
Java Persistence/Print version

Java



Persistence

A book relating to the Java programming language.

Contents

1. Preface
 2. What is Java persistence?
 1. What is Java?
 2. What is a database?
 3. What is JPA?
 4. What is new in JPA 2.0?
 5. Other Persistence Specs
 6. Why use JPA or ORM?
 3. Persistence Products, Which to Use?
 1. EclipseLink (Eclipse)
 2. TopLink (Oracle)
 3. Hibernate (RedHat)
 4. TopLink Essentials (Glassfish)
 5. Kodo (Oracle)
 6. Open JPA (Apache)
 7. Ebean (SourceForge)
 4. Mapping, Round Pegs into Square Holes
 1. Tables
 2. Identity, Primary Keys and Sequencing
 3. Inheritance
 4. Embeddables (Aggregates, Composite or Component Objects)
 5. Locking and Concurrency
 6. Basic Attributes
 7. Relationships
 1. OneToOne
 2. ManyToOne
 3. OneToMany
 4. ManyToMany
-

-
5. Embedded
 8. Advanced Mappings
 1. ElementCollection (Embeddable Collections, Basic Collections)
 2. Variable Relationships
 9. Advanced Topics
 1. Views
 2. #Stored Procedures/
 3. #Structured Object-Relational Data Types/
 4. #XML Data Types/
 5. #Filters/
 6. #History/
 7. #Logical Deletes/
 8. #Auditing/
 9. #Replication/
 5. Runtime, Doing the Hokey Pokey (EntityManager)
 1. Querying
 1. JPQL
 2. Persisting (Inserting, Updating, Merging)
 3. Transactions
 4. Caching
 5. EJB
 6. Security (User Authentication, Proxy Connections, VPD)
 7. #Servlets and JSPs/
 8. #Spring/
 9. #WebServices/
 6. #Packaging and Deploying/
 1. #Java EE/
 1. Oracle Weblogic
 2. IBM Websphere
 3. Redhat JBoss
 2. #Spring/
 3. #Tomcat/
 7. #Clustering/
 8. Databases
 1. Oracle
 2. PostgreSQL
 3. MySQL
 4. DB2
 5. SQL Server
 9. Debugging
 10. Performance
 11. Tools
 1. Eclipse JPA (Dali)
 2. TopLink Mapping Workbench
 12. Testing
-

Preface



What is this book about?

This book is meant to cover Java persistence, that is, storing stuff in the Java programming language to a persistent storage medium. Specifically using the Java Persistence API (JPA) to store Java objects to relational databases, but I would like it to have a somewhat wider scope than just JPA and concentrate more on general persistence patterns and use cases, after all JPA is just the newest of many failed Java persistence standards, this book should be able to evolve beyond JPA when it is replaced by the next persistence standard. I do not want this to be just a regurgitation of the JPA Spec, nor a User Manual to using one of the JPA products, but more focused on real-world use cases of users and applications trying to make use of JPA (or other Java persistence solution) and the patterns they evolved and pitfalls they made.



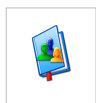
Intended Audience

This book is intended to be useful for or to anyone learning to, or developing Java applications that require persisting data to a database. It is mainly intended for Java developers intending to persist Java objects through the Java Persistence API (JPA) standard to a relational database. Please don't just read this book, if you're learning or developing with JPA please contribute your experiences to this book.



Style

This book is meant to be written in a casual manner. The goal is avoid sounding dry, overly technical, or impersonal. The book should sound casual, like a co-worker explaining to you how to use something, or a fellow consultant relating their latest engagement to another. Please refrain for being overly critical of any product, ranting about bugs, or marketing your own product or services.



Authors

Everyone is encouraged to participate in the ongoing development of this book. You do not need to be a Java persistence superstar to contribute, many times the best advice/information for other users comes from first time users who have not yet been conditioned to think something that may be confusing is obvious.

List of authors: (please contribute and sign your name)

James Sutherland : Currently working on Oracle TopLink and Eclipse EclipseLink, over 12 years of experience in object persistence and ORM.

Doug Clarke : Oracle TopLink and Eclipse EclipseLink, over 10 years of experience in the object persistence industry.

What is Java persistence?

Java persistence could be defined as storing anything to any persistence store in the Java programming language. Obviously this would be too broad of a concept for a single book, so this book is more focused on storing Java objects to relational databases. In particular using the Java Persistence API (JPA).

There are many ways to persist data in Java, to name a few, JDBC, serialization, file IO, JCA, object databases, or XML databases. However the majority of data in general is persisted in databases, specifically relational databases. Most things that you do on a computer or web site that involve storing data, involve accessing a relational database. Relational databases are the standard persistence store for most corporations from banking to industrial.

There are many things that can be stored in databases from Java. Java data includes strings, numbers, dates and byte arrays, images, XML and Java objects. Many Java applications use Java objects to model their application data, because Java is an Object Oriented language, storing Java objects is a natural and common approach to persisting data from Java.

There are many ways to access a relational database from Java, JPA is just the latest of many different specifications, but seems to be the direction that Java persistence is heading.

What is Java?

Java is an object oriented programming language first released by Sun Microsystems in 1995. It blended concepts from existing languages such as C++ and Smalltalk into a new programming language. It achieved its success over the many rival languages of the day because it was associated with this newish thing called the "Internet" in allowing Java applets to be embedded in web pages and run using Netscape. Its other main reason for success was unlike many of its competitors it was open, "free", and not embedded with an integrated development environment (IDE). Java also included the source code to its class library. This enabled Java to be adopted by many different companies producing their own Java development environments but sharing the same language, this open model fostered the growth in the Java language and continues today with the open sourcing of Java.

Java quickly moved from allowing developers to build dinky applets to being the standard server-side language running much of the Internet today. The Enterprise Edition (JEE) of Java was defined to provide an open model for server application to be written and portable across any compliant JEE platform provider. The JEE standard is basically a basket of other Java specifications brought together under one umbrella and has major providers including IBM WebSphere, RedHat JBoss, Sun Glassfish, BEA WebLogic, Oracle AS and many others.

Google Trends

- Programming Languages ^{[1][2]}
- JEE Servers ^{[3][4]}

[1] <http://www.google.com/trends?q=c%23%2C+php%2C+java%2C+c%2B%2B%2C+perl&ctab=0&geo=all&date=all&sort=0>

[2] Unfortunately hard to separate Java island from Java language.

[3] <http://www.google.com/trends?q=websphere%2C+weblogic%2C+jboss%2C+glassfish%2C+geronimo+apache&ctab=0&geo=all&date=all&sort=0>

[4] Could not include Oracle because of Oracle database hits.

See also

- Java Programming

What is a database?

A database is a program that stores data. There are many types of databases, flat-file, hierarchical, relational, object-relational, object-oriented, xml, and others. The original databases were mainly proprietary and non-standardized.

Relational database were the first databases to achieve great success and standardization, relational database are characterize by the SQL (structured query language) standard to query and modify the database, their client/server architecture, and relational table storage structure. Relational databases achieve great success because their standardization allowed many different vendors such as Oracle, IBM, and Sybase to produce interoperable products giving users the flexibility to switch their vendor and avoid vendor lock-in to a proprietary solution. Their client/server architecture allows the client programming language to be decoupled from the server, allowing the database server to support interface APIs into multiple different programming languages and clients.

Although relational database are relatively old technology they still dominate the industry. There have been many attempts to replace the relational model, first with object-oriented databases, then with object-relational databases, and finally with xml database, but none of the new database models achieved much success and relational database remain the overwhelming dominate database model.

The main relational databases used today are, Oracle, MySQL (Sun), PostgreSQL, DB2 (IBM), SQL Server (Microsoft).

- Google trend for databases (<http://www.google.com/trends?q=oracle,+sybase,+sql+server,+mysql,+db2&ctab=0&geo=all&date=all&sort=0>)

What is JPA?

The Java Persistence Architecture API (JPA) is a Java specification for accessing, persisting and managing data between Java objects / classes and the relational database. JPA was defined as part of the EJB 3.0 specification as a replacement to the EJB 2 CMP Entity Beans specification. It is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.

JPA is just a specification, it is not a product, and cannot perform persistence, or anything by itself. JPA is just a set of interfaces, and requires an implementation. There are open

source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use. JPA also requires a database to persist to.

JPA allows POJO (Plain Old Java Objects) to be easily persisted without requiring the classes to implement any interfaces or methods as the EJB 2 CMP specification required. JPA allows an object's object-relational mappings to be defined through standard annotations or XML defining how the Java class maps to a relational database table. JPA also defines a runtime EntityManager API for processing queries and transaction on the objects against the database. JPA defines an object-level query language JPQL to allow querying of the objects from the database.

JPA is the latest of several Java persistence specifications. The first was the OMG persistence service Java binding, which was never very successful and I'm not sure of any commercial products supporting it. Next came EJB 1.0 CMP Entity Beans, which was very successful in being adopted by the big Java EE providers (BEA, IBM), but there was a backlash against the spec by some users who thought the spec requirements on the Entity Beans overly complex and overhead and performance poor. EJB 2.0 CMP tried to reduce some of the complexity of Entity Beans through introducing local interfaces, but the majority of the complexity remained. EJB 2.0 also lacked portability, in that the deployment descriptors defining the object-relational mapping were not specified and all proprietary.

This backlash in part led to the creation of another Java persistence specification JDO (Java Data Objects). JDO obtained somewhat of a "cult" following of several independent vendors such as Kodo JDO, and several open source implementation, but never had much success with the big Java EE vendors.

Despite the two competing Java persistence standards of EJB CMP and JDO, the majority of users continued to prefer proprietary api solutions, mainly TopLink (which had been around for some time and had its own POJO API) and Hibernate (which was a relatively new open source product that also had its own POJO API and was quickly becoming the open source industry standard). The TopLink product formerly owned by WebGain was also acquired by Oracle, increasing it's influence on the Java EE community.

The EJB CMP backlash was only part of a backlash against all of Java EE which was seen as too complex in general and prompted such products as the Spring container. This led the EJB 3.0 specification to have a main goal of reducing the complexity, which lead the spec committee down the path of JPA. JPA was meant to unify the EJB 2 CMP, JDO, Hibernate, and TopLink API's and products, and seems to have been very successful in doing so.

Currently most of the persistence vendors have released implementations of JPA confirming its' adoption by the industry and users. These include Hibernate (acquired by JBoss, acquired by Red Hat), TopLink (acquired by Oracle), and Kodo JDO (acquired by BEA, acquired by Oracle). Other products that have added support for JPA include Cocobase (owned by Thought Inc.), and JPOX.

- EJB JPA Spec (<http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>)
- JPA ORM XML Schema (http://java.sun.com/xml/ns/persistence/orm_1_0.xsd)
- JPA Persistence XML Schema (http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd)
- JPA JavaDoc (<https://java.sun.com/javase/5/docs/api/javax/persistence/package-summary.html>)
- JPQL BNF

What is new in JPA 2.0?

The JPA 2.0 specification is under development. It plans to add several enhancements to the JPA 1.0 specification including:

- Extended Map support - Support for maintaining a key column for a `Basic`, `Embeddable`, or `Entity` key value in any collection relationship using a `Map`.
- Derived Identifiers
- Nested embedding
- New collection mappings - Support for collections of `Basic` or `Embeddable` types.
- Pessimistic Locking
- EntityManager API updates
- Cache APIs
- Standard Properties
- Ordered List mappings - Support for maintaining an index column in any collection relationship using a `List`.
- Orphan removal
- Metadata
- Query Criteria API
- JPQL enhancements

Resources

- JPA 2.0 Spec (<http://jcp.org/en/jsr/detail?id=317>)
- Eclipse EclipseLink to be JPA 2.0 Reference Implementation (http://www.eclipse.org/org/press-release/20080317_EclipseLink.php)
- JPA 2.0 Reference Implementation Development (on EclipseLink) (<http://wiki.eclipse.org/EclipseLink/Development/JPA>)

Other Persistence Specs

There are many specifications related to persistence in Java. The following table summaries each specification.^[1]

Spec	Version	Year of Last Release
JPA (Java Persistence API)	1.0 (EJB 3.0)	2006
JDO (Java Data Objects)	2.0	2006
SDO (Service Data Objects)	2.1	2006
JDBC (Java DataBase Connectivity)	4.0	2006
EJB CMP (Enterprise Java Beans, Container Managed Persistence)	2.1 (EJB)	2003
JCA (Java EE Connector Architecture)	1.5	2003

[1] Last updated 2008-04

Why use JPA or ORM?

This is an intriguing question. There are many reasons to use an ORM framework or persistence product, and many reasons to use JPA in particular.

Reasons for ORM

- Leverages large persistence library to avoid developing solutions to problems that others have already solved.
- Avoids low level JDBC and SQL code.
- Leverages object oriented programming and object model usage.
- Provides database and schema independence.
- Most ORM products are free and open source.
- Many enterprise corporations provide support and services for ORM products.
- Provides high end performance features such as caching and sophisticated database and query optimizations.

Reasons for JPA

- It is a standard and part of EJB3 and JEE.
- Many free and open source products with enterprise level support.
- Portability across application servers and persistence products (avoids vendor lock-in).
- A usable and functional specification.
- Supports both JEE and JSE.

ORM can be a hot topic for some people, and there are many ORM camps. There are those that endorse a particular standard or product. There are those that don't believe in ORM or even objects in general and prefer JDBC. There are those that still believe that object databases are the way to go. Personally I would recommend you use whatever technology you are most comfortable with, but if you have never used ORM or JPA, perhaps give it a try and see if you like it. The below list provides several discussions on why or why not to use JPA and ORM.

Discussions on JPA and ORM Usage

- Why do we need anything other than JDBC? (Java Ranch) (http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=78&t=003738)
- Why JPA? (BEA) (http://edocs.bea.com/kodo/docs41/full/html/ejb3_overview_why.html)
- JPA Explained (The Server Side) (http://www.theserverside.com/news/thread.tss?thread_id=44526)
- JPA vs JDO (The Server Side) (http://www.theserverside.com/news/thread.tss?thread_id=40965)

Persistence Products

There are many persistence products to choose from. Most persistence products now support a JPA interface, although there still are some exceptions. Which product you use depends on your preference, but most people would recommend you use the JPA standard whichever product you choose. This gives you the flexibility to switch persistence providers, or port your application to another server platform which may use a different persistence

provider.

Determining which persistence product to use involves many criteria. Valid things to consider include:

- Which persistence product does your server platform support and integrate with?
- What is the cost of the product, is it free and open source, can you purchase enterprise level support and services?
- Do you have an existing relationship with the company producing the product?
- Is the product active and does it have a large user base?
- How does the product perform and scale?
- Does the product integrate with your database platform?
- Does the product have active and open forums, do questions receive useful responses?
- Is the product JPA compliant, what functionality does the product offer beyond the JPA specification?

Existing Persistence Products

The following table summaries existing persistence products.^[1]

Product	JPA 1.0	JDO 2.0	CMP 2.1	Version	Year of Last Release	Open Source	Application Servers ^[2]	Forum Posts per Month ^[3]
Hibernate (Red Hat)	Yes			3.3.1	2008	Yes	JBoss	650 (http://forum.hibernate.org/viewforum.php?f=1)
EclipseLink (Eclipse)	Yes			1.0.2	2008	Yes	OracleAS (11g), Oracle Weblogic (10.3), Glassfish (v3)	81 (http://www.nabble.com/EclipseLink---Users-f26658.html)
TopLink (Oracle)	Yes		Yes	11g	2008		OracleAS (11g), Oracle Weblogic (10.3)	80 (http://forums.oracle.com/forums/forum.jspa?forumID=48)
OpenJPA (Apache)	Yes			1.0.3	2008	Yes	Geronimo	60 (http://www.nabble.com/OpenJPA-Users-f23252.html)
JPOX (http://www.jpox.org/) (SourceForge)	Yes	Yes		1.2.3	2008	Yes		21 (http://www.jpox.org/servlet/forum/index)
TopLink Essentials (java.net)	Yes			2.0	2007	Yes	Glassfish (v2), SunAS (9), OracleAS (10.1.3)	13 (http://www.nabble.com/java.net---glassfish-persistence-f13455.html)
Kodo (Oracle)	Yes	Yes		4.1	2007		Oracle WebLogic (10.3)	0 (http://forums.bea.com/bea/forum.jspa?forumID=500000029)

[1] Last updated 2008-10

[2] Application server that includes the product as their JPA implementation

[3] Number of user forum posts for 2008-10. Note this is only an estimate of community size and may be misleading due to several factors including users of open source products are more likely to post to forums, where as users of commercial products are more likely to use support. The estimate is also confused by the fact that some users post questions to generic EJB, JPA or ORM forums.

EclipseLink

EclipseLink is the open source Eclipse Persistence Services Project from the Eclipse Foundation. The product provides an extensible framework that allows Java developers to interact with various data services, including databases, XML, and Enterprise Information Systems (EIS). EclipseLink supports a number of persistence standards including the Java Persistence API (JPA), Java API for XML Binding (JAXB), Java Connector Architecture (JCA), and Service Data Objects (SDO).

EclipseLink is based on the TopLink product, which Oracle contributed the source code from to create the EclipseLink project. The original contribution was from TopLink's 11g code base, and the entire code-base/feature set was contributed, with only EJB 2 CMP and some minor Oracle AS specific integration removed. This differs from the TopLink Essentials Glassfish contribution, which did not include some key enterprise features. The package names were changed and some of the code was moved around.

The TopLink Mapping Workbench UI has also been contributed to the project.

EclipseLink is the intended path forward for persistence for Oracle and TopLink. It is intended that the next major release of Oracle TopLink will include EclipseLink as well as the next major release of Oracle AS.

EclipseLink supports usage in an OSGi environment.

EclipseLink was announced to be the JPA 2.0 reference implementation, and announced to be the JPA provider for Glassfish v3.

- EclipseLink Home (<http://www.eclipse.org/eclipselink/>)
- EclipseLink Newsgroup (<http://www.eclipse.org/newsportal/thread.php?group=eclipse.technology.eclipselink>)
- EclipseLink Wiki (<http://wiki.eclipse.org/EclipseLink>)

TopLink

TopLink is one of the leading Java persistence products and JPA implementations. TopLink is produced by Oracle and part of Oracle's OracleAS, WebLogic, and OC4J servers.

As of TopLink 11g, TopLink bundles the open source project EclipseLink for most of its functionality.

The TopLink 11g release supports the JPA 1.0 specification. TopLink 10.1.3 also supports EJB CMP and is the persistence provider for OracleAS OC4J 10.1.3 for both JPA and EJB CMP. TopLink provides advanced object-relational mapping functionality beyond the JPA specification, as well as providing persistence for object-relational data-types, and Enterprise Information Systems (EIS/mainframes). TopLink includes sophisticated object caching and performance features. TopLink provides a Grid extension that integrate with Oracle Coherence. TopLink provides object-XML mapping support and provides a JAXB implementation and web service integration. TopLink provides a Service Data Object (SDO) implementation.

TopLink provides a rich user interface through the TopLink Mapping Workbench. The Mapping Workbench allows for graphical mapping of an object model to a data model, as allows for generation of a data model from an object model, and generation of an object model from a data model, and auto-mapping of an existing object and data model. The TopLink Mapping Workbench functionality is also integrated with Oracle's JDeveloper IDE.

TopLink contributed part of its source code to become the JPA 1.0 reference implementation under the Sun java.net Glassfish project. This open-source product is called TopLink Essentials, and despite a different package name (oracle.toplink.essentials) it is basically a branch of the source code of the TopLink product with some advanced functionality stripped out.

TopLink contributed practically its entire source code to the Eclipse Foundation EclipseLink product. This is an open source product currently in incubation that represents the path forward for TopLink. The package name is different (org.eclipse.persistence) but the source code it basically a branch of the TopLink 11g release. Oracle also contributed its Mapping Workbench source code to the project. The TopLink Mapping Workbench developers also were major contributors to the Eclipse Dali project for JPA support.

TopLink was first developed in Smalltalk and ported to Java in the 90's, and has over 15 years worth of object persistence solutions. TopLink originally provided a proprietary POJO persistence API, when EJB was first released TopLink provided one of the most popular EJB CMP implementations, although it continued to recommend its POJO solution. TopLink also provided a JDO 1.0 implementation for a few releases, but this was eventually deprecated and removed once the JPA specification had been formed. Oracle and TopLink have been involved in each of the EJB, JDO and EJB3/JPA expert groups, and Oracle was the co-lead for the EJB3/JPA specification.

- Oracle TopLink Home (<http://www.oracle.com/technology/products/ias/toplink/index.html>)
- Oracle TopLink Forum (<http://forums.oracle.com/forums/forum.jspa?forumID=48>)
- Oracle TopLink Wiki (<http://wiki.oracle.com/page/TopLink>)

TopLink Resources

- TopLink Automatic Schema Generation Options (<http://docs.sun.com/app/docs/doc/819-3672/gbwmk?a=view>)

Hibernate

Hibernate was an open source project developed by a team of Java software developers around the world led by Gavin King. JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers and worked with them in supporting Hibernate.

The current version of Hibernate is Version 3.x. Hibernate provides both a proprietary POJO API, and JPA support.

TopLink Essentials

TopLink Essentials is an open source project from the Sun java.net Glassfish community. It is the EJB3 JPA 1.0 reference implementation, and is the JPA provider for the Sun Glassfish v1 application server.

TopLink Essentials was based on the TopLink product, which Oracle contributed some of the source code from to create the TopLink Essentials project. The original contribution was from TopLink's 10.1.3 code base, only some of the TopLink product source code was contributed, which did not include some key enterprise features. The package names were changed and some of the code was moved around.

TopLink Essentials has been somewhat replaced by the EclipseLink project. EclipseLink will be the JPA 2.0 reference implementation and be part of Sun Glassfish v3.

- TopLink Essentials Home (<https://glassfish.dev.java.net/javaee5/persistence/index.html>)
- TopLink Essentials Forum (<http://www.nabble.com/java.net--glassfish-persistence-f13455.html>)
- TopLink Essentials Wiki (<http://wiki.glassfish.java.net/Wiki.jsp?page=TopLinkEssentials>)

Kodo

Kodo (<http://www.bea.com/kodo/>), was originally developed as an Java Data Objects (JDO) implementation, by SolarMetric. BEA Systems acquired SolarMetric in 2005, where Kodo was expanded to be an implementation of both the JDO and JPA specifications. In 2006, BEA donated a large part of the Kodo source code to the Apache Software Foundation under the name OpenJPA. BEA (and Kodo) were acquired by Oracle.

Open JPA

OpenJPA is an Apache project for supporting the JPA specification. Its' source code was originally donated from (part of) BEA's Kodo product.

Ebean (<http://www.avaje.org>) is a Java ORM based on the JPA specification.

The goal of Ebean is to provide mapping compatible with JPA while providing a simpler API to use and learn.

- @EnumMapping
- @Formula

Mapping

The first thing that you need to do to persist something in Java is define how it is to be persisted. This is called the mapping process (details (http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch)). There have been many different solutions to the mapping process over the years, including some object-databases that didn't require you map anything, just lets you persist anything directly. Object-relational mapping tools that would generate an object model for a data model that included the mapping and persistence logic in it. ORM products that provided mapping tools to allow the mapping of

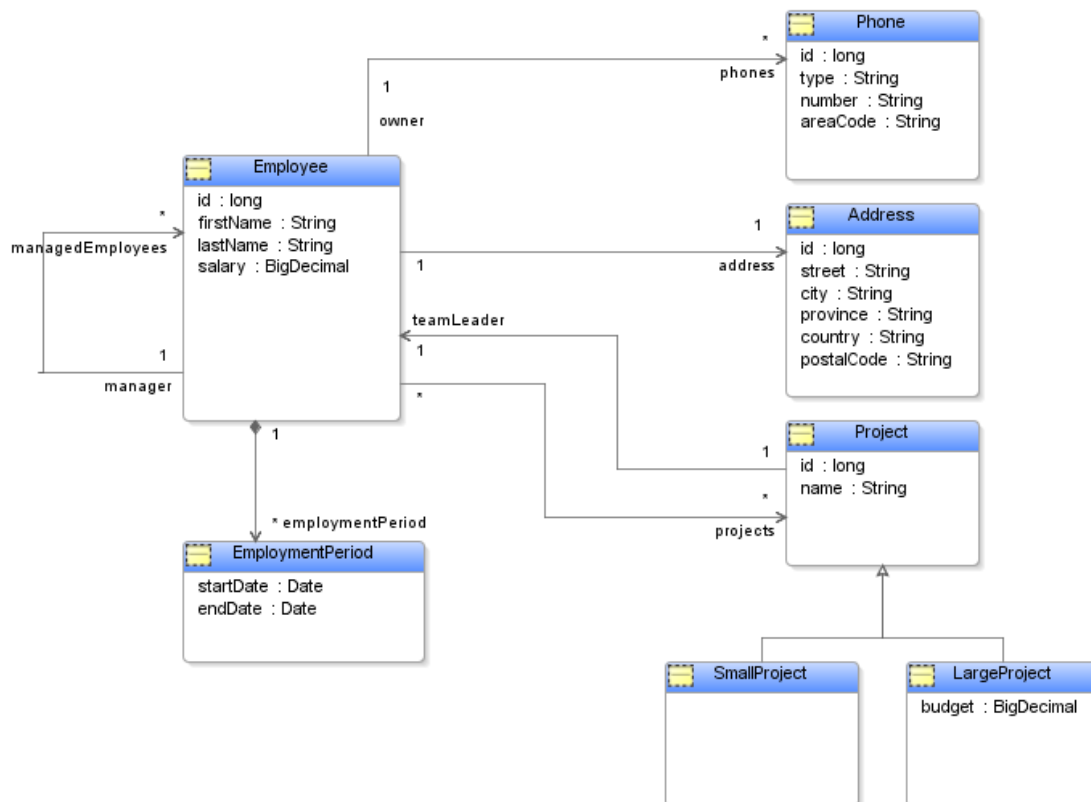
an existing object model to an existing data model and stored this mapping meta-data in flat files, database tables, XML and finally annotations.

In JPA mappings can either be stored through Java annotations, or in XML files. One significant aspect of JPA is that only the minimal amount of mapping is required. JPA implementations are required to provide defaults for almost all aspects of mapping and object.

The minimum requirement to mapping an object in JPA is to define which objects can be persisted. This is done through either marking the class with the `@Entity` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Entity.html>) annotation, or adding an `<entity>` tag for the class in the persistence unit's ORM XML file. Also the primary key, or unique identifier attribute(s) must be defined for the class. This is done through marking one of the class' fields or properties (get method) with the `@Id` annotation, or adding an `<id>` tag for the class' attribute in the ORM XML (http://java.sun.com/xml/ns/persistence/orm_1_0.xsd) file.

The JPA implementation will default all other mapping information, including defaulting the table name, column names for all defined fields or properties, cardinality and mapping of relationships, all SQL and persistence logic for accessing the objects. Most JPA implementations also provide the option of generating the database tables at runtime, so very little work is required by the developer to rapidly develop a persistent JPA application.

Example object model



Common Problems

My annotations are ignored

This typically occurs when you annotate both the fields and methods (properties) of the class. You must choose either field or property access, and be consistent. Also when annotating properties you must put the annotation on the get method, not the set method. Also ensure that you have not defined the same mappings in XML, which may be overriding the annotations. You may also have a classpath issue, such as having an old version of the class on the classpath.

Example of a persistent entity mappings in annotations

```
import javax.persistence.*;
...
@Entity
public class Employee {
    @Id
    private long id;
    private String firstName;
    private String lastName;
    private Address address;
    private List<Phone> phones;
    private Employee manager;
    private List<Employee> managedEmployees;
    ...
}
```

Example of a persistent entity mappings in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
orm_1_0.xsd">
    <description>The minimal mappings for a persistent entity in XML.</description>
    <entity name="Employee" class="org.acme.Employee" access="FIELD">
        <attributes>
            <id name="id"/>
        </attributes>
    </entity>
</entity-mappings>
```

Odd behavior

There are many reasons that odd behavior can occur with persistence. One common issue that can cause odd behavior is using property access and putting *side effects* in your get or set methods. For this reason it is generally recommended to use field access in mapping, i.e. putting your annotations on your variables not your get methods.

For example consider:

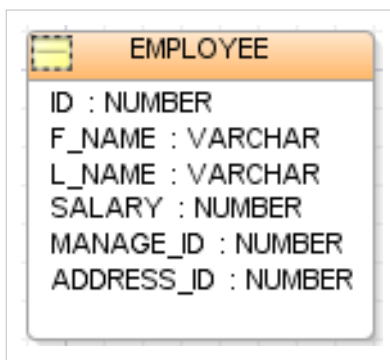
```
public void setPhones(List<Phone> phones) {
    for (Phone phone : phones) {
        phone.setOwner(this);
    }
    this.phones = phones;
}
```

This may look innocent, but these side effects can have unexpected consequences. For example if the relationship was lazy this would have the effect of always instantiating the collection when set from the database. It could also have consequences with certain JPA implementations for persisting, merging and other operations, causing duplicate inserts, missed updates, or a corrupt object model.

I have also seen simply incorrect property methods, such as a get method that always returns a new object, or a copy, or set methods that don't actually set the value.

In general if you are going to use property access, ensure your property methods are free of side effects. Perhaps even use different property methods than your application uses.

Tables



EMPLOYEE	
ID	: NUMBER
F_NAME	: VARCHAR
L_NAME	: VARCHAR
SALARY	: NUMBER
MANAGE_ID	: NUMBER
ADDRESS_ID	: NUMBER

A table is the basic persist structure of a relational database. A table contains a list of columns which define the table's structure, and a list of rows that define the table's data. Each column has a specific type and generally size. The standard set of relational types are limited to basic types including numeric, character, date-time, and binary (although most modern databases have additional types and typing systems). Tables can also have constraints that define the rules which restrict the row data, such as primary key, foreign key, and unique constraints. Tables also have other

artifacts such as indexes, partitions and triggers.

A typical mapping of a persist class will map the class to a single table. In JPA this is defined through the `@Table` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Table.html>) annotation or `<table>` XML element. If no table annotation is present, the JPA implementation will auto assign a table for the class, the JPA default table name is the name of the class as uppercase (minus the package). Each attribute of the class will be stored in a column in the table.

Example mapping annotations for an entity with a single table

```
...
@Entity
@Table(name="EMPLOYEE")
public class Employee {
    ...
}
```

Example mapping XML for an entity with a single table

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <table name="EMPLOYEE"/>
</entity/>
```

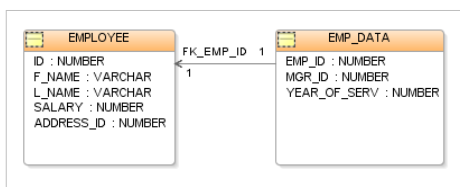
Advanced

Although in the ideal case each class would map to a single table, this is not always possible. Other scenarios include:

- Multiple tables : One class maps to 2 or multiple tables.
- Sharing tables : 2 or multiple classes are stored in the same table.
- Inheritance : A class is involved in inheritance and has an inherited and local table.
- Views : A class maps to a view.
- Stored procedures : A class maps to a set of stored procedures.
- Partitioning : Some instances of a class map to one table, and other instances to another table.
- Replication : A class's data is replicated to multiple tables.
- History : A class has historical data.

