  i = (int) l; // --- long is converted to int, casting is needed
```

The following table shows the conversions between primitive types, it shows the casting operation for explicit conversions:

	from byte	**from char**	**from short**	**from int**	**from long**	**from float**	**from double**	**from boolean**
to byte	-	(byte)	(byte)	(byte)	(byte)	(byte)	(byte)	N/A
to char		-	(char)	(char)	(char)	(char)	(char)	N/A
to short		(short)	-	(short)	(short)	(short)	(short)	N/A
to int				-	(int)	(int)	(int)	N/A
to long					-	(long)	(long)	N/A
to float						-	(float)	N/A
to double							-	N/A

	from byte	from char	from short	from int	from long	from float	from double	from boolean
to boolean	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-

18.7 Autoboxing/unboxing

Autoboxing/unboxing : Autoboxing and unboxing, language features since Java 1.5, make the programmer's life much easier when it comes to working with the primitive wrapper types. Consider this code fragment:

```
int age = 23;
Integer ageObject = new Integer(age);
```

Primitive wrapper objects were Java's way of allowing one to treat primitive data types as though they were objects. Consequently, one was expected to 'wrap' one's primitive data type with the corresponding primitive wrapper object, as shown above.

Autoboxing : Since Java 1.5, one may write as below and the compiler will automatically create the wrap object. The extra step of wrapping the primitive is no longer required. It has been 'automatically boxed up' on your behalf.

Points to ponder:
Keep in mind that the compiler still creates the missing wrapper code, so one doesn't really gain anything performance-wise. Consider this feature a programmer convenience, not a performance booster.

```
int age = 23;
Integer ageObject = age;
```

Unboxing : Uses the same process in reverse. Study the following code for a moment. The **if** statement requires a **boolean** primitive value, yet it was given a Boolean wrapper object. No problem! Java 1.5 will automatically 'unbox' this.

```
Boolean canMove = new Boolean(true);

if ( canMove )
{
  System.out.println( "This code is legal in Java 1.5" );
}
```

CATEGORY:JAVA PROGRAMMING[14]

14 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

19 java.lang.String

{{#ifeq: none
Types

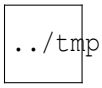


Figure 9
TYPES[1]
}}

Java Programming **String**

{{#ifeq: none
Classes, Objects and Types

CLASSES, OBJECTS AND TYPES[2]

Figure 10
}}

19.1 java.lang.String

String is a special class built into the Java language defined in the `java.lang` package.

The **String** class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class.

For example:

```
String str = "This is string literal";
```

On the right hand side a String object is created represented by the string literal. Its object reference is assigned to the `str` variable.

Strings are *immutable*; that is, they cannot be modified once created. Whenever it looks as if a String object was modified, a new String was actually created and the old one was thrown away.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. For example:

```
String str = "First part" + " second part";
// --- Is the same as:
String str = "First part second part";
```

Integers will also be converted to String after the (+) operator:

1 Chapter 18 on page 101
2 Chapter 12 on page 75

```
String str = "Age=" + 25;
```

Each Java object has the `String toString()` inherited from the **Object** class. This method provides a way to convert objects into `String`s. Most classes override the default behavior to provide more specific (and more useful) data in the returned `String`.

The `String` class provides a nice set of methods for string manipulation. Since `String` objects are immutable, all methods return a new `String` object. For example:

```
name = name.trim();
```

The `trim()` method returns a copy of the string with leading and trailing whitespace removed. Note that the following would do nothing useful:

```
name.trim();   // wrong!
```

This would create a new trimmed string and then throw it away. Study the `String` class and its methods carefully. Strings are ubiquitous in Java; it will serve you well to know how to manipulate them skillfully.

19.2 Using StringBuffer/StringBuilder to concatenate strings

Remember that `String` objects are immutable objects. Once a `String` is created, it can not be modified, takes up memory until garbage collected. Be careful of writing a method like this :

```
public String convertToString(Collection<String> coll)
{
    String str = "";
    for(String oneElem : coll) // loops through every element in coll
    {
       str = str + oneElem + " ";
    }
   return str;


}
```

On the (+) operation a new **String** object is created at each iteration. Suppose `coll` contains the elements `["Foo", "Bar", "Bam", "Baz"]`. The method creates five `String`s (`""`, `"Foo "`, `"Foo Bar "`, `"Foo Bar Bam "`, and `"Foo Bar Bam Baz"`) even though only last one is actually useful.

Instead use StringBuffer[3], as shown below, where only one StringBuilder[4] object is created:

To avoid unnecessary memory use like this, use the StringBuffer[5] or StringBuilder[6] class. They provide similar functionality to **String**s, but store their data in a mutable way. Also because object creation is time consuming, using StringBuffer or StringBuilder produces much faster code.

```
public String convertToString(Collection<String> coll)
{
    StringBuilder buf = new StringBuilder();
    for(String oneElem : coll) // loops through every element in coll
    {
       buf.append(oneElem);
       buf.append(" ");
    }
   return buf.toString();


}
```

`StringBuilder` was introduced in Java 5. Unlike `StringBuffer`, `StringBuilder` isn't thread safe, so you can't use it in more than one thread (see the chapter on Concurrency[7]). However, because it doesn't have to worry about synchronization, `StringBuilder`s are faster.

19.3 Comparing Strings

Comparing strings is not as easy as it may first seem. We cannot just use a simple equality statement such as:

```
if(myString == "Hello World!")       //Can't Use this.
{
     System.out.println("Match Found");


}
```

To test for equality, use the `equals(Object)` method inherited by every class and defined by `String` to return **true** if and only if the object passed in is a `String` containing the exact same data.

3 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.StringBuffer
4 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.StringBuilder
5 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.StringBuffer
6 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.StringBuilder
7 http://en.wikibooks.org/wiki/Java%20Programming%2FConcurrentProgramming

```
String greeting = "Hello World!";
if(greeting.equals("Hello World!")) { //true
    // ...

}
if(greeting.equals("hello world!")) { //false
    // ...

}
```

To order `String` objects, use the **compareTo()** method, which can be accessed wherever we use a String datatype. Let's take a look at an example:

```
String myString = "Hello World!";
//...
if(myString.compareTo("Hello World!") == 0 )
{
    System.out.println("Match Found");

}
```

This snippet of code is comparing the String variable `myString` to `"Hello World"`. The `compareTo` method returns a negative, zero, or positive number if the parameter is less than, equal to, or greater than the object on which it is called. If *myString* was to be different, even in the slightest manner we will get a value above or below 0 depending on the exact difference. The result is negative if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. Take a look at the JAVA API[8] for more details.

19.4 Splitting a String

Sometimes it is useful to split a string into separate strings, based on a *regular expression.* (For more information on regular expressions, see REGEX[9].) The **String** class has a `split()` method, since Java 1.4, that will return a String array.

See the following example:

8 HTTP://JAVA.SUN.COM/J2SE/1.4.2/DOCS/API/JAVA/LANG/STRING.HTML#COMPARETO(JAVA.LANG.STRING)

9 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%2FREGEX

```
String person = "Brown, John:100 Yonge Street,
Toronto:(416)777-9999";
...
String[] personData = person.split( ":" );
...
String name    = personData[0];
String address = personData[1];
String phone   = personData[2];
```

An other useful application could be to 'split' the String text based on the 'new line' character, so you could process the text line by line.

19.5 Creating substrings

It may also be sometimes useful to create **substrings**, or strings using the order of letters from an existing string. This can be done in two methods.

The first method involves creating a substring out of the characters of a string from a given index to the end.

For example:

```
String str    = "coffee";
String substr = str.substring(3);
```

In this example, `substr` would return "fee". As previously discussed, the index of the first character in a string is 0. By counting from there, it is apparent that the character in index 3 is the second "f" in "coffee". This is known as the `beginIndex`. All characters from the `beginIndex` until the end of the string will be copied into the new substring.

The second method involves a user-defined `beginIndex` and `endIndex`. For example:

```
String str = "supporting";
String substr = str.substring(3,7);
```

The string returned by `substr` would be "port". Please note that the endIndex is **not** inclusive. This means that the last character will be of the index `endIndex-1`. Therefore, in this example, every character from index 3 to index 6, inclusive, was copied into the substring.

Note: "Substring" is considered to be one word. This is why the method name does not seem to follow the common syntax of Java. It is easy to mistake the method `substr()` for `subStr()` (which does not exist and would return with a syntax error on compilation). Just remember that this style only applies to methods or other elements that are made up of more than one word.

19.6 Modifying String cases

The String Class also allows for the modification of cases. The two methods that make this possible are `toLowerCase()` and `toUpperCase()`. These methods are useful, for example, in the typical programming classroom assignment of evaluating whether or not a string is a palindrome.

```
String a = "WIKIBOOKS";
String b = "wikipedia";
```

In this example, a call to `a.toLowerCase()` would return a result of "wikibooks", and `b.toUpperCase()` would return "WIKIPEDIA".

19.7 See also

- JAVA API: JAVA.LANG.STRING[10]
- JAVA API: JAVA.LANG.STRINGBUFFER[11]
- JAVA API: JAVA.LANG.STRINGBUILDER[12]
- JAVA PROGRAMMING/API/JAVA.LANG.STRINGBUFFER[13]
- JAVA PROGRAMMING/API/JAVA.LANG.STRINGBUILDER[14]

10 HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/API/JAVA/LANG/STRING.HTML
11 HTTP://JAVA.SUN.COM/JAVASE/6/DOCS/API/JAVA/LANG/STRINGBUFFER.HTML
12 HTTP://JAVA.SUN.COM/JAVASE/6/DOCS/API/JAVA/LANG/STRINGBUILDER.HTML
13 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.STRINGBUFFER
14 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.STRINGBUILDER

20 Arrays

20.1 Intro to Arrays

An **array** is similar to a table of data, keyed by number. In Java an array is an object like all other objects. Look at the following program:

UNKNOWN TEMPLATE "Java_Code_File"

Copy the code and compile it. The program will ask you to enter some names then reprints the names in order. It demonstrates three major aspects of arrays: how to define an array, how to set data, and how to access it. The code `String[] names = new String[numNames];` tells Java to create an array of size *numNames* that will store Strings. To set data, use `names[x] =` **data** where *x* is the index to access. Note that all Java arrays start at 0 and go to (array size - 1). Thus, if you dimension an array to size 10, the highest index is 9.

20.2 Array Fundamentals

- To create an array, use the syntax **DataType[] variable = new DataType[ArraySize]**. Alternatively, if you know the data beforehand, you can write **DataType[] variable = {item 1, item 2,...item n}**
  - All elements of the array will be automatically initialized with the *default value* for that datatype. This is *false* for boolean's, *0* for all numeric primitive types, and *null* for all reference types. So for example, the previous note created an array of DataType references, all of which are initialized to *null.*
- To access an item, use the syntax **variable[i]** where *i* is the index
- To set an item, use the syntax **variable[i] = data**
- To find the length of an array, use the syntax **variable.length**

20.3 Two-Dimensional Arrays

A two dimensional array is represented by an array of an array. Because an array is also an object, like any other object having the **Object** as the super class, it can be used to create an array where the element of the array is also an array. In this way an array with any number of dimensions can be created. Here are examples of two dimensional arrays with initializer blocks:

```
String [][] twoDimArray = { {"00", "01", "02", "03", "04"},
                            {"10", "11", "12", "13", "14"},
                            {"20", "21", "22", "23", "23"} };
```

```
...
int [][] twoDimIntArray =  { {00, 01, 02, 03, 04},
                             {10, 11, 12, 13, 14},
                             {20, 21, 22, 23, 24} };
```

Note that the above "twoDimArray" is equivalent to the following more verbose code:

```
String [][] twoDimArray = new String[3][];
for (int i = 0; i < twoDimArray.length; i++) {
    twoDimArray[i] = new String[5];
    for (int j = 0; j < twoDimArray[i].length; j++)
        twoDimArray[i][j] = "" + i + j;
}
```

In the above example we defined an array which has three elements, each element contains an array having 5 elements. We could create the array having the 5 elements first and use that one in the initialize block.

```
String [] oneDimArray = {"00", "01", "02", "03", "04"};
String [][] twoDimArray = { oneDimArray ,
                            {"10", "11", "12", "13", "14"},
                            {"20", "21", "22", "23", "24"} };
```

Since they are arrays of array references, these multi-dimensional arrays can be "jagged" (i.e. subarrays can have different lenghts), or the subarray reference can even be null. Consider

```
String [][] weirdTwoDimArray = { {"10", "11", "12"},
                           null,
                           {"20", "21", "22", "23", "24"} };
```

20.4 Multidimensional Array

Going further any number of dimensional array can be defined.

```
<elementType>[][]...[] <arrayName>
or
<elementType><arrayName>[][]...[]
```

21 Data and Variables

A variable in Java can store two kinds of variables:

- JAVA PRIMITIVE TYPE VALUES[1]
- a reference to a JAVA OBJECT[2]

Java's primitive types are

- integers (whole numbers, declared as **byte**, **short**, **int**, or **long**; only int need be of interest to a beginner)
- floating-point numbers (decimal numbers, declared as **float** or **double**; only float need be of interest at first)
- characters (declared as **char**, representing one character like 'A' or ',')
- **boolean** (holding only *true* or *false* as values)

In addition, the Java language has special features for its String class, and strings can be treated very much like primitives for many purposes.

As in most languages, a variable is declared to be a particular type of data; the syntax for a declaration is:

```
variabletype variablename;
```

To store a value in a variable, a program statement like

```
variablename = data;
```

And can reference the variable (and use the data stored in it) by its name.

For example, to create an int primitive type value, named **year** that stores 2007;

```
year = 2007;
```

To access the data in *year*, use the variable in place of the number.

```
System.out.println(year);
```

Produces

1 Chapter 17 on page 99
2 Chapter 24 on page 131

```
2007
```

21.1 Strong Typing

Variables in Java are *strongly typed*, which means that the compiler checks the type of a variable matches the type of data stored in that variable. If you declare a variable to hold a String, for instance, you cannot assign an integer value to that variable. Some languages (such as C) define an interpretation of such a statement and use that interpretation without any warning; others (such as PL/I) define a conversion for almost all such statements and perform the conversion to complete the assignment. Strong typing is intended to prevent mistakes made by unwittingly assigning the wrong kind of value to a variable, and catching those mistakes when the program is compiled rather than waiting to find it when the program is running.

21.2 Case Conventions

Java is case-sensitive. A method called `mymethod` is completely separate from a method called `myMethod`. Be careful!

By convention, most identifiers that includes more than one word uses CAMEL CASE[3]. Classes begin with a capital letter; methods and variables do not. (Constructors have to start with a capital, because they must have the same name as the class.) Package names use lowercase, and do not use camel case. Thus:

```
package org.wikibooks.samplecode;

class CaseConventions {
    int variable;
    int multipleWordVariable;

    CaseConventions(String id) {
    }

    void method() {
    }

    void longMethodName() {
    }
}
```

21.3 Scope

Java uses **block scope**, which means that a variable is "un-defined" (and becomes useless) at the end of the block in which it is defined. A *block* is any section of code within curly braces.

3 HTTP://EN.WIKIPEDIA.ORG/WIKI/CAMELCASE

Common blocks include class definitions, methods and constructors, if/else blocks, and for, while, and do-while loops.

```
class BlockScope {
    int classScope; // valid in all of class BlockScope

    BlockScope(int param) { // param is valid only in this constructor
        int localVariable = 0; // valid only in this constructor
    }

    void someMethod() {
        int local = 42; // valid only in this method

        if(local > 0) {
            boolean positive = true; // valid only within the if block
        } else {
            // positive is not defined here!
        }
    }
}
```

There are three basic kinds of scope for variables in Java:

- local variable, declared within a method in a class, valid for (and occupying storage only for) the time that method is executing. Every time the method is called, a new copy of the variable is used.
- instance variable, declared within a class but outside any method. It is valid for and occupies storage for as long as the corresponding object is in memory; a program can instantiate multiple objects of the class, and each one gets its own copy of all instance variables. This is the basic data structure rule of Object-Oriented programming; classes are defined to hold data specific to a "class of objects" in a given system, and each instance holds its own data.
- static variable, declared within a class as *static*, outside any method. There is only one copy of such a variable no matter how many objects are instantiated from that class.

CATEGORY:JAVA PROGRAMMING[4]

4 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

22 Generics

Generics were added to the Java language syntax in version 1.5. This means that code using Generics will not compile with Java 1.4 and less.

Java was long criticized for the need to explicitly type-cast an element when it was taken out of a "container/collection" class. There was no way to enforce that a "collection" class contains only one type of object (e.g., to forbid *at compile time* that an `Integer` object is added to a `Collection` that should only contain `Strings`). This is now possible since Java 1.5.

In the first couple of years of Java evolution, Java did not have a real competitor. This has changed by the appearance of Microsoft C#. With Generics Java is better suited to compete against C#. Similar constructs to Java Generics exist in other languages, see GENERIC PROGRAMMING[1] for more information.

22.1 What are Generics?

Generics are so called because this language feature allows methods to be written generically, with no foreknowledge of the type on which they will eventually be called upon to carry out their behaviors. A better name might have been **type parameter argument**. Because, it is basically that, to pass a Type as a parameter to a class at creation time.

When an object is created, parameters can be passed to the created object, through the constructor. Now with Generics, we can also pass in Types. The type-place-holders will be replaced with the specified type, before the object is created.

Type parameter arguments can be set:

for a class : When an object is created from that class the type-parameter-argument will be replaced with the actual Type.

```
public class Person<T>
{
   private Person<T> person;
...
}
...
// --- Create an Employee person ---
Person<Employee> emplPerson = new Person<Employee>();
...
// --- Create a Customer person ---
Person<Customer> custPerson = new Person<Customer>();
```

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENERIC%20PROGRAMMING

for a method : Just like class declarations, method declarations can be generic--that is, parameterized by one or more type parameters.

```
public static <T> void assign( Person<T> person, T obj )
{
   person.setPerson( obj );
}
```

use of generics is optional : For backwards compatibility with pre-Generics code, it is okay to use generic classes without the generics type specification thing (<T>). In such a case, when you retrieve an object reference from a generic object, you will have to manually typecast it from type Object to the correct type. The compiler should also warn about unsafe operations.

22.2 Introduction

Java is a strongly typed language. That's one of the reasons why it is so easy to use. Many potential problems are caught by the compiler. One area where Java was criticized was regarding the container objects. Container objects are objects that contain other objects. Before Generics were introduced there was no way to ensure that a container object contains only one type of objects. When an object was added to a container, it was automatically cast to Java **Object**. When it was taken out an explicit cast was needed. Normally an explicit cast is checked by the compiler.

```
String st = "This is a String";
...
Integer integer = (Integer) st;  // --- Compilation Error --
```

But in the case of container classes, the compiler was not able to catch an invalid type casting.

```
1 Collection collString = new ArrayList();
2 collString.add( "This is a String" );
...
3 Integer integer = (Integer) collString.get(0);  // --- No Compilation Error; RunTime CastException
```

Just looking at line 3, we do not know what type of objects *collString* contains. If that contains Integers then the code is fine.

The below code using Generic:

```
Collection<String> collString = new ArrayList<String>();
collString.add( "This is a String" );
...
Integer integer = (Integer) collString.get(0);  // --- Compilation Error
```

collString is a container object, that can contain only *String* objects, nothing else, so when we get out an element it can be casted only to class that normally a String can be casted.

With Generics, Java strict type checking can be extended to container objects. Using Generics with container classes, gives an impression that a new container type is created, with each different type

parameter. Before Generics:

```
Collection collCustomer = new ArrayList();
collCustomer.add( new Customer() );
...
Collection collObject = collCustomer;  // --- No problem, both collObject and collCustomer have the same type
```

With generics:

```
Collection<Customer> collCustomer = new ArrayList<Customer>();
collCustomer.add( new Customer() );
...
Collection<Object> collObject = collCustomer;  // --- Compilation Error
```

Both *collObject* and *collCustomer* have the same type, **BUT it is against the Generic rule**, that is *collCustomer* can contain only Customer objects, and *collObject* can contain only Object object. So there is an additional check to the normal type checking, the type of the parameter type has to be matched too.

22.3 Note for C++ programmers

Java Generics are similar to C++ Templates in that both were added for the same reason. The syntax of Java Generic and C++ Template are also similar.

There are some differences however. The C++ template can be seen as a kind of macro, that generates code before compilation. The generated code depends on how the Template class is referenced. The amount of code generated depends on how many different types of classes are created from the Template. C++ Templates do not have any run-time mechanisms. The compiler creates normal code to substitute the template, similar to any 'hand-written' code.

In contrast, Java Generics are built into the language. The same Class object handles all the Generic type variations. No additional code is generated, no matter how many Generic objects are created with different type parameters. For example.

```
Collection<String>  collString  = new ArrayList<String>();

Collection<Integer> collInteger = new ArrayList<Integer>();
```

There is only one Class object created. In fact, at runtime, both these objects appear as the same type (both `ArrayList`'s). The generics type information is erased during compilation (type erasure). This means, for example, that if you had function that takes `Collection<T>` as an argument, and that collection happened to be empty, your function would have no way of instantiating another `T` object, because it doesn't know what `T` was.

The Class **class** itself is generic since Java 1.5.

```
public final class Class<T> extends Object
                        implements Serializable, GenericDeclaration, Type,
AnnotatedElement
{
...
}
```

The **T** type here represents the type that is handed to the Class object. The **T** type will be substituted with the class being loaded.

22.4 Class<T>

Since Java 1.5, the class java.lang.Class is generic. It is an interesting example of using genericness for something other than a container class.

For example, the type of String.class is `Class<String>`, and the type of Serializable.class is `Class<Serializable>`. This can be used to improve the type safety of your **reflection code**.

In particular, since the `newInstance()` method in Class now returns a T, you can get more precise types when creating objects reflectively.

Now we can use the `newInstance()` method to return a new object with exact type, without casting. :

```
Customer cust = Utility.createAnyObject(Customer.class);  // - No casting
...
public static <T> T createAnyObject(Class<T> cls)
{
    T ret = null;
    try
    {
        ret = cls.newInstance();
    }
    catch (Exception e)
    {
        // --- Exception Handling
    }
    return ret;
}
```

And the above code without Generics:

```
Customer cust = (Customer) Utility.createAnyObject(Customer.class);  // - Casting is needed
...
public static Object createAnyObject(Class cls)
{
    Object ret = null;
    try
    {
        ret = cls.newInstance();
    }
    catch (Exception e)
    {
        // --- Exception Handling
    }
```

```
    return ret;
}
```

Get exact type when getting JavaBean property, using reflection : See the following code where the method will return the exact type of the Java Bean property, based on how it will be called.

```
// --- Using reflection, get a Java Bean property by its name ---
public static <T> T getProperty(Object bean, String propertyName)
{
    if (bean == null ||
        propertyName == null ||
        propertyName.length() == 0)
    {
        return null;
    }
    // --- Based on the property name build the getter method name
---
    String methodName = "get" +
        propertyName.substring(0,1).toUpperCase() +
        propertyName.substring(1);
    T property = null;
    try
    {
        java.lang.Class c = bean.getClass();
        java.lang.reflect.Method m = c.getMethod(methodName, null);
        property = (T) m.invoke(bean, null);
    }
    catch (Exception e)
    {
        // --- Handle exception --
    }
    return property;
}
```

22.5 Variable Argument

Using Generics, it is very easy to define a method with a variable number of arguments. Before generics, this was not possible in Java—if a likewise feature was needed, this was mostly done by passing an array. The only requirement for using a variable number of arguments using Generics is that the arguments in the list must have the same type.

The following code illustrates a method that can be called with a variable number arguments:

```
/**
 * Method using variable-length argument list
 * @param <T>
 * @param args
 */
public static <T> List<T> makeAList(T... args)
{
    List<T> argList = new ArrayList<T>();
    for (int i = 0; i < args.length; i++)
    {
        argList.add(args[i]);
    }
```

```
    return argList;
}
```

The above method can be called with a variable number of arguments, for example:

```
List<String> list1 = makeAList("One", "Two", "Three");
List<String> list2 = makeAList("One", "Two", "Three", "Four");
```

In the above example calls, the arguments must be of type `String`. If we write `<? extends Object>` instead of `T`, then we can pass *any* kind of objects, regardles of their type:

```
List<? extends Object> list3 = makeAList("One", 10, new
StringBuffer(), new LinkedList());
```

Note: the number 10 in the above code will be converted (autoboxed) to Integer.

See also:

```
java.util.Arrays.asList(T... a)
```

22.6 Wildcard Types

As we have seen above, generics give the impression that a new container type is created with each different type parameter. We have also seen that in addition to the normal type checking, the type parameter has to match as well when we assign generics variables.

In some cases this is too restrictive. What if we would like to relax this additional checking? What if we would like to define a collection variable that can hold any generic collection, regardless of the parameter type it holds?

Wildcard : The wildcard type is represented by the character **<?>**, and pronounced **Unknown**, or **Any-Type**. This **Unknown** type matches anything, if it is used only by itself. Any-Type can be express also by `<? extends Object>`. Any-Type includes Interfaces, not only Classes.

So now we can define a collection whose element type matches anything. See below:

```
Collection<?> collUnknown;
```

Note that we can not add anything to this collection. We can only take out elements of type **Object** from it. So what is the use of this variable if we can not add anything to the collection it represents? The use of this new construct will be clear when you want to create a generic method that takes any collection.

```
public static void printElements( Collection<?> anycoll )
{
   Iterator<?> iter = coll.iterator();
   while ( iter.hasNext() )
   {
      System.out.print( iter.next() );
   }
}
```

Wildcard for a specific type of classes :

"<? extends ClassName>" specifies a restriction on the types of classes that may used.

For example, to create a collection that may only contain "Serializable" objects, specify:

```
Collection<? extends Serializable> serColl = new ArrayList<String>();
```

The above code is valid because, the **String** class is serializable. Use of a class that is not serializable would cause a compilation error.

The following collection can only contain objects that extend the class Animal.

```
class Dog extends Animal
{
 ...
}
...
// --- Create "Animal Collection" variable ---
Collection<? extends Animal> animalColl = new ArrayList<Dog>();
```

"<? super ClassName>" specifies a restriction on the types of classes that may be used.

For example, to declare a Comparator that can compare Dogs, you use

```
Comparator<? super Dog> myComparator;
```

Now suppose you define a comparator that can compare Animals:

```
class AnimalComparator implements Comparator<Animal>
{
  int compare(Animal a, Animal b) { //...
  }
}
```

Since Dogs are Animals, you can use this comparator to compare Dogs also. Comparators for any superclass of Dog can also compare Dog; but comparators for any strict subclass cannot.

```
Comparator<Animal> myAnimalComparator = new AnimalComparator();

static int compareTwoDogs(Comparator<? super Dog> comp, Dog dog1, Dog
```

```
dog2) {
  return comp.compare(dog1, dog2);
}
```

The above code is valid because, the `Animal` class is a supertype of the `Dog` class. Use of a class that is not a supertype would cause a compilation error.

CATEGORY:JAVA PROGRAMMING[2]

2 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

23 Defining Classes

23.1 Fundamentals

Every class in Java can be composed of the following elements

- **fields** - Fields are variables that hold data specific to each object. For example, an employee might have an ID number. (They are also called member variables.) There is one field for each object of a class.
- **member methods** - Member methods perform operations on an object. For example, an employee might have a method to issue his paycheck or to access his name.
- **static fields** - Static fields are common to any object of the same class. For example, a static field within the Employee class could keep track of the last ID number issued. Only one static field exists for one class.
- **static methods** - Static methods are methods that do not affect a specific object.
- **other classes** - Sometimes it is useful to contain a class within another one if it is useless outside of the class or should not be accessed outside the class.
- **Constructors** - A special method that generates a new object.
- **Parameterized types** - Since 1.5, 'parameterized types' can be assigned to a class during definition. The 'parameterized types' will be substituted with the types specified at the class's instantiation. It is done by the compiler. It is similar to the C language macro '#define' statement, where a preprocessor evaluates the macros.