These are all advanced cases, some are handled by the JPA Spec and many are not. The following sections investigate each of these scenarios further and include what is supported by the JPA spec, what can be done to workaround the issue within the spec, and how to use some JPA implementations extensions to handle the scenario.

Multiple tables



Sometimes a class maps to multiple tables. This typically occurs on legacy or existing data models where the object model and data model do not match. It can also occur in inheritance when subclass data is stored in additional tables. Multiple tables may also be used for performance, partitioning or security reasons.

JPA allows multiple tables to be assigned to a single class. The `@SecondaryTable` (<https://java.sun.com/javase/5/docs/api/javax/persistence/SecondaryTable.html>) and `SecondaryTables` annotations or `<secondary-table>` elements can be used. By default the `@Id` column(s) are assumed to be in both tables, such that the secondary table's `@Id` column(s) are the primary key of the table and a foreign key to the first table. If the first table's `@Id` column(s) are not named the same the `@PrimaryKeyJoinColumn` (<https://java.sun.com/javase/5/docs/api/javax/persistence/PrimaryKeyJoinColumn.html>) or

`<primary-key-join-column>` can be used to define the foreign key join condition.

In a multiple table entity, each mapping must define which table the mapping's columns are from. This is done using the `table` attribute of the `@Column` or `@JoinColumn` annotations or XML elements. By default the primary table of the class is used, so you only need to set the table for secondary tables. For inheritance the default table is the primary table of the subclass being mapped.

Example mapping annotations for an entity with multiple tables

```
...
@Entity
@Table(name="EMPLOYEE")
@SecondaryTable(name="EMP_DATA")
@PrimaryKeyJoinColumn(name="EMP_ID", referencedColumnName="ID")
public class Employee {
    ...
    @Column(name="YEAR_OF_SERV", table="EMP_DATA")
    private int yearsOfService;

    @OneToOne
    @JoinColumn(name="MGR_ID", table="EMP_DATA",
referencedColumnName="EMP_ID")
    private Employee manager;
    ...
}
```

Example mapping XML for an entity with multiple tables

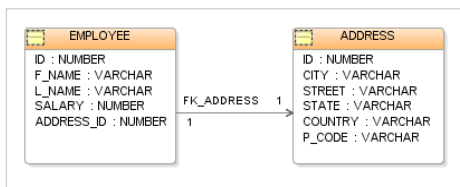
```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <table name="EMPLOYEE"/>
  <secondary-table name="EMP_DATA">
    <primary-key-join-column name="EMP_ID" referenced-column-name="ID"/>
  </secondary-table>
  <attributes>
    ...
    <basic name="yearsOfService">
      <column name="YEAR_OF_SERV" table="EMP_DATA"/>
    </basic>
    <one-to-one name="manager">
      <join-column name="MGR_ID" table="EMP_DATA" referenced-column-name="EMP_ID"/>
    </one-to-one>
  </attributes>
</entity>
```

With the `@PrimaryKeyJoinColumn` the name refers to the foreign key column in the secondary table and the `referencedColumnName` refers to the primary key column in the first table. If you have multiple secondary tables, they must always refer to the first table. When defining the table's schema typically you will define the join columns in the secondary table as the primary key of the table, and a foreign key to the first table. Depending how

you have defined your foreign key constraints, the order of the tables can be important, the order will typically match the order that the JPA implementation will insert into the tables, so ensure the table order matches your constraint dependencies.

For relationships to a class that has multiple tables the foreign key (join column) always maps to the primary table of the target. JPA does not allow having a foreign key map to a table other than the target object's primary table. Normally this is not an issue as foreign keys almost always map to the id/primary key of the primary table, but in some advanced scenarios this may be an issue. Some JPA products allow the column or join column to use the qualified name of the column (i.e. `@JoinColumn(referenceColumnName="EMP_DATA.EMP_NUM")`), to allow this type of relationship. Some JPA products may also support this through their own API, annotations or XML.

Multiple tables with foreign keys



Sometimes you may have a secondary table that is referenced through a foreign key from the primary table to the secondary table instead of a foreign key from the secondary table to the primary table. You may even have a foreign key between two of the secondary tables. Consider having an `EMPLOYEE` and `ADDRESS` table where `EMPLOYEE` refers to `ADDRESS` through an `ADDRESS_ID` foreign key, and (for some strange reason) you only want a single `Employee` class that has the data from both tables. The JPA spec does not cover this directly, so if you have this scenario the first thing to consider, if you have the flexibility, is to change your data model to stay within the confines of the spec. You could also change your object model to define a class for each table, in this case an `Employee` class and an `Address` class, which is typically the best solution. You should also check with your JPA implementation to see what extensions it supports in this area.

One way to solve the issue is simply to swap your primary and secondary tables. This will result in having the secondary table referencing the primary table's primary key and is within the spec. This however will have side-effects, one being that you now changed the primary key of your object from `EMP_ID` to `ADDRESS_ID`, and may have other mapping and querying implications. If you have more than 2 tables this also may not work.

Another option is to just use the foreign key column in the `@PrimaryKeyJoinColumn`, this will technically be backward, and perhaps not supported by the spec, but may work for some JPA implementations. However this will result in the table insert order not matching the foreign key constraints, so the constraints will need to be removed, or deferred.

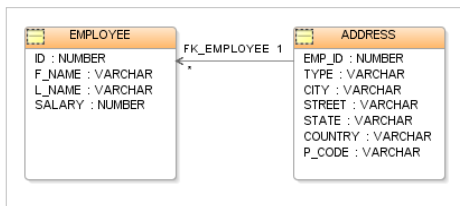
It is also possible to map the scenario through a database view. A view could be defined joining the two tables and the class could be mapped to the view instead of the tables. Views are read-only on some databases, but many also allow writes, or allow triggers to be used to handle writes.

Some JPA implementations provide extensions to handle this scenario.

`TopLink`, `EclipseLink` : Provides a proprietary API for its mapping model `ClassDescriptor.addForeignKeyFieldNameForMultipleTable()` that allows for arbitrary complex foreign key relationships to be defined among the secondary tables.

This can be configured through using a `@DescriptorCustomizer` annotation and `DescriptorCustomizer` class.

Multiple table joins



Occasionally the data model and object model do not get along very well at all. The database could be a legacy model and not fit very well with the new application model, or the DBA or object architect may be a little crazy. In these cases you may require advanced multiple table joins.

Examples of these include having two tables related not by their primary or foreign keys, but through some constant or computation. Consider having an `EMPLOYEE` table and an `ADDRESS` table, the `ADDRESS` table has an `EMP_ID` foreign key to the `EMPLOYEE` table, but there are several address for each employee and only the address with the `TYPE` of "HOME" is desired. In this case data from both of the tables is desired to be mapped in the `Employee` object. A join expression is required where the foreign key matches and the constant matches.

Again this scenario could be handled through redesigning the data or object model, or through using a view. Some JPA implementations provide extensions to handle this scenarios.

`TopLink`, `EclipseLink` : Provides a proprietary API for its mapping model `DescriptorQueryManager.setMultipleTableJoinExpression()` that allows for arbitrary complex multiple table joins to be defined. This can be configured through using a `@DescriptorCustomizer` annotation and `DescriptorCustomizer` class.

Multiple table outer joins

Another perversion of multiple table mapping is to desire to outer join the secondary table. This may be desired if the secondary may or may not have a row defined for the object. Typically the object should be read-only if this is to be attempted, as writing to a row that may or may not be there can be tricky.

This is not directly supported by JPA, and it is best to reconsider the data model or object model design if faced with this scenario. Again it is possible to map this through a database view, where an outer join is used to join the tables in the view.

Some JPA implementation support using outer joins for multiple tables.

`Hibernate` : This can be accomplished through using the `Hibernate @Table` annotation and set its `optional` attribute to `true`. This will configure `Hibernate` to use an outer join to read the table, and will not write to the table if all of the attributes mapping to the table are null.

`TopLink`, `EclipseLink` : If the database supports usage of outer join syntax in the where clause (`Oracle`, `Sybase`, `SQL Server`), then the multiple table join expression could be used to configure an outer join to be used to read the table.

Tables with special characters and mixed case

Some JPA providers may have issues with table and column names with special characters, such as spaces. In general it is best to use standard characters, no spaces, and all uppercase names. International languages should be ok, as long as the database and JDBC driver supports the character set.

It may be required to "quote" table and column names with special characters or in some cases with mixed case. For example if the table name had a space it could be defined as the following:

```
@Table("\Employee Data\")
```

Some databases support mixed case table and column names, and others are case insensitive. If your database is case insensitive, or you wish your data model to be portable, it is best to use all uppercase names. This is normally not a big deal with JPA where you rarely use the table and column names directly from your application, but can be an issue in certain cases if using native SQL queries.

Table qualifiers, schemas, or creators

A database table may require to be prefixed with a table qualifier, such as the table's creator, or its' namespace, schema, or catalog. Some databases also support linking table on other database, so the link name can also be a table qualifier.

In JPA a table qualifier can be set on a table through the `schema` or `catalog` attribute. Generally it does not matter which attribute is used as both just result in prefixing the table name. Technically you could even include the full name "schema.table" as the table's name and it would work. The benefit of setting the prefix in the schema or catalog is a *default* table qualifier can be set for the entire persistence unit, also not setting the real table name may impact native SQL queries.

If all of your tables require the same table qualifier, you can set the default in the `orm.xml`.

Example mapping annotations for an entity with a qualified table

```
...
@Entity
@Table(name="EMPLOYEE" schema="ACME")
public class Employee {
    ...
}
```

Example mapping XML for default (entire persistence unit) table qualifier

```
<entity-mappings>
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <schema name="ACME"/>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
  ...
```

```
</entity-mappings>
```

Example mapping XML for default (orm file) table qualifier

```
<entity-mappings>
  <schema name="ACME" />
  ...
</entity-mappings>
```

Identity

An object id (OID) is something that uniquely identifies an object. Within a VM this is typically the object's pointer. In a relational database table a row is uniquely identified in its' table by its' primary key. When persisting objects to a database you need a unique identifier for the objects, this allows you to query the object, define relationships to the object, and update and delete the object. In JPA the object id is defined through the `@Id` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Id.html>) annotation or `<id>` element and should correspond to the primary key of the object's table.

Example id annotation

```
...
@Entity
public class Employee {
    @Id
    private long id
    ...
}
```

Example id XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id name="id"/>
</entity/>
```

Common Problems

Strange behavior, unique constraint violation.

You must never change the id of an object. Doing so will cause errors, or strange behavior depending on your JPA provider. Also do not create two objects with the same id, or try persisting an object with the same id as an existing object. If you have an object that may be existing use the `EntityManager merge()` API, do not use `persist()` for an existing object, and avoid relating an un-managed existing object to other managed objects.

No primary key.

See No Primary Key.

Sequencing

An object id can either be a natural id or a generated id. A natural id is one that occurs in the object and has some meaning in the application. Examples of natural ids include user ids, email addresses, phone numbers, and social insurance numbers. A generated id is one that is generated by the system. A sequence number in JPA is a sequential id generated by the JPA implementation and automatically assigned to new objects. The benefits of using sequence numbers are that they are guaranteed to be unique, allow all other data of the object to change, are efficient values for querying and indexes, and can be efficiently assigned. The main issue with natural ids is that everything always changes at some point; even a person's social insurance number can change. Natural ids can also make querying, foreign keys and indexing less efficient in the database.

In JPA an `@Id` can be easily assigned a generated sequence number through the `@GeneratedValue` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/GeneratedValue.html>) annotation, or `<generated-value>` element.

Example generated id annotation

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private long id
    ...
}
```

Sequence Strategies

There are several strategies for generating unique ids. Some strategies are database agnostic and others make use of built-in databases support.

JPA provides support for several strategies for id generation defined through the `GenerationType` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/GenerationType.html>) enum, `TABLE`, `SEQUENCE` and `IDENTITY`.

The choice of which sequence strategy to use is important as it effect performance, concurrency and portability.

Table sequencing

Table sequencing uses a table in the database to generate unique ids. The table has two columns, one stores the name of the sequence, the other stores the last id value that was assigned. There is a row in the sequence table for each sequence object. Each time a new id is required the row for that sequence is incremented and the new id value is passed back to the application to be assigned to an object. This is just one example of a sequence table schema, for other table sequencing schemas see Customizing.

Table sequencing is the most portable solution because it just uses a regular database table, so unlike sequence and identity can be used on any database. Table sequencing also provides good performance because it allows for sequence pre-allocation, which is extremely important to insert performance, but can have potential concurrency issues.

In JPA the `@TableGenerator` ([https:// java. sun. com/ javase/ 5/ docs/ api/ javax/ persistence/ TableGenerator. html](https://java.sun.com/javase/5/docs/api/javax/persistence/TableGenerator.html)) annotation or `SEQ_NAMESEQ_COUNTTEMP_SEQ123PROJ_SEQ550`

Example table generator annotation

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=TABLE, generator="EMP_SEQ")
    @TableGenerator(name="EMP_SEQ", table="SEQUENCE_TABLE",
pkColumnName="SEQ_NAME",
    valueColumnName="SEQ_COUNT", pkColumnValue="EMP_SEQ")
    private long id;
    ...
}
```

Example generated id XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id name="id">
    <generated-value/>
  </id>
</entity/>
```

Example table generator XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id name="id">
    <generated-value strategy="TABLE" generator="EMP_SEQ"/>
    <table-generator name="EMP_SEQ" table="SEQUENCE_TABLE" pk-column-name="SEQ_NAME",
    value-column-name="SEQ_COUNT", pk-column-value="EMP_SEQ"/>
  </id>
</entity/>
```

Common Problems

Error when allocating a sequence number.

Errors such as *"table not found"*, *"invalid column"* can occur if you do not have a SEQUENCE table defined in your database, or its' schema does not match what your configured, or what your JPA provider is expecting by default. Ensure you create the sequence table correctly, or configure your `@TableGenerator` to match the table that you created, or let your JPA provider create you tables for you (most JPA provider support schema creation). You may also get an error such as *"sequence not found"*, this means you did not create a row in the table for your sequence. You must insert an

initial row in the sequence table for your sequence with the initial id (i.e. `INSERT INTO SEQUENCE_TABLE (SEQ_NAME, SEQ_COUNT) VALUES ("EMP_SEQ", 0)`), or let your JPA provider create your schema for you.

Deadlock or poor concurrency in the sequence table.

See concurrency issues.

Sequence objects

Sequence objects use special database objects to generate ids. Sequence objects are only supported in some databases, such as Oracle and Postgres. In Oracle a SEQUENCE object has a name, INCREMENT, and other database object settings. Each time the `<sequence>.NEXTVAL` is selected the sequence is incremented by the INCREMENT.

Sequence objects provide the optimal sequencing option, as they are the most efficient and have the best concurrency, however they are the least portable as most databases do not support them. Sequence objects support sequence preallocation through setting the INCREMENT on the database sequence object to the sequence preallocation size.

In JPA the `@SequenceGenerator` (<https://java.sun.com/javase/5/docs/api/javax/persistence/SequenceGenerator.html>) annotation or `<sequence-generator>` element is used to define a sequence object. The `SequenceGenerator` defines a `sequenceName` for the name of the database sequence object, and an `allocationSize` for the sequence preallocation size or sequence object INCREMENT.

Example sequence generator annotation

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=SEQUENCE, generator="EMP_SEQ")
    @SequenceGenerator(name="EMP_SEQ", sequenceName="EMP_SEQ",
allocationSize=100)
    private long id;
    ...
}
```

Example sequence generator XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id name="id">
    <generated-value strategy="SEQUENCE" generator="EMP_SEQ"/>
    <sequence-generator name="EMP_SEQ" sequence-name="EMP_SEQ" allocation-size="100"/>
  </id>
</entity/>
```


Common Problems

Error when allocating a sequence number.

Errors such as "*sequence not found*", can occur if you do not have a SEQUENCE object defined in your database. Ensure you create the sequence object, or let your JPA provider create your schema for you (most JPA providers support schema creation). When creating your sequence object, ensure the sequence's INCREMENT matches your SequenceGenerator's allocationSize. The DDL to create a sequence object depends on the database, for Oracle it is, CREATE SEQUENCE EMP_SEQ INCREMENT BY 100 START WITH 100.

Invalid, duplicate or negative sequence numbers.

This can occur if you sequence object's INCREMENT does not match your allocationSize. This results in the JPA provider thinking it got back more sequences than it really did, and ends up duplicating values, or with negative numbers. This can also occur on some JPA providers if you sequence object's STARTS WITH is 0 instead of a value equal or greater to the allocationSize.

Identity sequencing

Identity sequencing uses special IDENTITY columns in the database to allow the database to automatically assign an id to the object when its' row is inserted. Identity columns are supported in many databases, such as MySQL, DB2, SQL Server, Sybase and Postgres. Oracle does not support IDENTITY columns but they can be simulated through using sequence objects and triggers.

Although identity sequencing seems like the easiest method to assign an id, they have several issues. One is that since the id is not assigned by the database until the row is inserted the id cannot be obtained in the object until after commit or after a flush call. Identity sequencing also does not allow for sequence preallocation, so can require a select for each object that is inserted, potentially causing a major performance problem, so in general are not recommended.

In JPA there is no annotation or element for identity sequencing as there is no additional information to specify. Only the GeneratedValue's strategy needs to be set to IDENTITY.

Example identity annotation

```
...
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=IDENTITY)
    private long id;
    ...
}
```

Example identity XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id name="id">
    <generated-value strategy="IDENTITY"/>
  </id>
</entity/>
```

Common Problems

null is inserted into the database, or error on insert.

This typically occurs because the `@Id` was not configured to use an `@GeneratedValue(strategy="IDENTITY")`. Ensure it is configured correctly. It could also be that your JPA provider does not support identity sequencing on the database platform that you are using, or you have not configured your database platform. Most providers require that you set the database platform through a `persistence.xml` property, most provider also allow you to customize your own platform if it is not directly supported. It may also be that you did not set your primary key column in your table to be an identity type.

Object's id is not assign after persist.

Identity sequencing requires the insert to occur before the id can be assigned, so it is not assigned on `persist` like other types of sequencing. You must either call `commit()` on the current transaction, or call `flush()` on the `EntityManager`. It may also be that you did not set your primary key column in your table to be an identity type.

Child's id is not assign from parent on persist.

A common issue is that the generated Id is part of a child object's Id through a `OneToOne` or `ManyToOne` mapping. In this case, because JPA requires that the child define a duplicate `Basic` mapping for the Id, its Id will be inserted as null. One solution to this is to mark the `Column` on the Id mapping in the child as `insertable=false`, `updateable=false`, and define the `OneToOne` or `ManyToOne` using a normal `JoinColumn` this will ensure the foreign key field is populated by the `OneToOne` or `ManyToOne` not the `Basic`. Another option is to first persist the parent, then call `flush()` before persisting the child.

Poor insert performance.

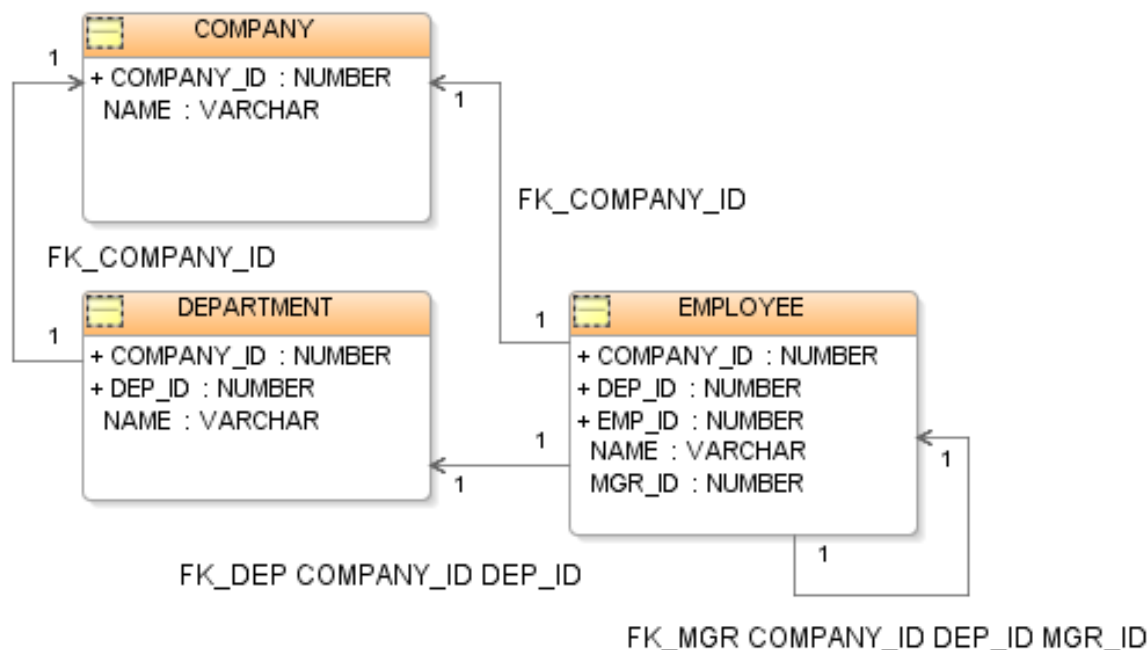
Identity sequencing does not support sequence preallocation, so requires a `select` after each insert, in some cases doubling the insert cost. Consider using a sequence table, or sequence object to allow sequence preallocation.

Advanced

Composite Primary Keys

A composite primary key is one that is made up of several columns in the table. A composite primary key can be used if no single column in the table is unique. In general it is normally more efficient and much simpler to have a singleton primary key, such as a generated

sequence number, but sometimes a composite primary key is desirable and unavoidable.



Composite primary keys can be common in legacy database schemas, where *cascaded keys* can sometimes be used. This is where you have a model where dependent objects include their parent's primary key, i.e. COMPANY's primary key is COMPANY_ID, DEPARTMENT's primary key is composed of COMPANY_ID, and DEP_ID, EMPLOYEE's primary key is composed of COMPANY_ID, DEP_ID, and EMP_ID, and so on. Some OO Java designers may find this type of model disgusting, but some DBA's actually think it is the correct model. Issues with the model include the obvious fact that Employee's cannot switch departments, but also foreign key relationships become more complex and all primary key queries, updates, deletes, caching become less efficient. On the plus side, each department has control over their own ids, and if you need to partition the database EMPLOYEE table, you can easily do so based on the COMPANY_ID or DEP_ID, as these are included in every query.

Other common usages of composite primary key include many-to-many relationships where the join table has additional columns, so the table is mapped to an object, whose primary key consists of both foreign key columns. Also dependent or aggregate one-to-many relationships where the child object's primary key consists of its' parent's primary key and a locally unique field.

There are two methods of declaring a composite primary key in JPA, `IdClass` and `EmbeddedId`.

Id Class

An `IdClass` defines a separate Java class to represent the primary key. It is defined through the `@IdClass` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/IdClass.html>) annotation or `<id-class>` XML element. The `IdClass` must define an attribute (field/property) that mirrors each `Id` attribute in the entity. It must have the same attribute name and type. When using an `IdClass` you still require to mark each `Id` attribute in the entity with `@Id`.

The main purpose of the `IdClass` is to be used as the structure passed to the `EntityManager find()` and `getReference()` API. Some JPA products also use the `IdClass`

as a cache key to track an object's identity. Because of this, it is required (depending on JPA product) to implement an `equals()` and `hashCode()` method on the `IdClass`. Ensure that the `equals()` method checks each part of the primary key, and correctly uses `equals` for objects and `==` for primitives. Ensure that the `hashCode()` method will return the same value for two equal objects.

TopLink / EclipseLink : Do not require the implementation of `equals()` or `hashCode()` in the id class.

Example id class annotation

```
...
@Entity
@IdClass(EmployeePK.class)
public class Employee {
    @Id
    private long employeeId

    @Id
    private long companyId

    @Id
    private long departmentId
    ...
}
```

Example id class XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <id-class class="org.acme.EmployeePK"/>
  <id name="employeeId"/>
  <id name="companyId"/>
  <id name="departmentId"/>
</entity/>
```

Example id class

```
...
public class EmployeePK {
    private long employeeId;

    private long companyId;

    private long departmentId;

    public EmployeePK(long employeeId, long companyId, long
departmentId) {
        this.employeeId = employeeId;
        this.companyId = companyId;
        this.departmentId = departmentId;
    }
}
```

```
public boolean equals(Object object) {
    if (object instanceof EmployeePK) {
        EmployeePK pk = (EmployeePK)object;
        return employeeId == pk.employeeId && companyId ==
pk.companyId && departmentId == pk.departmentId;
    } else {
        return false;
    }
}

public int hashCode() {
    return employeeId + companyId + departmentId;
}
}
```

Embedded Id

An `EmbeddedId` defines a separate `Embeddable` Java class to contain the entities primary key. It is defined through the `@EmbeddedId` (<https://java.sun.com/javase/5/docs/api/javax/persistence/EmbeddedId.html>) annotation or `<embedded-id>` XML element. The `EmbeddedId`'s `Embeddable` class must define each id attribute for the entity using `Basic` mappings. All attributes in the `EmbeddedId`'s `Embeddable` are assumed to be part of the primary key.

The `EmbeddedId` is also used as the structure passed to the `EntityManager find()` and `getReference()` API. Some JPA products also use the `EmbeddedId` as a cache key to track an object's identity. Because of this, it is required (depending on JPA product) to implement an `equals()` and `hashCode()` method on the `EmbeddedId`. Ensure that the `equals()` method checks each part of the primary key, and correctly uses `equals` for objects and `==` for primitives. Ensure that the `hashCode()` method will return the same value for two equal objects.

TopLink / EclipseLink : Do not require the implementation of `equals()` or `hashCode()` in the id class.

Example embedded id annotation

```
...
@Entity
public class Employee {
    @EmbeddedId
    private EmployeePK id
    ...
}
```

Example embedded id XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <embedded-id class="org.acme.EmployeePK"/>
</entity/>
<embeddable name="EmployeePK" class="org.acme.EmployeePK" access="FIELD">
  <basic name="employeeId"/>
  <basic name="companyId"/>
  <basic name="departmentId"/>
</embeddable/>
```

Example embedded id class

```
...
@Embeddable
public class EmployeePK {
    @Basic
    private long employeeId

    @Basic
    private long companyId

    @Basic
    private long departmentId

    public EmployeePK(long employeeId, long companyId, long
departmentId) {
        this.employeeId = employeeId;
        this.departmentId = companyId;
        this.departmentId = departmentId;
    }

    public boolean equals(Object object) {
        if (object instanceof EmployeePK) {
            EmployeePK pk = (EmployeePK)object;
            return employeeId == pk.employeeId && companyId ==
pk.companyId && departmentId == pk.departmentId;
        } else {
            return false;
        }
    }

    public int hashCode() {
        return employeeId + companyId + departmentId;
    }
}
```

Primary Keys through OneToOne Relationships

A common model is to have a dependent object share the primary key of its parent. In the case of a `OneToOne` the child's primary key is the same as the parent, and in the case of a `ManyToOne` the child's primary key is composed of the parent's primary key and another locally unique field.

Unfortunately JPA does not handle this model well, and things become complicated, so to make your life a little easier you may consider defining a generated unique id for the child. It would be simple if JPA allowed the `@Id` annotation on a `OneToOne` or `ManyToOne` mapping, but it does not. JPA requires that all `@Id` mappings be `Basic` mappings, so if your `Id` comes from a foreign key column through a `OneToOne` or `ManyToOne` mapping, you must also define a `Basic @Id` mapping for the foreign key column. The reason for this is in part that the `Id` must be a simple object for identity and caching purposes, and for use in the `IdClass` or the `EntityManager find()` API.

Because you now have two mappings for the same foreign key column you must define which one will be written to the database (it must be the `Basic` one), so the `OneToOne` or `ManyToOne` foreign key must be defined to be read-only. This is done through setting the `JoinColumn` attributes `insertable` and `updatable` to `false`, or by using the `@PrimaryKeyJoinColumn` instead of the `@JoinColumn`.

A side effect of having two mappings for the same column is that you now have to keep the two in synch. This is typically done through having the set method for the `OneToOne` attribute also set the `Basic` attribute value to the target object's id. This can become very complicated if the target object's primary key is a `GeneratedValue`, in this case you must ensure that the target object's id has been assigned *before* relating the two objects.

Some times I think that JPA primary keys would be much simpler if they were just defined on the entity using a collection of `Columns` instead of mixing them up with the attribute mapping. This would leave you free to map the primary key field in any manner you desired. A generic `List` could be used to pass the primary key to `find()` methods, and it would be the JPA provider's responsibility for hashing and comparing the primary key correctly instead of the user's `IdClass`. But perhaps for simple singleton primary key models the JPA model is more straight forward.

`TopLink / EclipseLink` : Allow the primary key to be specified as a list of columns instead of using `Id` mappings. This allows `OneToOne` and `ManyToOne` mapping foreign keys to be used as the primary key without requiring a duplicate mapping. It also allows the primary key to be defined through any other mapping type. This is set through using a `DescriptorCustomizer` and the `ClassDescriptor addPrimaryKeyFieldName` API.

`Hibernate / Open JPA`: Allows the `@Id` annotation to be used on a `OneToOne` or `ManyToOne` mapping.

Example OneToOne id annotation

```
...
@Entity
public class Address {
    @Id
    @Column(name="OWNER_ID")
    private long ownerId;
```

```

    @OneToOne
    @PrimaryKeyJoinColumn(name="OWNER_ID",
referencedColumnName="EMP_ID")
    private Employee owner;
    ...

    public void setOwner(Employee owner) {
        this.owner = owner;
        this.ownerId = owner.getId();
    }
    ...
}

```

Example OneToOne id XML

```

<entity name="Address" class="org.acme.Address" access="FIELD">
  <id name="ownerId">
    <column name="OWNER_ID"/>
  </id>
  <one-to-one name="owner">
    <primary-key-join-column name="OWNER_ID" referencedColumnName="EMP_ID"/>
  </one-to-one>
</entity/>

```

Example ManyToOne id annotation

```

...
@Entity
@IdClass({PhonePK.class})
public class Phone {
    @Id
    @Column(name="OWNER_ID")
    private long ownerId;

    @Id
    private String type;

    @ManyToOne
    @PrimaryKeyJoinColumn(name="OWNER_ID",
referencedColumnName="EMP_ID")
    private Employee owner;
    ...

    public void setOwner(Employee owner) {
        this.owner = owner;
        this.ownerId = owner.getId();
    }
    ...
}

```



```
}
```

Example ManyToOne id XML

```
<entity name="Address" class="org.acme.Address" access="FIELD">
  <id-class class="org.acme.PhonePK"/>
  <id name="ownerId">
    <column name="OWNER_ID"/>
  </id>
  <id name="type"/>
  <many-to-one name="owner">
    <primary-key-join-column name="OWNER_ID" referencedColumnName="EMP_ID"/>
  </many-to-one>
</entity/>
```

Advanced Sequencing

Concurrency and Deadlocks

One issue with table sequencing is that the sequence table can become a concurrency bottleneck, even causing deadlocks. If the sequence ids are allocated in the same transaction as the insert, this can cause poor concurrency, as the sequence row will be locked for the duration of the transaction, preventing any other transaction that needs to allocate a sequence id. In some cases the entire sequence table or the table page could be locked causing even transactions allocating other sequences to wait or even deadlock. If a large sequence pre-allocation size is used this becomes less of an issue, because the sequence table is rarely accessed. Some JPA providers use a separate (non-JTA) connection to allocate the sequence ids in, avoiding or limiting this issue. In this case, if you use a JTA data-source connection, it is important to also include a non-JTA data-source connection in your persistence.xml.

Running Out of Numbers

One paranoid delusional fear that programmers frequently have is running out of sequence numbers. Since most sequence strategies just keep incrementing a number it is unavoidable that you will eventually run out. However as long a large enough numeric precision is used to store the sequence id this is not an issue. For example if you stored your id in a NUMBER(5) column, this would allow 99,999 different ids, which on most systems would eventually run out. However if you store your id in a NUMBER(10) column, which is more typical, this would store 9,999,999,999 ids, or one id each second for about 300 years (longer than most databases exist). But perhaps your system will process a lot of data, and (hopefully) be around a very long time. If you store your id in a NUMBER(20) this would be 99,999,999,999,999,999,999 ids, or one id each millisecond for about 3,000,000,000 years, which is pretty safe.

But you also need to store this id in Java. If you store the id in a Java int, this would be a 32 bit number, which is 4,294,967,296 different ids, or one id each second for about 200 years. If you instead use a long, this would be a 64 bit number, which is 18,446,744,073,709,551,616 different ids, or one id each millisecond for about 600,000,000 years, which is pretty safe.

Guaranteeing Sequential Ids

Table sequencing also allows for truly sequential ids to be allocated. Sequence and identity sequencing are non-transactional and typically cache values on the database, leading to large gaps in the ids that are allocated. Typically this is not an issue and desired to have good performance, however if performance and concurrency are less of a concern, and true sequential ids are desired then a table sequence can be used. By setting the `allocationSize` of the sequence to 1 and ensuring the sequence ids are allocated in the same transaction of the insert, you can guarantee sequence ids without gaps (but generally it is much better to live with the gaps and have good performance).

Customizing

JPA supports three different strategies for generating ids, however there are many other methods. Normally the JPA strategies are sufficient, so you would only use a different method in a legacy situation.

Sometimes the application has an application specific strategy for generating ids, such as prefixing ids with the country code, or branch number. There are several ways to integrate a customize ids generation strategy, the simplest is just define the id as a normal id and have the application assign the id value when the object is created.

Some JPA products provide additional sequencing and id generation options, and configuration hooks.

TopLink, EclipseLink : Several additional sequencing options are provided. A `UnaryTableSequence` allows a single column table to be used. A `QuerySequence` allows for custom SQL or stored procedures to be used. An API also exists to allow a user to supply their own code for allocating ids.

Hibernate : A GUID id generation options is provided through the `@GenericGenerator` annotation.

Primary Keys through Triggers

A database table can be defined to have a trigger that automatically assign its' primary key. Generally this is normally not a good idea (although some DBAs may think it is), and it is better to use a JPA provider generated sequence id, or assign the id in the application. The main issue with the id being assigned in a trigger is that the application and object require this value back. For non-primary key values assigned through triggers it is possible to refresh the object after committing or flushing the object to obtain the values back. However this is not possible for the id, as the id is required to refresh an object.

If you have an alternative way to select the id generated by the trigger, such as selecting the object's row using another unique field, you could issue this SQL `select` after the insert to obtain the id and set it back in the object. You could perform this `select` in a JPA `@PostPersist` (<https://java.sun.com/javase/5/docs/api/javax/persistence/PostPersist.html>) event. Some JPA providers may not allow/like a query execution during an event, they also may not pick up a change to an object during an event callback, so there may be issues with doing this. Also some JPA providers may not allow the primary key to be un-assigned/null when not using a `GeneratedValue`, so you may have issues. Some JPA providers have built-in support for returning values assigned in a trigger (or stored procedure) back into the object.

TopLink / EclipseLink : Provide a `ReturningPolicy` that allows for any field values including the primary key to be returned from the database after an insert or update. This is defined through the `@ReturnInsert`, `@ReturnUpdate` annotations, or the `<return-insert>`, `<return-update>` XML elements in the `eclipselink-orm.xml`.

Primary Keys through Events

If the application generates its' own id instead of using a JPA `GeneratedValue`, it is sometimes desirable to perform this id generation in a JPA event, instead of the application code having to generate and set the id. In JPA this can be done through the `@PrePersist` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/PrePersist.html>) event.

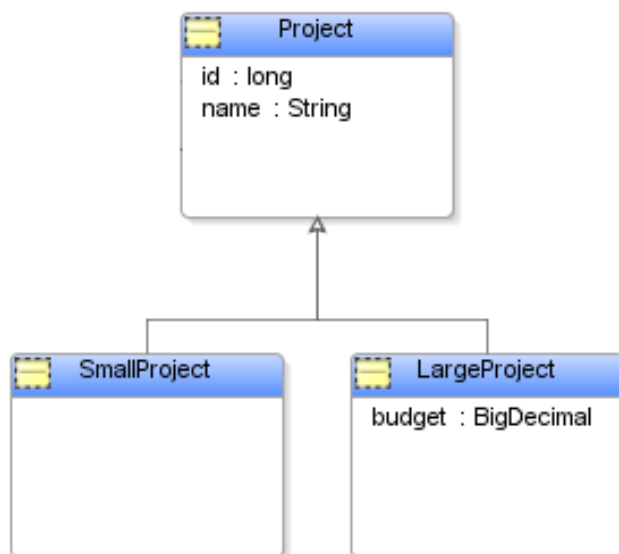
No Primary Key

Sometimes your object or table has no primary key. The best solution in this case is normally to add a generated id to the object and table. If you do not have this option, sometimes there is a column or set of columns in the table that make up a unique value. You can use this unique set of columns as your id in JPA. The JPA Id does not always have to match the database table primary key constraint, nor is a primary key or a unique constraint required.

If your table truly has no unique columns, then use all of the columns as the id. Typically when this occurs the data is read-only, so even if the table allows duplicate rows with the same values, the objects will be the same anyway, so it does not matter that JPA thinks they are the same object. The issue with allowing updates and deletes is that there is no way to uniquely identify the object's row, so all of the matching rows will be updated or deleted.

If your object does not have an id, but its' table does, this is fine. Make the object and `Embeddable` object, `embeddable` objects do not have ids. You will need a `Entity` that contains this `Embeddable` to persist and query it.

Inheritance



Inheritance is a fundamental concept of object-oriented programming and Java. Relational databases have no concept of inheritance, so persisting inheritance in a database can be

tricky. Because relational databases have no concept of inheritance, there is no standard way of implementing inheritance in database, so the hardest part of persisting inheritance is choosing how to represent the inheritance in the database.

JPA defines several inheritance mechanisms, mainly defined through the `@Inheritance` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Inheritance.html>) annotation or the `<inheritance>` element. There are three inheritance strategies defined from the `InheritanceType` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/InheritanceType.html>) enum, `SINGLE_TABLE`, `TABLE_PER_CLASS` and `JOINED`.

Single table inheritance is the default, and *table per class* is an *optional* feature of the JPA spec, so not all providers may support it. JPA also defines a mapped superclass concept defined through the `@MappedSuperclass` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/MappedSuperclass.html>) annotation or the `<mapped-superclass>` element. A mapped superclass is not a persistent class, but allow common mappings to be define for its subclasses.

Single Table Inheritance

Single table inheritance is the simplest and typically the best performing and best solution. In single table inheritance a single table is used to store all of the instances of the entire inheritance hierarchy. The table will have a column for *every* attribute of *every* class in the hierarchy. A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value.

Example single table inheritance table

PROJECT (table)

ID	PROJ_TYPE	NAME	BUDGET
1	L	Accounting	50000
2	S	Legal	null

Example single table inheritance annotations

```
@Entity
@Inheritance
@DiscriminatorColumn(name="PROJ_TYPE")
@Table(name="PROJECT")
public abstract class Project {
    @Id
    private long id;
    ...
}

@Entity
@DiscriminatorValue("L");
public class LargeProject extends Project {
    private BigDecimal budget;
}
```

```
@Entity
@DiscriminatorValue("S");
public class SmallProject extends Project {
}
```

Example single table inheritance XML

```
<entity name="Project" class="org.acme.Project" access="FIELD">
  <table name="PROJECT"/>
  <inheritance/>
  <discriminator-column name="PROJ_TYPE"/>
  <attributes>
    <id name="id"/>
    ...
  </attributes>
</entity/>
```

```
<entity name="LargeProject" class="org.acme.LargeProject" access="FIELD">
  <discriminator-value>L</discriminator-value>
  ...
</entity/>
```

```
<entity name="SmallProject" class="org.acme.SmallProject" access="FIELD">
  <discriminator-value>S</discriminator-value>
</entity/>
```

Common Problems

No class discriminator column

If you are mapping to an existing database schema, your table may not have a class discriminator column. Some JPA providers do not require a class discriminator when using a *joined* inheritance strategy, so this may be one solution. Otherwise you need some way to determine the class for a row. Sometimes the inherited value can be computed from several columns, or there is an discriminator but not a one to one mapping from value to class. Some JPA providers provide extended support for this. Another option is to create a database view that manufactures the discriminator column, and then map your hierarchy to this view instead of the table. In general the best solution is just to add a discriminator column to the table (truth be told, ALTER TABLE is your best friend in ORM).

TopLink / EclipseLink : Support computing the inheritance discriminator through Java code. This can be done through using a `DescriptorCustomizer` and the `ClassDescriptor's InheritancePolicy's setClassExtractor()` method.

Hibernate : This can be accomplished through using the Hibernate `@DiscriminatorFormula` annotation. This allows database specific SQL or functions to be used to compute the discriminator value.

Non nullable attributes

Subclasses cannot define attributes as not allowing null, as the other subclasses must insert null into those columns. A workaround to this issue is instead of defining a not null constraint on the column, define a table constraint that check the discriminator value and the not nullable value. In general the best solution is to just live without the constraint (odds are you have enough constraints in your life to deal with as it is).

Joined, Multiple Table Inheritance

Joined inheritance is the most logical inheritance solution because it mirrors the object model in the data model. In joined inheritance a table is defined for *each* class in the inheritance hierarchy to store *only* the local attributes of that class. Each table in the hierarchy must also store the object's id (primary key), which is *only* defined in the root class. All classes in the hierarchy must share the same id attribute. A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value.

Some JPA providers support joined inheritance with or without a discriminator column, some required the discriminator column, and some do not support the discriminator column. So joined inheritance does not seem to be fully standardized yet.

Hibernate : A discriminator column on joined inheritance is not supported. (<http://opensource.atlassian.com/projects/hibernate/browse/ANN-140>)

Example joined inheritance tables

PROJECT (table)

ID	PROJ_TYPE	NAME
1	L	Accounting
2	S	Legal

SMALLPROJECT (table)

ID
2

LARGEPROJECT (table)

ID	BUDGET
1	50000

Example joined inheritance annotations

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="PROJ_TYPE")
@Table(name="PROJECT")
public abstract class Project {
    @Id
    private long id;
```

```

...
}

@Entity
@DiscriminatorValue("L");
@Table(name="LARGEPROJECT")
public class LargeProject extends Project {
    private BigDecimal budget;
}

@Entity
@DiscriminatorValue("S");
@Table(name="SMALLPROJECT")
public class SmallProject extends Project {
}

```

Example joined inheritance XML

```

<entity name="Project" class="org.acme.Project" access="FIELD">
  <table name="PROJECT"/>
  <inheritance strategy="JOINED"/>
  <discriminator-column name="PROJ_TYPE"/>
  <attributes>
    <id name="id"/>
    ...
  </attributes>
</entity/>

<entity name="LargeProject" class="org.acme.LargeProject" access="FIELD">
  <table name="LARGEPROJECT"/>
  <discriminator-value>L</discriminator-value>
  ...
</entity/>

<entity name="SmallProject" class="org.acme.SmallProject" access="FIELD">
  <table name="SMALLPROJECT"/>
  <discriminator-value>S</discriminator-value>
</entity/>

```

Common Problems

Poor query performance

The main disadvantage to the joined model is that to query any class join queries are required. Querying the root or branch classes is even more difficult as either multiple queries are required, or outer joins or unions are required. One solution is to use single table inheritance instead, this is good if the classes have a lot in common, but if it is a big hierarchy and the subclasses have little in common this may not be desirable. Another solution is to remove the inheritance and instead use a MappedSuperclass, but this means that you can no longer query or have relationships to the class.

The poorest performing queries will be those to the root or branch classes. Avoiding queries and relationships to the root and branch classes will help to alleviate this burden. If you must query the root or branch classes there are two methods that JPA providers use, one is to outer join all of the subclass tables, the second is to first query the root table, then query only the required subclass table directly. The first method has the advantage of only requiring one query, the second has the advantage of avoiding outer joins which typically have poor performance in databases. You may wish to experiment with each to determine which mechanism is more efficient in your application and see if your JPA provider supports that mechanism. Typically the multiple query mechanism is more efficient, but this generally depends on the speed of your database connection.

TopLink / EclipseLink : Support both querying mechanisms. The multiple query mechanism is used by default. Outer joins can be used instead through using a `DescriptorCustomizer` and the `ClassDescriptor's InheritancePolicy's setShouldOuterJoinSubclasses()` method.

Do not have/want a table for every subclass

Most inheritance hierarchies do not fit with either the *joined* or the *single table* inheritance strategy. Typically the desired strategy is somewhere in between, having joined tables in some subclasses and not in others. Unfortunately JPA does not directly support this. One workaround is to map your inheritance hierarchy as single table, but then add the additional tables in the subclasses, either through defining a `Table` or `SecondaryTable` in each subclass as required. Depending on your JPA provider, this may work (don't forget to sacrifice the chicken). If it does not work, then you may need to use a JPA provider specific solution if one exists for your provider, otherwise live within the constraints of having either a single table or one per subclass. You could also change your inheritance hierarchy so it matches your data model, so if the subclass does not have a table, then collapse its' class into its' superclass.

No class discriminator column

If you are mapping to an existing database schema, your table may not have a class discriminator column. Some JPA providers do not require a class discriminator when using a *joined* inheritance strategy, so this may be one solution. Otherwise you need some way to determine the class for a row. Sometimes the inherited value can be computed from several columns, or there is an discriminator but not a one to one mapping from value to class. Some JPA providers provide extended support for this. Another option is to create a database view that manufactures the discriminator column, and then map your hierarchy to this view instead of the table.

TopLink / EclipseLink : Support computing the inheritance discriminator through Java code. This can be done through using a `DescriptorCustomizer` and the `ClassDescriptor's InheritancePolicy's setClassExtractor()` method.

Hibernate : This can be accomplished through using the `Hibernate @DiscriminatorFormula` annotation. This allows database specific SQL or functions to be used to compute the discriminator value.

Advanced

Table Per Class Inheritance

Table per class inheritance allows inheritance to be used in the object model, when it does not exist in the data model. In table per class inheritance a table is defined for *each* concrete class in the inheritance hierarchy to store *all* the attributes of that class and *all* of its' superclasses. Be cautious using this strategy as it is optional in the JPA spec, and querying root or branch classes can be very difficult and inefficient.

Example table per class inheritance tables

SMALLPROJECT (table)

ID	NAME
2	Legal

LARGEPROJECT (table)

ID	NAME	BUDGET
1	Accounting	50000

Example table per class inheritance annotations

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Project {
    @Id
    private long id;
    ...
}
```

```
@Entity
@Table(name="LARGEPROJECT")
public class LargeProject extends Project {
    private BigDecimal budget;
}
```

```
@Entity
@Table(name="SMALLPROJECT")
public class SmallProject extends Project {
}
```

Example table per class inheritance XML

```
<entity name="Project" class="org.acme.Project" access="FIELD">
  <inheritance strategy="TABLE_PER_CLASS"/>
  <attributes>
    <id name="id"/>
    ...
  </attributes>
```

```
<entity/>
<entity name="LargeProject" class="org.acme.LargeProject" access="FIELD">
  <table name="LARGEPROJECT"/>
  ...
<entity/>
<entity name="SmallProject" class="org.acme.SmallProject" access="FIELD">
  <table name="SMALLPROJECT"/>
<entity/>
```

Common Problems

Poor query performance

The main disadvantage to the table per class model is queries or relationships to the root or branch classes become expensive. Querying the root or branch classes require multiple queries, or unions. One solution is to use single table inheritance instead, this is good if the classes have a lot in common, but if it is a big hierarchy and the subclasses have little in common this may not be desirable. Another solution is to remove the table per class inheritance and instead use a `MappedSuperclass`, but this means that you can no longer query or have relationships to the class.

Issues with ordering and joins

Because table per class inheritance requires multiple queries, or unions, you cannot join to, fetch join, or traverse them in queries. Also when ordering is used the results will be ordered by class, then by the ordering. These limitations depend on your JPA provider, some JPA provider may have other limitations, or not support table per class at all as it is optional in the JPA spec.

Mapped Superclasses

Mapped superclass inheritance allows inheritance to be used in the object model, when it does not exist in the data model. It is similar to table per class inheritance, but does not allow querying, persisting, or relationships to the superclass. Its' main purpose is to allow mappings information to be inherited by its' subclasses. The subclasses are responsible for defining the table, id and other information, and can modify any of the inherited mappings. A common usage of a mapped superclass is to define a common `PersistentObject` for your application to define common behavior and mappings such as the id and version. A mapped superclass normally should be an abstract class. A mapped superclass is *not* an `Entity` but is instead defined through the `@MappedSuperclass` (<https://java.sun.com/javase/5/docs/api/javax/persistence/MappedSuperclass.html>) annotation or the `<mapped-superclass>` element.

Example mapped superclass tables

SMALLPROJECT (table)

ID	NAME
2	Legal

LARGEPROJECT (table)

ID	PROJECT_NAME	BUDGET
1	Accounting	50000

Example mapped superclass annotations

```
@MappedSuperclass
```

```
public abstract class Project {
    @Id
    private long id;
    @Column(name="NAME")
    private String name;
    ...
}
```

```
@Entity
```

```
@Table(name="LARGEPROJECT")
```

```
@AttributeOverride(name="name", column=@Column(name="PROJECT_NAME"))
```

```
public class LargeProject extends Project {
    private BigDecimal budget;
}
```

```
@Entity
```

```
@Table("SMALLPROJECT")
```

```
public class SmallProject extends Project {
}
```

Example mapped superclass XML

```
<mapped-superclass class="org.acme.Project" access="FIELD">
```

```
  <attributes>
```

```
    <id name="id"/>
```

```
    <basic name="name">
```

```
      <column name="NAME"/>
```

```
    </basic>
```

```
    ...
```

```
  </attributes>
```

```
</mapped-superclass/>
```

```
<entity name="LargeProject" class="org.acme.LargeProject" access="FIELD">
```

```
  <table name="LARGEPROJECT"/>
```

```
  <attribute-override>
```

```
    <column name="NAME"/>
```

```
    </attribute-override>
    ...
<entity/>

<entity name="SmallProject" class="org.acme.SmallProject" access="FIELD">
    <table name="SMALLPROJECT"/>
<entity/>
```

Common Problems

Cannot query, persist, or have relationships

The main disadvantage of mapped superclasses is that they cannot be queried or persisted. You also cannot have a relationship to a mapped superclass. If you require any of these then you must use another inheritance model, such as table per class, which is virtually identical to a mapped superclass except it (may) not have these limitations. Another alternative is to change your model such that your classes do not have relationships to the superclass, such as changing the relationship to a subclass, or removing the relationship and instead querying for its value by querying each possible subclass and collecting the results in Java.

Subclass does not want to inherit mappings

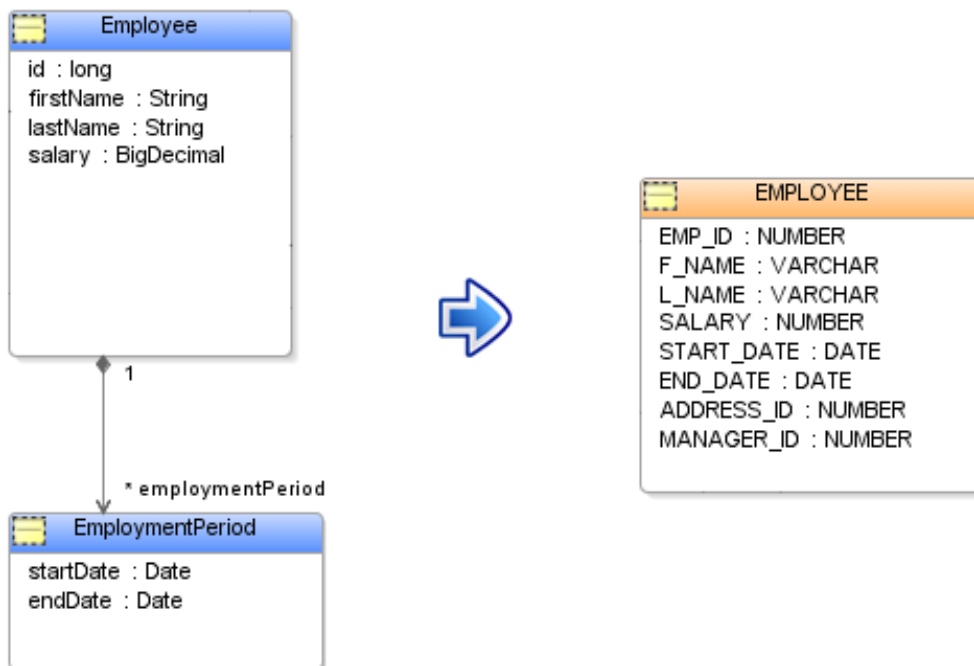
Sometimes you have a subclass that needs to be mapped differently than its parent, or is similar to its' parent but does not have one of the fields, or uses it very differently. Unfortunately it is very difficult not to inherit everything from your parent in JPA, you can override a mapping, but you cannot remove one, or change the type of mapping, or the target class. If you define your mappings as properties (get methods), or through XML, you may be able to attempt to override or mark the inherited mapping as Transient, this may work depending on your JPA provider (don't forget to sacrifice a chicken).

Another solution is to actually fix your inheritance in your object model. If you inherit `foo` from `Bar` but don't want to inherit it, then remove it from `Bar`, if the other subclasses need it, either add it to each, or create a `FooBar` subclass of `Bar` that has the `foo` and have the other subclasses extend this.

Some JPA providers may provide ways to be less stringent on inheritance.

TopLink / EclipseLink : Allow a subclass remove a mapping, redefine a mapping, or be entirely independent of its superclass. This can be done through using a `DescriptorCustomizer` and removing the `ClassDescriptor`'s mapping, or adding a mapping with the same attribute name, or removing the `InheritancePolicy`.

Embeddables



In an application object model some objects are considered independent, and others are considered dependent parts of other objects. In UML a relationship to a dependent object is considered an aggregate or composite association. In a relational database this kind of relationship could be modeled in two ways, the dependent object could have its own table, or its data could be *embedded* in the independent object's table.

In JPA a relationship where the target object's data is embedded in the source object's table is considered an embedded relationship, and the target object is considered an Embeddable object. Embeddable objects have different requirements and restrictions than Entity objects and are defined by the `@Embeddable` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Embeddable.html>) annotation or `<embeddable>` element.

An embeddable object cannot be directly persisted, or queried, it can only be persisted or queried in the context of its parent. An embeddable object does not have an id or table. The JPA spec does not support embeddable objects having relationships or inheritance, although some JPA providers may allow this.

Relationships to embeddable objects are defined through the `@Embedded` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Embedded.html>) annotation or `<embedded>` element. The JPA spec only allows references to embeddable objects, and does not support collection relationships to embeddable objects, although some JPA providers may allow this.

Example of an Embeddable object annotations

```
@Embeddable
public class EmploymentPeriod {
    @Column(name="START_DATE")
    private java.sql.Date startDate;

    @Column(name="END_DATE")
    private java.sql.Date endDate;
    ...
}
```

Example of an Embeddable object XML

```
<embeddable class="org.acme.EmploymentPeriod" access="FIELD">
  <attributes>
    <basic name="startDate">
      <column name="START_DATE"/>
    </basic>
    <basic name="endDate">
      <column name="END_DATE"/>
    </basic>
  </attributes>
</embeddable>
```

Example of an embedded relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @Embedded
    private EmploymentPeriod period;
    ...
}
```

Example of an embedded relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <embedded name="period"/>
  </attributes>
</entity>
```

Advanced

Sharing

An embeddable object can be shared between multiple classes. Consider a `Name` object, that both an `Employee` and a `User` contain. Both `Employee` and a `User` have their own tables, with different column names that they desire to store their name in. Embeddables support this through allowing each embedded mapping to override the columns used in the embeddable. This is done through the `@AttributeOverride` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/AttributeOverride.html>) annotation or `<attribute-override>` element.

Note that an embeddable cannot be shared between multiple instances. If you desire to share an embeddable object instance, then you must make it an independent object with its own table.

Example shared embeddable annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="startDate",
column=@Column(name="START_DATE")),
        @AttributeOverride(name="endDate", column=@Column(name="END_DATE"))
    })
    private Name name;
    ...
}
```

```
@Entity
public class User {
    @Id
    private long id;
    ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="startDate", column=@Column(name="SDATE")),
        @AttributeOverride(name="endDate", column=@Column(name="EDATE"))
    })
    private Name name;
    ...
}
```

Example shared embeddable XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <embedded name="name">
      <attribute-override name="startDate">
        <column name="START_DATE"/>
      </attribute-override>
      <attribute-override name="endDate">
        <column name="END_DATE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
<entity name="User" class="org.acme.User" access="FIELD">
  <attributes>
    <id name="id"/>
    <embedded name="name">
      <attribute-override name="startDate">
        <column name="SDATE"/>
      </attribute-override>
      <attribute-override name="endDate">
        <column name="EDATE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
```

Embedded Ids

An `EmbeddedId` is an embeddable object that contains the `Id` for an entity.

See: [Embedded Id](#)

Nulls

An embeddable object's data is contained in several columns in its parent's table. Since there is no single field value, there is no way to know if a parent's reference to the embeddable is null. One could assume that if every field value of the embeddable is null, then the reference should be null, but then there is no way to represent an embeddable with all null values. JPA does not allow embeddables to be null, but some JPA providers may support this.

`TopLink` / `EclipseLink` : Support an embedded reference being null. This is set through using a `DescriptorCustomizer` and the `AggregateObjectMapping.setIsNullAllowed` API.

Nesting

A nested embeddable is a relationship to an embeddable object from another embeddable. The JPA spec only allows `Basic` relationships in an embeddable object, so nested embeddables are not supported, however some JPA products may support them. Technically there is nothing preventing the `@Embedded` annotation being used in an embeddable object, so this may just work depending on your JPA provider (don't forget to sacrifice a chicken).

TopLink / EclipseLink : Support embedded mappings from embeddables. The existing `@Embedded` annotation or `<embedded>` element can be used.

A workaround to having a nested embeddable, and for embeddables in general is to use property access, and add get/set methods for all of the attributes of the nested embeddable object.