```
public class Employee                // This defines the Employee class.
{                                    // The public modifier indicates
 that
                                       // it can be accessed by any
  other class

     private static int nextID;      // Define a static field. Only
  one copy of this will exist,
                                       // no matter how many Employees
  are created.
```

```
   private int myID;                // Define fields that will be
stored
   private String myName;           // for each Employee. The private
modifier indicates that
                                    // only code inside the Employee
class can access it.

   public Employee(String name)     // This is a constructor. You can
pass a name to the constructor
   {                                // and it will give you a newly
created Employee object.
       myName = name;
       myID = nextID;               // Automatically assign an ID to
the object
       nextID++;                    // Increment the ID counter
   }

   public String getName()          // This is a member method that
returns the
   {                                // Employee object's name.
       return myName;               // Note how it can access the
private field myName.
   }

   public int getID()               // This is another member method.
   {
       return myID;
   }

   public static int getNextID()    // This is a static method that
returns the next ID
   {                                // that will be assigned if
another Employee is created.
       return nextID;
   }
```

}

The following Java code

```
public class EmployeeList {
    public static void main(String[] args) {
```

```
        System.out.println(Employee.getNextID());

        Employee a = new Employee("John Doe");
        Employee b = new Employee("Jane Smith");
        Employee c = new Employee("Sally Brown");

        System.out.println(Employee.getNextID());

        System.out.println(a.getID() + " : " + a.getName());
        System.out.println(b.getID() + " : " + b.getName());
        System.out.println(c.getID() + " : " + c.getName());
    }
```

```
}
```

would produce this output:

```
0
3
0 : John Doe
1 : Jane Smith
2 : Sally Brown
```

CATEGORY:JAVA PROGRAMMING[1]

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

24 Creating Objects

24.1 Introduction

Before a Java object can be created the class byte code must be loaded from the file system (with .class extension) to memory. This process of locating the byte code for a given class name and converting that code into a Java CLASS[1] class instance is known as class loading. There is one CLASS[2] created for each type of Java class.

All objects in java programs are created on heap memory. An object is created based on its class. You can consider a class as a blueprint, template, or a description how to create an object. When an object is created, memory is allocated to hold the object properties. An object reference pointing to that memory location is also created. To use the object in the future, that object reference has to be stored as a local variable or as an object member variable.

The Java Virtual Machine (JVM), keeps track of the usage of object references. If there are no more reference to the object, the object can not be used any more and becomes garbage. After a while the heap memory will be full of unused objects. The JVM collects those garbage objects and frees the memory they allocated, so the memory can be reused again when a new object is created. See below a simple example:

```
{
  // --- Create an object ---
  MyObject obj = new MyObject();

  // --- Use the object ---
  obj.printMyValues();
}
```

The `obj` contains the object reference pointing to an object created from the MyObject class. The `obj` object reference is in scope inside the { }. After the } the object becomes garbage. Object references can be passed in to methods, object references can be returned from methods.

24.2 Creating object with the new keyword

99% of new objects are created using the **new** keyword.

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.CLASS
2 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.CLASS

```
{
   // --- Create an 'MyObject' for the first time the application
started --
   MyObject obj = new MyObject();
}
```

When an object from the MyObject class is created for the first time. The JVM searches the file system for the definition of the class, that is the Java byte code. The file has the extention of '*.class'. The CLASSPATH environment variable contains locations where Java classes are stored. The JVM is looking for the 'MyObject.class' file. Depending on which package the class belongs to, the package name will be translated to a directory path.

When the 'MyObject.class' file is found, the JVM's class loader loads the class in memory, and creates a Class object. The JVM stores the code in memory, allocates memory for the **static** variables, and executes any static initialize block. Memory is not allocated for the object member variables at this point, memory will be allocated for them when an instance of the class, an object, is created.

There is no limit on how many objects from the same class can be created. Code and **static** variables are stored only once, no matter how many objects are created. Memory is allocated for the object member variables when the object is created. Thus, the size of an object is determined not by its code's size but by the memory it needs for its member variables to be stored.

24.3 Creating object by cloning an object

Cloning is not automatically available to classes. There is some help though, as all Java objects inherit the `protected Object clone()` method. This base method would allocate the memory and do the bit by bit copying of the object's states.

You may ask why we need this clone method. Couldn't I create a constructor and just passing in the same object, and do the copying variable by variable? Lets see:

```
public class MyObject
{
   private int memberVar;
...
   MyObject( MyObject obj )
   {
      this.memberVar = obj.memberVar;
    ...
   }
...
}
```

You might think that accessing the private memberVar variable of obj would fail but as this is in the same class this code is legal. The `clone()` method copies the whole object's memory in one operation. This is much faster than using the new keyword. Object creation with the **new** keyword is expensive, so if you need to create lots of objects with the same type, performance will be better if you create one object and clone new ones from it. See below a factory method that will return a new object using cloning.

```
HashTable _cacheTemplate = new HashTable;
...
/** Clone Customer object for performance reason */
public Customer createCustomerObject()
{
   // --- See if a template object exists in our cache ---
   Customer template = _cacheTemplate.get( "Customer" );
   if ( template == null )
   {
      // --- Create template ---
      template = new Customer();
      _cacheTemplate.put( "Customer", template );
   }
  return template.clone();
}
```

Now, lets see how to make the Customer object cloneable.

- First the Customer class needs to implement the Cloneable Interface.
- Override and make the `clone()` method **public**, as that is **protected** in the Object class.
- Call the `super.clone()` method at the beginning of your `clone` method.
- Override the `clone()` method in all the subclasses of Customer.

```
public class Customer implements Cloneable
{
 ...
    public Object clone() throws CloneNotSupportedException
    {
        Object obj = super.clone();

       return obj;
    }
}
```

In the above example we used cloning for speed up object creation.

An other use of cloning could be to take a snapshot of an object that can change in time. Lets say we want to store Customer objects in a collection, but we want to disassociate them from the 'live' objects . So before adding the object, we clone them, so if the original object changes from that point forward, the added object won't. Also lets say that the Customer object has a reference to an Activity object that contains the customer activities. Now we are facing a problem, it is not enough to clone the Customer object, we also need to clone the referenced objects. The solution:

- Make the Activity class also cloneable
- Make sure that if the Activity class has other 'changeable' object references, those has to be cloned as well, as seen below
- Change the Customer class `clone()` method as follows:

```
public class Customer implements Cloneable
{
  Activity _activity;
  ...
    public Customer clone() throws CloneNotSupportedException
    {
        Customer clonedCustomer = (Customer) super.clone();

        // -- Clone the object referenced objects ---
        if ( _activity != null )
```

```
        {
            clonedCustomer.setActivity( (Activity) _activity.clone()
);
        }
       return clonedCustomer;
    }
}
```

Note that only mutable objects needs to be cloned. References to unchangeable objects such as String be used in the cloned object without worry.

24.4 Creating object receiving from a remote source

When an object is sent through a network, the object needs to be **recreated** at the receiving host.

Object Serialization : The term Object Serialization refers to the act of converting the object to a byte stream. The byte stream can be stored on the file system, or can be sent through a network.

At the later time the object can be re-created from that stream of bytes. The only requirement is that the same class has to be available at both times, when the object is serialized and also when the object is re-created. If that happens in different servers, then the same class must be available on both servers. Same class means that exactly the same version of the class must be available, otherwise the object won't be able to be re-created. This is a maintenance problem to those applications where java serialization is used to persist object or sent the object through the network.
When a class is modified, there could be a problem re-creating those objects that were serialized using an earlier version of the class.

Java has built in support for serialization, using the Serializable interface; however, a class must first implement the Serializable interface.

By default, a class will have all of its fields serialized when converted into a data stream (with TRANSIENT[3] fields being skipped.) If additional handling is required beyond the default of writing all fields, you need to provide an implementation for two methods:

```
private void writeObject(java.io.ObjectOutputStream out) throws
IOException;
private void readObject(java.io.ObjectInputStream in) throws
IOException, ClassNotFoundException;
private void readObjectNoData() throws ObjectStreamException;
```

If the object needs to write or provide a replacement object during serialization, it needs to impement the following two methods, with any access specifier:

```
Object writeReplace() throws ObjectStreamException;
```

3 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FTRANSIENT

```
Object readResolve() throws ObjectStreamException;
```

Normally, a minor change to the class can cause the serialization to fail. You can still allow the class to be loaded by defining the serialization version id:

```
private static final long serialVersionUID = 42L;
```

CATEGORY:JAVA PROGRAMMING[4]

4 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

25 Interfaces

25.1 Interfaces

Java does not allow you to create a subclass from two classes. There is no multiple inheritance. The major benefit of that is that all java objects can have a common ancestor. That class is called **Object**. All java classes can be up-casted to **Object**. Example:

```
class MyObject
{
 ...
}
```

When you type the above code, it actually means the following:

```
class MyObject extends Object // -- The compiler adds 'extends Object'. if not
specified
{
 ...
}
```

So it can be guaranteed that certain methods are available in all java classes. This makes the language simpler.

To mimic multiple inheritance, java offers *interfaces*, which are similar to abstract classes. In interfaces all methods are abstract by default, without the *abstract* key word. Interfaces have no implementation and no variables, but constant values can be defined in interfaces - however, a single class can implement as many interfaces as required.

```
public interface MyInterface
{
  public static final String CONSTANT = "This value can not be changed";

  public String methodOne(); // This method must be implemented by the class that implements this interface
  ...
}

...

public class MyObject implements MyInterface
{
  // Implement MyInterface interface
  public String methodOne()
  {
   ...
  }
}
```

25.2 External links

- INTERFACES, INTERACTIVE JAVA LESSON[1]
- INTERFACES 2, INTERACTIVE JAVA LESSON[2]

CATEGORY:JAVA PROGRAMMING[3]

1 HTTP://JAVALESSONS.COM/CGI-BIN/FUN/JAVA-TUTORIALS-MAIN.CGI?SES=AO789&CODE=IF1&SUB=FUN
2 HTTP://JAVALESSONS.COM/CGI-BIN/FUN/JAVA-TUTORIALS-MAIN.CGI?SES=AO789&CODE=IF2&SUB=FUN
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

26 Using Static Members

26.1 What does static mean?

When you declare a

- method, or
- member variable

static, it is independent of any particular, but rather it is shared among all instances of a class. To access a static method or member variable, no instance needs to be created.

The **static** keyword is used to declare a method, or member variable static.

26.2 What can it be used for?

- Static variables can be used as data sharing amongst objects of the same class. For example to implement a counter that stores the number of objects created at a given time can be defined as so:

```
public AClass
{
   static private int counter;
...
   public AClass()
   {
    ...
      counter += 1;
   }
...
   public int getNumberOfObjectsCreated()
   {
      return counter;
   }
}
```

The `counter` variable is incremented each time an object is created.

Public static variable should not be used, as these become GLOBAL variables that can be accessed from everywhere in the program. Global constants can be used, however. See below:

```
static public final String CONSTANT_VAR = "Const";
```

- Static methods can be used for utility functions or for functions that do not belong to any particular object. For example:

```
public Match
{
 ...
   public static addTwoNumbers( int par1, int par2 )
   {
        return par1 + par2;
   }
}
```

26.3 Danger of static variables

Use static variables only for:

- data sharing (be careful)
- defining global constants

Use static methods for:

- utility functions

Using static variables and/or method for other purposes goes against object orientation, and will make your code either harder to read or maintain.

26.4 External links

- STATIC, INTERACTIVE JAVA LESSON[1]

CATEGORY:JAVA PROGRAMMING[2]

1 HTTP://JAVALESSONS.COM/CGI-BIN/FUN/JAVA-TUTORIALS-MAIN.CGI?SES=AO789&CODE=STC&SUB=FUN

2 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

27 Destroying Objects

Unlike in many other object-oriented programming languages, Java performs automatic garbage collection - any unreferenced objects are automatically erased from memory - and prohibits the user from manually destroying objects.

27.1 finalize()

When an object is garbage-collected, the programmer may want to manually perform cleanup, such as closing any open input/output streams. To accomplish this, the finalize() method is used. Note that finalize() should never be manually called, except to call a super class' finalize method from a derived class' finalize method. Also, we can not rely on when the finalize() method will be called. If the java application exits before the object is garbage-collected, the finalize() method may never be called.

```
protected void finalize() throws Throwable
{
    try {
        doCleanup();        // Perform some cleanup.  If it fails for
some reason, it is ignored.
    } finally {
        super.finalize(); //Call finalize on the parent object
    }
}
```

The garbage-collector thread runs in a lower priority than the other threads. If the application creates objects faster than the garbage-collector can claim back memory, the program can run out of memory.

The finalize method is required only if there are resources beyond the direct control of the Java Virtual Machine that need to be cleaned up. In particular, there is no need to explicitly close an OutputStream, since the OutputStream will close itself when it gets finalized. Instead, the finalize method is used to release either native or remote resources controlled by the class.

CATEGORY:JAVA PROGRAMMING[1]

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

28 Overloading Methods and Constructors

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be overloaded. This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the throws clauses of two methods with the same name but different signatures.

Methods are overridden on a signature-by-signature basis.

If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method. In this respect, the Java programming language differs from C++.

When a method is invoked, the number of actual arguments and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked . If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup.

CATEGORY:JAVA PROGRAMMING[1]

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

29 Arrays

29.1 Intro to Arrays

An **array** is similar to a table of data, keyed by number. In Java an array is an object like all other objects. Look at the following program:

UNKNOWN TEMPLATE "Java_Code_File"

Copy the code and compile it. The program will ask you to enter some names then reprints the names in order. It demonstrates three major aspects of arrays: how to define an array, how to set data, and how to access it. The code `String[] names = new String[numNames];` tells Java to create an array of size *numNames* that will store Strings. To set data, use `names[x] =` **data** where *x* is the index to access. Note that all Java arrays start at 0 and go to (array size - 1). Thus, if you dimension an array to size 10, the highest index is 9.

29.2 Array Fundamentals

- To create an array, use the syntax **DataType[] variable = new DataType[ArraySize]**. Alternatively, if you know the data beforehand, you can write **DataType[] variable = {item 1, item 2,...item n}**
  - All elements of the array will be automatically initialized with the *default value* for that datatype. This is *false* for boolean's, *0* for all numeric primitive types, and *null* for all reference types. So for example, the previous note created an array of DataType references, all of which are initialized to *null.*
- To access an item, use the syntax **variable[i]** where *i* is the index
- To set an item, use the syntax **variable[i] = data**
- To find the length of an array, use the syntax **variable.length**

29.3 Two-Dimensional Arrays

A two dimensional array is represented by an array of an array. Because an array is also an object, like any other object having the **Object** as the super class, it can be used to create an array where the element of the array is also an array. In this way an array with any number of dimensions can be created. Here are examples of two dimensional arrays with initializer blocks:

```
String [][] twoDimArray = { {"00", "01", "02", "03", "04"},
                            {"10", "11", "12", "13", "14"},
                            {"20", "21", "22", "23", "23"} };
```

```
...
int [][] twoDimIntArray =  { {00, 01, 02, 03, 04},
                             {10, 11, 12, 13, 14},
                             {20, 21, 22, 23, 24} };
```

Note that the above "twoDimArray" is equivalent to the following more verbose code:

```
String [][] twoDimArray = new String[3][];
for (int i = 0; i < twoDimArray.length; i++) {
    twoDimArray[i] = new String[5];
    for (int j = 0; j < twoDimArray[i].length; j++)
        twoDimArray[i][j] = "" + i + j;
}
```

In the above example we defined an array which has three elements, each element contains an array having 5 elements. We could create the array having the 5 elements first and use that one in the initialize block.

```
String [] oneDimArray = {"00", "01", "02", "03", "04"};
String [][] twoDimArray = { oneDimArray ,
                            {"10", "11", "12", "13", "14"},
                            {"20", "21", "22", "23", "24"} };
```

Since they are arrays of array references, these multi-dimensional arrays can be "jagged" (i.e. subarrays can have different lenghts), or the subarray reference can even be null. Consider

```
String [][] weirdTwoDimArray = { {"10", "11", "12"},
                           null,
                           {"20", "21", "22", "23", "24"} };
```

29.4 Multidimensional Array

Going further any number of dimensional array can be defined.

```
<elementType>[][]...[] <arrayName>
or
<elementType><arrayName>[][]...[]
```

30 Collection Classes

Collections are a group of objects bound together by a common characteristic. Java has various built-in classes to support the collection of objects, either of the same type or general. The most basic construct to work with is the *array* of which you will come to know of in a SECTION[1] later in this chapter.

30.1 Introduction to Collections

The most basic collection interface is called `Collection`. This interface gives the user a generic usage of a collection.

```
import java.util.Collection;   // Interface
import java.util.ArrayList;    // Implementation
import java.util.HashSet;      // Implementation
...
Collection coll1 = new ArrayList();
Collection coll2 = new HashSet();
...
< Use coll1 & coll2 >
```

In the above there are two collections. The usage of the collections are the same, the implementations are different. If the existing collection implementations do not meet your needs, you can write your version of the implementation. Your version of the implementation just needs to implement the same `java.util.Collection` interface, then you can switch to using your implementation and the code that is using the collection does not need to be changed.