Example of using properties to define a nested embeddable

```
@Embeddable
public class EmploymentDetails {
    private EmploymentPeriod period
    private int yearsOfService;
    private boolean fullTime;
    ....
    public EmploymentDetails() {
        this.period = new EmploymentPeriod();
    }
    @Transient
    public EmploymentPeriod getEmploymentPeriod() {
        return period;
    }
    @Basic
    public Date getStartDate() {
        return getEmploymentPeriod().getStartDate();
    }
    public void setStartDate(Date startDate) {
        getEmploymentPeriod().setStartDate(startDate);
    }
    @Basic
    public Date getEndDate() {
        return getEmploymentPeriod().getEndDate();
    }
    public void setEndDate(Date endDate) {
        getEmploymentPeriod().setStartDate(endDate);
    }
    ....
}
```

Inheritance

Embeddable inheritance is when one embeddable class subclasses another embeddable class. The JPA spec does not allow inheritance in embeddable objects, however some JPA products may support this. Technically there is nothing preventing the `@DiscriminatorColumn` annotation being used in an embeddable object, so this may just work depending on your JPA provider (cross your fingers). Inheritance in embeddables is always *single table* as an embeddable must live within its' parent's table. Generally attempting to mix inheritance between embeddables and entities is not a good idea, but may work in some cases.

TopLink / EclipseLink : Support inheritance with embeddables. This is set through using a `DescriptorCustomizer` and the `InheritancePolicy`.

Relationships

A relationship is when an embeddable has a `OneToOne` or other such mapping to an entity. The JPA spec only allows `Basic` mappings in an embeddable object, so relationships from embeddables are not supported, however some JPA products may support them. Technically there is nothing preventing the `@OneToOne` annotation or other relationships from being used in an embeddable object, so this may just work depending on your JPA provider (cross your fingers).

TopLink / EclipseLink : Support relationship mappings from embeddables. The existing relationship annotations or XML elements can be used.

Relationships to embeddable objects from entities other than the embeddable's parent are typically not a good idea, as an embeddable is a private dependent part of its parent. Generally relationships should be to the embeddable's parent, not the embeddable. Otherwise, it would normally be a good idea to make the embeddable an independent entity with its own table. If an embeddable has a bi-directional relationship, such as a `OneToMany` that requires an inverse `ManyToOne` the inverse relationship should be to the embeddable's parent.

A workaround to having a relationship from an embeddable is to define the relationship in the embeddable's parent, and define property get/set methods for the relationship that set the relationship into the embeddable.

Example of setting a relationship in an embeddable from its parent

```
@Entity
public class Employee {
    ....
    private EmploymentDetails details;
    ....
    @Embedded
    public EmploymentDetails getEmploymentDetails() {
        return details;
    }
    @OneToOne
    public Address getEmploymentAddress() {
        return getEmploymentDetails().getAddress();
    }
}
```

```
}  
public void setEmploymentAddress(Address address) {  
    getEmploymentDetails().setAddress(address);  
}  
}
```

One special relationship that is sometimes desired in an embeddable is a relationship to its parent. JPA does not support this, but some JPA providers may.

 Hibernate : Supports a `@Parent` annotation in embeddables to define a relationship to its parent.

A workaround to having a parent relationship from an embeddable is to set the parent in the property set method.

Example of setting a relationship in an embeddable to its parent

```
@Entity  
public class Employee {  
    ....  
    private EmploymentDetails details;  
    ....  
    @Embedded  
    public EmploymentDetails getEmploymentDetails() {  
        return details;  
    }  
    public void setEmploymentDetails(EmploymentDetails details) {  
        this.details = details;  
        details.setParent(this);  
    }  
}
```

Collections

A collection of embeddable objects is similar to a `OneToMany` except the target objects are embeddables and have no Id. This allows for a `OneToMany` to be defined without a inverse `ManyToOne`, as the parent is responsible for storing the foreign key in the target object's table. JPA 1.0 does not support collections of embeddable objects, but some JPA providers support this.

 TopLink / EclipseLink : Support collections of embeddables. This is set through using a `DescriptorCustomizer` and the `AggregateCollectionMapping`.

 Hibernate : Supports collections of embeddables through the `@CollectionOfElements` annotation.

Typically the primary key of the target table will be composed of the parent's primary key, and some unique field in the embeddable object. The embeddable should have a unique field within its parent's collection, but does not need to be unique for the entire class. It could still have a unique id and still use sequencing, or if it has no unique fields, its id could be composed of all of its fields. The embeddable collection object will be different than a typical embeddable object as it will not be stored in the parent's table, but in its own table. Embeddables are strictly privately owned objects, deletion of the parent will cause deletion

of the embeddables, and removal from the embeddable collection should cause the embeddable to be deleted. Embeddables cannot be queried directly, and are not independent objects as they have no Id.

Querying

Embeddable objects cannot be queried directly, but they can be queried in the context of their parent. Typically it is best to select the parent, and access the embeddable from the parent. This will ensure the embeddable is registered with the persistence context. If the embeddable is selected in a query, the resulting objects will be detached, and changes will not be tracked.

Example of querying an embeddable

```
SELECT employee.period from Employee employee where  
employee.period.endDate = :param
```

Locking

Locking is perhaps the most ignored persistence consideration. Most applications tend to ignore thinking about concurrency issues during development, and then smush in a locking mechanism before going into production. Considering the large percentage of software projects that fail or are canceled, or never achieve a large user base, perhaps this is logical. However, locking and concurrency is a critical or at least a very important issue for most applications, so probably should be something considered earlier in the development cycle.

If the application will have concurrent writers to the same objects, then a locking strategy is critical so that data corruption can be prevented. There are two strategies for preventing concurrent modification of the same object/row; optimistic and pessimistic locking. Technically there is a third strategy, *ostrich* locking, or no locking, which means put your head in the sand and ignore the issue.

There are various ways to implement both optimistic and pessimistic locking. JPA has support for version optimistic locking, but some JPA providers support other methods of optimistic locking, as well as pessimistic locking.

Locking and concurrency can be a confusing thing to consider, and there are a lot of misconcepts out there. Correctly implementing locking in your application typically involves more than setting some JPA or database configuration option (although that is all many applications that *think* they are using locking do). Locking may also involve application level changes, and ensuring other applications accessing the database also do so correctly for the locking policy being used.

Optimistic Locking

Optimistic locking assumes that the data will not be modified between when you read the data until you write the data. This is the most common style of locking used and recommended in today's persistence solutions. The strategy involves checking that one or more values from the original object read, are still the same when updating it. This verifies that the object has not changed by another user in between the read and the write.

JPA supports using an optimistic locking version field that gets updated on each update. The field can either be numeric or a timestamp value. A numeric value is recommended as a numeric value is more precise, portable, performant and easier to deal with than a timestamp.

The `@Version` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Version.html>) annotation or `<version>` element is used to define the optimistic lock version field. The annotation is defined on the version field or property for the object, similar to an Id mapping. The object must contain an attribute to store the version field.

The object's version attribute is automatically updated by the JPA provider, and should not normally be modified by the application. The one exception is if the application reads the object in one transaction, sends the object to a client, and updates/merges the object in another transaction. In this case the application must ensure that the original object version is used, otherwise any changes in between the read and write will not be detected. The `EntityManager merge()` API will always merge the version, so the application is only responsible for this if manually merging.

When a locking contention is detected an `OptimisticLockException` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/OptimisticLockException.html>) will be thrown. This could be wrapped insider a `RollbackException`, or other exceptions if using JTA, but it should be set as the cause of the exception. The application can handle the exception, but should normally report the error to the user, and let them determine what to do.

Example of Version annotation

```
@Entity
public abstract class Employee{
    @Id
    private long id;
    @Version
    private long version;
    ...
}
```

Common Locking Mistakes, Questions and Problems

Not sending version to client, only locking on the server

Probably the most common mistake in locking in general is locking the wrong section of code. This is true no matter what form of locking is used, whether it be optimistic or pessimistic. The basic scenario is:

1. User requests some data, the server reads the data from the database and sends it to the user in their client (doesn't matter if the client is html, rmi, web service).
2. The user edits the data in the client.
3. The user submits the data back to the server.
4. The server begins a transaction, reads the object, merges the data and commits the transaction.

The issues is that the original data was read in step 1, but the lock was not obtained until step 4, so any changes made to the object in between steps 1 and 4 would not result in a conflict. This means there is little point to using any locking.

A key point is that when using database pessimistic locking or database transaction isolation, this will always be the case, the database locks will only occur in step 4, and any conflicts will not be detected. This is the main reason why using database locking does not scale to web applications, for the locking to be valid, the database transaction must be started at step 1 and not committed until step 4. This means the a live database connection and transaction must be held open while waiting for the web client, as well as locks, since there is no guarantee that the web client will not sit on the data for hours, go to lunch, or disappear of the face of the earth, holding database resources and locking data for all other users can be very undesirable.

For optimistic locking the solution is relatively simple, the object's version must be sent to the client along with the data (or kept in the http session). When the user submits the data back, the original version must be merged into the object read from the database, to ensure that any changes made between step 1 and 4 will be detected.

Example of Version XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <version name="version"/>
    ...
  </attributes>
</entity/>
```

Handling optimistic lock exceptions

Unfortunately programmers can frequently be too clever for their own good. The first issue that comes up when using optimistic locking is what to do when an `OptimisticLockException` occurs. The typical response of the friendly neighborhood super programmer, is to automatically handle the exception. They will just create a new transaction, refresh the object to reset its version, and merge the data back into the object and re-commit it. Presto problem solved, or is it?

This actually defeats the whole point of locking in the first place. If this is what you desire, you may as well use no locking. Unfortunately, the `OptimisticLockException` should rarely be automatically handled, and you really need to bother the user about the issue. You should report the conflict to the user, and either say "your sorry but an edit conflict occurred and they are going to have to redo their work", or in the best case, refresh the object and present the user with the current data and the data that they submitted and help them merge the two if appropriate.

Some automated merge tools will compare the two conflicting versions of the data and if none of the individual fields conflict, then the data will just be automatically merged without the user's aid. This is what most software version control systems do. Unfortunately the user is typically better able to decide when something is a conflict than the program, just because two versions of the .java file did not change the same line of code does not mean there was no conflict, the first user could have deleted a method that the other user added a method to reference, and several other possible issues that cause the typically nightly build to break every so often.

Paranoid Delusionment

Locking can prevent most concurrency issues, but be careful of going overboard in over analyzing to death every possible hypothetical occurrence. Sometimes in an concurrent application (or any software application) bad stuff can happen. Users are pretty used to this by now, and I don't think anyone out there thinks computer are perfect.

A good example is a source code control system. Allowing users to overwrite each other changes is a bad thing; so most systems avoid this through versioning the source files. If a user submits changes to a file that originated from a older version than the current version, the source code control system will raise a conflict and make the user merge the two files. This is essentially optimistic locking. But what if one user removes, or renames a method in one file, then another user adds a new method or call in another file to that old method. No source code control system that I know of will detect this issue, it is a conflict and will cause the build to break. The solution to this is to start locking or checking the lock on every file in the system (or at least every possible related file). Similar to using optimistic read locking on every possible related object, or pessimistically locking every possible related object. This could be done, but would probably be very expensive, and more importantly would now raise possible conflicts every time a user checked in, so would be entirely useless.

So, in general be careful of being too paranoid, such that you sacrifice the usability of your system.

Other applications accessing same data

Any form of locking that is going to work requires that all applications accessing the same data follow the same rules. If you use optimistic locking in one application, but no locking in another accessing the same data, they will still conflict. One fake solution is to configure an update trigger to always increment the version value (unless incremented in the update). This will allow the new application to avoid overwriting the old application's changes, but the old application will still be able to overwrite the new application's changes. This still may be better than no locking at all, and perhaps the old application will eventually go away.

One common misconception is that if you use pessimistic locking, instead of adding a version field, you will be ok. Again pessimistic locking requires that all applications accessing the same data use the same form of locking. The old application can still read data (without locking), then update the data after the new application reads, locks, and updates the same data, overwriting its changes.

Isn't database transaction isolation all I need?

Possibly, but most likely not. Most databases default to *read committed* transaction isolation. This means that you will never see uncommitted data, but this does not prevent concurrent transactions from overwriting the same data.

1. Transaction A reads row x.
2. Transaction B reads row x.
3. Transaction A writes row x.
4. Transaction B writes row x (and overwrites A's changes).
5. Both commit successfully.

This is the case with *read committed*, but with *serializable* this conflict would not occur. With serializable either Transaction B would lock on the select for B and wait (perhaps a long time) until Transaction A commits. In some databases Transaction A may not wait, but would fail on commit. However, even with serializable isolation the typical web application would still have a conflict. This is because each server request operates in a different database transaction. The web client reads the data in one transaction, then updates it in another transaction. So optimistic locking is really the only viable locking option for the typical web application. Even if the read and write occurs in the same transaction, serializable is normally not the solution because of concurrency implications and deadlock potential.

See Serializable Transaction Isolation

What happens if I merge an object that was deleted by another user?

What *should* happen is the merge should trigger an `OptimisticLockException` because the object has a version that is not null and greater than 0, and the object does not exist. But this is probably JPA provider specific, some may re-insert the object (this would occur without locking), or throw a different exception.

If you called `persist` instead of `merge`, then the object would be re-inserted.

What if my table doesn't have a version column?

The best solution is probably just to add one. Field locking is another solution, as well as pessimistic locking in some cases.

See Field Locking

What about relationships?

See Cascaded Locking

Can I use a timestamp?

See Timestamp Locking

Do I need a version in each table for inheritance or multiple tables?

The short answer is no, only in the root table.

See Multiple Versions

Advanced**Timestamp Locking**

Timestamp version locking is supported by JPA and is configured the same as numeric version locking, except the attribute type will be a `java.sql.Timestamp` or other date/time type. Be cautious in using timestamp locking as timestamps have different levels of precision in different databases, and some database do not store a timestamp's milliseconds, or do not store them precisely. In general timestamp locking is less efficient than numeric version locking, so numeric version locking is recommended.

Timestamp locking is frequently used if the table already has a *last updated* timestamp column, and is also a convenient way to auto update a *last updated* column. The timestamp version value can be more useful than a numeric version, as it includes the relevant information on when the object was last updated.

The timestamp value in timestamp version locking can either come from the database, or from Java (mid-tier). JPA does not allow this to be configured, however some JPA providers may provide this option. Using the database's current timestamp can be very expensive, as it requires a database call to the server.

Multiple Versions

An object can only have one version in JPA. Even if the object maps to multiple tables, only the primary table will have the version. If any fields in any of the tables changes, the version will be updated. If you desire multiple versions, you may need to map multiple version attributes in your object and manually maintain the duplicate versions, perhaps through events. Technically there is nothing preventing your from annotating multiple attributes with `@Version`, and potentially some JPA providers may support this (don't forget to sacrifice a chicken).

Cascaded Locking

Locking objects is different than locking rows in the database. An object can be more complex than a simple row; an object can span multiple tables, have inheritance, have relationships, and have dependent objects. So determining when an object has *changed* and needs to update its' version can be more difficult than determining when a row has changed.

JPA does define that when any of the object's tables changes the version is updated. However it is less clear on relationships. If Basic, Embedded, or a foreign key relationship (OneToOne, ManyToOne) changes, the version will be updated. But what about OneToMany, ManyToMany, and a target foreign key OneToOne? For changes to these relationships the update to the version may depend on the JPA provider.

What about changes made to dependent objects? JPA does not have a cascade option for locking, and has no direct concept of dependent objects, so this is not an option. Some JPA providers may support this. One way to simulate this is to use write locking. JPA defines the

`EntityManager lock()` ([https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#lock\(java.lang.Object, javax.persistence.LockModeType\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#lock(java.lang.Object, javax.persistence.LockModeType))) API. You can define a version only in your root parent objects, and when a child (or relationship) is changed, you can call the lock API with the parent to cause a `WRITE` lock. This will cause the parent version to be updated. You may also be able to automate this through persistence events.

Usage of cascaded locking depends on your application. If in your application you consider one user updating one dependent part of an object, and another user updating another part of the object to be a locking contention, then this is what you want. If your application does not consider this to be a problem, then you do not want cascaded locking. One of the advantages of cascaded locking is you have fewer version fields to maintain, and only the update to the root object needs to check the version. This can make a difference in optimizations such as batch writing, as the dependent objects may not be able to be batched if they have their own version that must be checked.

`TopLink` / `EclipseLink` : Support cascaded locking through their `@OptimisticLocking` and `@PrivateOwned` annotations and XML.

Field Locking

If you do not have a version field in your table, optimistic field locking is another solution. Field locking involves comparing certain fields in the object when updating. If those fields have changed, then the update will fail. JPA does not support field locking, but some JPA providers do support it.

Field locking can also be used when a finer level of locking is desired. For example if one user changes the object's name and another changes the objects address, you may desire for these updates to not conflict, and only desire optimistic lock errors when users change the same fields. You may also only be concerned about conflicts in changes to certain fields, and not desire lock errors from conflicts in the other fields.

Field locking can also be used on legacy schemas, where you cannot add a version column, or to integrate with other applications accessing the same data which are not using optimistic locking (note if the other applications are not also using field locking, you can only detect conflicts in one direction).

There are several types of field locking:

- All fields compared in the update - This can lead to a very big where clause, but will detect any conflicts.
- Selected fields compared in the update - This is useful if conflicts in only certain fields are desired.
- Changed fields compared in the update - This is useful if only changes to the same fields are considered to be conflicts.

If your JPA provider does not support field locking, it is difficult to simulate, as it requires changes to the update SQL. Your JPA provider may allow overriding the update SQL, in which case, `All` or `Selected` field locking may be possible (if you have access to the original values), but `Changed` field locking is more difficult because the update must be dynamic. Another way to simulate field locking is to flush your changes, then refresh the object using a separate `EntityManager` and connection and compare the current values with your original object.

When using field locking it is important to keep the original object that was read. If you read the object in one transaction and send it to a client, then update in another, you are not really locking. Any changes made between the read and write will not be detected. You must keep the original object read managed in an `EntityManager` for your locking to have any effect.

TopLink / EclipseLink : Support field locking through their `@OptimisticLocking` annotation and XML.

Read and Write Locking

It is sometimes desirable to lock something that you did not change. Normally this is done when making a change to one object, that is based on the state of another object, and you wish to ensure that the other object represents the current state of the database at the point of the commit. This is what serializable transaction isolation gives you, but optimistic read and write locking allow this requirement to be met declaratively and optimistically (and without deadlock, concurrency, and open transaction issues).

JPA supports read and write locks through the `EntityManager.lock()` ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#lock\(java.lang.Object, javax.persistence.LockModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#lock(java.lang.Object, javax.persistence.LockModeType))) API. The `LockModeType` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/LockModeType.html>) argument can either be `READ` or `WRITE`. A `READ` lock will ensure that the state of the object does not change on commit. A `WRITE` lock will ensure that this transaction conflicts with any other transaction changing or locking the object. Essentially the `READ` lock check the optimistic version field, and the `WRITE` checks and increments it.

Example of Using the Lock API

```
Employee employee = entityManager.find(Employee.class, id);
employee.setSalary(employee.getSalary() / 2);
entityManager.lock(employee.getManager(), LockModeType.READ);
```

Write locking can also be used to provide object-level locks. If you desire for a change to a dependent object to conflict with any change to the parent object, or any other of its dependent objects, this can be done through write locking. This can also be used to lock relationships, when you change a `OneToMany` or `ManyToMany` relationship you can also force the parent's version to be incremented.

Example of Using the Lock API for Cascaded Locks

```
Employee employee = entityManager.find(Employee.class, id);
employee.getAddress().setCity("Ottawa");
entityManager.lock(employee, LockModeType.WRITE);
```

No Locking a.k.a *Ostrich Locking*

Conceptually people may scoff and be alarmed at the thought of no locking, but it probably the most common form of locking in use. Some call it Ostrich locking as the strategy is to stick your head in the sand and ignore the issue. Most prototypes or small applications frequently do not have the requirement or in most cases the need for locking, and handling what to do when a locking contention does occur is beyond the scope of the application, so best to just ignore the issue.

In general it is probably best in JPA to enable optimistic locking always, as it is fairly simple to do, as least in concept, but what does occur on a conflict without any form of locking? Essentially it is last in wins, so if two users edit the same object, at the same time, the last one to commit will have their changes reflected in the database. This is true, at least for users editing the same fields, but if two users edit different fields in the same object, it depends on the JPA implementation. Some JPA providers only update exactly the fields that were changed, where as other update all fields in the object. So in one case the first user's changes would be overridden, but in the second they would not.

Pessimistic Locking

Pessimistic locking means acquiring a lock on the object before you begin to edit the object, to ensure that no other users are editing the object. Pessimistic locking is typically implemented through using database row locks, such as through the `SELECT ... FOR UPDATE` SQL syntax. The data is read and locked, the changes are made and the transaction is committed, releasing the locks.

JPA does not support pessimistic locking, but some JPA providers do. It is also possible to use JPA native SQL queries to issue `SELECT ... FOR UPDATE` and use pessimistic locking. When using pessimistic locking you must ensure that the object is refreshed when it is locked, locking a potentially stale object is of no use. The SQL syntax for pessimistic locking is database specific, and different databases have different syntax and levels of support, so ensure your database properly supports your locking requirements.

TopLink / EclipseLink : Support pessimistic locking through the "eclipselink.pessimistic-lock" query hint.

The main issues with pessimistic locking is they use database resources, so require a database transaction and connection to be held open for the duration of the edit. This is typically not desirable for interactive web applications. Pessimistic locking can also have concurrency issues and cause deadlocks. The main advantages of pessimistic locking is that once the lock is obtained, it is fairly certain that the edit will be successful. This can be desirable in highly concurrent applications, where optimistic locking may cause too many optimistic locking errors.

There are other ways to implement pessimistic locking, it could be implemented at the application level, or through serializable transaction isolation.

Application level pessimistic locking can be implemented through adding a *locked* field to your object. Before an edit you must update the field to locked (and commit the change). Then you can edit the object, and set the locked field back to false. To avoid conflicts in acquiring the lock, you should also use optimistic locking, to ensure the lock field is not updated to true by another user at the same time.

Serializable Transaction Isolation

Serializable transaction isolation guarantees that anything read in the transaction will not be updated by any other user. Through using serializable transaction isolation and ensuring the data being edited is read in the same transaction, you can achieve pessimistic locking. It is important to ensure the objects are refreshed from the database in the transaction, as editing cached or potentially stale data defeats the point of locking.

Serializable transaction isolation can typically be enabled on the database, some databases even have this as the default. It can also be set on the JDBC Connection, or through native

SQL, but this is database specific and different databases have different levels of support. The main issues with serializable transaction isolation are the same as using `SELECT ... FOR UPDATE` (see above for the gory details), in addition everything read is locked, so you cannot decide to only lock certain objects at certain times, but lock everything all the time. This can be a major concurrency issue for transactions with common read-only data, and can lead to deadlocks.

How database implement serializable transaction isolation differs between databases. Some databases (such as Oracle) can perform serializable transaction isolation in more of an optimistic sense, than the typically pessimistic implementation. Instead of each transaction requiring locks on all the data as it is read, the row versions are not checked until the transaction is committed, if any of the data changed an exception is thrown and the transaction is not allowed to commit.

Basics

A basic attribute is one where the attribute class is a simple type such as `String`, `Number`, `Date` or a primitive. A basic attribute's value can map directly to the column value in the database. The following table summarizes the basic types and the database types they map to.

Java type	Database type
<code>String (char, char[])</code>	<code>VARCHAR (CHAR, VARCHAR2, CLOB, TEXT)</code>
<code>Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte)</code>	<code>NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)</code>
<code>int, long, float, double, short, byte</code>	<code>NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)</code>
<code>byte[]</code>	<code>VARBINARY (BINARY, BLOB)</code>
<code>boolean (Boolean)</code>	<code>BOOLEAN (BIT, SMALLINT, INT, NUMBER)</code>
<code>java.util.Date</code>	<code>TIMESTAMP (DATE, DATETIME)</code>
<code>java.sql.Date</code>	<code>DATE (TIMESTAMP, DATETIME)</code>
<code>java.sql.Time</code>	<code>TIME (TIMESTAMP, DATETIME)</code>
<code>java.sql.Timestamp</code>	<code>TIMESTAMP (DATETIME, DATE)</code>
<code>java.util.Calendar</code>	<code>TIMESTAMP (DATETIME, DATE)</code>
<code>java.lang.Enum</code>	<code>NUMERIC (VARCHAR, CHAR)</code>
<code>java.util.Serializable</code>	<code>VARBINARY (BINARY, BLOB)</code>

In JPA a basic attribute is mapped through the `@Basic` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Basic.html>) annotation or the `<basic>` element. The types and conversions supported depend on the JPA implementation and database platform. Some JPA implementations may support conversion between many different data-types or additional types, or have extended type conversion support, see the advanced section for more details. Any basic attribute using a type that does not map directly to a database type can be serialized to a binary database type.

The easiest way to map a basic attribute in JPA is to do nothing. Any attributes that have no other annotations and do not reference other entities will be automatically mapped as

basic, and even serialized if not a basic type. The column name for the attribute will be defaulted, named the same as the attribute name, as uppercase. Sometimes auto-mapping can be unexpected if you have an attribute in your class that you did not intend to have persisted. You must mark any such non-persistent fields using the `@Transient` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Transient.html>) annotation or `<transient>` element.

Although auto-mapping makes rapid prototyping easy, you typically reach a point where you want control over your database schema. To specify the column name for a basic attribute the `@Column` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Column.html>) annotation or `<column>` element is used. The column annotation also allows for other information to be specified such as the database type, size, and some constraints.

Example of basic mapping annotations

```
@Entity
public class Employee {
    // Id mappings are also basic mappings.
    @Id
    @Column(name="ID")
    private long id;
    @Basic
    @Column(name="F_NAME")
    private String firstName;
    // The @Basic is not required in general because it is the default.
    @Column(name="L_NAME")
    private String lastName;
    // Any un-mapped field will be automatically mapped as basic and
    column name defaulted.
    private BigDecimal salary;
    // Non-persistent fields must be marked as transient.
    @Transient
    private EmployeeService service;
    ...
}
```

Common Problems

Translating Values

See Conversion

Example of basic mapping XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="ID"/>
    </id>
    <basic name="firstName">
```

```
        <column name="F_NAME" />
    </basic>
    <basic name="lastName">
        <column name="L_NAME" />
    </basic>
    <transient name="service" />
</attributes>
</entity>
```

Truncated Data

A common issue is that data, such as Strings, written from the object are truncated when read back from the database. This is normally caused by the column length not being large enough to handle the object's data. In Java there is no maximum size for a String, but in a database VARCHAR field, there is a maximum size. You must ensure that the length you set in your column when you create the table is large enough to handle any object value. For very large Strings CLOBs can be used, but in general CLOBs should not be over used, as they are less efficient than a VARCHAR.

If you use JPA to generate your database schema, you can set the column length through the Column annotation or element, see Column Definition and Schema Generation.

How to map timestamp with timezones?

See Timezones

How to map XML data-types?

See Custom Types

How to map Struct and Array types?

See Custom Types

How to map custom database types?

See Custom Types

How to excluded fields from INSERT or UPDATE statements, or default values in triggers?

See Insertable, Updatable

Advanced

Temporal, Dates, Times, Timestamps and Calendars

Dates, times, and timestamps are common types both in the database and in Java, so in theory mappings these types should be simple, right? Well sometimes this is the case and just a normal `Basic` mapping can be used, however sometimes it becomes more complex.

Some databases do not have `DATE` and `TIME` types, only `TIMESTAMP` fields, however some do have separate types, and some just have `DATE` and `TIMESTAMP`. Originally in Java 1.0, Java only had a `java.util.Date` type, which was both a date, time and milliseconds. In Java 1.1 this was expanded to support the common database types with `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`, then to support internationalization Java created the `java.util.Calendar` type and virtually deprecated (almost all of the methods) the old date types (which JDBC still uses).

If you map a Java `java.sql.Date` type to a database `DATE`, this is just a basic mapping and you should not have any issues (ignore Oracle's `DATE` type that is/was a timestamp for now). You can also map `java.sql.Time` to `TIME`, and `java.sql.Timestamp` to `TIMESTAMP`. However if you have a `java.util.Date` or `java.util.Calendar` in Java and wish to map it to a `DATE` or `TIME`, you may need to indicate that the JPA provider perform some sort of conversion for this. In JPA the `@Temporal` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Temporal.html>) annotation or `<temporal>` element is used to map this. You can indicate that just the `DATE` or `TIME` portion of the date/time value be stored to the database. You could also use `Temporal` to map a `java.sql.Date` to a `TIMESTAMP` field, or any other such conversion.