```
import java.util.Collection;
import com.yourcomp.util.YourCollectionImpl;
...
Collection coll1 = new YourCollectionImpl();
Collection coll2 = new YourCollectionImpl();
...
< Use coll1 & coll2 >
```

The Java JDK collection implementations are quite powerful and good, so it is unlikely that you will need to write your own.

All collections contain object references. Because of that, if the object is changed after it was put in the collection, the object that is 'in' the collection also 'changes'. The object is not really in the collection, only the object reference is. It is not guaranteed that the objects 'inside' the collections

1 Chapter 29 on page 145

won't change. This is an issue only if you put an actively used object in the collection. In that case when you are adding an object that could change any time you need to make a copy or clone of the object. A new object will be created and its reference will be put in the collection. In that case there will be no object references outside of the collection, so the objects 'inside' the collection can only be changed if we take out an object reference from the collection.

Aside from the `java.util.Collection` interface, the Java JDK has the `java.util.Map` interface as well. This defines key value mappings. Implementations of the Map interface do not contain collections of objects. Instead they contain collections of key->value mappings.

```
import java.util.Map;
import java.util.Hashtable;
...
Map map = new Hashtable();
...
map.put( key, value );
```

All collections need to have the same basic operations. Those are:

- Adding element(s) to the collection
- Removing element(s) from the collection
- Obtaining the number of elements in the collection
- Listing the contents of the collection, (Iterating through the collection)

Before selecting a particular collection implementation, ask the following question:

- Can my collection contain the same elements, i.e. are duplicates allowed?
- Can my collection contain the **null** element?
- Should the collection maintain the order of the elements? Is the order important in any way?
- How do you want to access an element? By index, key or just with an iterator?
- Does the collection need to be synchronized?
- From a performance perspective, which one needs to be faster, updates or reads?
- From a usage perspective, which operation will be more frequent, updates or reads?

Once you know your needs, you can select an exsisting implementation. But first decide if you need a 'Collection', or a 'Map'.

30.2 Generics

Since JDK version 1.5 an enhancement to the type system of the Java language has been added. It is called *Generics.* Most often *Generics* will be used with the collection classes.

parameterized type <E> :

All collection implementations since 1.5 now have one 'parameterized type <E>' added. The 'E' refers to an *Element* type. When a collection is created the actual 'Element type' will replace the E.

Objects put into a collection are upcasted to **Object** class. It means that you need to cast the object reference back when you get an element out from the collection. It also means that **you need to know** the type of the object when you taking it out. For this reason we usually add objects of the

same type to a collection. If a collection contains different types of objects, we will have difficulty finding out the type of the objects obtained from a collection at run time.

With the use of the 'parameterized type <E>', the 'Element-type' that can be put into the collection can be specified when the collection object is created.

```
Collection<Integer> ageList = new ArrayList<Integer>();
ageList.add( new Integer(46) );     // --- Integer can be added
ageList.add( "50" );  // --- Compilation error, ageList can have only
Integers inside
```

`ageList` is a collection that can contain only Integer objects as elements. No casting is required when we take out an element.

```
Integer age = ageList.get(0);
```

For more information about Generics, please go to JAVA PROGRAMMING/GENERICS[2].

30.3 Collection or Map

The Java JDK contains two high level Interfaces:

- java.util.Collection
- java.util.Map

Implementations for those interfaces are not interchangeable. Collections are used for collecting Java objects. Maps are used for mapping key/value pairs.

30.3.1 Collection

Use the Collection interface if you need to keep related (usually the same type of) objects together in a collection where you can:

- Search for a particular element
- List the elements
- Maintain and/or change the order of the elements by using the collection basic operations (Add, Remove, Update,..)
- Access the elements by an index number

The advantages of using the `Collection` interface are:

- Gives a generic usage, as we talked about above, it is easy to switch implementation
- It makes it easy to convert one type of collection to an other.

The `Collection` interface defines the following basic operations:

- **boolean** add(E o);

2 Chapter 22 on page 119

- **boolean** addAll(Collection c);
- **boolean** remove(**Object** o);
- **boolean** removeAll(Collection c);
- **boolean** retainAll(Collection c);

The methods above return **true** if the collection has changed due to the operation. Note that in addAll() we can add any type of collection. This is the beauty of using the Collection interface. You can have a LinkedList and just call the `addAll(list)` method, passing in a list. You can pass in a `Vector`, an `ArrayList`, a `HashSet`, a `TreeSet`, a `YourImpOfCollection`, ... All those different type of collections will be **magically** converted to a `LinkList`.

Lets have a closer look at this *magic*. The conversion is easy because the `Collection` interface defines a standard way of looping through the elements. The following code is a possible implementation of `addAll()` method of the LinkList.

```
import java.util.Collection
import java.util.Iterator
...
public String addAll( Collection coll )
{
   int sizeBefore = _linkList.size();
   Iterator iter = coll.iterator();
   while( iter.hasNext() )
   {
      _linkList.add( iter.next() );
   }
   if ( sizeBefore > _linkList.size() )
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

The above code just iterates through the passed in collection and adds the elements to the link list. You do not have to do that, since that is already defined. What you might need to code for is to loop through a 'Customer' collection:

```
import java.util.Collection
import java.util.Iterator
import java.yourcompany.Customer
...
public String printCustomerNames( Collection customerColl )
{
   StringBuffer buf = new StringBuffer();

   Iterator iter = customerColl.iterator();
   while( iter.hasNext() )
   {
      Customer cust = (Customer) iter.next();
      buf.append( cust.getName() );
      buf.append( "\n" );
   }
  return buf.toString();
}
```

Notice two things:

- The above code will work for all type of collections.
- We have to know the type of objects inside the collection, because we call a method on it.

30.3.2 Map

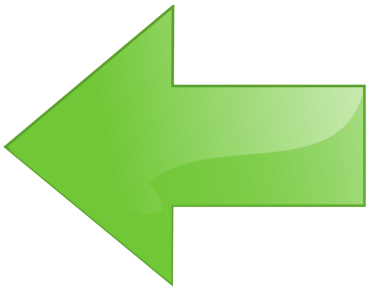


Figure 11: Figure 1:Map Interfaces

A map, sometimes also called an *Associated Array* or a *Dictionary*, can be thought of as an array where the index need not be an integer.

Use the Map interface if you need to keep related objects together in a Map where you can:

- Access an element by a key object
- Map one object to other

java.util.Map<K,V> : maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The key is usually a non-mutable object. The value object however can be a mutable object.

java.util.SortedMap<K,V> : same as the Map interface, plus the keys in the Map are sorted.

30.4 Set or List or Queue

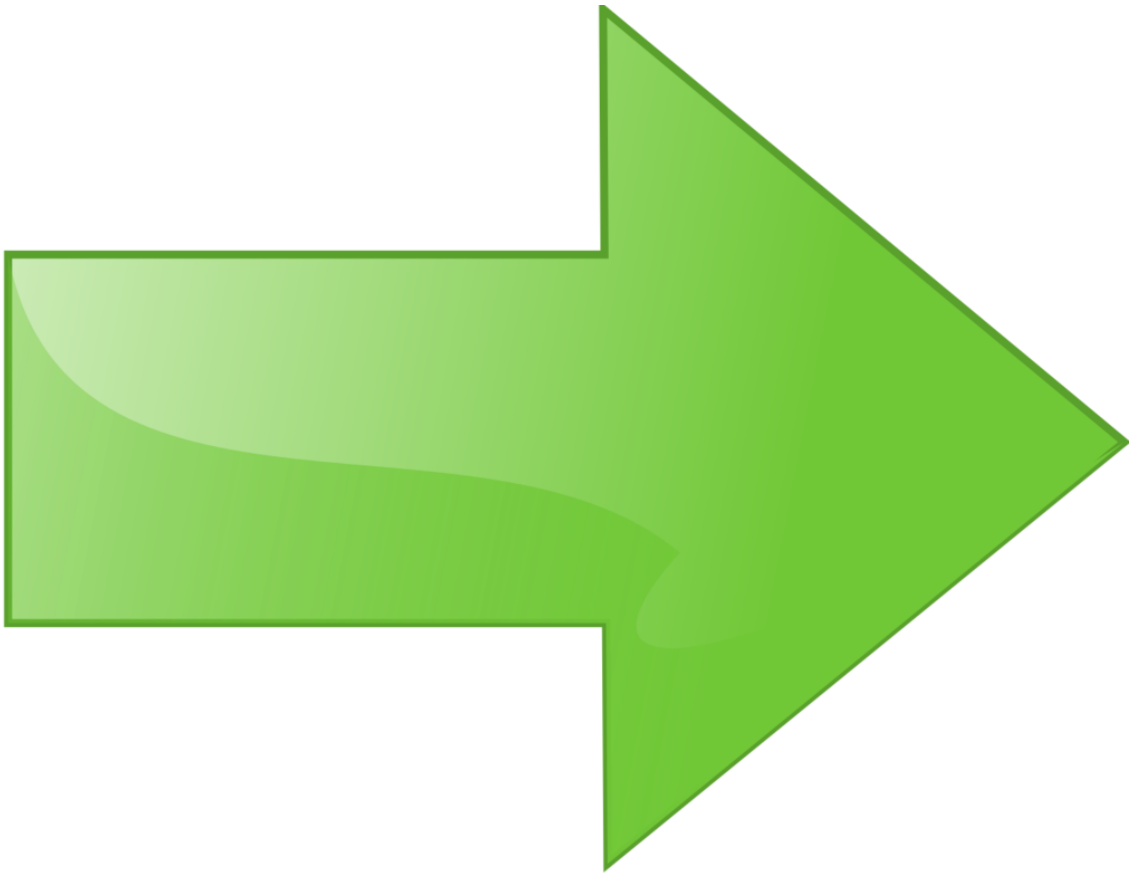


Figure 12: Figure 2:

There is no direct implementation for the `java.util.Collection` interface. The Collection interface has five sub interfaces. Those are:

java.util.Set<E> : contains unique elements, so duplicates not allowed. It is similar to a mathematical Set.

java.util.List<E> : elements are put in the list in a certain order, and can be accessed by an index. Duplicates are allowed, the same element can be added to a list.

java.util.SortedSet<E> : same as the Set interface; plus the elements in the SortedSet are sorted

java.util.Queue<E> : queues provide additional insertion, extraction, and inspection operations. There are FIFO (first in, first out) and LIFO (last in, first out) queues.

java.util.BlockingQueue<E> : waits for the queue to become non-empty when retrieving an element, and waits for space to become available in the queue when storing an element. Best used for producer-consumer queues.

30.4.1 Set

The basic implementation of the `Set` interface is the `HashSet`.

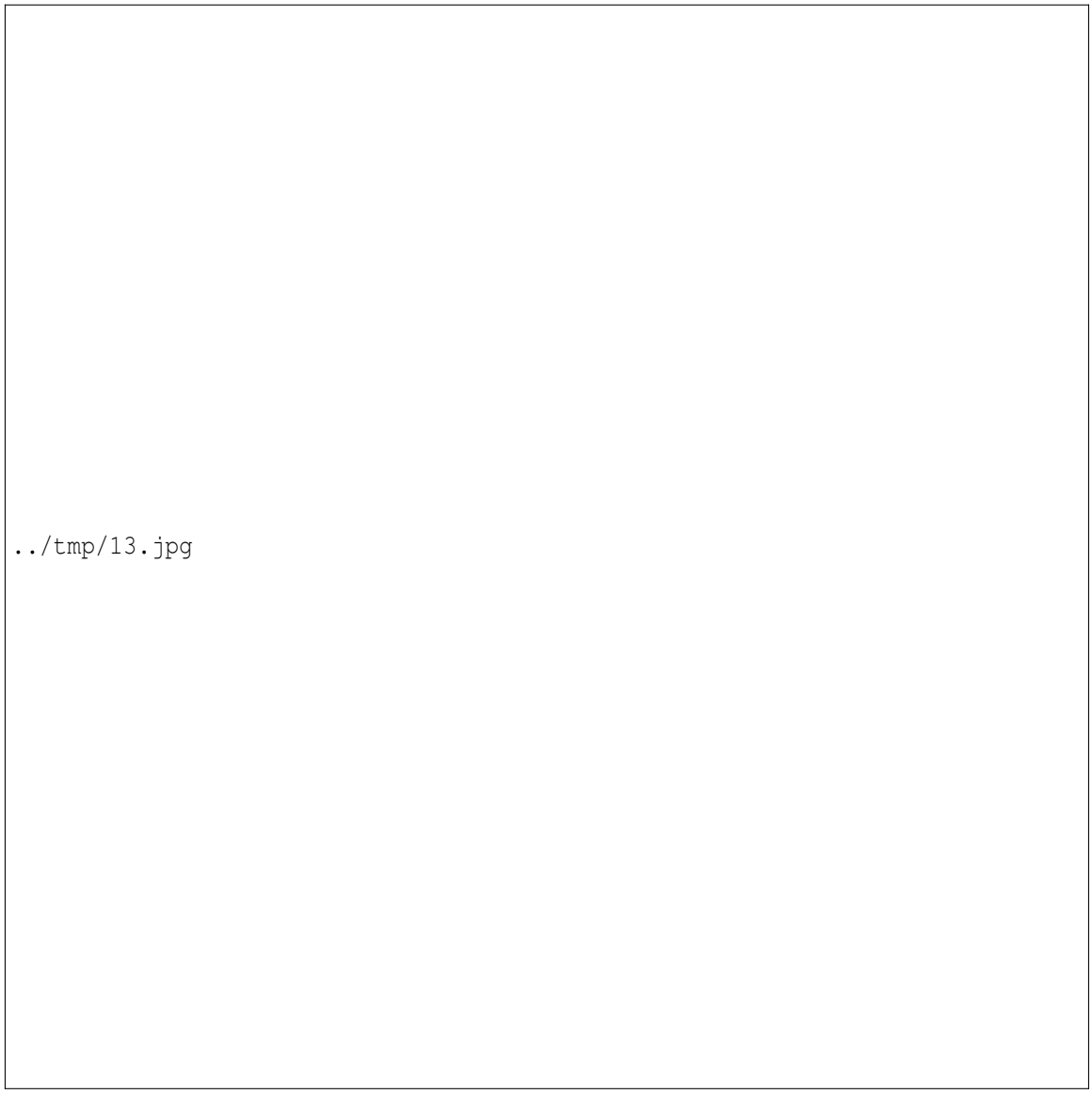


Figure 13

java.util.TreeSet<E>: Elements are sorted, not synchronized. **null** not allowed

java.util.HashSet<E> : Not synchronized. Allows the **null** elements

java.util.CopyOnWriteArraySet<E> : Thread safe, a fresh copy is created during modification operation. Add, update, delete are expensive.

java.util.EnumSet<E extends Enum<E>> : All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors.

java.util.LinkedHashSet<E> : Same as HashSet, plus defines the iteration ordering, which is the order in which elements were inserted into the set.

Detecting duplicate objects in Sets

`Set` cannot have duplicates in it. You may wonder how duplicates are detected when we are adding an object to the `Set`. We have to see if that object exists in the Set or not. It is not enough to check the object references, the objects' values have to be checked as well.

To do that, fortunately, each java object has the `boolean equal(Object obj);`[3], method available inherited from **Object**. You need to override it. That method will be called by the Set implementation to compare the two objects to see if they are equal or not.

There is a problem, though. What if I put two different type of objects to the Set. I put an Apple and an Orange. They can not be compared. Calling the `equal()`[4] method would cause a `ClassCastException`. There are two solutions to this:

- **Solution one** : Override the `int hashCode()`[5] method and return the same values for the same type of objects and return different values for different type of objects. The `equal()`[6] method is used to compare objects only with the same value of hashCode. So before an object is added, the Set implementation needs to:
  - find all the objects in the Set that has the same hashCode as the candidate object hashCode
  - and for those, call the `equal()`[7] methods passing in the candidate object
  - if any of them returns true, the object is not added to the Set.
- **Solution two** : Create a super class for the Apple and Orange, let's call it Fruit class. Put Fruits in the Set. You need to do the following:
  - Do not override the `equals()` and `hashCode()` methods in the Apple and Orange classes
  - Create `appleEquals()` method in the Apple class, and create `orangeEquals()` method in the Orange class
  - Override the `hashCode()` method in the Fruit class and return the same value, so the `equals()` is called by the Set implementation
  - Override the `equals()` method in the Fruit class for something like this.

```
public boolean equals( Object obj )
{
```

3 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.Object%23equals%28%29%20Method
4 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.Object%23
5 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.Object%23hashCode%28%29%20Method
6 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.Object%23
7 http://en.wikibooks.org/wiki/Java%20Programming%2FAPI%2Fjava.lang.Object%23

```
    boolean ret = false;
    if ( this instanceof Apple &&
          obj instanceof Apple )
    {
        ret = this.appleEquals(obj);
    }
    else if ( this instanceof Orange &&
              obj  instanceof Orange )
    {
        ret = this.orangeEquals(obj);
    }
    else
    {
        // --- Can not compare Orange to Apple ---
       ret = false;
    }
    return ret;


}
```

Note:

- only the objects that have the same hashCode will be compared.
- you are responsible to override the `equal()` and `hashCode()` methods. The default implementations in Object won't work.
- Only override the `hashCode()`[8] method if you want to eliminate value duplicates.
- Do not override the `hashCode()`[9] method if you know that the values of your objects are different, or if you only want to prevent adding the exactly same object.
- Beware that the `hashCode()`[10] may be used in other collection implementaions, like in a Hashtable to find an object fast. Overriding the default `hashCode()` method may affect performance there.
- the default hashCodes are unique for each object created, so if you decide not to override the `hashCode()` method, there is no point overriding the `equal()` method, as it won't be called.

SortedSet

The SortedSet interface extends the Set Interface. All elements in the SortedSet must implement the Comparable Interface, futher more all elements must be mutually comparable.

Note that the ordering maintained by a sorted set must be consistent with equals if the sorted set is to correctly implement the Set interface. This is so because the Set interface is defined in terms of the equals operation, but a sorted set performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal.

8 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.OBJECT%23HASHCODE%28%29%20METHOD

9 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.OBJECT%23HASHCODE%28%29%20METHOD

10 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FAPI%2FJAVA.LANG.OBJECT%23HASHCODE%28%29%20METHOD

The SortedSet interface has additional methods due to the sorted nature of the 'Set'. Those are:

- `E first();` -- returns the first element
- `E last();` -- returns the last element
- `SortedSet headSet(E toElement);` -- returns from the first, to the exclusive toElement
- `SortedSet tailSet(E fromElement);` -- returns from the inclusive fromElement to the end
- `SortedSet subSet(E fromElement, E toElement);` -- returns elements range from fromElement, inclusive, to toElement, exclusive. (If fromElement and toElement are equal, the returned sorted set is empty.)

30.4.2 List

The List has the following implemenations:

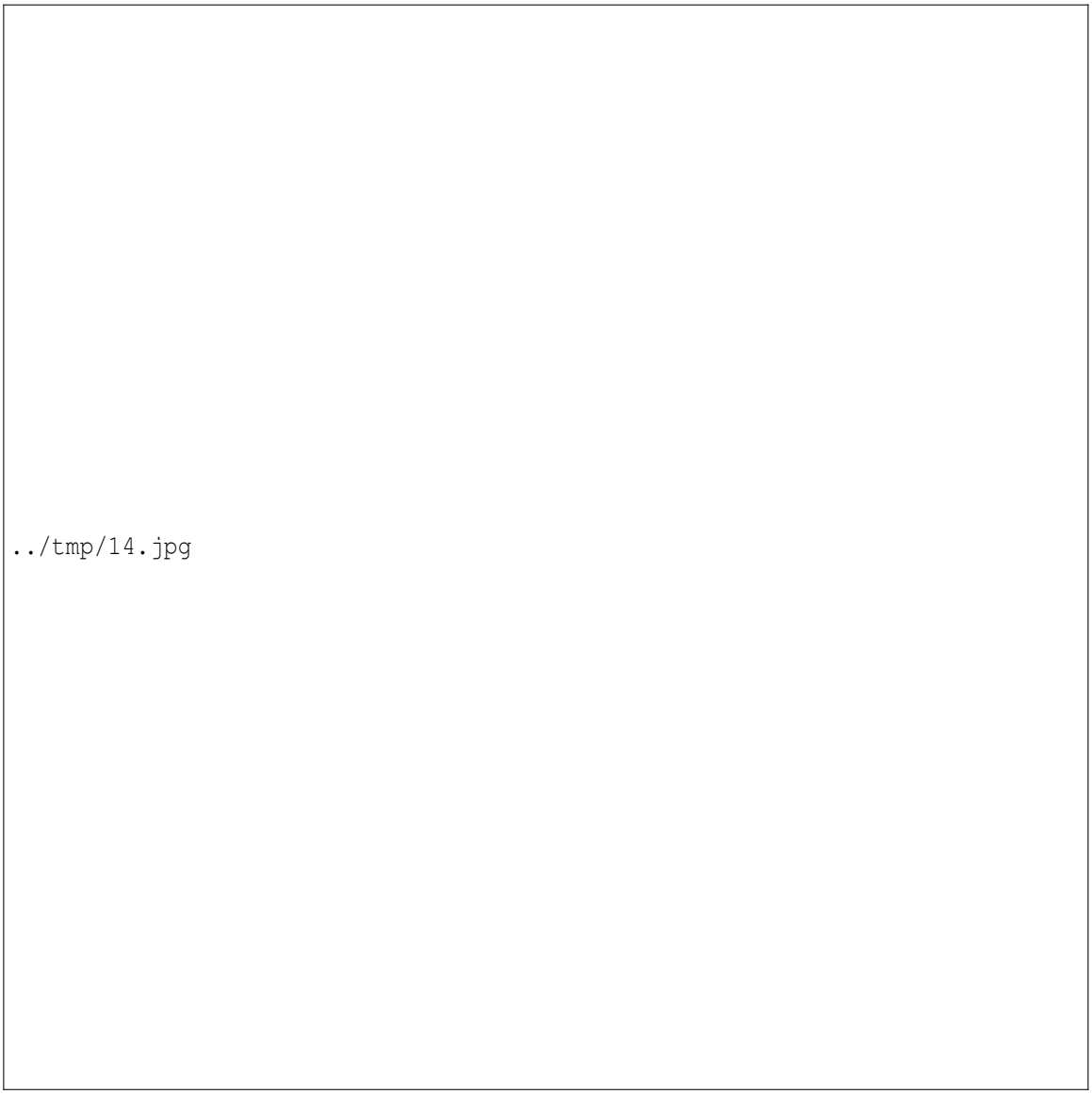


Figure 14

java.util.Vector<E> : Syncronized, use in multiple thread access, otherwise use ArrayList

java.util.Stack<E> : It extends class Vector with five operations that allow a vector to be treated as a stack. It represents a last-in-first-out (LIFO) stack of objects.

java.util.ArrayList<E> : Non-syncronized, use in single thread environment, otherwise use Vector

java.util.LinkedList<E> : Non-syncronized, update operation is faster than other lists, easy to use for stacks, queues, double-ended queues.

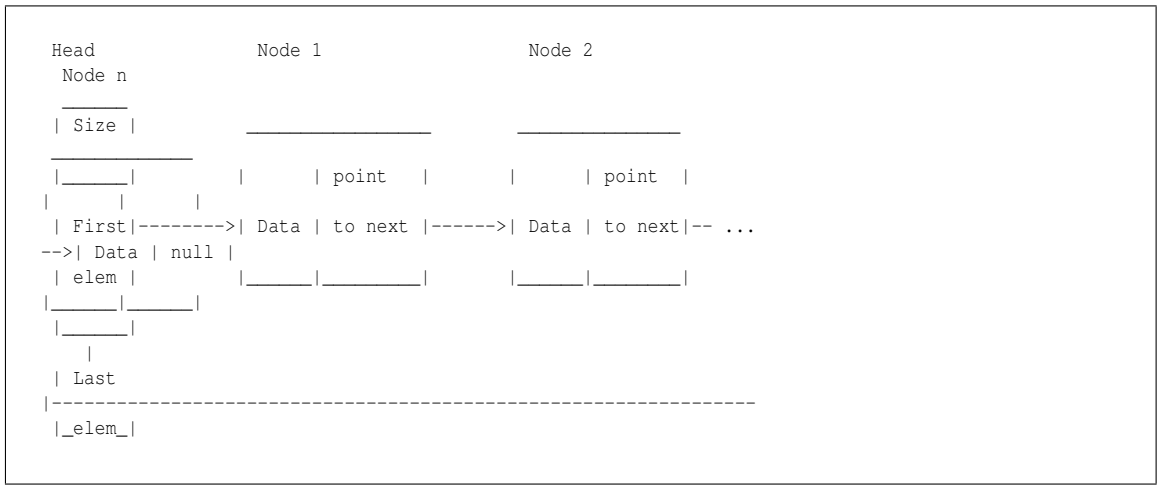
javax.management.AtributeList<E> : Represents a list of values for attributes of an MBean. The methods used for the insertion of Attribute objects in the AttributeList overrides the corresponding methods in the superclass ArrayList. This is needed in order to insure that the objects contained in the AttributeList are only Attribute objects.

javax.management.relation.RoleList<E> : A RoleList represents a list of roles (Role objects). It is used as parameter when creating a relation, and when trying to set several roles in a relation (via 'setRoles()' method). It is returned as part of a RoleResult, to provide roles successfully retrieved.

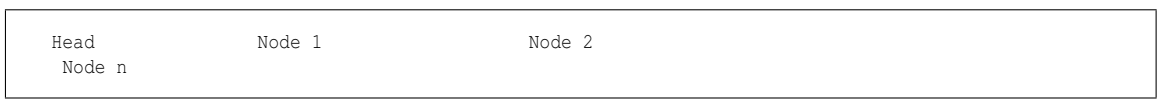
javax.management.relation.RoleUnresolvedList<E> : A RoleUnresolvedList represents a list of RoleUnresolved objects, representing roles not retrieved from a relation due to a problem encountered when trying to access (read or write to roles).

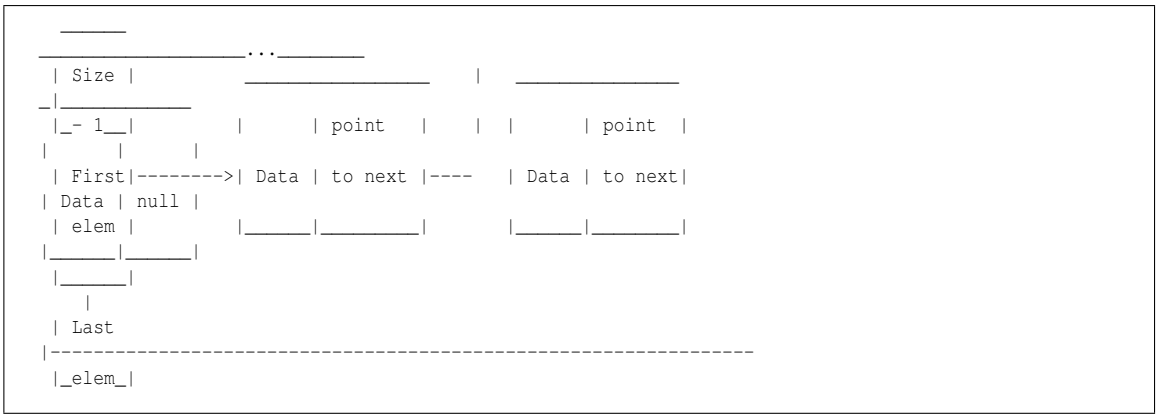
The basic implemenation of the `List` interface is the `ArrayList`. The ArrayList is not syncronized, not thread safe. `Vector` is syncronized, and thread safe. `Vector` is slower, because of the extra overhead to make it thread safe. When only one thread is accessing the list, use the ArrayList. Whenever you insert or remove an element from the list, there are extra overhead to reindex the list. When you have a large list, and you have lots of insert and remove, consider using the `LinkedList`.

The name `LinkList` implies a special data structure where the elements/nodes are connected by pointers. To remove an element from the link list the pointers need to be rearranged.



After removing Node 2:





30.4.3 Queue

The Queue interface adds the following operations to the Collection interface:

E element()	Retrieves, but does not remove, the head of this queue. This method differs from the peek method only in that it throws an exception if this queue is empty
boolean offer(E o)	Inserts the specified element into this queue, if possible.
E peek()	Retrieves, but does not remove, the head of this queue, returning null if this queue is empty
E poll()	Retrieves and removes the head of this queue, or null if this queue is empty
E remove()	Retrieves and removes the head of this queue. This method differs from the poll method in that it throws an exception if this queue is empty.