Example of temporal annotation

```
@Entity
public class Employee {
    ...
    @Basic
    @Temporal (DATE)
    private Calendar startDate;
    ...
}
```

Example of temporal XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    ...
    <basic name="startDate">
      <temporal>DATE</temporal>
    </basic>
  </attributes>
</entity>
```


Milliseconds

The precision of milliseconds is different for different temporal classes and database types, and on different databases. The `java.util.Date` and `Calendar` classes support milliseconds. The `java.sql.Date` and `java.sql.Time` classes do not support milliseconds. The `java.sql.Timestamp` class supports nanoseconds.

On many databases the `TIMESTAMP` type supports milliseconds. On Oracle prior to Oracle 9, there was only a `DATE` type, which was a date and a time, but had no milliseconds. Oracle 9 added a `TIMESTAMP` type that has milliseconds (and nanoseconds), and now treats the old `DATE` type as only a date, so be careful using it as a timestamp. MySQL has `DATE`, `TIME` and `DATETIME` types. DB2 has a `DATE`, `TIME` and `TIMESTAMP` types, the `TIMESTAMP` supports microseconds. Sybase and SQL Server just have a `DATETIME` type which has milliseconds, but at least on some versions has precision issues, it seems to store an estimate of the milliseconds, not the exact value.

If you use timestamp version locking you need to be very careful of your milliseconds precision. Ensure your database supports milliseconds precisely otherwise you may have issues, especially if the value is assigned in Java, then differs what gets stored on the database, which will cause the next update to fail for the same object.

In general I would not recommend using a timestamp and as primary key or for version locking. There are too many database compatibility issues, as well as the obvious issue of not supporting two operations in the same millisecond.

Timezones

Temporals become a lot more complex when you start to consider time zones, internationalization, eras, locals, day-light savings time, etc. In Java only `Calendar` supports time zones. Normally a `Calendar` is assumed to be in the local time zone, and is stored and retrieved from the database with that assumption. If you then read that same `Calendar` on another computer in another time zone, the question is if you will have the same `Calendar` or will you have the `Calendar` of what the original time would have been in the new time zone? It depends on if the `Calendar` is stored as the GMT time, or the local time, and if the time zone was stored in the database.

Some databases support time zones, but most database types do not store the time zone. Oracle has two special types for timestamps with time zones, `TIMESTAMPTZ` (time zone is stored) and `TIMESTAMPLTZ` (local time zone is used). Some JPA providers may have extended support for storing `Calendar` objects and time zones.

TopLink, EclipseLink : Support the Oracle `TIMESTAMPTZ` and `TIMESTAMPLTZ` types using the `@TypeConverter` annotation and XML.

Forum Posts

- Investigation of storing timezones in MySQL (<http://www.nabble.com/MySQL's-datetime-and-time-zones--td21006801.html>)

Enums

Java Enums (<https://java.sun.com/java/5/docs/api/java/lang/Enum.html>) are typically used as constants in an object model. For example an `Employee` may have a `gender` of enum type `Gender` (`MALE`, `FEMALE`).

By default in JPA an attribute of type Enum will be stored as a `Basic` to the database, using the integer Enum values as codes (i.e. 0, 1). JPA also defines an `@Enumerated` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Enumerated.html>) annotation and `<enumerated>` element (on a `<basic>`) to define an Enum attribute. This can be used to store the Enum as the `STRING` value of its name (i.e. "MALE", "FEMALE").

For translating Enum types to values other than the integer or String name, such as character constants, see [Translating Values](#).

Example of enumerated annotation

```
public enum Gender {
    MALE,
    FEMALE
}

@Entity
public class Employee {
    ...
    @Basic
    @Enumerated(STRING)
    private Gender gender;
    ...
}
```

Example of enumerated XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    ...
    <basic name="gender">
      <enumerated>STRING</enumerated>
    </basic>
  </attributes>
</entity>
```

LOBs, BLOBs, CLOBs and Serialization

A LOB is a Large Object, such as a BLOB (Binary LOB), or a CLOB (Character LOB). It is a database type that can store a large binary or string value, as the normal `VARCHAR` or `VARBINARY` types typically have size limitations. A LOB is often stored as a locator in the database table, with the actual data stored outside of the table. In Java a `CLOB` will normally map to a `String`, and a `BLOB` will normally map to a `byte[]`, although a `BLOB` may also represent some serialized object.

By default in JPA any `Serializable` attribute that is not a relationship or a basic type (String, Number, temporal, primitive), will be serialized to a BLOB field.

JPA defines the `@Lob` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Lob.html>) annotation and `<lob>` element (on a `<basic>`) to define that an attribute maps to a LOB type in the database. The annotation is just a hint to the JPA implementation that this attribute will be stored in a LOB, as LOBs may need to be persisted specially. Sometimes just mapping the LOB as a normal `Basic` will work fine as well.

Various databases and JDBC drivers have various limits for LOB sizes. Some JDBC drivers have issues beyond 4k, 32k or 1meg. The Oracle thin JDBC drivers had a 4k limitation in some versions for binding LOB data. Oracle provided a workaround for this limitation, which some JPA providers support. For reading LOBs, some JDBC drivers prefer using streams, some JPA providers also support this option.

Typically the entire LOB will be read and written for the attribute. For very large LOBs reading the value always, or reading the entire value may not be desired. The fetch type of the `Basic` could be set to `LAZY` to avoid reading a LOB unless accessed. Support for `LAZY` fetching on `Basic` is optional in JPA, so some JPA providers may not support it. A workaround, which is often a good idea in general given the large performance cost of LOBs, is to store the LOB in a separate table and class and define a `OneToOne` to the LOB object instead of a `Basic`. If the entire LOB is never desired to be read, then it should not be mapped. It is best to use direct JDBC to access and stream the LOB in this case. It may be possible to map the LOB to a `java.sql.Blob`/`java.sql.Clob` in your object to avoid reading the entire LOB, but these require a live connection, so may have issues with detached objects.

Example of lob annotation

```
@Entity
public class Employee {
    ...
    @Basic(fetch=FetchType.LAZY)
    @Lob
    private Image picture;
    ...
}
```

Example of lob XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    ...
    <basic name="picture" fetch="LAZY">
      <lob/>
    </basic>
  </attributes>
</entity>
```

Lazy Fetching

The `fetch` attribute can be set on a `Basic` mapping to use LAZY fetching. By default all `Basic` mappings are EAGER, which means the column is selected whenever the object is selected. By setting the `fetch` to LAZY, the column will not be selected with the object. If the attribute is accessed, then the attribute value will be selected in a separate database select. Support for LAZY is an optional feature of JPA, so some JPA providers may not support it. Typically support for lazy on basics will require some form of byte code weaving, or dynamic byte code generation, which may have issues in certain environments or JVMs, or may require preprocessing your application's persistence unit jar.

Only attributes that are rarely accessed should be marked lazy, as accessing the attribute causes a separate database select, which can hurt performance. This is especially true if a large number of objects is queried. The original query will require one database select, but if each object's lazy attribute is accessed, this will require n database selects, which can be a major performance issue.

Using lazy fetching on basics is similar to the concept of fetch groups. Lazy basics is basically support for a single default fetch group. Some JPA providers support fetch groups in general, which allow more sophisticated control over what attributes are fetched per query.

TopLink, EclipseLink : Support lazy basics and fetch groups. Fetch groups can be configured through the EclipseLink API using the `FetchGroup` class.

Optional

A `Basic` attribute can be `optional` if its value is allowed to be null. By default everything is assumed to be optional, except for an `Id`, which can not be optional. `Optional` is basically only a hint that applies to database schema generation, if the persistence provider is configured to generate the schema. It adds a `NOT NULL` constraint to the column if `false`. Some JPA providers also perform validation of the object for optional attributes, and will throw a validation error before writing to the database, but this is not required by the JPA specification. `Optional` is defined through the `optional` attribute of the `Basic` annotation or element.

Column Definition and Schema Generation

There are various attributes on the `Column` (<https://java.sun.com/javase/5/docs/api/javax/persistence/Column.html>) annotation and element for database schema generation. If you do not use JPA to generate your schema you can ignore these. Many JPA providers do provide the feature of auto generation of the database schema. By default the Java types of the object's attributes are mapped to their corresponding database type for the database platform you are using. You may require configuring your database platform with your provider (such as a `persistence.xml` property) to allow schema generation for your database, as many database use different type names.

The `columnDefinition` attribute of `Column` can be used to override the default database type used, or enhance the type definition with constraints or other such DDL. The `length`, `scale` and `precision` can also be set to override defaults. Since the defaults for the `length` are just defaults, it is normally a good idea to set these to be correct for your data model's expected data, to avoid data truncation. The `unique` attribute can be used to define

a unique constraint on the column, most JPA providers will automatically define primary key and foreign key constraints based on the `Id` and relationship mappings.

JPA does not define any options to define an index. Some JPA providers may provide extensions for this. You can also create your own indexes through native queries

Example of column annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="ID")
    private long id;
    @Column(name="SSN" unique=true optional=false)
    private long ssn;
    @Column(name="F_NAME" length=100)
    private String firstName;
    @Column(name="L_NAME" length=200)
    private String lastName;
    @Column(name="SALARY" scale=10 precision=2)
    private BigDecimal salary;
    @Column(name="S_TIME" columnDefinition="TIMESTAMPTZ")
    private Calendar startTime;
    @Column(name="E_TIME" columnDefinition="TIMESTAMPTZ")
    private Calendar endTime;
    ...
}
```

Example of column XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="ID"/>
    </id>
    <basic name="ssn">
      <column name="SSN" unique="true" optional="false"/>
    </basic>
    <basic name="firstName">
      <column name="F_NAME" length="100"/>
    </basic>
    <basic name="lastName">
      <column name="L_NAME" length="200"/>
    </basic>
    <basic name="startTime">
      <column name="S_TIME" columnDefinition="TIMESTAMPTZ"/>
    </basic>
    <basic name="endTime">
      <column name="E_TIME" columnDefinition="TIMESTAMPTZ"/>
    </basic>
  </attributes>
</entity>
```

```
</attributes>
</entity>
```

Insertable, Updatable / Read Only Fields / Returning

The `Column` annotation and XML element defines `insertable` and `updatable` options. These allow for this column, or foreign key field to be omitted from the SQL INSERT or UPDATE statement. These can be used if constraints on the table prevent insert or update operations. They can also be used if multiple attributes map to the same database column, such as with a foreign key field through a `ManyToOne` and `Id` or `Basic` mapping. Setting both `insertable` and `updatable` to false, effectively mark the attribute as read-only.

`insertable` and `updatable` can also be used in the database table defaults, or auto assigns values to the column on insert or update. Be careful in doing this though, as this means that the object's values will be out of synch with the database, unless it is refreshed. For `IDENTITY` or auto assigned id columns a `GeneratedValue` should normally be used, instead of setting `insertable` to false. Some JPA providers also support returning auto assigned fields values from the database after insert or update operations. The cost of refreshing or returning fields back into the object can effect performance, so it is normally better to initialize field values in the object model, not in the database.

`TopLink`, `EclipseLink` : Support returning insert and update values back into the object using the `ReturnInsert` and `ReturnUpdate` annotations and XML elements.

Conversion

A common problem in storing values to the database is that the value desired in Java differs from the value used in the database. Common examples include using a `boolean` in Java and a `0`, `1` or a `'T'`, `'F'` in the database. Other examples are using a `String` in Java and a `DATE` in the database.

One way to accomplish this is to translate the data through property get/set methods.

```
@Entity
public class Employee {
    ...
    private boolean isActive;
    ...
    @Transient
    public boolean getIsActive() {
        return isActive;
    }
    public void setIsActive(boolean isActive) {
        this.isActive = isActive;
    }
    @Basic
    private String getIsActiveValue() {
        if (isActive) {
            return "T";
        } else {
            return "F";
        }
    }
}
```

```
    }  
  }  
  private void setIsActiveValue(String isActive) {  
    this.isActive = "T".equals(isActive);  
  }  
}
```

Also for translating date/times see, Temporals.

As well some JPA providers have support for this.

TopLink, EclipseLink : Support translation using the @Convert, @Converter, @ObjectConverter and @TypeConverter annotations and XML.

Custom Types

JPA defines support for most common database types, however some databases and JDBC driver have additional types that may require additional support.

Some custom database types include:

- TIMESTAMPTZ, TIMESTAMPLTZ (Oracle)
- TIMESTAMP WITH TIMEZONE (Postgres)
- XMLTYPE (Oracle)
- XML (DB2)
- NCHAR, NVARCHAR, NCLOB (Oracle)
- Struct (STRUCT Oracle)
- Array (VARRAY Oracle)
- BINARY_INTEGER, DEC, INT, NATURAL, NATURALN, BOOLEAN (Oracle)
- POSITIVE, POSITIVEN, SIGNTYPE, PLS_INTEGER (Oracle)
- RECORD, TABLE (Oracle)
- SDO_GEOMETRY (Oracle)
- LOBs (Oracle thin driver)

To handle persistence to custom database types either custom hooks are required in your JPA provider, or you need to mix raw JDBC code with your JPA objects. Some JPA provider provide custom support for many custom database types, some also provide custom hooks for adding your own JDBC code to support a custom database type.

TopLink, EclipseLink : Support several custom database types including, TIMESTAMPTZ, TIMESTAMPLTZ, XMLTYPE, NCHAR, NVARCHAR, NCLOB, object-relational Struct and Array types, PLSQL types, SDO_GEOMETRY and LOBs.

Relationships

A relationship is a reference from one object to another. In Java, relationships are defined through object references (pointers) from a source object to the target object. Technically, in Java there is no difference between a relationship to another object and a "relationship" to a data attribute such as a String or Date (primitives are different), as both are pointers; however, logically and for the sake of persistence, data attributes are considered part of the object, and references to other persistent objects are considered relationships.

In a relational database relationships are defined through foreign keys. The source row contains the primary key of the target row to define the relationship (and sometimes the

inverse). A query must be performed to read the target objects of the relationship using the foreign key and primary key information.

In Java, if a relationship is to a collection of other objects, a `Collection` or array type is used in Java to hold the contents of the relationship. In a relational database, collection relations are either defined by the target objects having a foreign key back to the source object's primary key, or by having an intermediate join table to store the relationship (both objects' primary keys).

All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA Relationship Types

- `OneToOne` - A unique reference from one object to another, inverse of a `OneToOne`.
- `ManyToOne` - A reference from one object to another, inverse of a `OneToMany`.
- `OneToMany` - A `Collection` or `Map` of objects, inverse of a `ManyToOne`.
- `ManyToMany` - A `Collection` or `Map` of objects, inverse of a `ManyToMany`.
- `Embedded` - A reference to an object that shares the same table of the parent.
- `ElementCollection` - JPA 2.0, a `Collection` or `Map` of `Basic` or `Embeddable` objects, stored in a separate table.

This covers the majority of types of relationships that exist in most object models. Each type of relationship also covers multiple different implementations, such as `OneToMany` allowing either a join table, or foreign key in the target, and collection mappings also allow `Collection` types and `Map` types. There are also other possible complex relationship types, see [Advanced Relationships](#).

Lazy Fetching

The cost of retrieving and building an object's relationships, far exceeds the cost of selecting the object. This is especially true for relationships such as `manager` or `managedEmployees` such that if any employee were selected it would trigger the loading of every employee through the relationship hierarchy. Obviously this is a bad thing, and yet having relationships in objects is very desirable.

The solution to this issue is lazy fetching (lazy loading). Lazy fetching allows the fetching of a relationship to be deferred until it is accessed. This is important not only to avoid the database access, but also to avoid the cost of building the objects if they are not needed.

In JPA lazy fetching can be set on any relationship using the `fetch` attribute. The `fetch` can be set to either `LAZY` or `EAGER` as defined in the `FetchType` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/FetchType.html>) enum. The default fetch type is `LAZY` for all relationships except for `OneToOne` and `ManyToOne`, but in general it is a good idea to make every relationship `LAZY`. The `EAGER` default for `OneToOne` and `ManyToOne` is for implementation reasons (more difficult to implement), not because it is a good idea. Technically in JPA `LAZY` is just a hint, and a JPA provider is not required to support it, however in reality all main JPA providers support it, and they would be pretty useless if they did not.

Example of a lazy one to one relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="ADDR_ID")
    private Address address;
    ...
}
```

Example of a lazy one to one relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-one name="address" fetch="LAZY">
      <join-column name="ADDR_ID"/>
    </one-to-one>
  </attributes>
</entity>
```

Magic

Lazy fetching normally involves some sort of *magic* in the JPA provider to transparently fault in the relationships as they are accessed. The typical magic for collection relationships is for the JPA provider to set the relationships to its own `Collection`, `List`, `Set` or `Map` implementation. When any (or most) method is accessed on this collection proxy, it loads the real collection and forwards the method. This is why JPA requires that all collection relationships use one of the collection interfaces (although some JPA providers support collection implementations too).

For `OneToOne` and `ManyToOne` relationships the magic normally involves some sort of byte code manipulation of the entity class, or creation of a subclass. This allows the access to the field or get/set methods to be intercepted, and for the relationships to be first retrieved before allowing access to the value. Some JPA providers use different methods, such as wrapping the reference in a proxy object, although this can have issues with null values and primitive methods. To perform the byte code magic normally an agent or post-processor is required. Ensure that you correctly use your providers agent or post-processor otherwise lazy may not work. You may also notice additional variables when in a debugger, but in general debugging will still work as normal.

Basics

A `Basic` attribute can also be made `LAZY`, but this is normally a different mechanism than lazy relationships, and should normally be avoided unless the attribute is rarely accessed.

See Basic Attributes : Lazy Fetching.

Serialization, and Detaching

A major issue with lazy relationships, is ensuring that the relationship is still available after the object has been detached, or serialized. For most JPA providers, after serialization any lazy relationship that was not instantiated will be broken, and either throw an error when accessed, or return null.

The naive solution is to make every relationship eager. Serialization suffers from the same issue as persistence, in that you can very easily serialize your entire database if you have no lazy relationships. So lazy relationships are just a necessary for serialization, as they are for database access, however you need to ensure you have everything you will need after serialization instantiated upfront. You may mark only the relationships that you think you will need after serialization as `EAGER`, this will work, however there are probably may cases when you do not need these relationships.

A second solution is to access any relationship you will need before returning the object for serialization. This has the advantage of being use case specific, so different use cases can instantiate different relationships. For collection relationships sending `size()` is normally the best way to ensure a lazy relationship is instantiated. For `OneToOne` and `ManyToOne` relationships, normally just accessing the relationship is enough (i.e. `employee.getAddress()`), although for some JPA providers that use proxies you may need to send the object a message (i.e. `employee.getAddress().hashCode()`).

A third solution is to use the JPQL `JOIN FETCH` for the relationship when querying the objects. A join fetch will normally ensure the relationship has been instantiated. Some caution should be used with join fetch however, as it can become inefficient if used on collection relationships, especially multiple collection relationships as it requires an n^2 join on the database.

Some JPA providers may also provide certain query hints, or other such serialization options.

The same issue can occur without serialization, if a detached object is accessed after the end of the transaction. Some JPA providers allow lazy relationship access after the end of the transaction, or after the `EntityManager` has been closed, however some do not. If your JPA provider does not, then you may require that you ensure you have instantiated all the lazy relationships that you will need before ending the transaction.

Eager Join Fetching

One common misconception is that `EAGER` means that the relationship should be join fetched, i.e. retrieved in the same SQL `SELECT` statement as the source object. Some JPA providers do implement eager this way. However, just because something is desired to be loaded, does not mean that it should be join fetched. Consider `Employee - Phone`, a `Phone`'s `employee` reference is made `EAGER` as the `employee` is almost always loaded before the `phone`. However when loading the `phone`, you do not want to join the `employee`, the `employee` has already been read and is already in the cache or persistence context. Also

just because you want two collection relationships loaded, does not mean you want them join fetch which would result in a very inefficient join that would return n^2 data.

Join fetching is something that JPA currently only provides through JPQL, which is normally the correct place for it, as each use case has different relationship requirements. Some JPA providers also provide a join fetch option at the mapping level to always join fetch a relationship, but this is normally not the same thing as EAGER. Join fetching is not normally the most efficient way to load a relationship anyway, normally batch reading a relationship is much more efficient when supported by your JPA provider.

See Join Fetching

See Batch Reading

Cascading

Relationship mappings have a `cascade` option that allows the relationship to be cascaded for common operations. `cascade` is normally used to model dependent relationships, such as `Order -> OrderLine`. Cascading the `orderLines` relationship allows for the `Order`'s `-> OrderLines` to be persisted, removed, merged along with their parent.

The following operations can be cascaded, as defined in the `CascadeType` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/CascadeType.html>) enum:

- **PERSIST** - Cascaded the `EntityManager.persist()` operation. If `persist()` is called on the parent, and the child is also new, it will also be persisted. If it is existing, nothing will occur, although calling `persist()` on an existing object will still cascade the `persist` operation to its dependents. If you persist an object, and it is related to a new object, and the relationship does not cascade `persist`, then an exception will occur. This may require that you first call `persist` on the related object before relating it to the parent. General it may seem odd, or be desirable to always cascade the `persist` operation, if a new object is related to another object, then it should probably be persisted. There is most likely not a major issue with always cascading `persist` on every relationship, although it may have an impact on performance. Calling `persist` on a related object is not required, on commit any related object whose relationship is `cascade persist` will automatically be persisted. The advantage of calling `persist` up front is that any generated ids will (unless using identity) be assigned, and the `prePersist` event will be raised.
- **REMOVE** - Cascaded the `EntityManager.remove()` operation. If `remove()` is called on the parent then the child will also be removed. This should only be used for dependent relationships. Note that only the `remove()` operation is cascaded, if you remove a dependent object from a `OneToMany` collection it will not be deleted, JPA requires that you explicitly call `remove()` on it. Some JPA providers may support an option to have objects removed from dependent collection deleted, JPA 2.0 also defines an option for this.
- **MERGE** - Cascaded the `EntityManager.merge()` operation. If `merge()` is called on the parent, then the child will also be merged. This should normally be used for dependent relationships. Note that this only effects the cascading of the merge, the relationship reference itself will always be merged. This can be a major issue if you use `transient` variables to limit serialization, you may need to manually merge, or reset transient relationships in this case. Some JPA providers provide additional merge operations.
- **REFRESH** - Cascaded the `EntityManager.refresh()` operation. If `refresh()` is called on the parent then the child will also be refreshed. This should normally be used for

dependent relationships. Be careful enabling this for all relationships, as it could cause changes made to other objects to be reset.

- ALL - Cascaded all the above operations.

Example of a cascaded one to one relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="ADDR_ID")
    private Address address;
    ...
}
```

Example of a cascaded one to one relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-one name="address">
      <join-column name="ADDR_ID"/>
      <cascade>
        <cascade-all/>
      </cascade>
    </one-to-one>
  </attributes>
</entity>
```

Target Entity

Relationship mappings have a `targetEntity` attribute that allows the reference class (target) of the relationship to be specified. This is normally not required to be set as it is defaulted from the field type, get method return type, or collection's generic type.

This can also be used if your field uses a public interface type, for example field is interface `Address`, but the mapping needs to be to implementation class `AddressImpl`. Another usage is if your field is a superclass type, but you want to map the relationship to a subclass.

Example of a target entity relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(targetEntity=Phone.class)
    @JoinColumn(name="OWNER_ID")
    private List phones;
}
```

```
...  
}
```

Example of a target entity relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">  
  <attributes>  
    <id name="id"/>  
    <one-to-many name="phones" target-entity="org.acme.Phone">  
      <join-column name="OWNER_ID"/>  
    </one-to-many>  
  </attributes>  
</entity>
```

Collections

Collection mappings include `OneToMany`, `ManyToMany`, and in JPA 2.0 `ElementCollection`. JPA requires that the type of the collection field or get/set methods be one of the Java collection interfaces, `Collection`, `List`, `Set`, or `Map`.

Collection Implementations

Your field should not be of a collection implementation type, such as `ArrayList`. Some JPA providers may support using collection implementations, many support `EAGER` collection relationships to use the implementation class. You can set any implementation as the instance value of the collection, but when reading an object from the database, if it is `LAZY` the JPA provider will normally put in a special `LAZY` collection.

Duplicates

A `List` in Java supports duplicate entries, and a `Set` does not. In the database, duplicates are generally not supported. Technically it could be possible if a `JoinTable` is used, but JPA does not require duplicates to be supported, and most providers do not.

If you require duplicate support, you may need to create an object that represents and maps to the join table. This object would still require a unique `Id`, such as a `GeneratedValue`. See [Mapping a Join Table with Additional Columns](#).

Ordering

JPA allows the collection values to be ordered by the database when retrieved. This is done through the `@OrderBy` (<https://java.sun.com/javase/5/docs/api/javax/persistence/OrderBy.html>) annotation or `<order-by>` XML element.

The value of the `OrderBy` is a JPQL `ORDER BY` string. The can be an attribute name followed by `ASC` or `DESC` for ascending or descending ordering. You could also use a path or nested attribute, or a `"` for multiple attributes. If no `OrderBy` value is given it is assumed to be the `Id` of the target object.

The `OrderBy` value must be a mapped attribute of the target object. If you want to have an ordered `List` you need to add an `index` attribute to your target object and an `index` column to it's table. You will also have to ensure you set the index values. JPA 2.0 will have extended support for an ordered `List` using an `OrderColumn`.

Note that using an `OrderBy` does not ensure the collection is ordered in memory. You are responsible for adding to the collection in the correct order. Java does define a `SortedSet` interface and `TreeSet` collection implementation that does maintain an order. JPA does not specifically support `SortedSet`, but some JPA providers may allow you to use a `SortedSet` or `TreeSet` for your collection type, and maintain the correct ordering. By default these require your target object to implement the `Comparable` interface, or set a `Comparator`. You can also use the `Collections.sort()` method to sort a `List` when required. One option to sort in memory is to use property access and in your set and add methods call `Collections.sort()`.

Example of a collection order by annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany
    @OrderBy("areaCode")
    private List<Phone> phones;
    ...
}
```

Example of a collection order by XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phones">
      <order-by>areaCode</order-by>
    </one-to-many>
  </attributes>
</entity>
```

Order Column (JPA 2.0)

JPA 2.0 adds support for an `OrderColumn`. An `OrderColumn` can be used to define an order `List` on any collection mapping. It is defined through the `@OrderColumn` annotation or `<order-column>` XML element.

The `OrderColumn` is maintained by the mapping and should not be an attribute of the target object. The table for the `OrderColumn` depends on the mapping. For a `OneToMany` mapping it will be in the target object's table. For a `ManyToMany` mapping or a `OneToMany` using a `JoinTable` it will be in the join table. For an `ElementCollection` mapping it will be in the target table.

Example of a collection order column annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany
    @OrderColumn(name="INDEX")
    private List<Phone> phones;
    ...
}
```

Example of a collection order column XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id" />
        <one-to-many name="phones">
            <order-column name="INDEX" />
        </one-to-many>
    </attributes>
</entity>
```

Common Problems

Object corruption, one side of the relationship is not updated after updating the other side

A common problem with bi-directional relationships is the application updates one side of the relationship, but the other side does not get updated, and becomes out of synch. In JPA, as in Java in general it is the responsibility of the application, or the object model to maintain relationships. If your application adds to one side of a relationship, then it must add to the other side.

This is commonly resolved through `add` or `set` methods in the object model that handle both sides of the relationships, so the application code does not need to worry about it. There are two ways to go about this, you can either only add the relationship maintenance code to one side of the relationship, and only use the setter from one side (such as making the other side protected), or add it to both sides and ensure you avoid a infinite loop.

For example:

```
public class Employee {
    private List phones;
    ...
    public void addPhone(Phone phone) {
        this.phones.add(phone);
        if (phone.getOwner() != this) {
            phone.setOwner(this);
        }
    }
}
```

```
    ...
}

public class Phone {
    private Employee owner;
    ...
    public void setOwner(Employee employee) {
        this.owner = employee;
        if (!employee.getPhones().contains(this)) {
            employee.getPhones().add(this);
        }
    }
}
    ...
}
```

The code is similar for bi-directional `OneToOne` and `ManyToMany` relationships.

Some expect the JPA provider to have magic that automatically maintains relationships. This was actually part of the EJB CMP 2 specification. However the issue is if the objects are detached or serialized to another VM, or new objects are related before being managed, or the object model is used outside the scope of JPA, then the magic is gone, and the application is left figuring things out, so in general it may be better to add the code to the object model. However some JPA providers do have support for automatically maintaining relationships.

In some cases it is undesirable to instantiate a large collection when adding a child object. One solution is to not map the bi-directional relationship, and instead query for it as required. Also some JPA provides optimize their lazy collection objects to handle this case, so you can still add to the collection without instantiating it.

Poor performance, excessive queries

This most common issue leading to poor performance is the usage of `EAGER` relationships. This requires the related objects to be read when the source objects are read. So for example reading the president of the company with `EAGER managedEmployees` will cause every `Employee` in the company to be read. The solution is to always make all relationships `LAZY`. By default `OneToMany` and `ManyToMany` are `LAZY` but `OneToOne` and `ManyToOne` are not, so make sure you configure them to be. See, [Lazy Fetching](#). Sometimes you have `LAZY` configured but it does not work, see [Lazy is not working](#)

Another common problems is the $n+1$ issue. For example consider that you read all `Employee` objects then access their `Address`. Since each `Address` is accessed separately this will cause $n+1$ queries, which can be a major performance problem. This can be solved through [Join Fetching](#) and [Batch Reading](#).

Lazy is not working

Lazy OneToOne and ManyToOne relationships typically require some form of weaving or byte-code generation. Normally when running in JSE an agent option is required to allow the byte-code weaving, so ensure you have the agent configured correctly. Some JPA providers perform dynamic subclass generation, so do not require an agent.

Example agent

```
java -javaagent:eclipselink.jar ...
```

Some JPA providers also provide static weaving instead, or in addition to dynamic weaving. For static weaving some preprocessor must be run on your JPA classes.

When running in JEE lazy should normally work, as the class loader hook is required by the EJB specification. However some JEE providers may not support this, so static weaving may be required.

Also ensure that you are not accessing the relationship when you shouldn't be. For example if you use property access, and in your set method access the related lazy value, this will cause it to be loaded. Either remove the set method side-effects, or use field access.

Broken relationships after serialization

If your relationship is marked as lazy then if it has not been instantiated before the object is serialized, then it may not get serialized. This may cause an error, or return null if it is accessed after deserialization.

See, [Serialization](#), and [Detaching](#)

Dependent object removed from OneToMany collection is not deleted

When you remove an object from a collection, if you also want the object deleted from the database you must call `remove()` on the object. In JPA 1.0 even if your relationship is cascade REMOVE, you still must call `remove()`, only the remove of the parent object is cascaded, not removal from the collection.

JPA 2.0 will provide an option for having removes from the collection trigger deletion. Some JPA providers support an option for this in JPA 1.0.

See, [Cascading](#)

My relationship target is an interface

If your relationship field's type is a public interface of your class, and only has a single implementer, then this is simple to solve, you just need to set a `targetEntity` on your mapping. See, [Target Entity](#).

If your interface has multiple implementers, then this is more complex. JPA does not directly support mapping interfaces. One solution is to convert the interface to an abstract class and use inheritance to map it. You could also keep the interface, create the abstract class and make sure each implementer extends it, and set the `targetEntity` to be the abstract class.

Another solution is to define virtual attributes using get/set methods for each possible implementer, and map these separately, and mark the interface get/set as transient. You could also not map the attribute, and instead query for it as required.

Some JPA providers have support for interfaces and variable relationships.

TopLink, EclipseLink : Support variable relationships through their @VariableOneToOne annotation and XML. Mapping to and querying interfaces are also supported through their ClassDescriptor's InterfacePolicy API.

Advanced

Advanced Relationships

JPA 2.0 Relationship Enhancements

- ElementCollection - A Collection or Map of Embeddable or Basic values.
- Map Columns - A OneToMany or ManyToMany or ElementCollection that has a Basic, Embeddable or Entity key not part of the target object.
- Order Columns - A OneToMany or ManyToMany or ElementCollection can now have a OrderColumn that defines the order of the collection when a List is used.
- Unidirectional OneToMany - A OneToMany no longer requires the ManyToOne inverse relationship to be defined.

Other Types of Relationships

- Variable OneToOne, ManyToOne - A reference to an interface or common unmapped inheritance class that has multiple distinct implementors.
- Variable OneToMany, ManyToMany - A Collection or Map of heterogeneous objects that share an interface or common unmapped inheritance class that has multiple distinct implementors.
- Nested collection relationships, such as an array of arrays, Listof Lists, or Map of Maps, or other such combinations. Object-Relational Data Type - Relationships stored in the database using STRUCT, VARRAY, REF, or NESTEDTABLE types. XML relationships - Relationships stored as XML documents.

Maps

Java defines the Map interface to represent collections whose values are indexed on a key. There are several Map implementations, the most common is HashMap, but also Hashtable and TreeMap.

JPA allows a Map to be used for any collection mapping including, OneToMany, ManyToMany and ElementCollection. JPA requires that the Map interface be used as the attribute type, although some JPA providers may also support using Map implementations.

In JPA 1.0 the map key must be a mapped attribute of the collection values. The @MapKey (<https://java.sun.com/javaee/5/docs/api/javax/persistence/MapKey.html>) annotation or <map-key> XML element is used to define a map relationship. If the MapKey is not specified it defaults to the target object's Id.

Example of a map key relationship annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    @MapKey(name="type")
    private Map<String, PhoneNumber> phoneNumbers;
    ...
}

@Entity
public class PhoneNumber {
    @Id
    private long id;
    @Basic
    private String type; // Either "home", "work", or "fax".
    ...
    @ManyToOne
    private Employee owner;
    ...
}
```

Map Key Columns (JPA 2.0)

JPA 2.0 allows for a `Map` where the key is not part of the target object to be persisted. The `Map` key can be any of the following:

- A `Basic` value, stored in the target's table or join table.
- An `Embedded` object, stored in the target's table or join table.
- A foreign key to another `Entity`, stored in the target's table or join table.