- java.util.PriorityQueue<E> : orders elements according to an order/priority specified at construction time, null element is not allowed.
- java.util.concurrent.ArrayBlockingQueue<E> : orders elements FIFO; syncronized, thread safe.
- java.util.concurrent.SyncronousQueue<E> : each put must wait for a take, and vice versa, does not have any internal capacity, not even a capacity of one, an element is only present when you try to take it; you cannot add an element (using any method) unless another thread is trying to remove it

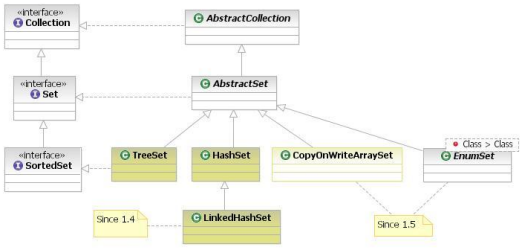


Figure 15

30.5 Map Classes

The Map interface has the following implementations:

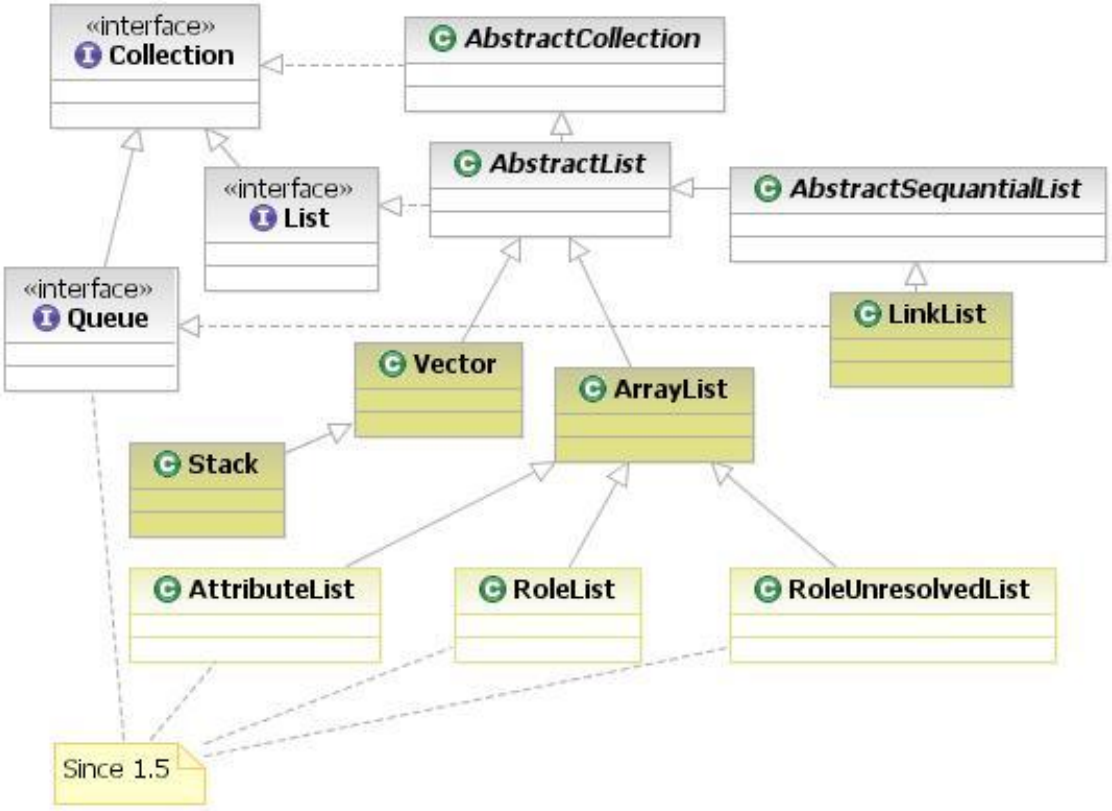


Figure 16

java.util.TreeMap<E>: guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class, not-syncronized.

java.util.HashMap<E> : is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls

java.util.concurrent.ConcurrentHashMap : same Hashtable, plus retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove).

java.util.Hashtable<E> : Syncronized, null can not be used as key

java.util.WeakHashMap<E> : entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. Nonsyncronized.

java.util.LinkedHashMap<E> : This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map.

java.util.IdentityHashMap : This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values). In other words, in an IdentityHashMap, two keys k1 and k2 are considered equal if and only if

(k1==k2). (In normal Map implementations (like HashMap) two keys k1 and k2 are considered equal if and only if (k1==null ? k2==null : k1.equals(k2)).) Not-syncronized.

java.util.EnumMap : All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created. Enum maps are represented internally as arrays. This representation is extremely compact and efficient. Not-syncronized.

30.6 Thread Safe Collections

It is also called Concurrent Collections. Most of the popular collection classes have implementations for both single thread and multiple thread environments. The non-synchronized implementations are always faster. You can use the non-synchronized implementations in multiple thread environments, when you make sure that only one thread updating the collection at any given time.

A new Java JDK package was introduced at Java 1.5, that is `java.util.concurrent`. This package supplies a few Collection implementations designed for use in multithreaded environments.

The following table list all the syncronized collection classes:

	syncronized	**non-syncronized**
List	java.util.Vector	java.util.ArrayList
	java.util.Stack	
		java.util.LinkList
	java.util.concurrent.CopyOnWriteArrayList	
Set		java.util.TreeSet
		java.util.HashSet
		java.util.LinkHashSet
	java.util.concurrent.CopyOnWriteArraySet	
Map		java.util.TreeMap
	java.util.Hashtable	java.util.HashMap
	java.util.concurrent.ConcurrentHashMap	
		java.util.LinkedHashMap
		java.util.IdentityHashMap
		java.util.EnumMap

30.7 Classes Diagram (UML)

The following UML class diagram shows the **Collection** interfaces and their implementations.

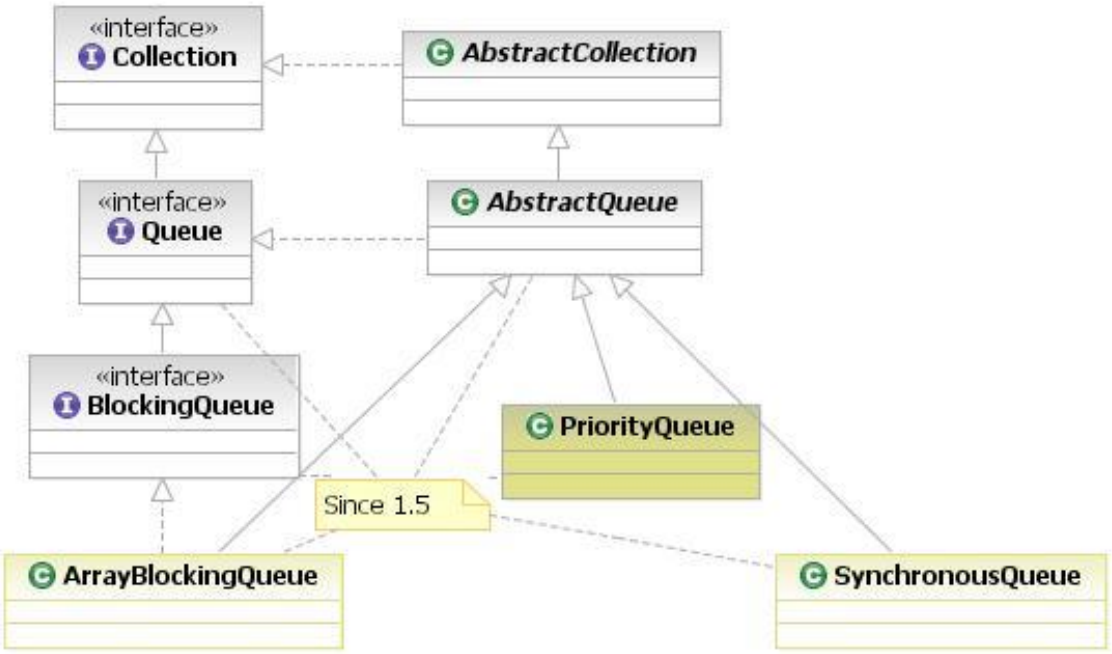


Figure 17

The following UML class diagram shows the **Map** interfaces and their implementations.

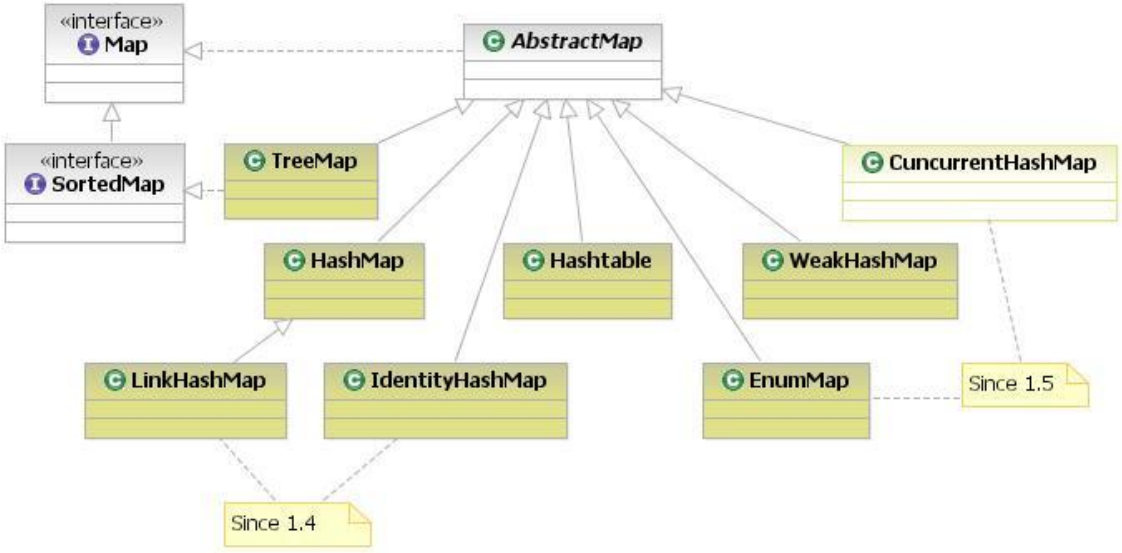


Figure 18

30.8 External links

- JAVA TUTORIAL ON COLLECTIONS[11]

CATEGORY:JAVA PROGRAMMING[12]

11 HTTP://JAVA.SUN.COM/DOCS/BOOKS/TUTORIAL/COLLECTIONS/INTERFACES/COLLECTION.HTML

12 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

31 Throwing and Catching Exceptions

In Java, there are two main flow of code executions.

- Normal main sequential code execution, the program doing what it meant to accomplish
- Exception handling code execution, the main program flow was interrupted by an error or some other condition that prevent the continuation of the normal main sequential code execution. After the interruption, that is when the exception is "thrown", the search for the first matching catch block/clause begins, the search is going backward against the main flow, until a match is found.

The exception handling is built in to the language. There is a keyword to **throw** an exception, there is keyword to **catch** an exception, and there is keyword(**throws**) to declare that a method could throw an exception that the caller has to deal with.

An exception can be thrown from anywhere in the code by the **throw** keyword, plus specifying a Java object, that is instance of the `Throwable` class. Applications should sub-class the `Exception` class as an exception object to be thrown.

The following is an example of throwing an instance of the ExceptionClass.

```
throw new ExceptionClass("Something happened, the main program flow
 has to be interrupted, the caller has to be informed about this");
```

At the above line the current method execution is interrupted and the created exception object is thrown back to the caller. The exception object should contain the information about the reason of the interruption.

The caller of the method catches the exception by the **catch** keyword.

```
catch (ExceptionClass e)
{
  // --- What to do with this interruption ??? ---
}
```

Sometimes just the name of the exception class is enough to know the reason of the interruption.

31.1 Catching Matching rules

When an exception is "thrown", the search for the first matching catch block/clause begins.

There are five main catching matching rules:

1. A thrown exception object, can be caught by the **catch** keyword with specifying the exception object's class or its super-class.
2. When there are a list of catch clauses, it is evaluated sequentially, applying the first rule. If there is a catch, the following catch clauses are ignored.
3. If there is a catch clause in the list, which will never be executed, because its super-class are listed before it, the compiler will give an error message.
4. The compiler enforce that all `Exception` and its sub-class exceptions must be handled by the programmer, except the `RuntimeException` and its sub-classes.
5. If the try block would never throw an exception that is specified in the catch list, the compiler gives an error.

31.1.1 Rule number 1

A thrown exception object, can be caught by the **catch** keyword with specifying the exception object's class or its super-class.

The `Throwable` class is the super-class of all exception class. So specifying with the catch keyword we can catch all type of exceptions.

The following piece of code would catch all type of exception, but is not recommended for normal use:

```
catch (Throwable th) {
    // --- I caught all type of exception ---
    log_it( th );
    // --- I was not suppose to catch Throwable so I throw it again
 ---
    throw th;
}
```

There are some special exceptions that used by the JVM, those are the sub-classes of `java.lang.Error`. We are not suppose the catch them. So we should use the following code to catch all application, and runtime exceptions.

```
catch (Exception e) {
    // --- I caught all application and runtime exceptions ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
```

31.1.2 Rule number 2

When there are a list of catch clauses, it is evaluated sequentially, applying the first rule, for each clause. If there is a catch, the following catch clauses are ignored.

The following piece of code demonstrates rule number 2.

The `NullPointerException` is caught by the first catch clause, the following catch clauses are ignored. The second catch clause will catch all `RuntimeException`'s, except the `NullPointerException`, because that will be caught by the first clause. All exceptions that are defined by the application, will be caught by the last catch clause, see below.

```
catch (NullPointerException e) {
    // --- I caught a Nulpointer ex. ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
catch (RuntimeException e) {
    // --- I caught runtime ex. but not NullPointer ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
catch (Exception e) {
    // --- I caught all application but not Runtime exceptions ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
```

31.1.3 Rule number 3

If there is a catch clause in the list which will never be executed, because its super-class are listed before it, the compiler will give an error message.

The following piece of code will not compile, it will give an error message saying that the last catch clause will never be executed. This demonstrates that the exception handling is built into the language. The compiler checks the list of catch clauses and reports an error if class follows any of it super-class on the list.

```
catch (NullPointerException e) {
    // --- I caught a Nullpointer ex. ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
catch (Exception e) {
    // --- I caught all application but not Runtime exceptions ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ---
    throw e;
}
catch (RuntimeException e) {  // -- COMPILATION ERROR, THIS CODE WILL
 NEVER BE EXECUTED ---
    // --- I caught runtime ex. but not NullPointer ---
    log_it( e );
    // --- I don't know how to handle this, so I throw it again ----
    throw e;
}
```

31.1.4 Rule number 4

The compiler enforce that all `Exception` and its sub-class exceptions must be handled by the programmer, except the `RuntimeException` and its sub-classes.

From the application's point of view there are two types of exceptions. One is called `RuntimeException`, that suppose to be thrown when there is an anticipated bug in the code. For example, if the program tries to access an array, with a number that is out side of the array size, then a `RuntimeException` is thrown. There are many other possibilities which indicate a coding bug.

The other type of exception is called the application exception. During application design a list of possible exception is defined that can anticipated during executing the business logic code. For example, the application tries to retrieve a customer, but the customer does not exist. This may not necessarily be a bug. In this case an application exception should be thrown. Application exceptions are sub-classes of the `Exception` class.

The java language enforces that all application exceptions are dealt with by the application programmer.

If a method throws an application exception, that must be declared in the method signature by the **throws** keyword. The caller of that method can either catch the exception by a catch block or it can make sure that it is being declared to be thrown in the current method, and so on. The search for a catch block goes backward on the methods track trace, until a match is found. The compiler ensures that someone has to handle the exception.

The following method can throw a `CustomerNotFoundException`, so it must declare it at the method signature.

```
public String method() throws CustomerNotFoundException
{
   ...
   throw new CustomerNotFoundException();
   ...
}
```

31.1.5 Rule number 5

If the try block would never throw an exception that is specified in the catch list, the compiler gives an error.

The following code would not compile because inside the try block no `Exception` is thrown.

```
try {
  ...
} catch ( Exception e ) {
}
```

The above code won't compile, because the try block does not throw exception. The compiler can be fooled with the following code.

```
try {
  if ( 1 == 0 ) throw new Exception();
  ...
} catch ( Exception e ) {
}
```

The above code will compile now.

31.2 Example of handling exceptions

Let's examine the following code:

```
public void methodA() throws SomeException, AnotherException
{
    //methodbody
}
public void methodB() throws CustomException
{
    //Methodbody
}
public void methodC()
{
    methodB();
    methodA();
}
```

In the code sample, methodC is invalid. Because methodA and methodB pass (or throw) exceptions, methodC must be prepared to handle them. This can be handled in two ways: a **try** - **catch** block, which will handle the exception within the method and a **throws** clause which would in turn throw the exception to the caller to handle. The above example will cause a compilation error, as Java is very strict about exception handling. So the programmer forced to handle any possible error condition at some point.

A method can do two things with an exception. Ask the calling method to handle it by the **throws** declaration. Or handle the exception inside the method by the **try**-**catch** block.