`Map` columns can be used for any collection mapping including, `OneToMany`, `ManyToMany` and `ElementCollection`.

This allows for great flexibility and complexity in the number of different models that can be mapped. The type of mapping used is always determined by the value of the `Map`, not the key. So if the key is a `Basic` but the value is an `Entity` a `OneToMany` mapping is still used. But if the value is a `Basic` but the key is an `Entity` a `ElementCollection` mapping is used.

This allows some very sophisticated database schemas to be mapped. Such as a three way join table, can be mapped using a `ManyToMany` with a `MapKeyJoinColumn` for the third foreign key. For a `ManyToMany` the key is always stored in the `JoinTable`. For a `OneToMany` it is stored in the `JoinTable` if defined, otherwise it is stored in the target `Entity`'s table, even though the target `Entity` does not map this column. For an `ElementCollection` the key is stored in the element's table.

The `@MapKeyColumn` annotation or `<map-key-column>` XML element is used to define a map relationship where the key is a `Basic` value, the `@MapKeyEnumerated` and `@MapKeyTemporal` can also be used with this for `Enum` or `Calendar` types. The

`@MapKeyJoinColumn` annotation or `<map-key-join-column>` XML element is used to define a map relationship where the key is an Entity value, the `@MapKeyJoinColumns` can also be used with this for composite foreign keys. The annotation `@MapKeyClass` or `<map-key-class>` XML element can be used when the key is an Embeddable or to specify the target class or type if generics are not used.

Example of a map key column relationship annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    @MapKeyColumn(name="PHONE_TYPE")
    private Map<String, PhoneNumber> phoneNumbers;
    ...
}

@Entity
public class PhoneNumber {
    @Id
    private long id;
    ...
    @ManyToOne
    private Employee owner;
    ...
}
```

Example of a map key relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phoneNumbers" mapped-by="owner">
      <map-key name="type"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="PhoneNumber" class="org.acme.PhoneNumber" access="FIELD">
  <attributes>
    <id name="id"/>
    <basic name="type"/>
    <many-to-one name="owner"/>
  </attributes>
</entity>
```

Example of a map key column relationship XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phoneNumbers" mapped-by="owner">
      <map-key-column name="PHONE_TYPE"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="PhoneNumber" class="org.acme.PhoneNumber" access="FIELD">
  <attributes>
    <id name="id"/>
    <many-to-one name="owner"/>
  </attributes>
</entity>

```

Example of a map key join column relationship annotation

```

@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    @MapKeyJoinColumn(name="PHONE_TYPE_ID")
    private Map<PhoneType, PhoneNumber> phoneNumbers;
    ...
}

@Entity
public class PhoneNumber {
    @Id
    private long id;
    ...
    @ManyToOne
    private Employee owner;
    ...
}

@Entity
public class PhoneType {
    @Id
    private long id;
    ...
    @Basic
    private String type;
    ...
}

```

Example of a map key join column relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-many name="phoneNumbers" mapped-by="owner">
      <map-key-join-column name="PHONE_TYPE_ID"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="PhoneNumber" class="org.acme.PhoneNumber" access="FIELD">
  <attributes>
    <id name="id"/>
    <many-to-one name="owner"/>
  </attributes>
</entity>
<entity name="PhoneType" class="org.acme.PhoneType" access="FIELD">
  <attributes>
    <id name="id"/>
    <basic name="type"/>
  </attributes>
</entity>
```

Example of a map key class embedded relationship annotation

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany
    @MapKeyClass(PhoneType.class)
    private Map<PhoneType, PhoneNumber> phoneNumbers;
    ...
}

@Entity
public class PhoneNumber {
    @Id
    private long id;
    ...
}

@Embeddable
public class PhoneType {
    @Basic
    private String type;
    ...
}
```

Example of a map key class embedded relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id" />
    <one-to-many name="phoneNumbers">
      <map-key-class>PhoneType</map-key-class>
    </one-to-many>
  </attributes>
</entity>
<entity name="PhoneNumber" class="org.acme.PhoneNumber" access="FIELD">
  <attributes>
    <id name="id" />
    <many-to-one name="owner" />
  </attributes>
</entity>
<embeddable name="PhoneType" class="org.acme.PhoneType" access="FIELD">
  <attributes>
    <basic name="type" />
  </attributes>
</embeddable>
```

Join Fetching

Join fetching is a query optimization technique for reading multiple objects in a single database query. It involves joining the two object's tables in SQL and selecting both object's data. Join fetching is commonly used for OneToOne relationships, but also can be used for any relationship including OneToMany and ManyToMany.

Join fetching is one solution to the classic ORM $n+1$ performance problem. The issue is if you select n Employee objects, and access each of their addresses, in basic ORM (including JPA) you will get 1 database select for the Employee objects, and then n database selects, one for each Address object. Join fetching solves this issue by only requiring one select, and selecting both the Employee and its Address.

JPA supports join fetching through JPQL using the JOIN FETCH syntax.

Example of JPQL Join Fetch

```
SELECT emp FROM Employee emp JOIN FETCH emp.address
```

This causes both the Employee and Address data to be selected in a single query.

Outer Joins

Using the JPQL JOIN FETCH syntax a normal INNER join is performed. This has the side effect of filtering any Employee from the result set that did not have an address. An OUTER join in SQL is one that does not filter absent rows on the join, but instead joins a row of all null values. If your relationship allows null or an empty collection for collection relationships, then you can use an OUTER join fetch, this is done in JPQL using the LEFT syntax.

Note that OUTER joins can be less efficient on some databases, so avoid using an OUTER if it is not required.

Example of JPQL Outer Join Fetch

```
SELECT emp FROM Employee emp LEFT JOIN FETCH emp.address
```

Mapping Level Join Fetch and EAGER

JPA has no way to specify that a join fetch always be used for a relationship. Normally it is better to specify the join fetch at the query level, as some use cases may require the related objects, and other use cases may not. JPA does support an EAGER option on mappings, but this means that the relationship will be loaded, not that it will be joined. It may be desirable to mark all relationships as EAGER as everything is desired to be loaded, but join fetching everything in one huge select could result in a inefficient, overly complex, or invalid join on the database.

Some JPA providers do interpret EAGER as join fetch, so this may work on some JPA providers. Some JPA providers support a separate option for always join fetching a relationship.

TopLink, EclipseLink : Support a @JoinFetch annotation and XML on a mapping to define that the relationship always be join fetched.

Nested Joins

JPA 1.0 does not allow nested join fetches in JPQL, although this may be supported by some JPA providers. You can join fetch multiple relationships, but not nested relationships.

Example of Multiple JPQL Join Fetch

```
SELECT emp FROM Employee emp LEFT JOIN FETCH emp.address LEFT JOIN  
FETCH emp.phoneNumbers
```

Duplicate Data and Huge Joins

One issue with join fetching is that duplicate data can be returned. For example consider join fetching an Employee's phoneNumbers relationship. If each Employee has 3 Phone objects in its phoneNumbers collection, the join will require to bring back $n*3$ rows. As there are 3 phone rows for each employee row, the employee row will be duplicated 3 times. So you are reading more data than if you have selected the objects in $n+1$ queries. Normally the fact that your executing fewer queries makes up for the fact that you may be reading duplicate data, but if you consider joining multiple collection relationships you can start to get back $j*i$ duplicate data which can start to become an issue. Even with ManyToOne relationships you can be selecting duplicate data. Consider join fetching an Employee's manager, if all or most employee's have the same manager, you will end up select this manager's data many times, in this case you would be better off not using join fetch, and allowing a single query for the manager.

If you start join fetching every relationship, you can start to get some pretty huge joins. This can sometimes be an issue for the database, especially with huge outer joins.

One alternative solution to join fetch that does not suffer from duplicate data is using Batch Reading.

Batch Reading

Batch Reading is a query optimization technique for reading multiple related objects in a finite set of database queries. It involves executing the query for the root objects as normal. But for the related objects the original query is joined with the query for the related objects, allowing all of the related objects to be read in a single database query. Batch Reading can be used for any type of relationship.

Batch Reading is one solution to the classic ORM $n+1$ performance problem. The issue is if you select n `Employee` objects, and access each of their addresses, in basic ORM (including JPA) you will get 1 database select for the `Employee` objects, and then n database selects, one for each `Address` object. Batch Reading solves this issue by requiring one select for the `Employee` objects and one select for the `Address` objects.

Batch reading is more optimal for reading collection relationships and multiple relationships as it does not require selecting duplicate data as in join fetching.

JPA does not support batch reading, but some JPA providers do.

TopLink, EclipseLink : Support a "eclipselink.batch" query hint to enable batch reading. Batch reading can also be configured on a relationship mapping using the API.

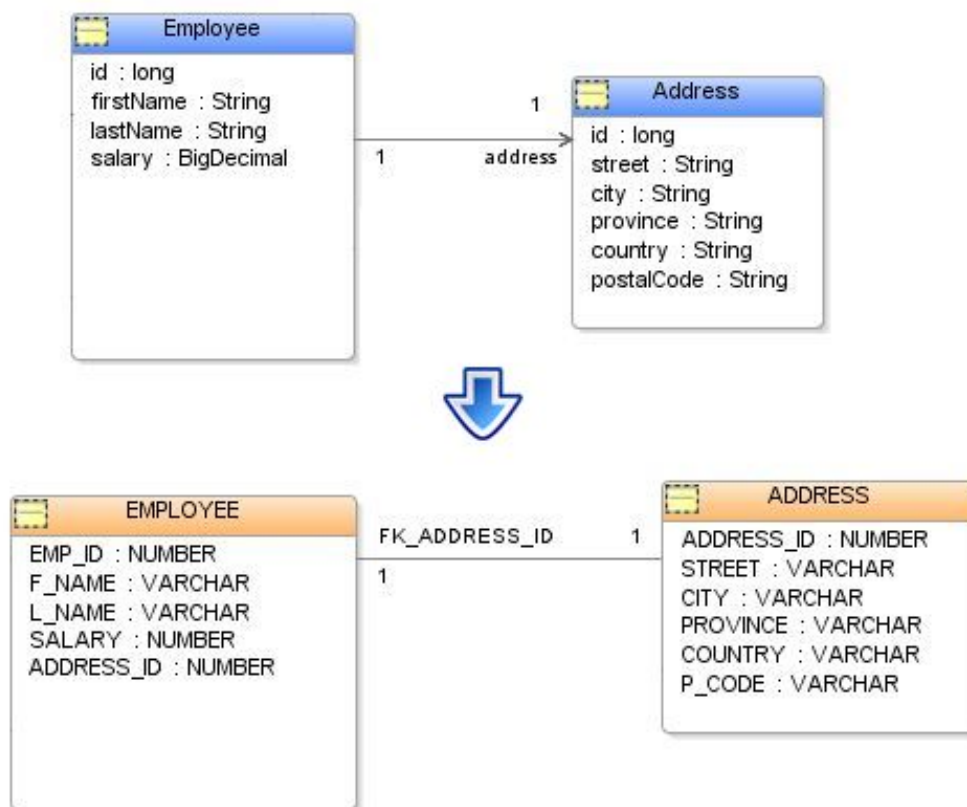
Filtering, Complex Joins

Normally a relationship is based on a foreign key in the database, but on occasion it is always based on other conditions. Such as `Employee` having many `PhoneNumbers` but also a single home phone, or one of his phones that has the "home" type, or a collection of "active" projects, or other such condition.

JPA does not support mapping these types of relationships, as it only supports mappings defined by foreign keys, not based on other columns, constant values, or functions. Some JPA providers may support this. Workarounds include, mapping the foreign key part of the relationship, then filtering the results in your get/set methods of your object. You could also query for the results, instead of defining a relationship.

TopLink, EclipseLink : Support filtering and complex relationships through several mechanisms. You can use a `DescriptorCustomizer` to define a `selectionCriteria` on any mapping using the `Expression` criteria API. This allows for any condition to be applied including constants, functions, or complex joins. You can also use a `DescriptorCustomizer` to define the SQL or define a `StoredProcedureCall` for the mapping's `selectionQuery`.

OneToOne



A `OneToOne` relationship in Java is where the source object has an attribute that references another target object and (if that target object had the inverse relationship back to the source object it would also be a `OneToOne` relationship). All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a `ManyToOne` relationship, which is similar to a `OneToOne` relationship except that the inverse relationship (if it were defined) is a `OneToMany` relationship. The main difference between a `OneToOne` and a `ManyToOne` relationship in JPA is that a `ManyToOne` always contains a foreign key from the source object's table to the target object's table, whereas a `OneToOne` relationship the foreign key may either be in the source object's table or the target object's table. If the foreign key is in the target object's table JPA requires that the relationship be bi-directional (must be defined in both objects), and the source object must use the `mappedBy` attribute to define the mapping.

In JPA a `OneToOne` relationship is defined through the `@OneToOne` (<https://java.sun.com/javase/5/docs/api/javax/persistence/OneToOne.html>) annotation or the `<one-to-one>` element. A `OneToOne` relationship typically requires a `@JoinColumn` (<https://java.sun.com/javase/5/docs/api/javax/persistence/JoinColumn.html>).

Example of a OneToOne relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="ADDRESS_ID")
    private Address address;
    ...
}
```

Inverse Relationships, Target Foreign Keys and Mapped By

A typical object centric perspective of a OneToOne relationship has the data model mirror the object model, in that the *source* object has a pointer to the *target* object, so the database *source* table has a foreign key to the *target* table. This is not how things always work out in the database though, in fact many database developers would think having the foreign key in the *target* table to be logical, as this enforces the uniqueness of the OneToOne relationship. Personally I prefer the object perspective, however you will most likely encounter both.

To start with consider a bi-directional OneToOne relationship, you do not require two foreign keys, one in each table, so a single foreign key in the *owning* side of the relationship is sufficient. In JPA the *inverse* OneToOne *must* use the `mappedBy` attribute (with some exceptions), this makes the JPA provider use the foreign key and mapping information in the *source* mapping to define the *target* mapping.

See also, Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys.

The following gives an example of what the inverse address relationship would look like.

Example of an inverse OneToOne relationship annotations

```
@Entity
public class Address {
    @Id
    private long id;
    ...
    @OneToOne(fetch=FetchType.LAZY, mappedBy="address")
    private Employee owner;
    ...
}
```

Example of a OneToOne relationship XML

```
<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="EMP_ID"/>
    </id>
    <one-to-one name="address" fetch="LAZY">
      <join-column name="ADDRESS_ID"/>
    </one-to-one>
  </attributes>
</entity>
```

Example of an inverse OneToOne relationship XML

```
<entity name="Address" class="org.acme.Address" access="FIELD">
  <attributes>
    <id name="id"/>
    <one-to-one name="owner" fetch="LAZY" mapped-by="address"/>
  </attributes>
</entity>
```

See Also

- Relationships
 - Cascading
 - Lazy Fetching
 - Target Entity
 - Join Fetching
 - Batch Reading
 - Common Problems
- ManyToOne

Common Problems

Foreign key is also part of the primary key.

See Primary Keys through OneToOne Relationships.

Foreign key is also mapped as a basic.

If you use the same field in two different mappings, you typically require to make one of them read-only using `insertable`, `updateable = false`.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys.

Constraint error on insert.

This typically occurs because you have incorrectly mapped the foreign key in a OneToOne relationship.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys.

It can also occur if your JPA provider does not support referential integrity, or does not resolve bi-directional constraints. In this case you may either need to remove the constraint, or use `EntityManager flush()` to ensure the order your objects are written in.

Foreign key value is null

Ensure you set the value of the object's `OneToOne`, if the `OneToOne` is part of a bi-directional `OneToOne` relationship, ensure you set `OneToOne` in both object's, JPA does not maintain bi-directional relationships for you.

Also check that you defined the `JoinColumn` correctly, ensure you did not set `insertable`, `updateable = false` or use a `PrimaryKeyJoinColumn`, or `mappedBy`.

Advanced

Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys

If a `OneToOne` relationship uses a *target foreign key* (the foreign key is in the target table, not the source table), then JPA requires that you define a `OneToOne` mapping in both directions, and that the *target foreign key* mapping use the `mappedBy` attribute. The reason for this, is the mapping in the source object only affects the row the JPA writes to the source table, if the foreign key is in the target table, JPA has no easy way to write this field.

There are other ways around this problem however. In JPA the `JoinColumn` defines an `insertable` and `updateable` attribute, these can be used to instruct the JPA provider that the foreign key is actually in the target object's table. With these enabled JPA will not write anything to the source table, most JPA providers will also infer that the foreign key constraint is in the target table to preserve referential integrity on insertion. JPA also defines the `@PrimaryKeyJoinColumn` (<https://java.sun.com/javase/5/docs/api/javax/persistence/PrimaryKeyJoinColumn.html>) that can be used to define the same thing. You still must map the foreign key in the target object in some fashion though, but could just use a `Basic` mapping to do this.

Some JPA providers may support an option for a unidirectional `OneToOne` mapping for target foreign keys.

Target foreign keys can be tricky to understand, so you might want to read this section twice. They can get even more complex though. If you have a data model that cascades primary keys then you can end up with a single `OneToOne` that has both a logical foreign key, but has some fields in it that are logically target foreign keys.

For example consider `Company`, `Department`, `Employee`. `Company`'s id is `COM_ID`, `Department`'s id is a composite primary key of `COM_ID` and `DEPT_ID`, and `Employee`'s id is a composite primary key of `COM_ID`, `DEP_ID`, and `EMP_ID`. So for an `Employee` its relationship to `company` uses a normal `ManyToOne` with a foreign key, but its relationship to `department` uses a `ManyToOne` with a foreign key, but the `COM_ID` uses `insertable`, `updateable = false` or `PrimaryKeyJoinColumn`, because it is actually mapped through the `company` relationship. The `Employee`'s relationship to its `address` then uses a normal foreign key for `ADD_ID` but a target foreign key for `COM_ID`, `DEP_ID`, and `EMP_ID`.

This may work in some JPA providers, others may require different configuration, or not support this type of data model.

Example of cascaded primary keys and mixed OneToOne and ManyToOne mapping annotations

```

@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long employeeId;
    @Id
    @Column(name="DEP_ID" insertable=false, updateable=false)
    private long departmentId;
    @Id
    @Column(name="COM_ID" insertable=false, updateable=false)
    private long companyId;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="COM_ID")
    private Company company;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="DEP_ID")
    @JoinColumn(name="COM_ID" insertable=false, updateable=false)
    private Department department;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="MNG_ID")
    @JoinColumn(name="DEP_ID" insertable=false, updateable=false)
    @JoinColumn(name="COM_ID" insertable=false, updateable=false)
    private Employee manager;
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="ADD_ID")
    @JoinColumn(name="EMP_ID" insertable=false, updateable=false)
    @JoinColumn(name="DEP_ID" insertable=false, updateable=false)
    @JoinColumn(name="COM_ID" insertable=false, updateable=false)
    private Address address;
    ...
}

```

Example of cascaded primary keys and mixed OneToOne and ManyToOne mapping annotations using PrimaryKeyJoinColumn

```

@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long employeeId;

```

```
@Id
@Column(name="DEP_ID" insertable=false, updateable=false)
private long departmentId;
@Id
@Column(name="COM_ID" insertable=false, updateable=false)
private long companyId;
...
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="COM_ID")
private Company company;
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="DEP_ID")
@PrimaryKeyJoinColumn(name="COM_ID")
private Department department;
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="MNG_ID")
@PrimaryKeyJoinColumn(name="DEP_ID")
@PrimaryKeyJoinColumn(name="COM_ID")
private Employee manager;
@OneToOne(fetch=FetchType.LAZY)
@JoinColumn(name="ADD_ID")
@PrimaryKeyJoinColumn(name="EMP_ID")
@PrimaryKeyJoinColumn(name="DEP_ID")
@PrimaryKeyJoinColumn(name="COM_ID")
private Address address;
...
}
```

Mapping a OneToOne Using a Join Table

In some data models, you may have a `OneToOne` relationship defined through a *join table*. For example consider you had existing `EMPLOYEE` and `ADDRESS` tables with no foreign key, and wanted to define a `OneToOne` relationship without changing the existing tables. To do this you could define an intermediate table that contained the primary key of both objects. This is similar to a `ManyToMany` relationship, but if you add a unique constraint to each foreign key you can enforce that it is `OneToOne` (or even `OneToMany`).

JPA defines a join table using the `@JoinTable` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinTable.html>) annotation and `<join-table>` XML element. A `JoinTable` can be used on a `ManyToMany` or `OneToMany` mappings, but the JPA 1.0 specification is vague whether it can be used on a `OneToOne`. The `JoinTable` documentation does not state that it can be used in a `OneToOne`, but the XML schema for `<one-to-one>` does allow a nested `<join-table>` element. Some JPA providers may support this, and others may not.

If your JPA provider does not support this, you can workaround the issue by instead defining a `OneToMany` or `ManyToMany` relationship and just define a `get/set` method that returns/sets the first element on the collection.

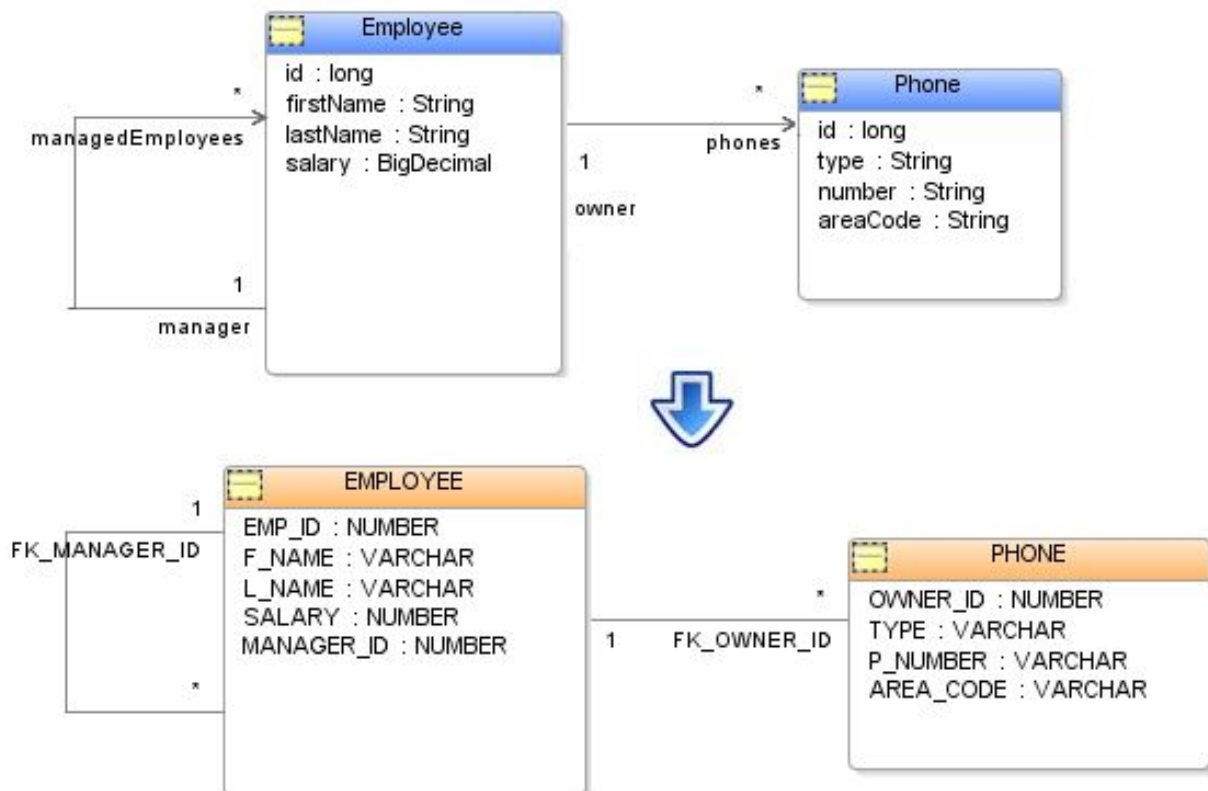
Example of a OneToOne using a JoinTable

```
@OneToOne(fetch=FetchType.LAZY)
@JoinTable(
    name="EMP_ADD"
    joinColumns=
        @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID"),
    inverseJoinColumns=
        @JoinColumn(name="ADDR_ID", referencedColumnName="ADDRESS_ID"))
private Address address;
...
```

Example of simulating a OneToOne using a OneToMany JoinTable

```
@OneToMany
@JoinTable(
    name="EMP_ADD"
    joinColumns=
        @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID"),
    inverseJoinColumns=
        @JoinColumn(name="ADDR_ID", referencedColumnName="ADDRESS_ID"))
private List<Address> addresses;
...
public Address getAddress() {
    if (this.addresses.isEmpty()) {
        return null;
    }
    return this.addresses.get(0);
}
public void setAddress(Address address) {
    if (this.addresses.isEmpty()) {
        this.addresses.add(address);
    } else {
        this.addresses.set(1, address);
    }
}
...
```


ManyToOne



A **ManyToOne** relationship in Java is where the source object has an attribute that references another target object and (if) that target object had the inverse relationship back to the source object it would be a **OneToMany** relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a **OneToOne** relationship, which is similar to a **ManyToOne** relationship except that the inverse relationship (if it were defined) is a **OneToOne** relationship. The main difference between a **OneToOne** and a **ManyToOne** relationship in JPA is that a **ManyToOne** always contains a foreign key from the source object's table to the target object's table, whereas a **OneToOne** relationship the foreign key may either be in the source object's table or the target object's table.

In JPA a **ManyToOne** relationship is defined through the `@ManyToOne` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/ManyToOne.html>) annotation or the `<many-to-one>` element.

In JPA a **ManyToOne** relationship is always (well almost always) required to define a **OneToMany** relationship, the **ManyToOne** always defines the foreign key (`JoinColumn`) and the **OneToMany** must use a `mappedBy` to define its inverse **ManyToOne**.

Example of a ManyToOne relationship annotations

```
@Entity
public class Phone {
    @Id
    private long id;
    ...
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="OWNER_ID")
    private Employee owner;
    ...
}
```

See Also

- Relationships
 - Cascading
 - Lazy Fetching
 - Target Entity
 - Join Fetching
 - Batch Reading
 - Common Problems
- OneToOne
- OneToMany

Example of a ManyToOne relationship XML