To construct a throws declaration, add `throws ExceptionName` (additional exceptions can be added with commas). To construct a **try** - **catch** block, use the following syntax

```
 ...
try {
   //Possibly exception-causing code
}
 catch (TheException e) {
   //Handle the exception
}
finally {
    //Optional.  Executes regardless of exceptions thrown
}
```

The original code can be modified to work correctly in multiple ways. For example, the following:

```
public void methodC() throws CustomException, SomeException
{
    try {
       methodB();
    }
    catch (AnotherException e) {
      //handle it
    }
    methodA();
}
```

The AnotherException from methodB will be handled locally, while CustomException and SomeException will be thrown to the caller to handle it.

31.3 Application Exceptions

Application Exception classes should extend the `java.lang.Exception` class. Some of the JDK classes also throw exception objects inherited from `java.lang.Exception`. If any of those Exception object is thrown, **it must be caught by the application** some point, by a catch-block. The compiler will enforce that there is a catch-block associated with an exception thrown, if the thrown exception object is inherited from `java.lang.Exception` and it is not the `java.lang.RuntimeException` or its inherited objects. However, `java.lang.RuntimeException` or its inherited objects, can be caught by the application, but that is not enforced by the compiler.

Lets see what is the catching criteria for a catch block to catch the "thrown" exception.

A catch-block will "catch" a thrown exception if and only if:

- the thrown exception object is the same as the exception object specified by the catch-block
- the thrown exception object is the subtype of the exception object specified by the catch-block

```
try {
    throw new Exception( "This will be caught below" );
}
catch( Exception e ) {
   // --- The "thrown" object is the same what is specified at the
catch-block --
}
```

```
try {
    throw new NullPointerException( "This will be caught below" );
}
catch( Exception e ) {
   // --- NullPointerException is subclass of the Exception class --
}
```

There can be more than one catch-block for a try-block. The catching blocks evaluated sequentially one by one. If a catch-block catch the exception, the others will not be evaluated.

Example:

```
try {
     throw new NullPointerException( "This will be caught below" );
}
catch( Exception e ) {
   // --- The above NullPointerException will catched by here --
}
catch(<strike> NullPointerException e </strike>) {  // --- THIS CODE IS
NEVER EXECUTED - Compiler error
   // ---
}
```

Because NullPointerException is subclass of the Exception class. All NullPointerExceptions will be caught by the first catch-block.

Instead the above code should be rewritten as follows:

```
try {
     throw new NullPointerException( "This will be caught below" );
}
catch( NullPointerException e ) {
   // --- The above NullPointerException will caught  here ---
}
catch( Exception e ) {
   // --- Any other exception except the NullPointerException will be
caught here --
}
```

31.4 Runtime Exceptions

The `java.lang.RuntimeException` exception class is inherited from `java.lang.Exception`. It is a special exception class, because catching this exception class or its subclasses are not enforced by the Java compiler.

runtime exception : Runtime exceptions are usually caused by data errors, like arithmetic overflow, divide by zero, ... Runtime exceptions are not business related exceptions. In a well debugged code, runtime exceptions should not occur. Runtime exceptions should only be used in the case that the exception could be thrown by and only by something hard-coded into the program. These should not be able to be triggered by the software's user(s).

31.4.1 NullPointerException

NullPointerException is a RuntimeException. In Java, a special **null** can be assigned to an object reference. NullPointerException is thrown when an application attempts to use an object reference, having the **null** value. These include:

- Calling an instance method on the object referred by a null reference.

- Accessing or modifying an instance field of the object referred by a null reference.
- If the reference type is an array type, taking the length of a null reference.
- If the reference type is an array type, accessing or modifying the slots of a null reference.
- If the reference type is a subtype of Throwable, throwing a null reference.

Applications should throw instances of this class to indicate other illegal uses of the null object.

```
Object obj = null;
...
obj.toString();  // --- This will throw a NullPointerException ---
```

The above code shows one of the pitfall of Java, and the most common source of bugs. No object is created and the compiler does not detect it. NullPointerException is one of the most common exceptions thrown in Java.

why do we need **null** ? :

The reason we need it is because many times we need to create an object reference, before the object itself is created. Object references cannot exist without a value, so we assign the **null** value to it.

```
public Customer getCustomer()
{
  Customer customer = null;
  try {
     ...
     customer = createCustomer();
     ...
  }
  catch ( Exception e)
  {
  ...
  }
return customer;
}
```

In the above code we want to create the Customer inside the try-block, but we also want to return the object reference to the caller, so we need to create the object reference outside of the try-block, because of the scoping rule in Java. This is one of the pitfall of Java.

31.5 Main Exception Classes

- java.lang.Throwable : The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

  A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a cause: another throwable that caused this throwable to get thrown. The cause facility is new in release 1.4. It is also known as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another

- java.lang.Error : An Error indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

- java.lang.Exception : The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch. Also this is the class that a programmer may want to extend when adding business logic exceptions.

- java.lang.RuntimeException : RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

- java.lang.NullPointerException : Thrown when an application attempts to use null in a case where an object is required.

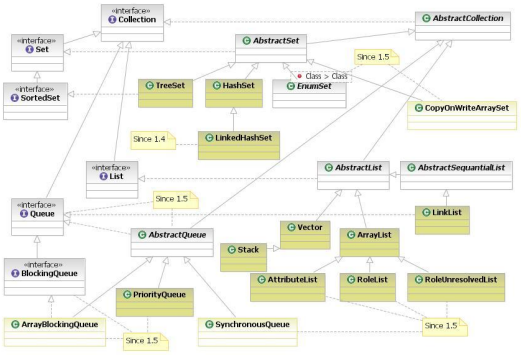


Figure 19

31.6 See Also

- JAVA PROGRAMMING/KEYWORDS/TRY[1]
- JAVA PROGRAMMING/KEYWORDS/CATCH[2]
- JAVA PROGRAMMING/KEYWORDS/THROWS[3]
- JAVA PROGRAMMING/KEYWORDS/THROW[4]

CATEGORY:JAVA PROGRAMMING[5]

This page describes some techniques for preventing NullPointerException.

It does not describe general techniques for how you should program Java. It is of some use, to make you more aware of null values, and to be more careful about generating them yourself.

Note that this list is not complete - there are no rules for preventing NullPointerException entirely in Java, because the standard libraries have to be used, and they can cause NullPointerExceptions. Also, it is possible to observe an uninitialised final field in Java, so you can't even treat a final field as being completely trusted during the object's creation.

A good approach is to learn how to deal with NullPointerExceptions first, and become competent with that. These suggestions will help you to cause less NullPointerExceptions, but they don't replace the need to know about NullPointerExceptions.

31.7 Minimize the use of the keyword 'null' in assignment statements

This means not doing things like:

```
String s=null;
while (something)
    if (something2)
        s="yep";

if (s!=null)
    something3(s);
```

You can replace this with:

```
boolean done=false;

while (!done && something)
    if (something2)
    {
       done=true;
       something3("yep");
    }
```

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FTRY
2 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FCATCH
3 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FTHROWS
4 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING%2FKEYWORDS%2FTHROW
5 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING

You might also consider replacing null with "" in the first example, but default values bring about bugs caused by default values being left in place. A NullPointerException is actually better, as it allows the runtime to tell you about the bug, rather than just continue with a default value.

31.8 Minimize the use of the new Type[int] syntax for creating arrays of objects

An array created using new Object[10] has 10 null pointers. That's 10 more than we want, so use collections instead, or explicitly fill the array at initialisation with:

```
Object[] objects={"blah",5,new File("/usr/bin")};
```

or:

```
Object[] objects;
objects=new Object[]{"blah",5,new File("/usr/bin")};
```

31.9 Check all references obtained from 'untrusted' methods

Many methods that can return a reference can return a null reference. Make sure you check these. For example:

```
File file=new File("/etc");
File[] files=file.listFiles();
if (files!=null)
{
    stuff
}
```

File.listFiles() can return null if "/etc" is not a directory.

You can decide to trust some methods not to return null, if you like, but that's an assumption you're making. Some methods that don't specify that they might return null, actually do, instead of throwing an exception.

31.10 Comparing string variable with a string literal

When you compare a variable with a string literal, always put the string literal first. For example do:

```
if ( "OK".equals( state ) )
{
  ...
```

```
  }
```

and do not do:

<strike>

```
if ( state.equals( "OK" ) )
{
  ...
}
```

</strike>

If the 'state' variable is null, you get a NullPointerException in the second example, but not in the first one.

CATEGORY:JAVA PROGRAMMING[6]

31.11 See also

- JAVA PLATFORM[7]
- JAVA API[8]
- JAVA VIRTUAL MACHINE[9]
- GCC[10] (includes a Java to machine language compiler)
- COMPARISON OF JAVA TO C++[11].
- JINI[12]
- ECLIPSE[13] IDE[14] http://eclipse.org/
- NETBEANS[15] (Another open source IDE)
- OPTIMIZATION OF JAVA[16]

6 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AJAVA%20PROGRAMMING
7 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20PLATFORM
8 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20API
9 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20VIRTUAL%20MACHINE
10 HTTP://EN.WIKIPEDIA.ORG/WIKI/GNU%20COMPILER%20COLLECTION
11 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPARISON%20OF%20JAVA%20TO%20CPLUSPLUS
12 HTTP://EN.WIKIPEDIA.ORG/WIKI/JINI
13 HTTP://EN.WIKIPEDIA.ORG/WIKI/ECLIPSE%20%28COMPUTING%29
14 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPEN%20SOURCE%20INTEGRATED%20DEVELOPMENT%20ENVIRONMENT
15 HTTP://EN.WIKIPEDIA.ORG/WIKI/NETBEANS
16 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPTIMIZATION%20OF%20JAVA

32 Links

32.1 External References

- JAVA CERTIFICATION MOCK EXAMS[1] 500+ questions with exam simulator (this is the older 1.4 version of the exam)
- JAVA LANGUAGE SPECIFICATION, 3RD EDITION[2].
- THINKING IN JAVA[3]
- JAVA 5 SDK DOCUMENTATION[4]
- JAVA 5 SDK DOCUMENTATION IN CHM FORMAT[5]
- JAVA 5 API DOCUMENTATION[6]
- THE JAVA TUTORIAL[7]
- Sun Developer Network NEW TO JAVA CENTER[8]
- A SIMPLE JAVA TUTORIAL [9]
- TWO SEMESTERS OF COLLEGE-LEVEL JAVA LECTURES--FREE[10]
- JAVA LESSONS - INTERACTIVE JAVA PROGRAMMING TUTORIALS BASED ON EXAMPLES[11]
- JAVA TUTORIALS FOR KIDS AND ADULTS[12]

32.2 External links

- JAVA CERTIFICATION MOCK EXAMS[13] 500+ questions with exam simulator
- SWINGWIKI[14] - Open documentation project containing tips, tricks and best practices for Java Swing development
- JAVATIPS[15] - Blog project containing best JAVA tips and tricks
- FREE JAVA/ ADVANCED JAVA BOOKS[16]

---

1 HTTP://WWW.CERTIFICATION4CAREER.COM
2 HTTP://JAVA.SUN.COM/DOCS/BOOKS/JLS/THIRD_EDITION/HTML/J3TOC.HTML
3 HTTP://WWW.MINDVIEW.NET/BOOKS/TIJ/
4 HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/INDEX.HTML
5 HTTP://WWW.ZEUSEDIT.COM/FORUM/VIEWTOPIC.PHP?T=10
6 HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/API/INDEX.HTML
7 HTTP://JAVA.SUN.COM/DOCS/BOOKS/TUTORIAL/INDEX.HTML
8 HTTP://JAVA.SUN.COM/DEVELOPER/ONLINETRAINING/NEW2JAVA/INDEX.HTML
9 HTTP://WWW.ALNAJA7.NET/PROGRAMMER/393/ITCS-393.HTM
10 HTTP://CURMUDGEON99.GOOGLEPAGES.COM/
11 HTTP://JAVALESSONS.COM
12 HTTP://WWW.KIDWARESOFTWARE.COM
13 HTTP://WWW.CERTIFICATION4CAREER.COM
14 HTTP://WWW.SWINGWIKI.ORG
15 HTTP://WWW.AKKIDI.COM
16 HTTP://WWW.FREEBOOKCENTRE.NET/JAVATECH/JAVACATEGORY.HTML

- FREE JAVA AND J2EE EBOOKS[17]
- JAVA BOOKS AVAILABLE FOR FREE DOWNLOADS[18]
- ROEDY GREEN'S JAVA & INTERNET GLOSSARY[19] A comprehensive reference that's also an excellent starting point for beginners
- C2: JAVA LANGUAGE[20]
- NETBEANS IDE[21]
- ECLIPSE IDE[22]
- ZEUS FOR WINDOWS IDE[23]
- OFFICIAL JAVA HOME SITE[24]
- ORIGINAL JAVA WHITEPAPER[25]
- COMPLETE JAVA PROGRAMMING TUTORIALS[26]
- JAVAPASSION, JAVA COURSE[27] - The Javapassion Site, Java Course, driven by Sang Shin from Sun
- BEANSHELL[28] Interpreted version
- THE JAVA LANGUAGE SPECIFICATION, THIRD EDITION[29] "This book attempts a complete specification of the syntax and semantics of the language."
- THE JAVA VIRTUAL MACHINE SPECIFICATION, SECOND EDITION[30] and AMENDMENTS[31]
- A PURE JAVA DESKTOP[32]
- JAVAPEDIA PROJECT[33]
- Bruce Eckel Thinking in Java Third edition -- HTTP://WWW.MINDVIEW.NET/BOOKS/TIJ/[34] (Bruce has an C/C++ free book available on-line too)
- JAVAGAMEDEVELOPMENT[35] Daily news and articles on Java Game Development
- JAVA CERTIFICATIONS SITE(SCJP,SCWCD,SCBCD,JAVA 5.0,SCEA[36]
- JAVA PROGRAMMING FAQS AND TUTORIALS[37]
- MORE RESOURCES[38]
- JAVA LESSONS[39]
- ONLINE JAVA TUTORIAL[40]

17 HTTP://WWW.BESTEBOOKSWORLD.COM/DEFAULT.ASP?CAT=55
18 HTTP://WWW.TECHBOOKSFORFREE.COM/JAVA.SHTML
19 HTTP://WWW.MINDPROD.COM/JGLOSS/JGLOSS.HTML
20 HTTP://C2.COM/CGI/WIKI?JAVALANGUAGE
21 HTTP://WWW.NETBEANS.ORG
22 HTTP://WWW.ECLIPSE.ORG
23 HTTP://WWW.ZEUSEDIT.COM/JAVA.HTML
24 HTTP://JAVA.SUN.COM
25 HTTP://JAVA.SUN.COM/DOCS/WHITE/LANGENV/
26 HTTP://WWW.ROSEINDIA.NET/JAVA/
27 HTTP://WWW.JAVAPASSION.COM/JAVAINTRO/
28 HTTP://WWW.BEANSHELL.ORG
29 HTTP://JAVA.SUN.COM/DOCS/BOOKS/JLS/THIRD_EDITION/HTML/J3TOC.HTML
30 HTTP://JAVA.SUN.COM/DOCS/BOOKS/VMSPEC/2ND-EDITION/HTML/VMSPECTOC.DOC.HTML
31 HTTP://JAVA.SUN.COM/DOCS/BOOKS/VMSPEC/2ND-EDITION/JVMS-CLARIFY.HTML
32 HTTP://WWW.JDISTRO.COM/
33 HTTP://WIKI.JAVA.NET/BIN/VIEW/JAVAPEDIA/
34 HTTP://WWW.MINDVIEW.NET/BOOKS/TIJ/
35 HTTP://JAVAGAMEDEVELOPMENT.NET
36 HTTP://WWW.JAVABEAT.NET
37 HTTP://WWW.APL.JHU.EDU/~{}HALL/JAVA/FAQS-AND-TUTORIALS.HTML
38 HTTP://FINDSHELL.COM
39 HTTP://WWW.LANDOFCODE.COM/JAVA/
40 HTTP://COMPUTER.FREEONLINEBOOKSTORE.ORG/SHOWBOOK.PHP?SUBCATEGORYID=17

- Full Java Tutorial[41] - A collection of free premium programming tutorials
- Java Certification Practice Tests and Articles[42]
- Kode Java - Learn Java Programming by Examples[43]
- Games Programming Wiki[44] - Java tutorials and lessons based on game programming
- WikiJava[45] - Examples and tutorials in Java
- Download Free java ebooks from 83 ebooks collection[46] - Free Java Ebooks to download from ebookslab.info
- Download Free Sun Certified Developer for Java Web Services[47] - Free Java Ebooks to download from ebooks.mzwriter.net
- Code Conventions for the Java Programming Language[48] - At SUN

Newsgroups:

news:comp.lang.java comp.lang.java (Google's web interface[49])

Links[50]

41 http://www.meshplex.org/wiki/Java/Introduction_to_Java
42 http://www.ucertify.com/vendors/Sun.html
43 http://www.kodejava.org/
44 http://gpwiki.org/
45 http://www.wikijava.org/
46 http://www.ebookslab.info/download-free-java-ebooks
47 http://ebooks.mzwriter.net/e-books/share_ebook-310-220-sun-certified-developer-for-java-web-
48 http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
49 http://groups.google.com/groups?group=comp.lang.java
50 http://en.wikibooks.org/wiki/Category%3AJava%20Programming

33 License

34 GNU Free Documentation License

1. REDIRECT WIKIBOOKS:GNU FREE DOCUMENTATION LICENSE[1]

DE:JAVA[2] DEUTSCH J.SE[3]

1 HTTP://EN.WIKIBOOKS.ORG/WIKI/GNU%20FREE%20DOCUMENTATION%20LICENSE
2 HTTP://DE.WIKIBOOKS.ORG/WIKI/JAVA
3 HTTP://DE.WIKIBOOKS.ORG/WIKI/JAVA%20STANDARD

35 Authors

Edits	User
1	AdRiley[1]
24	Adrignola[2]
2	Alainr345[3]
1	Albmont[4]
1	Alexander.orlov[5]
2	Antidrugue[6]
1	Aphonik[7]
1	Arsenalfan[8]
69	Arunreginald[9]
7	Ashmailit[10]
2	Az1568[11]
1	Beno1000[12]
1	Ceclauson[13]
4	Colfulus[14]
1	Colindaven[15]
1	Cspurrier[16]
1	Dallas1278[17]
3	Dan Polansky[18]
2	Darklama[19]
5	Darkxxxxillusion[20]
3	DavidCary[21]

1 http://en.wikibooks.org/w/index.php?title=User:AdRiley
2 http://en.wikibooks.org/w/index.php?title=User:Adrignola
3 http://en.wikibooks.org/w/index.php?title=User:Alainr345
4 http://en.wikibooks.org/w/index.php?title=User:Albmont
5 http://en.wikibooks.org/w/index.php?title=User:Alexander.orlov
6 http://en.wikibooks.org/w/index.php?title=User:Antidrugue
7 http://en.wikibooks.org/w/index.php?title=User:Aphonik
8 http://en.wikibooks.org/w/index.php?title=User:Arsenalfan
9 http://en.wikibooks.org/w/index.php?title=User:Arunreginald
10 http://en.wikibooks.org/w/index.php?title=User:Ashmailit
11 http://en.wikibooks.org/w/index.php?title=User:Az1568
12 http://en.wikibooks.org/w/index.php?title=User:Beno1000
13 http://en.wikibooks.org/w/index.php?title=User:Ceclauson
14 http://en.wikibooks.org/w/index.php?title=User:Colfulus
15 http://en.wikibooks.org/w/index.php?title=User:Colindaven
16 http://en.wikibooks.org/w/index.php?title=User:Cspurrier
17 http://en.wikibooks.org/w/index.php?title=User:Dallas1278
18 http://en.wikibooks.org/w/index.php?title=User:Dan_Polansky
19 http://en.wikibooks.org/w/index.php?title=User:Darklama
20 http://en.wikibooks.org/w/index.php?title=User:Darkxxxxillusion
21 http://en.wikibooks.org/w/index.php?title=User:DavidCary

1 Devourer09[22]
1 Dickdickdick[23]
5 Dirk Hünniger[24]
3 Dirk gently[25]
79 Djb[26]
1 Dmonego[27]
1 Drawde83[28]
646 Ervinn[29]
3 Exabyte[30]
3 Explanator[31]
3 Felipeochoa0918[32]
3 Fishpi[33]
1 Fkereki[34]
1 Flatipac[35]
2 Fredmaranhao[36]
4 Ftiercel[37]
2 GoLLoXp[38]
2 Grokus[39]
1 Grunny[40]
1 Guanaco[41]
5 Guangpu.huang[42]
1 HMPerson1[43]
1 Hagindaz[44]
1 Hermione1980[45]
1 Hrođulf[46]

22 http://en.wikibooks.org/w/index.php?title=User:Devourer09
23 http://en.wikibooks.org/w/index.php?title=User:Dickdickdick
24 http://en.wikibooks.org/w/index.php?title=User:Dirk_H%C3%BCnniger
25 http://en.wikibooks.org/w/index.php?title=User:Dirk_gently
26 http://en.wikibooks.org/w/index.php?title=User:Djb
27 http://en.wikibooks.org/w/index.php?title=User:Dmonego
28 http://en.wikibooks.org/w/index.php?title=User:Drawde83
29 http://en.wikibooks.org/w/index.php?title=User:Ervinn
30 http://en.wikibooks.org/w/index.php?title=User:Exabyte
31 http://en.wikibooks.org/w/index.php?title=User:Explanator
32 http://en.wikibooks.org/w/index.php?title=User:Felipeochoa0918
33 http://en.wikibooks.org/w/index.php?title=User:Fishpi
34 http://en.wikibooks.org/w/index.php?title=User:Fkereki
35 http://en.wikibooks.org/w/index.php?title=User:Flatipac
36 http://en.wikibooks.org/w/index.php?title=User:Fredmaranhao
37 http://en.wikibooks.org/w/index.php?title=User:Ftiercel
38 http://en.wikibooks.org/w/index.php?title=User:GoLLoXp
39 http://en.wikibooks.org/w/index.php?title=User:Grokus
40 http://en.wikibooks.org/w/index.php?title=User:Grunny
41 http://en.wikibooks.org/w/index.php?title=User:Guanaco
42 http://en.wikibooks.org/w/index.php?title=User:Guangpu.huang
43 http://en.wikibooks.org/w/index.php?title=User:HMPerson1
44 http://en.wikibooks.org/w/index.php?title=User:Hagindaz
45 http://en.wikibooks.org/w/index.php?title=User:Hermione1980
46 http://en.wikibooks.org/w/index.php?title=User:Hro%C3%B0ulf

2 IMacWin95[47]
1 Itsborin[48]
1 J36miles[49]
33 Jguk[50]
1 Jimmyatic[51]
1 Jk33[52]
7 Jomegat[53]
2 Jonathan Webley[54]
1 Jpkotta[55]
1 Kejia[56]
1 Kenj0418[57]
1 Kriak[58]
1 Lcawte[59]
6 Malfist[60]
2 MatrixFrog[61]
11 Mattylaws[62]
1 MaxBowsher[63]
3 MetaBohemian[64]
1 Mhayes[65]
42 Mikm[66]
1 Ms2ger[67]
1 Mstenta[68]
2 Omzig89[69]
4 Panic2k4[70]
1 Pengo[71]

47 http://en.wikibooks.org/w/index.php?title=User:IMacWin95
48 http://en.wikibooks.org/w/index.php?title=User:Itsborin
49 http://en.wikibooks.org/w/index.php?title=User:J36miles
50 http://en.wikibooks.org/w/index.php?title=User:Jguk
51 http://en.wikibooks.org/w/index.php?title=User:Jimmyatic
52 http://en.wikibooks.org/w/index.php?title=User:Jk33
53 http://en.wikibooks.org/w/index.php?title=User:Jomegat
54 http://en.wikibooks.org/w/index.php?title=User:Jonathan_Webley
55 http://en.wikibooks.org/w/index.php?title=User:Jpkotta
56 http://en.wikibooks.org/w/index.php?title=User:Kejia
57 http://en.wikibooks.org/w/index.php?title=User:Kenj0418
58 http://en.wikibooks.org/w/index.php?title=User:Kriak
59 http://en.wikibooks.org/w/index.php?title=User:Lcawte
60 http://en.wikibooks.org/w/index.php?title=User:Malfist
61 http://en.wikibooks.org/w/index.php?title=User:MatrixFrog
62 http://en.wikibooks.org/w/index.php?title=User:Mattylaws
63 http://en.wikibooks.org/w/index.php?title=User:MaxBowsher
64 http://en.wikibooks.org/w/index.php?title=User:MetaBohemian
65 http://en.wikibooks.org/w/index.php?title=User:Mhayes
66 http://en.wikibooks.org/w/index.php?title=User:Mikm
67 http://en.wikibooks.org/w/index.php?title=User:Ms2ger
68 http://en.wikibooks.org/w/index.php?title=User:Mstenta
69 http://en.wikibooks.org/w/index.php?title=User:Omzig89
70 http://en.wikibooks.org/w/index.php?title=User:Panic2k4
71 http://en.wikibooks.org/w/index.php?title=User:Pengo

2 Pitel[72]
12 Prodoc[73]
3 QuiteUnusual[74]
1 Ralphcook[75]
7 Rappo[76]
1 Ravichandar84[77]
3 Recent Runes[78]
10 Ricky clarkson[79]
3 SBJohnny[80]
5 Samwilson[81]
1 Seanja[82]
1 Shahidsidd[83]
65 Sigma 7[84]
8 Spongebob88[85]
16 Spoon![86]
1 Stephanvaningen[87]
4 Sundar22in[88]
3 Sunnychan[89]
7 Superfly Jon[90]
1 Swift[91]
2 Tanminivan[92]
1 Tarose.trevor[93]
1 TheDaveRoss[94]
2 Thephilwells[95]
2 UNV[96]

72 http://en.wikibooks.org/w/index.php?title=User:Pitel
73 http://en.wikibooks.org/w/index.php?title=User:Prodoc
74 http://en.wikibooks.org/w/index.php?title=User:QuiteUnusual
75 http://en.wikibooks.org/w/index.php?title=User:Ralphcook
76 http://en.wikibooks.org/w/index.php?title=User:Rappo
77 http://en.wikibooks.org/w/index.php?title=User:Ravichandar84
78 http://en.wikibooks.org/w/index.php?title=User:Recent_Runes
79 http://en.wikibooks.org/w/index.php?title=User:Ricky_clarkson
80 http://en.wikibooks.org/w/index.php?title=User:SBJohnny
81 http://en.wikibooks.org/w/index.php?title=User:Samwilson
82 http://en.wikibooks.org/w/index.php?title=User:Seanja
83 http://en.wikibooks.org/w/index.php?title=User:Shahidsidd
84 http://en.wikibooks.org/w/index.php?title=User:Sigma_7
85 http://en.wikibooks.org/w/index.php?title=User:Spongebob88
86 http://en.wikibooks.org/w/index.php?title=User:Spoon%21
87 http://en.wikibooks.org/w/index.php?title=User:Stephanvaningen
88 http://en.wikibooks.org/w/index.php?title=User:Sundar22in
89 http://en.wikibooks.org/w/index.php?title=User:Sunnychan
90 http://en.wikibooks.org/w/index.php?title=User:Superfly_Jon
91 http://en.wikibooks.org/w/index.php?title=User:Swift
92 http://en.wikibooks.org/w/index.php?title=User:Tanminivan
93 http://en.wikibooks.org/w/index.php?title=User:Tarose.trevor
94 http://en.wikibooks.org/w/index.php?title=User:TheDaveRoss
95 http://en.wikibooks.org/w/index.php?title=User:Thephilwells
96 http://en.wikibooks.org/w/index.php?title=User:UNV

1 Vinay h[97]
1 Webaware[98]
1 Whiteknight[99]
6 WikiWizard[100]
2 Wikialt[101]
1 Willem Souwer[102]
1 WiseEyes[103]
1 Wur-dene[104]
2 Wutzofant[105]
1 YMS[106]
7 Yuuki Mayuki[107]
9 ZeroOne[108]
1 [109]

97 http://en.wikibooks.org/w/index.php?title=User:Vinay_h
98 http://en.wikibooks.org/w/index.php?title=User:Webaware
99 http://en.wikibooks.org/w/index.php?title=User:Whiteknight
100 http://en.wikibooks.org/w/index.php?title=User:WikiWizard
101 http://en.wikibooks.org/w/index.php?title=User:Wikialt
102 http://en.wikibooks.org/w/index.php?title=User:Willem_Souwer
103 http://en.wikibooks.org/w/index.php?title=User:WiseEyes
104 http://en.wikibooks.org/w/index.php?title=User:Wur-dene
105 http://en.wikibooks.org/w/index.php?title=User:Wutzofant
106 http://en.wikibooks.org/w/index.php?title=User:YMS
107 http://en.wikibooks.org/w/index.php?title=User:Yuuki_Mayuki
108 http://en.wikibooks.org/w/index.php?title=User:ZeroOne
109 http://en.wikibooks.org/w/index.php?title=User:%E7%A0%B2%E7%81%AB_%E4%B8%87%E7%89%A9%E3%81%AE%E9%9C%8A%E9%95%B7

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.
- EPL: Eclipse Public License. http://www.eclipse.org/org/documents/epl-v10.php

1	PETER CAMPBELL[110]	GFDL
2	USER:ARUNREGINALD[111]	GFDL
3	USER:ARUNREGINALD[112]	
4	USER:ARUNREGINALD[113]	
5		
6		
7		
8		
9		
10		
11	ARUN REGINALD[114]	GFDL
12	ARUN REGINALD[115]	GFDL
13	Original version created by B:USER:ERVINN[116], SVG version created by MYSELF[117]	cc-by-sa-2.5
14	Original version created by B:USER:ERVINN[118], SVG version created by MYSELF[119]	cc-by-sa-2.5
15		
16		
17		
18		
19		
20		
21		

110 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3APETER%20CAMPBELL
111 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AARUNREGINALD
112 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AARUNREGINALD
113 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AARUNREGINALD
114 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AARUNREGINALD
115 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AARUNREGINALD
116 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AERVINN
117 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMIKM
118 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AERVINN
119 HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3AMIKM