```
<entity name="Phone" class="org.acme.Phone" access="FIELD">
  <attributes>
    <id name="id"/>
    <many-to-one name="owner" fetch="LAZY">
      <join-column name="OWNER_ID"/>
    </many-to-one>
  </attributes>
</entity>
```

Common Problems

Foreign key is also part of the primary key.

See Primary Keys through OneToOne Relationships.

Foreign key is also mapped as a basic.

If you use the same field in two different mappings, you typically require to make one of them read-only using `insertable`, `updateable = false`.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys.

Constraint error on insert.

This typically occurs because you have incorrectly mapped the foreign key in a OneToOne relationship.

See Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys.

It can also occur if your JPA provider does not support referential integrity, or does not resolve bi-directional constraints. In this case you may either need to remove the constraint, or use `EntityManager flush()` to ensure the order your objects are written in.

Foreign key value is null

Ensure you set the value of the object's `OneToOne`, if the `OneToOne` is part of a bi-directional `OneToMany` relationship, ensure you set the object's `OneToOne` when adding an object to the `OneToMany`, JPA does not maintain bi-directional relationships for you.

Also check that you defined the `JoinColumn` correctly, ensure you did not set `insertable, updateable = false` or use a `PrimaryKeyJoinColumn`.

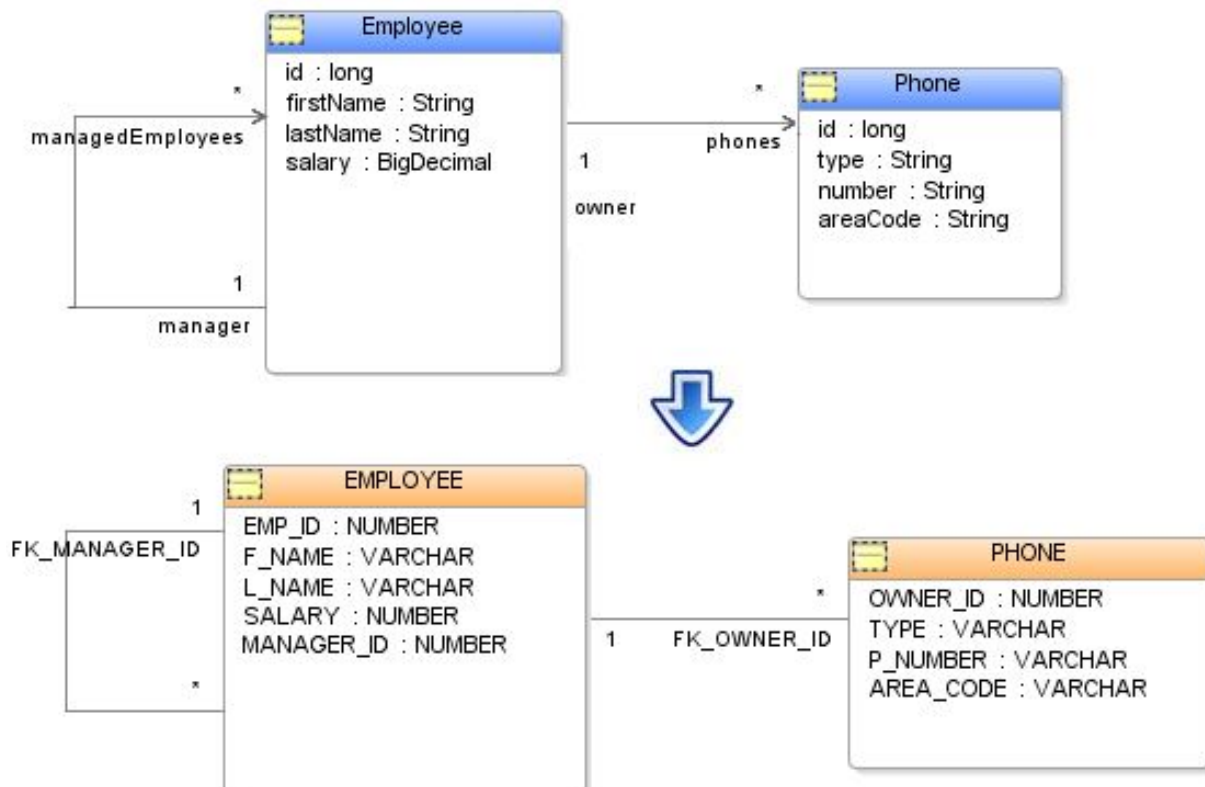
Advanced

Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys

In complex data models it may be required to use a target foreign key, or read-only `JoinColumn` in mapping a `ManyToOne` if the foreign key/`JoinColumn` is shared with other `ManyToOne` or `Basic` mappings.

See, Target Foreign Keys, Primary Key Join Columns, Cascade Primary Keys

OneToMany



A `OneToMany` relationship in Java is where the source object has an attribute that stores a collection of target objects and (if) those target objects had the inverse relationship back to the source object it would be a `ManyToOne` relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a `ManyToMany` relationship, which is similar to a `OneToMany` relationship except that the inverse relationship (if it were defined) is a `ManyToMany` relationship. The main difference between a `OneToMany` and a `ManyToMany` relationship in JPA is that a `ManyToMany` always makes use of an intermediate relational join table to store the relationship, whereas a `OneToMany` can either use a join table, or a foreign key in target object's table referencing the source object table's primary key. If the `OneToMany` uses a foreign key in the target object's table JPA requires that the relationship be bi-directional (inverse `ManyToOne` relationship must be defined in the target object), and the source object must use the `mappedBy` attribute to define the mapping.

In JPA a `OneToMany` relationship is defined through the `@OneToMany` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/OneToMany.html>) annotation or the `<one-to-many>` element.

Example of a `OneToMany` relationship annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany(mappedBy="owner")
    private List<Phone> phones;
    ...
}
```

Join Table

A common mismatch between objects and relational tables is that a `OneToMany` does not require a back reference in Java, but requires a back reference foreign key in the database. Normally it is best to define the `ManyToOne` back reference in Java, if you cannot or don't want to do this, then you can use an intermediate join table to store the relationship. This is similar to a `ManyToMany` relationship, but if you add a unique constraint to the target foreign key you can enforce that it is `OneToMany`.

JPA defines a join table using the `@JoinTable` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinTable.html>) annotation and `<join-table>` XML element. A `JoinTable` can be used on a `ManyToMany` or `OneToMany` mappings.

See also, `Undirectional OneToMany`

Example of a OneToMany using a JoinTable annotation

```

@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToMany
    @JoinTable
    (
        name="EMP_PHONE"
        joinColumns={ @JoinColumn(name="EMP_ID",
referencedColumnName="EMP_ID") }
        inverseJoinColumns={ @JoinColumn(name="PHONE_ID",
referencedColumnName="PHONE_ID") }
    )
    private List<Phone> phones;
    ...
}

```

Example of a OneToMany relationship XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
    <attributes>
        <id name="id"/>
        <one-to-many name="phones" mapped-by="owner"/>
    </attributes>
</entity>

```

Note this @OneToMany mapping requires an inverse @ManyToOne mapping to be complete, see ManyToOne.

Getters and Setters

As the relationship is bi-directional so as the application updates one side of the relationship, the other side should also get updated, and be in synch. In JPA, as in Java in general it is the responsibility of the application, or the object model to maintain relationships. If your application adds to one side of a relationship, then it must add to the other side.

This can be resolved through add or set methods in the object model that handle both sides of the relationships, so the application code does not need to worry about it. There are two ways to go about this, you can either only add the relationship maintenance code to one side of the relationship, and only use the setter from one side (such as making the other side protected), or add it to both sides and ensure you avoid a infinite loop.

For example:

```

public class Employee {
    private List phones;
    ...
}

```

```

    public void addPhone(Phone phone) {
        this.phones.add(phone);
        if (phone.getOwner() != this) {
            phone.setOwner(this);
        }
    }
    ...
}

public class Phone {
    private Employee owner;
    ...
    public void setOwner(Employee employee) {
        this.owner = employee;
        if (!employee.getPhones().contains(this)) {
            employee.getPhones().add(this);
        }
    }
    ...
}

```

Some expect the JPA provider to have magic that automatically maintains relationships. This was actually part of the EJB CMP 2 specification. However the issue is if the objects are detached or serialized to another VM, or new objects are related before being managed, or the object model is used outside the scope of JPA, then the magic is gone, and the application is left figuring things out, so in general it may be better to add the code to the object model. However some JPA providers do have support for automatically maintaining relationships.

In some cases it is undesirable to instantiate a large collection when adding a child object. One solution is to not map the bi-directional relationship, and instead query for it as required. Also some JPA provides optimize their lazy collection objects to handle this case, so you can still add to the collection without instantiating it.

Example of a OneToMany using a JoinTable XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="EMP_ID"/>
    </id>
    <one-to-many name="phones">
      <join-table name="EMP_PHONE">
        <join-column name="EMP_ID" referenced-column-name="EMP_ID"/>
        <inverse-join-column name="PHONE_ID" referenced-column-name="PHONE_ID"/>
      </join-table>
    </one-to-many>
  </attributes>
</entity>

```

See Also

- Relationships
 - Cascading
 - Lazy Fetching
 - Target Entity
 - Collections
 - Maps
 - Join Fetching
 - Batch Reading
 - Common Problems
- ManyToOne
- ManyToMany

Common Problems

Object not in collection after refresh.

See Object corruption.

Advanced

Unidirectional OneToMany, No Inverse ManyToOne, No Join Table (JPA 2.0)

JPA 1.0 does not support a unidirectional `OneToMany` relationship without a `JoinTable`. JPA 2.0 will have support for a unidirectional `OneToMany`. In JPA 2.0 a `JoinColumn` can be used on a `OneToMany` to define the foreign key, some JPA providers may support this already.

The main issue with an unidirectional `OneToMany` is that the foreign key is owned by the target object's table, so if the target object has no knowledge of this foreign key, inserting and updating the value is difficult. In a unidirectional `OneToMany` the source object take ownership of the foreign key field, and is responsible for updating its value.

The target object in a unidirectional `OneToMany` is an independent object, so it should not rely on the foreign key in any way, i.e. the foreign key cannot be part of its primary key, nor generally have a not null constraint on it. You can model a collection of objects where the target has no foreign key mapped, but uses it as its primary key, or has no primary key using a `Embeddable` collection mapping, see `Embeddable Collections`.

If your JPA provider does not support unidirectional `OneToMany` relationships, then you will need to either add a back reference `ManyToOne` or a `JoinTable`. In general it is best to use a `JoinTable` if you truly want to model a unidirectional `OneToMany` on the database.

There are some creative workarounds to defining a unidirectional `OneToMany`. One is to map it using a `JoinTable`, but make the target table the `JoinTable`. This will cause an extra join, but work for the most part for reads, writes of course will not work correctly, so this is only a read-only solution and a hacky one at that.

Example of a JPA 2.0 unidirectional OneToMany relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
    @OneToMany
    @JoinColumn(name="EMP_ID", referencedColumnName="OWNER_ID")
    private List<Phone> phones;
    ...
}
```

ManyToMany

A ManyToMany relationship in Java is where the source object has an attribute that stores a collection of target objects and (if) those target objects had the inverse relationship back to the source object it would also be a ManyToMany relationship. All relationships in Java and JPA are unidirectional, in that if a source object references a target object there is no guarantee that the target object also has a relationship to the source object. This is different than a relational database, in which relationships are defined through foreign keys and querying such that the inverse query always exists.

JPA also defines a OneToMany relationship, which is similar to a ManyToMany relationship except that the inverse relationship (if it were defined) is a ManyToOne relationship. The main difference between a OneToMany and a ManyToMany relationship in JPA is that a ManyToMany always makes use of an intermediate relational join table to store the relationship, whereas a OneToMany can either use a join table, or a foreign key in target object's table referencing the source object table's primary key.

In JPA a ManyToMany relationship is defined through the @ManyToMany (<https://java.sun.com/javase/5/docs/api/javax/persistence/ManyToMany.html>) annotation or the <many-to-many> element.

All ManyToMany relationships require a JoinTable. The JoinTable is defined using the @JoinTable (<https://java.sun.com/javase/5/docs/api/javax/persistence/JoinTable.html>) annotation and <join-table> XML element. The JoinTable defines a foreign key to the source object's primary key (joinColumns), and a foreign key to the target object's primary key (inverseJoinColumns). Normally the primary key of the JoinTable is the combination of both foreign keys.

Example of a ManyToMany relationship annotations

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private long id;
    ...
}
```



```

@ManyToMany
@JoinTable(
    name="EMP_PROJ"
    joinColumns={@JoinColumn(name="EMP_ID",
referencedColumnName="EMP_ID")}
    inverseJoinColumns={@JoinColumn(name="PROJ_ID",
referencedColumnName="PROJ_ID")})
    private List<Project> projects;
    ...
}

```

Bi-directional Many to Many

...

Example of a ManyToMany relationship XML

```

<entity name="Employee" class="org.acme.Employee" access="FIELD">
  <attributes>
    <id name="id">
      <column name="EMP_ID"/>
    </id>
    <many-to-many name="projects">
      <join-table name="EMP_PROJ">
        <join-column name="EMP_ID" referenced-column-name="EMP_ID"/>
        <inverse-join-column name="PROJ_ID" referenced-column-name="PROJ_ID"/>
      </join-table>
    </many-to-many>
  </attributes>
</entity>

```

See Also

- Relationships
 - Cascading
 - Lazy Fetching
 - Target Entity
 - Collections
 - Maps
 - Join Fetching
 - Batch Reading
 - Common Problems
- OneToMany

Common Problems

Object not in collection after refresh.

If you have a bi-directional `ManyToMany` relationship, ensure that you add to both sides of the relationship.

See `Object corruption`.

Additional columns in join table.

See `Mapping a Join Table with Additional Columns`

Duplicate rows inserted into the join table.

If you have a bidirectional `ManyToMany` relationship, you must use `mappedBy` on one side of the relationship, otherwise it will be assumed to be two difference relationships and you will get duplicate rows inserted into the join table.

Advanced

Mapping a Join Table with Additional Columns

A frequent problem is that two classes have a `ManyToMany` relationship, but the relational join table has additional data. For example if `Employee` has a `ManyToMany` with `Project` but the `PROJ_EMP` join table also has an `IS_TEAM_LEAD` column. In this case the best solution is to create a class that models the join table. So an `ProjectAssociation` class would be created. It would have a `ManyToOne` to `Employee` and `Project`, and attributes for the additional data. `Employee` and `Project` would have a `OneToMany` to the `ProjectAssociation`. Some JPA providers also provide additional support for mapping to join tables with additional data.

Unfortunately mapping this type of model becomes more complicated in JPA because it requires a composite primary key. The association object's `Id` is composed of the `Employee` and `Project` ids. The JPA spec does not allow an `Id` to be used on a `ManyToOne` so the association class must have two duplicate attributes to also store the ids, and use an `IdClass`, these duplicate attributes must be kept in synch with the `ManyToOne` attributes. Some JPA providers may allow a `ManyToOne` to be part of an `Id`, so this may be simpler with some JPA providers. To make your life simpler, I would recommend adding a generated `Id` attribute to the association class. This will give the object a simpler `Id` and not require duplicating the `Employee` and `Project` ids.

This same pattern can be used no matter what the additional data in the join table is. Another usage is if you have a `Map` relationship between two objects, with a third unrelated object or data representing the `Map` key. The JPA spec requires that the `Map` key be an attribute of the `Map` value, so the *association object* pattern can be used to model the relationship.

If the additional data in the join table is only required on the database and not used in Java, such as auditing information, it may also be possible to use database triggers to automatically set the data.

Example join table association object annotations

```
@Entity
public class Employee {
    @Id
    private long id;
    ...
    @OneToMany
    private List<ProjectAssociation> projects;
}
```

```
@Entity
public class Project {
    @Id
    private long id;
    ...
    @OneToMany
    private List<ProjectAssociation> employees;
    ...
    // Add an employee to the project.
    // Create an association object for the relationship and set its'
data.
    public void addEmployee(Employee employee, boolean teamLead) {
        ProjectAssociation association = new ProjectAssociation();
        association.setEmployee(employee);
        association.setProject(this);
        association.setEmployeeId(employee.getId());
        association.setProjectId(this.getId());
        association.setIsTeamLead(teamLead);

        employees.add(association);
    }
}
```

```
@Entity
@Table(name="PROJ_EMP")
@IdClass(ProjectAssociationId.class)
public class ProjectAssociation {
    @Id
    private long employeeId;
    @Id
    private long projectId;
    @Column("IS_PROJECT_LEAD")
    private boolean isProjectLead;
    @ManyToOne
    @PrimaryKeyJoinColumn(name="EMPLOYEEID", referencedColumnName="ID")
    private Employee employee;
    @ManyToOne
    @PrimaryKeyJoinColumn(name="PROJECTID", referencedColumnName="ID")
```

```
private Project project;
...
}

public class ProjectAssociationId {
    @Id
    private long employeeId;
    @Id
    private long projectId;
    ...
}
```

Runtime

Once you have mapped your object model the second step in persistence development is to access and process your objects from your application, this is referred to as the runtime usage of persistence. Various persistence specifications have had various runtime models. The most common model is to have a runtime API; a runtime API typically will define API for connecting to a data-source, querying and transactions.

Entity Manager

JPA provides a runtime API defined by the `javax.persistence` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html>) package. The main runtime class is the `EntityManager` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html>) class. The `EntityManager` provides API for creating queries, accessing transactions, and finding, persisting, merging and deleting objects. The JPA API can be used in any Java environment including JSE and JEE.

An `EntityManager` can be created through an `EntityManagerFactory` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManagerFactory.html>), or can be *injected* into an instance variable in an EJB `SessionBean`, or can be looked up in JNDI in a JEE server.

JPA is used differently in Java Standard Edition (JSE) versus Java Enterprise Edition (JEE).

Java Standard Edition

In JSE an `EntityManager` is accessed from the `JPA Persistence` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/Persistence.html>) class through the `createEntityManagerFactory` API. The *persistent unit name* is passed to the `createEntityManagerFactory`, this is the name given in the persistence unit's `persistence.xml` file. All JSE JPA applications must define a `persistence.xml` file. The file defines the persistence unit including the name, classes, orm files, datasource, vendor specific properties.

Oddly enough JPA does not define a standard way of specifying how to connect to the database in JSE. Each JPA provider defines their own persistence properties for setting the JDBC driver manager class, URL, user and password. JPA has a standard way of setting the `DataSource` JNDI name, but this is mainly used in JEE.

The JPA application is typically required to be packaged into a persistence unit jar file. This is a normal jar, that has the `persistence.xml` file in the `META-INF` directory. Typically a JPA provider will require something special be done in JSE to enable certain features such as lazy fetching, such as static weaving (byte-code processing) of the jar, or using an agent JVM option.

In JSE the `EntityManager` must be closed when your application is done with it. The life-cycle of the `EntityManager` is typically per client, or per request. The `EntityManagerFactory` can be shared among multiple threads or users, but the `EntityManager` should not be shared.

Example persistence.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             persistence_1_0.xsd"
             version="1.0">
  <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
    <provider>org.acme.jpa.PersistenceProvider</provider>
    <exclude-unlisted-classes>>false</exclude-unlisted-classes>
    <properties>
      <property name="acme.driver" value="org.acme.db.Driver"/>
      <property name="acme.url" value="jdbc:acmedb://localhost/acme"/>
      <property name="acme.user" value="acmeuser"/>
      <property name="acme.password" value="loonytunes"/>
    </properties>
  </persistence-unit>
</persistence>
```

Example of accessing an EntityManager from an EntityManagerFactory

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory("acme");
EntityManager entityManager = factory.createEntityManager();
...
entityManager.close();
```

Java Enterprise Edition

In JEE the `EntityManager` or `EntityManagerFactory` can either be looked up in JNDI, or *injected* into a `SessionBean`. To look up the `EntityManager` in JNDI it must be published in JNDI such as through a `<persistence-context-ref>` in a `SessionBean`'s `ejb-jar.xml` file. To inject an `EntityManager` or `EntityManagerFactory` the annotation `@PersistenceContext` or `@PersistenceUnit` are used.

In JEE an `EntityManager` can either be *managed* (container-managed) or *non-managed* (application-managed). A managed `EntityManager` has a different life-cycle than an

EntityManager managed by the application. A managed EntityManager should never be closed, and integrates with JTA transactions so local transaction cannot be used. Across each JTA transaction boundary all of the entities read or persisted through a managed EntityManager become detached. Outside of a JTA transaction a managed EntityManager's behavior is sometimes odd, so typically should be used inside a JTA transaction.

A non-managed EntityManager is one that is created by the application through a EntityManagerFactory or directly from Persistence. A non-managed EntityManager must be closed, and typically does not integrate with JTA, but this is possible through the joinTransaction API. The entities in a non-managed EntityManager do not become detached after a transaction completes, and can continue to be used in subsequent transactions.

Example SessionBean ejb-jar.xml file with persistence context

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
  version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>EmployeeService</ejb-name>
      <business-remote>org.acme.EmployeeService</business-remote>
      <ejb-class>org.acme.EmployeeServiceBean</ejb-class>
      <session-type>Stateless</session-type>
      <persistence-context-ref>
        <persistence-context-ref-name>persistence/acme/entity-manager</persistence-context-ref-name>
        <persistence-unit-name>acme</persistence-unit-name>
      </persistence-context-ref>
      <persistence-unit-ref>
        <persistence-unit-ref-name>persistence/acme/factory</persistence-unit-ref-name>
        <persistence-unit-name>acme</persistence-unit-name>
      </persistence-unit-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Example of looking up an EntityManager in JNDI from a SessionBean

```
InitialContext context = new InitialContext();
EntityManager entityManager =
(EntityManager)context.lookup("java:comp/env/persistence/acme/entity-manager");
...
```

Example of looking up an EntityManagerFactory in JNDI from a SessionBean

```
InitialContext context = new InitialContext();
EntityManagerFactory factory =
(EntityManagerFactory)context.lookup("java:comp/env/persistence/acme/factory");
...
```

Example of injecting an EntityManager and EntityManagerFactory in a SessionBean

```
@Stateless(name="EmployeeService", mappedName="acme/EmployeeService")
@Remote(EmployeeService.class)
public class EmployeeServiceBean implements EmployeeService {

    @PersistenceContext(unitName="acme")
    private EntityManager entityManager;

    @PersistenceUnit(unitName="acme")
    private EntityManagerFactory factory;

    ...
}
```

Querying

Querying is a fundamental part of persistence. Being able to persist something is not very useful without being able to query it back. There are many querying languages and frameworks; the most common query language is SQL used in relational databases. JPA uses the Java Persistence Querying Language (JPQL), which is based on the SQL language and evolved from the EJB Query Language (EJBQL). It basically provides the SQL syntax at the object level instead of at the data level.

Other querying languages and frameworks include:

- SQL
- EJBQL
- JDOQL
- Query By Example (QBE)
- TopLink Expressions
- Hibernate Criteria
- Object Query Language (OQL)

JPA provides querying through JPQL, the Query interface, and the @NamedQuery annotation and <named-query> element.

JPQL is similar in syntax to SQL and can be defined through its BNF definition.

Named Queries

There are two main types of queries in JPA, named queries and dynamic queries. A named query is used for a static query that will be used many times in the application. The advantage of a named query is that it can be defined once, in one place, and reused in the application. Most JPA providers also pre-parse/compile named queries, so they are more optimized than dynamic queries which typically must be parsed/compiled every time they are executed. Since named queries are part of the persistence meta-data they can also be optimized or overridden in the `orm.xml` without changing the application code.

Named queries are defined through the `@NamedQuery` (<https://java.sun.com/javase/5/docs/api/javax/persistence/NamedQuery.html>) and `@NamedQueries` (<https://java.sun.com/javase/5/docs/api/javax/persistence/NamedQueries.html>) annotations, or `<named-query>` XML element.

Named queries can be defined on any annotated class, but are typically defined on the Entity that they query for. The name of the named query must be unique for the entire persistence unit, they name is not local to the Entity. In the `orm.xml` named queries can be defined either on the `<entity-mappings>` or on any `<entity>`.

Named queries are typically parametrized, so they can be executed with different parameter values. Parameters are defined in JPQL using the `:<name>` syntax for named parameters, or the `?` syntax for positional parameters.

A collection of query hints can also be provided to a named query. Query hints can be used to optimize or to provide special configuration to a query. Query hints are specific to the JPA provider. Query hints are defined through the `@QueryHint` (<https://java.sun.com/javase/5/docs/api/javax/persistence/QueryHint.html>) annotation or `query-hint` XML element.

Example named query annotation

```
@NamedQuery(  
    name="findAllEmployeesInCity",  
    query="Select emp from Employee emp where emp.address.city = :city"  
    hints={@QueryHint(name="acme.jpa.batch", value="emp.address")}  
)  
public class Employee {  
    ...  
}
```

Example named query XML

```
<entity-mappings>  
  <entity name="Employee" class="org.acme.Employee" access="FIELD">  
    <named-query name="findAllEmployeesInCity">  
      <query>Select emp from Employee emp where emp.address.city = :city</query>  
      <hint name="acme.jpa.batch" value="emp.address"/>  
    </named-query>  
    <attributes>  
      <id name="id"/>  
    </attributes>  
  </entity>  
</entity-mappings>
```



```
</entity>
</entity-mappings>
```

Example named query execution

```
EntityManager em = getEntityManager();
Query query = em.createNamedQuery("findAllEmployeesInCity");
query.setParameter("city", "Ottawa");
List<Employee> employees = query.getResultList();
...

```

Dynamic Queries

Dynamic queries are normally used when the query depends on the context. For example, depending on which items in the query form were filled in, the query may have different parameters. Dynamic queries are also useful for uncommon queries, or prototyping. Because JPQL is a string based language, dynamic queries using JPQL typically involve string concatenation. Some JPA providers provide more dynamic query languages, and in JPA 2.0 a Criteria API will be provided to make dynamic queries easier.

Dynamic queries can use parameters, and query hints the same as named queries.

Example dynamic query execution

```
EntityManager em = getEntityManager();
Query query = em.createQuery("Select emp from Employee emp where
emp.address.city = :city");
query.setParameter("city", "Ottawa");
query.setHint("acme.jpa.batch", "emp.address");
List<Employee> employees = query.getResultList();
...

```

Common Queries

Inverse ManyToMany, all employees for a given project

To query *all employees for a given project* where the employee project relationship is a ManyToMany.

If the relationship is bi-directional you could use:

```
Select project.employees from Project project where project.name = :name
```

If it is uni-directional you could use:

```
Select employee from Employee employee, Project project where
project.name = :name and project member of employee.projects
```

or,

```
Select employee from Employee employee join employee.projects project
where project.name = :name
```

How to simulate casting to a subclass

To query *all employees who have a large project with a budget greater than 1,00,000* where the employee only has a relationship to Project, not to the LargeProject subclass. JPA 1.0 JPQL does not define a cast operation (JPA 2.0 may define this), so querying on an attribute of a subclass is not obvious. This can be done indirectly however, if you add a secondary join to the subclass to the query.

```
Select employee from Employee employee join employee.projects project,
LargeProject lproject where project = lproject and lproject.budget >
1000000
```

Advanced

Flush Mode

Within a transaction context in JPA, changes made to the managed objects are normally not flushed (written) to the database until commit. So if a query were executed against the database directly, it would not see the changes made within the transaction, as these changes are only made in memory within the Java. This can cause issues if new objects have been persisted, or objects have been removed or changed, as the application may expect the query to return these results. Because of this JPA requires that the JPA provider performs a flush of all changes to the database before any query operation. This however can cause issues if the application is not expecting that a flush as a side effect of a query operation. If the application changes are not yet in a state to be flushed, a flush may not be desired. Flushing also can be expensive and causes the database transaction, and database locks are other resources to be held for the duration of the transaction, which can effect performance and concurrency.

JPA allows the flush mode for a query to be configured using the FlushModeType (<https://java.sun.com/javaee/5/docs/api/javax/persistence/FlushModeType.html>) enum and the Query.setFlushMode() ([https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setFlushMode\(javax.persistence.FlushModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html#setFlushMode(javax.persistence.FlushModeType))) API. The flush mode is either AUTO the default which means flush before every query execution, or COMMIT which means only flush on commit. The flush mode can also be set on an EntityManager using the EntityManager.setFlushMode() ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#setFlushMode\(javax.persistence.FlushModeType\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#setFlushMode(javax.persistence.FlushModeType))) API, to affect all queries executed with the EntityManager. The EntityManager.flush() ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#flush\(\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#flush())) API can be called directly on the EntityManager anytime that a flush is desired.

Some JPA providers also let the flush mode be configured through persistence unit properties, or offer alternatives to flushing, such as performing the query against the in memory objects.

Native SQL Queries

Typically queries in JPA are defined through JPQL. JPQL allows the queries to be defined in terms of the object model, instead of the data model. Since developers are programming in Java using the object model, this is normally more intuitive. This also allows for data abstraction and database schema and database platform independence. JPQL supports much of the SQL syntax, but some aspects of SQL, or specific database extensions or functions may not be possible through JPQL, so native SQL queries are sometimes required. Also some developers have more experience with SQL than JPQL, so may prefer to use SQL queries. Native queries can also be used for calling some types of stored procedures or executing DML or DDL operations.

Native queries are defined through the `@NamedNativeQuery` (<https://java.sun.com/javase/5/docs/api/javax/persistence/NamedNativeQuery.html>) and `@NamedNativeQueries` (<https://java.sun.com/javase/5/docs/api/javax/persistence/NamedNativeQueries.html>) annotations, or `<named-native-query>` XML element. Native queries can also be defined dynamically using the `EntityManager.createNativeQuery()` ([https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#createNativeQuery\(java.lang.String\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#createNativeQuery(java.lang.String))) API.

A native query can be for a query for instances of a class, a query for raw data, an update or DML or DDL operation, or a query for a complex query result. If the query is for a class, the `resultClass` attribute of the query must be set. If the query result is complex, a Result Set Mapping can be used.

Native queries can be parameterized, so they can be executed with different parameter values. Parameters are defined in SQL using the `?` syntax for positional parameters, JPA does not require native queries support named parameters, but some JPA providers may.

A collection of query hints can also be provided to a native query. Query hints can be used to optimize or to provide special configuration to a query. Query hints are specific to the JPA provider. Query hints are defined through the `@QueryHint` (<https://java.sun.com/javase/5/docs/api/javax/persistence/QueryHint.html>) annotation or `query-hint` XML element.

Example native named query annotation

```
@NamedNativeQuery(  
    name="findAllEmployeesInCity",  
    query="SELECT E.* from EMP E, ADDRESS A WHERE E.EMP_ID = A.EMP_ID AND  
A.CITY = ?",  
    resultClass=Employee.class  
)  
public class Employee {  
    ...  
}
```

Result Set Mapping

Example native named query XML

```
<entity-mappings>
  <entity name="Employee" class="org.acme.Employee" access="FIELD">
    <named-native-query name="findAllEmployeesInCity" result-class="org.acme.Employee">
      <query>SELECT E.* from EMP E, ADDRESS A WHERE E.EMP_ID = A.EMP_ID AND
A.CITY = ?</query>
    </named-native-query>
    <attributes>
      <id name="id"/>
    </attributes>
  </entity>
</entity-mappings>
```

Example native named query execution

```
EntityManager em = getEntityManager();
Query query = em.createNamedQuery("findAllEmployeesInCity");
query.setParameter(0, "Ottawa");
List<Employee> employees = query.getResultList();
...
```

Example dynamic native query execution

```
EntityManager em = getEntityManager();
Query query = em.createNativeQuery("SELECT E.* from EMP E, ADDRESS A
WHERE E.EMP_ID = A.EMP_ID AND A.CITY = ?", Employee.class);
query.setParameter(0, "Ottawa");
List<Employee> employees = query.getResultList();
...
```

Stored Procedures

A stored procedure is a database-defined procedure typically written in a proprietary database language, such as PL/SQL in Oracle.

JPQL BNF

The following defines the structure of the JPQL query language. For further examples and usage see Querying.

```
| = or
[] = optional
* = repeatable
{} = optional
```

```
QL_statement ::= select_statement | update_statement | delete_statement
```

Select

```
Select employee from Employee employee join employee.address address
where address.city = :city and employee.firstName like :name order by
employee.firstName
```

```
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
[having_clause] [orderby_clause]
```

From

```
FROM Employee employee JOIN FETCH employee.address LEFT OUTER JOIN
FETCH employee.phones JOIN employee.manager manager, Employee ceo
```

```
from_clause ::= FROM identification_variable_declaration {,
{identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS] identification_variable
join ::= join_spec join_association_path_expression [AS] identification_variable
fetch_join ::= join_spec FETCH join_association_path_expression
association_path_expression ::= collection_valued_path_expression |
single_valued_association_path_expression
join_spec ::= [ LEFT [OUTER] | INNER ] JOIN
join_association_path_expression ::= join_collection_valued_path_expression |
join_single_valued_association_path_expression
join_collection_valued_path_expression ::=
identification_variable.collection_valued_association_field
join_single_valued_association_path_expression ::=
identification_variable.single_valued_association_field
collection_member_declaration ::= IN (collection_valued_path_expression) [AS]
identification_variable
single_valued_path_expression ::= state_field_path_expression |
single_valued_association_path_expression
state_field_path_expression ::= {identification_variable |
single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
identification_variable.{single_valued_association_field.*} single_valued_association_field
collection_valued_path_expression ::=
identification_variable.{single_valued_association_field.*}collection_valued_association_field
state_field ::= {embedded_class_state_field.*}simple_state_field
```

Select Clause

```
SELECT employee.id, employee.phones
```

```
SELECT DISTINCT employee.address.city, NEW  
com.acme.EmployeeInfo(AVG(employee.salary), MAX(employee.salary))
```

```
select_clause ::= SELECT [DISTINCT] select_expression {, select_expression}*
```

```
select_expression ::= single_valued_path_expression | aggregate_expression |  
identification_variable | OBJECT(identification_variable) | constructor_expression
```

```
constructor_expression ::= NEW constructor_name ( constructor_item {,  
constructor_item}* )
```

```
constructor_item ::= single_valued_path_expression | aggregate_expression
```

```
aggregate_expression ::= { AVG | MAX | MIN | SUM } ([DISTINCT]  
state_field_path_expression) | COUNT ([DISTINCT] identification_variable |  
state_field_path_expression | single_valued_association_path_expression)
```

Where

```
WHERE employee.firstName = :name AND employee.address.city LIKE 'Ott%'  
ESCAPE '/' OR employee.id IN (1, 2, 3) AND (employee.salary * 2) > 40000
```

```
where_clause ::= WHERE conditional_expression
```

```
conditional_expression ::= conditional_term | conditional_expression OR conditional_term
```

```
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
```

```
conditional_factor ::= [ NOT ] conditional_primary
```

```
conditional_primary ::= simple_cond_expression | (conditional_expression)
```

```
simple_cond_expression ::= comparison_expression | between_expression | like_expression |  
in_expression | null_comparison_expression | empty_collection_comparison_expression |  
collection_member_expression | exists_expression
```

```
between_expression ::= arithmetic_expression [NOT] BETWEEN arithmetic_expression  
AND arithmetic_expression | string_expression [NOT] BETWEEN string_expression AND  
string_expression | datetime_expression [NOT] BETWEEN datetime_expression AND  
datetime_expression
```

```
in_expression ::= state_field_path_expression [NOT] IN ( in_item {, in_item}* | subquery)
```

```
in_item ::= literal | input_parameter
```

```
like_expression ::= string_expression [NOT] LIKE pattern_value [ESCAPE  
escape_character]
```

```
null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS  
[NOT] NULL
```

```
empty_collection_comparison_expression ::= collection_valued_path_expression IS [NOT]  
EMPTY
```

```
collection_member_expression ::= entity_expression [NOT] MEMBER [OF]  
collection_valued_path_expression
```

```
exists_expression ::= [NOT] EXISTS (subquery)
```

```
all_or_any_expression ::= { ALL | ANY | SOME } (subquery)
```

```

comparison_expression ::= string_expression comparison_operator {string_expression |
all_or_any_expression} | boolean_expression { =|<> } {boolean_expression |
all_or_any_expression} | enum_expression { =|<> } {enum_expression |
all_or_any_expression} | datetime_expression comparison_operator {datetime_expression |
all_or_any_expression} | entity_expression { = | <> } {entity_expression |
all_or_any_expression} | arithmetic_expression comparison_operator
{arithmetic_expression | all_or_any_expression}
comparison_operator ::= = | > | >= | < | <= | <>
arithmetic_expression ::= simple_arithmetic_expression | (subquery)
simple_arithmetic_expression ::= arithmetic_term | simple_arithmetic_expression { + | - }
arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term { * | / } arithmetic_factor
arithmetic_factor ::= [{ + | - }] arithmetic_primary
arithmetic_primary ::= state_field_path_expression | numeric_literal |
(simple_arithmetic_expression) | input_parameter | functions_returning_numerics |
aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::= state_field_path_expression | string_literal | input_parameter |
functions_returning_strings | aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::= state_field_path_expression | input_parameter |
functions_returning_datetime | aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::= state_field_path_expression | boolean_literal | input_parameter |
enum_expression ::= enum_primary | (subquery)
enum_primary ::= state_field_path_expression | enum_literal | input_parameter |
entity_expression ::= single_valued_association_path_expression | simple_entity_expression
simple_entity_expression ::= identification_variable | input_parameter

```

Functions

```
LENGTH(SUBSTRING(UPPER(CONCAT('FOO', :bar))), 1, 5))
```

```

functions_returning_numerics ::= LENGTH(string_primary) | LOCATE(string_primary,
string_primary[, simple_arithmetic_expression]) | ABS(simple_arithmetic_expression) |
SQRT(simple_arithmetic_expression) | MOD(simple_arithmetic_expression,
simple_arithmetic_expression) | SIZE(collection_valued_path_expression)

```

```

functions_returning_datetime ::= CURRENT_DATE | CURRENT_TIME |
CURRENT_TIMESTAMP

```

```

functions_returning_strings ::= CONCAT(string_primary, string_primary) |
SUBSTRING(string_primary, simple_arithmetic_expression, simple_arithmetic_expression) |
TRIM([[trim_specification] [trim_character] FROM] string_primary) |
LOWER(string_primary) | UPPER(string_primary)

```

```
trim_specification ::= LEADING | TRAILING | BOTH
```

Group By

```
GROUP BY employee.address.country, employee.address.city HAVING
COUNT(employee.id) > 500
```

groupby_clause ::= GROUP BY groupby_item {, groupby_item}*

groupby_item ::= single_valued_path_expression | identification_variable

having_clause ::= HAVING conditional_expression

Order By

```
ORDER BY employee.address.country, employee.address.city DESC
```

orderby_clause ::= ORDER BY orderby_item {, orderby_item}*

orderby_item ::= state_field_path_expression [ASC | DESC]

Subquery

```
WHERE employee.salary = (SELECT MAX(wellPaid.salary) FROM Employee
wellPaid)
```

subquery ::= simple_select_clause subquery_from_clause [where_clause] [groupby_clause] [having_clause]

subquery_from_clause ::= FROM subselect_identification_variable_declaration {, subselect_identification_variable_declaration}*

subselect_identification_variable_declaration ::= identification_variable_declaration | association_path_expression [AS] identification_variable | collection_member_declaration

simple_select_clause ::= SELECT [DISTINCT] simple_select_expression

simple_select_expression ::= single_valued_path_expression | aggregate_expression | identification_variable

Update

```
UPDATE Employee SET salary = salary * 2 WHERE address.city = :city
```

update_statement ::= update_clause [where_clause]

update_clause ::= UPDATE abstract_schema_name [[AS] identification_variable] SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field | single_valued_association_field} = new_value

new_value ::= simple_arithmetic_expression | string_primary | datetime_primary | boolean_primary | enum_primary simple_entity_expression | NULL

Delete

```
DELETE FROM Employee WHERE address.city = :city
```

```
delete_statement ::= delete_clause [where_clause]
```

```
delete_clause ::= DELETE FROM abstract_schema_name [[AS] identification_variable]
```

Persisting

JPA uses the `EntityManager` (<https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html>) API for runtime usage. The `EntityManager` represents the application session or dialog with the database. Each request, or each client will use its own `EntityManager` to access the database. The `EntityManager` also represents a transaction context, and in a typical stateless model a new `EntityManager` is created for each transaction. In a stateful model, an `EntityManager` may match the lifecycle of a client's session.

The `EntityManager` provides API for all required persistence operations. These includes CRUD operations including:

- `persist` ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist(java.lang.Object))) (INSERT)
- `merge` ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#merge\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#merge(java.lang.Object))) (UPDATE)
- `remove` ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#remove\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#remove(java.lang.Object))) (DELETE)
- `find` ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#find\(java.lang.Class,java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#find(java.lang.Class,java.lang.Object))) (SELECT)

The `EntityManager` is an object-oriented API, so does not map directly onto database SQL or DML operations. For example to update an object, you just need to read the object and change its state through its `set` methods, and then call `commit` on the transaction. The `EntityManager` figures out which objects you changed and performs the correct updates to the database, there is no explicit update operation in JPA.

Persist

The `EntityManager.persist()` ([https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist\(java.lang.Object\)](https://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html#persist(java.lang.Object))) operation is used to insert a new object into the database. `persist` does not directly insert the object into the database, it just registers it as new in the persistence context (transaction). When the transaction is committed, or if the persistence context is *flushed*, then the object will be inserted into the database.

If the object uses a generated `Id`, the `Id` will normally be assigned to the object when `persist` is called, so `persist` can also be used to have an object's `Id` assigned. The one exception is if `IDENTITY` sequencing is used, in this case the `Id` is only assigned on `commit` or `flush` because the database will only assign the `Id` on `INSERT`. If the object does not use a generated `Id`, you should normally assign its `Id` before calling `persist`.

The `persist` operation can only be called within a transaction, an exception will be thrown outside of a transaction. The `persist` operation is in-place, in that the object being persisted will become part of the persistence context. The state of the object at the point of

the commit of the transaction will be persisted, not its state at the point of the `persist` call.

`persist` should normally only be called on new objects. It is allowed to be called on existing objects if they are part of the persistence context, this is only for the purpose of cascading `persist` to any possible related new objects. If `persist` is called on an existing object that is not part of the persistence context, then an exception may be thrown, or it may be attempted to be inserted and a database constraint error may occur, or if no constraints are defined, it may be possible to have duplicate data inserted.

`persist` can only be called on `Entity` objects, not on `Embeddable` objects, or collections, or non-persistent objects. `Embeddable` objects are automatically persisted as part of their owning `Entity`.

Calling `persist` is not always required. If you related a new object to an existing object that is part of the persistence context, and the relationship is `cascade persist`, then it will be automatically inserted when the transaction is committed, or when the persistence context is flushed.

Example `persist`

```
EntityManager em = getEntityManager();
em.getTransaction().begin();

Employee employee = new Employee();
employee.setFirstName("Bob");
Address address = new Address();
address.setCity("Ottawa");
employee.setAddress(address);

em.persist(employee);

em.getTransaction().commit();
```

Cascading `Persist`

Calling `persist` on an object will also cascade the `persist` operation to across any relationship that is marked as `cascade persist`. If a relationship is not `cascade persist`, and a related object is new, then an exception may be thrown if you do not first call `persist` on the related object. Intuitively you may consider marking every relationship as `cascade persist` to avoid having to worry about calling `persist` on every objects, but this can also lead to issues.

One issue with marking all relationships `cascade persist` is performance. On each `persist` call all of the related objects will need to be traversed and checked if they reference any new objects. This can actually lead to n^2 performance issues if you mark all relationships `cascade persist`, and `persist` a large new graph of objects. If you just call `persist` on the root object, this is ok. However, if you call `persist` on each object in the graph, then you will traverse the entire graph for each object in the graph, and this can lead to a major performance issue. The JPA spec should probably define `persist` to only apply to new objects, not already part of the persistence context, but it requires `persist` apply to all objects, whether new, existing, or already persisted, so can have this issue.

A second issue is that if you remove an object to have it deleted, if you then call `persist` on the object, it will resurrect the object, and it will become persistent again. This may be desired if it is intentional, but the JPA spec also requires this behavior for `cascade persist`. So if you remove an object, but forget to remove a reference to it from a `cascade persist` relationship, the `remove` will be ignored.

I would recommend only marking relationships that are composite or privately owned as `cascade persist`.

Merge

The `EntityManager.merge()` ([https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#merge\(java.lang.Object\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#merge(java.lang.Object))) operation is used to merge the changes made to a detached object into the persistence context. `merge` does not directly update the object into the database, it merges the changes into the persistence context (transaction). When the transaction is committed, or if the persistence context is *flushed*, then the object will be updated in the database.

Normally `merge` is not required, although it is frequently misused. To update an object you simply need to read it, then change its state through its `set` methods, then commit the transaction. The `EntityManager` will figure out everything that has been changed and update the database. `merge` is only required when you have a detached copy of a persistence object. A *detached* object is one that was read through a different `EntityManager` (or in a different transaction in a JEE managed `EntityManager`), or one that was cloned, or serialized. A common case is a stateless `SessionBean` where the object is read in one transaction, then updated in another transaction. Since the update is processed in a different transaction, with a different `EntityManager`, it must first be merged. The `merge` operation will look-up/find the managed object for the detached object, and copy each of the detached objects attributes that changed into the managed object, as well as cascading any related objects marked as `cascade merge`.

The `merge` operation can only be called within a transaction, an exception will be thrown outside of a transaction. The `merge` operation is not in-place, in that the object being merged will never become part of the persistence context. Any further changes must be made to the managed object returned by the `merge`, not the detached object.

`merge` is normally called on existing objects, but can also be called on new objects. If the object is new, a new copy of the object will be made and registered with the persistence context, the detached object will not be persisted itself.

`merge` can only be called on `Entity` objects, not on `Embeddable` objects, or collections, or non-persistent objects. `Embeddable` objects are automatically merged as part of their owning `Entity`.

Example merge

```
EntityManager em = createEntityManager();
Employee detached = em.find(Employee.class, id);
em.close();
...
em = createEntityManager();
em.getTransaction().begin();
Employee managed = em.merge(detached);
```

```
em.getTransaction().commit();
```

Cascading Merge

Calling `merge` on an object will also cascade the `merge` operation across any relationship that is marked as cascade merge. Even if the relationship is not cascade merge, the reference will still be merged. If the relationship is cascade merge the relationship and each related object will be merged. Intuitively you may consider marking every relationship as cascade merge to avoid having to worry about calling `merge` on every objects, but this is normally a bad idea.

One issue with marking all relationships cascade merge is performance. If you have an object with a lot of relationships, the each `merge` call can require to traverse a large graph of objects.

Another issues is that your detached object is corrupt in one way or another. By this I mean you have, for example, an `Employee` who has a `manager`, but that `manager` has a different copy of the detached `Employee` object as its `managedEmployee`. This may cause the same object to be merged twice, or at least may not be consistent which object will be merged, so you may not get the changes you expect merged. The same is true if you didn't change an object at all, but some other user did, if `merge` cascades to this unchanged object, it will revert the other user's changes, or throw an `OptimisticLockException` (depending on your locking policy). This is normally not desirable.

I would recommend only marking relationships that are composite or privately owned as cascade merge.

Transient Variables

Another issue with `merge` is transient variables. Since `merge` is normally used with object serialization, if a relationship was marked as transient (Java transient, not JPA transient), then the detached object will contain `null`, and `null` will be merged into the object, even though it is not desired. This will occur even if the relationship was not cascade merge, as `merge` always merges the references to related objects. Normally transient is required when using serialization to avoid serializing the entire database when only a single, or small set of objects are required.

One solution is to avoid marking anything transient, and instead use LAZY relationships is JPA to limit what is serialized (lazy relationships that have not been accessed, will normally not be serialized). Another solution is to manually merge in your own code.

Some JPA provides provide extended `merge` operations, such as allowing a *shallow* merge or *deep* merge, or merging without merging references.

Remove

The `EntityManager.remove()` ([https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#remove\(java.lang.Object\)](https://java.sun.com/javase/5/docs/api/javax/persistence/EntityManager.html#remove(java.lang.Object))) operation is used to delete an object from the database. `remove` does not directly delete the object from the database, it marks the object to be deleted in the persistence context (transaction). When the transaction is committed, or if the persistence context is *flushed*, then the object will be deleted from the database.

The `remove` operation can only be called within a transaction, an exception will be thrown outside of a transaction. The `remove` operation must be called on a managed object, not on a detached object. Generally you must first `find` the object before removing it, although it is possible to call `EntityManager.getReference()` on the object's `Id` and call `remove` on the reference. Depending on how your JPA provider optimizes `getReference` and `remove`, it may not require reading the object from the database.

`remove` can only be called on `Entity` objects, not on `Embeddable` objects, or collections, or non-persistent objects. `Embeddable` objects are automatically removed as part of their owning `Entity`.

Example merge

```
EntityManager em = getEntityManager();
em.getTransaction().begin();
Employee employee = em.find(Employee.class, id);
em.remove(employee);
em.getTransaction().commit();
```

Cascading Remove

Calling `remove` on an object will also cascade the `remove` operation across any relationship that is marked as `cascade remove`.

Note that `cascade remove` only effects the `remove` call. If you have a relationship that is `cascade remove`, and `remove` an object from the collection, or dereference an object, it will *not* be removed. You must explicitly call `remove` on the object to have it deleted. Some JPA providers provide an extension to provide this behavior, and in JPA 2.0 there will be an `orphanRemoval` option on `OneToMany` and `OneToOne` mappings to provide this.

Reincarnation

Normally an object that has been removed, stays removed, but in some cases you may need to bring the object back to life. This normally occurs with natural ids, not generated ones, where a new object would always get a new id. Generally the desire to reincarnate an object occurs from a bad object model design, normally the desire to change the class type of an object (which cannot be done in Java, so a new object must be created). Normally the best solution is to change your object model to have your object hold a *type* object which defines its type, instead of using inheritance. But sometimes reincarnation is desirable.

When done in two separate transactions, this is normally fine, first you `remove` the object, then you `persist` it back. This can be more complex if you wish to `remove` and `persist` an object with the same `Id` in the same transaction. If you call `remove` on an object, then call `persist` on the same object, it will simply no longer be removed. If you call `remove` on an object, then call `persist` on a different object with the same `Id` the behavior may depend on your JPA provider, and probably will not work. If you call `flush` after calling `remove`, then call `persist`, then the object should be successfully reincarnated. Note that it will be a different row, the existing row will have been deleted, and a new row inserted. If you wish the same row to be updated, you may need to resort to using a native SQL update query.

Advanced

Get Reference

Clear

Get Delegate

A transaction is a set of operations that either fail or succeed as a unit. Transactions are a fundamental part of persistence. A database transaction consists of a set of SQL DML (Data Manipulation Language) operations that are committed or rolled back as a single unit. An object level transaction is one in which a set of changes made to a set of objects are committed to the database as a single unit.

JPA provides two mechanisms for transactions. When used in JEE JPA provides integration with JTA (Java Transaction API). JPA also provides its own `Transaction` implementation for JSE and for use in a non-managed mode in JEE. Transactions in JPA are always at the object level, this means that all changes made to all persistent objects in the persistence context are part of the transaction.

Caching

Caching is the most important performance optimization technique. There are many things that can be cached in persistence, objects, data, database connections, database statements, query results, meta-data, relationships, to name a few. Caching in object persistence normally refers to the caching of objects or their data. Caching also influences object identity, that is that if you read an object, then read the same object again you should get the identical object back (same reference).

JPA does not define a server object cache, JPA providers can support a server object cache or not, however most do. Caching in JPA is required within a transaction or within an extended persistence context to preserve object identity, but JPA does not require that caching be supported across transactions or persistence contexts.

There are two types of object caching. You can cache the objects themselves including all of their structure and relationships, or you can cache their database row data. Both provide a benefit, however just caching the row data is missing a huge part of the caching benefit as the retrieval of each relationship typically involves a database query, and the bulk of the cost of reading an object is spent in retrieving its relationships.

Data Cache

Object Identity

Stale Data

Refreshing

Caching in a Cluster

Databases

MySQL

Using MySQL with Java Persistence API is quite straightforward as the JDBC driver is available directly from MySQL web site (<http://dev.mysql.com/>).

MySQL Connector/J is the official JDBC driver for MySQL and has a good documentation available directly on the web site.

In this page we will see some aspects of managing MySQL using the Persistence API.

Installing

Installation is straightforward and consists in making the .jar file downloaded from MySQL web site visible to the JVM. It can be already installed if you are using Apache or JBOSS.

Configuration tips

You can learn a lot from the documentation on MySQL web site (<http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-configuration-properties.html>).

Creating the database automatically

If you intend to create table and a database automatically , you will need to have the correct user rights but also inform the Persistence API about the name of the Database you want to create. For example with TopLink, if you used:

```
<property name="toplink.ddl-generation" value="create-tables"/>
```

in the property.xml, you will be likely need to create a new database. To create the database "NewDB" automatically, you need to give the following URL to the jdbc connection:

```
<property name="toplink.jdbc.url"
value="jdbc:mysql://localhost:3306/NewDB?createDatabaseIfNotExist=true"/>
```

If not, the persistence API will complain that the database does not exist.

References



Resources

- EJB 3.0 JPA 1.0 Spec (<http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>)
- JPA 2.0 Spec (<http://jcp.org/en/jsr/detail?id=317>)
- JPA 1.0 ORM XML Schema (http://java.sun.com/xml/ns/persistence/orm_1_0.xsd)
- JPA 1.0 Persistence XML Schema (http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd)
- JPA 1.0 JavaDoc (<https://java.sun.com/javaee/5/docs/api/javax/persistence/package-summary.html>)
- JPQL BNF
- JPA 2.0 Reference Implementation Development (<http://wiki.eclipse.org/EclipseLink/Development/JPA>)
- Java Programming



Forums

- Sun EJB Forum (<http://forum.java.sun.com/forum.jspa?forumID=13>)
- JavaRanch ORM Forum (<http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=forum&f=78>)
- Nabble JPA Forum (<http://www.nabble.com/JPA-f27109.html>)
- EclipseLink Forum (<http://www.nabble.com/EclipseLink-f26430.html>)
- EclipseLink Newsgroup (<http://www.eclipse.org/newsportal/thread.php?group=eclipse.rt.eclipselink>)
- Oracle TopLink Forum (<http://forums.oracle.com/forums/forum.jspa?forumID=48>)
- Hibernate Forum (<http://forum.hibernate.org/>)
- TopLink Essentials Mailing List (Glassfish persistence) (<http://www.nabble.com/java.net---glassfish-persistence-f13455.html>)



Wikis

- EclipseLink Wiki (<http://wiki.eclipse.org/EclipseLink>)
- Oracle TopLink Wiki (<http://wiki.oracle.com/page/TopLink>)
- Glassfish TopLink Essentials Wiki (<http://wiki.glassfish.java.net/Wiki.jsp?page=TopLinkEssentials>)
- Hibernate Wiki (<http://www.hibernate.org/37.html>)
- JPA on Wikipedia (http://en.wikipedia.org/wiki/Java_Persistence_API)
- JPA on Javapedia (<http://wiki.java.net/bin/view/Javapedia/JPA>)
- JPA on freebase (<http://www.freebase.com/view/guid/9202a8c04000641f8000000004666d33>)
- JPA on DMOZ (Open Directory) (http://www.dmoz.org/Computers/Programming/Languages/Java/Databases_and_Persistence/Object_Persistence/JPA/)



Products

- Oracle TopLink Home (<http://www.oracle.com/technology/products/ias/toplink/index.html>)
- EclipseLink Home (<http://www.eclipse.org/eclipselink/>)
- TopLink Essentials Home (<https://glassfish.dev.java.net/javaee5/persistence/>)
- Hibernate Home (<http://www.hibernate.org/>)
- Open JPA Home (<http://openjpa.apache.org/>)
- HiberObjects (<http://objectgeneration.com/eclipse/>)

Blogs

- Java Persistence (<http://java-persistence.blogspot.com/>) (Doug Clarke)
- System.out (<http://jroller.com/mkeith/>) (Mike Keith)
- On TopLink (<http://ontoplink.blogspot.com/>) (Shaun Smith)
- EclipseLink (<http://eclipselink.blogspot.com/>)
- Hibernate Blog (<http://blog.hibernate.org/>)

Books

- Pro EJB 3 (<http://www.amazon.com/gp/product/1590596455/102-2412923-9620152?v=glance&n=283155>)

Article Sources and Contributors

Java Persistence/Print version *Source:* <http://en.wikibooks.org/w/index.php?oldid=1569890> *Contributors:* Jamessss

Image Sources, Licenses and Contributors

Image:Java-persistence.PNG *Source:* <http://en.wikibooks.org/w/index.php?title=File:Java-persistence.PNG> *License:* Public Domain *Contributors:* Jamessss

Image:Vraagteken.svg *Source:* <http://en.wikibooks.org/w/index.php?title=File:Vraagteken.svg> *License:* Public Domain *Contributors:* Roland Geider

Image:Crystal Clear app Community Help.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Crystal_Clear_app_Community_Help.png *License:* unknown *Contributors:* Abu badali, CyberSkull, It Is Me Here, Martin Kozák, OsamaK

Image:Crystal Clear app kcoloredit.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Crystal_Clear_app_kcoloredit.png *License:* unknown *Contributors:* AVRS, CyberSkull, It Is Me Here

Image:Litchar.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:Litchar.png> *License:* GNU Lesser General Public License *Contributors:* User:Pegship

Image:Employee-model.PNG *Source:* <http://en.wikibooks.org/w/index.php?title=File:Employee-model.PNG> *License:* Public Domain *Contributors:* Jamessss

Image:EMPLOYEE Table (Database).PNG *Source:* [http://en.wikibooks.org/w/index.php?title=File:EMPLOYEE_Table_\(Database\).PNG](http://en.wikibooks.org/w/index.php?title=File:EMPLOYEE_Table_(Database).PNG) *License:* Public Domain *Contributors:* User:Jamessss

Image:Emp Tables (Database).PNG *Source:* [http://en.wikibooks.org/w/index.php?title=File:Emp_Tables_\(Database\).PNG](http://en.wikibooks.org/w/index.php?title=File:Emp_Tables_(Database).PNG) *License:* Public Domain *Contributors:* User:Jamessss

Image:Emp Add Tables (Database).PNG *Source:* [http://en.wikibooks.org/w/index.php?title=File:Emp_Add_Tables_\(Database\).PNG](http://en.wikibooks.org/w/index.php?title=File:Emp_Add_Tables_(Database).PNG) *License:* Public Domain *Contributors:* User:Jamessss

Image:Emp Adds Tables (Database).PNG *Source:* [http://en.wikibooks.org/w/index.php?title=File:Emp_Adds_Tables_\(Database\).PNG](http://en.wikibooks.org/w/index.php?title=File:Emp_Adds_Tables_(Database).PNG) *License:* Public Domain *Contributors:* User:Jamessss

File:Crystal_Clear_app_password.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Crystal_Clear_app_password.png *License:* unknown *Contributors:* Color probe, CyberSkull

Image:Cascaded-keys.PNG *Source:* <http://en.wikibooks.org/w/index.php?title=File:Cascaded-keys.PNG> *License:* Public Domain *Contributors:* User:Jamessss

Image:Inheritance.PNG *Source:* <http://en.wikibooks.org/w/index.php?title=File:Inheritance.PNG> *License:* Public Domain *Contributors:* Jamessss

Image:embeddable.PNG *Source:* <http://en.wikibooks.org/w/index.php?title=File:Embeddable.PNG> *License:* Public Domain *Contributors:* Jamessss

Image:ObjectRelational-OneToOne.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:ObjectRelational-OneToOne.jpg> *License:* Public Domain *Contributors:* User:Jamessss

Image:ObjectRelational-ManyToOne.jpg *Source:* <http://en.wikibooks.org/w/index.php?title=File:ObjectRelational-ManyToOne.jpg> *License:* Public Domain *Contributors:* User:Jamessss

Image:Crystal Clear app reminders.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Crystal_Clear_app_reminders.png *License:* unknown *Contributors:* CyberSkull, Ysangkok

Image:Wikipedia.png *Source:* <http://en.wikibooks.org/w/index.php?title=File:Wikipedia.png> *License:* logo *Contributors:* Avatar, Mac, Schaengel89, i0-8-15!

Image:Nuvola apps edu languages.svg *Source:* http://en.wikibooks.org/w/index.php?title=File:Nuvola_apps_edu_languages.svg *License:* GNU Lesser General Public License *Contributors:* User:Stannered

Image:Crystal Clear app desktopshare.png *Source:* http://en.wikibooks.org/w/index.php?title=File:Crystal_Clear_app_desktopshare.png *License:* unknown *Contributors:* CyberSkull

License

Creative Commons Attribution-Share Alike 3.0 Unported
<http://creativecommons.org/licenses/by-sa/3.0/>