

Persistence in the Enterprise

A Guide to Persistence
Technologies

Roland Barcia, Geoffrey Hambrick, Kyle Brown,
Robert Peterson, Kulvir Singh Bhogal

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

© Copyright 2008 by International Business Machines Corporation. All rights reserved.

Note to U.S. Government Users: Documentation related to restricted right. Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

IBM Press Program Managers: Tara Woodman, Ellice Uffer

Cover design: IBM Corporation
Associate Publisher: Greg Wiegand
Marketing Manager: Kourtayne Sturgeon
Publicist: Heather Fox
Acquisitions Editor: Katherine Bull
Development Editor: Kevin Howard
Managing Editor: Gina Kanouse
Designer: Alan Clements
Senior Project Editor: Lori Lyons
Copy Editor: Cheri Clark
Indexer: Erika Millen
Senior Compositor: Gloria Schurick
Proofreader: Lori Lyons
Manufacturing Buyer: Dan Uhrig
Published by Pearson plc
Publishing as IBM Press

IBM Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both: IBM, the IBM logo, IBM Press, Cloudscape, DB2, DB2 Universal Database, developerWorks, Informix, Rational, VisualAge, and WebSphere. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.



This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days. Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it. To gain 45-day Safari Enabled access to this book:

- Go to <http://www.awprofessional.com/safarienabled>.
- Complete the brief registration form.
- Enter the coupon code 59M6-YSIL-3W4E-VPLD-PQTZ

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Library of Congress Cataloging-in-Publication Data

Persistence in the enterprise : a guide to persistence technologies / Roland Barcia ... [et al.].

p. cm.

ISBN 0-13-158756-0 (pbk. : alk. paper) 1. Object-oriented databases. 2. Java (Computer program language) I. Barcia, Roland.

QA76.9.D3P494 2008

005.75'7—dc22

2008001531

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-13-158756-4

ISBN-10: 0-13-158756-0

Text printed in the United States on recycled paper at Courier Westford in Westford, Massachusetts.
First printing May 2008

Introduction

Why You Should Steal This Book—with Apologies to Abbie Hoffman

Wise men often say that the first step to wisdom is learning that we know nothing—not even the right questions to ask. The implication is that we must listen to those with experience and contemplate the deeper meaning of their questions and answers. And in the end, a new experience may undo much of what everyone thought to be true.

This observation applies not only to sages on a mountaintop contemplating esoteric subjects such as philosophy, but also to those of us needing to know about a highly technical subject like providing persistent data within enterprise Java applications.

Imagine you were asked to choose the persistence mechanism for new Java applications within your company, and then review your architectural decision with various stakeholders in the company. Right at the start of the review, you would have to be prepared to answer some questions of general interest:

- Which persistence mechanisms were evaluated and why?
- Of those evaluated, which ones were chosen for use and why?

Then, depending on the exact role of the stakeholders invited to the review, you would have to be prepared to drill down and answer specific questions about the details of the proposed architecture that are relevant to each reviewer and the job each must do. For example:

- **Manager and Executive** roles evaluate the costs associated with making the transition to a new technology in terms of time, money, and other resources. They are interested in answers to questions about such topics as which vendors market a particular technology, the kinds of licensing agreements involved (and the restrictions those bring), the availability of skills, and references of successful applications using a given mechanism.

- **Analyst and Architect** roles need to assess whether a given mechanism can support the business and IT requirements of applications expected by the **End Users and Operators** for whom they serve as advocates. They are interested in questions about functionality, such as whether the mechanism supports relationships, a wide variety of attribute types, triggers, and constraints. They are also interested in whether the mechanism can scale up to response time, throughput, and other “nonfunctional” goals of the application.
- **Developer and Tester** roles are most impacted by the complexity of the framework and API in terms of the code they must write when implementing the services as objects mapped to a relational database layer. They are curious about how to handle specific coding tasks like creating connections, and reading/updating data in the context of the detailed database design for representative use cases.

The reality is that architecture of any type is more of an art than a science—which is what makes it such a challenging job to do well. It requires not just deep knowledge about methods and best practices that have worked in the past, but also a good sense of intuition about when to try something innovative.

A search of the Web for “Java and relational databases” returns a number of useful links to articles and books, but most are specific to a given mechanism—such as Java Database Connectivity (JDBC), Enterprise JavaBeans (EJB), Hibernate, or the new Java Persistence API (JPA). Some of these references go into detail on how to design the databases. Others are mainly guides on how to use the APIs to build a sample application. None of these references takes an end-to-end application architecture view that helps you understand the issues involved with choosing a persistence mechanism for relational data stores, and then helps you make a choice.

The reason that this end-to-end view is important is that, as noted above, good architects know the answers to the kinds of questions that will be asked by various stakeholder roles during a review. The better ones anticipate the questions and use those questions to drive their approach to design in the first place. But the best architects document these questions and answers in a cookbook form such that they guide every phase of an application development project, including analysis, design, construction, test, deployment, operations, and maintenance. Having the answers to these questions documented in a reusable form not only increases the quality of the applications, but also accelerates the development process—because the amount of time the team spends “reinventing the wheel” through trial and error is drastically reduced.

That end-to-end view is what makes this book different. We are all consultants with the IBM Software Services for WebSphere (ISSW) team. Our job is to help clients fully exploit our products, such as IBM WebSphere Application Server, WebSphere Portal Server, WebSphere Commerce Server, and WebSphere Process Server. We are often involved in proof of technology and head-to-head bake-offs that pair us with a client’s architects who are making build-versus-buy decisions or vendor and technology selections; so we are experts at taking this broader application-centric view and finding answers to tough questions.

This book looks at persistence frameworks, mostly Java based, in the same way so that you can propose a solution that satisfies all the stakeholders—from your CTO to your fellow architects, to the developers, testers, and operations team. And if you play one of these specific roles and find yourself in a review, this book will help you ask the right kinds of questions and be able to interpret the answers.

Another thing that makes this book different is that we endeavor to capture our unique on-the-job mentoring-based consulting approach. In a nutshell, we like to both “give you a fish” and “teach you how to catch it” at the same time. The ideal result of an ISSW engagement is that you have an early success with our products yet become self-sufficient for the future.

Although a book will never fully substitute for a live consultant directly engaging with you on a project, we make an attempt by informally dividing this book into two parts:

- **Part I, “A Question of Persistence,”** teaches you about fishing so that you can “eat for a lifetime.” Specifically, it helps you understand what the issues and trade-offs are in choosing a Java persistence mechanism for relational data stores. These issues are organized into chapters based on the questions asked by the various stakeholder roles described previously. Specifically, there are three chapters that cover the following topics: Chapter 1 provides a brief history of relevant persistence mechanisms; Chapter 2 covers business drivers and associated IT requirements; and Chapter 3 discusses implementation issues associated with object-relational mapping. We end this part of the book with Chapter 4, which extracts a questionnaire from the issues and trade-offs discussed in the first three chapters so that each mechanism can be evaluated with a consistent “yardstick” and best-practice-based approach.
- **Part II, “Comparing Apples to Apples,”** gives you some fish so that you can “eat for today.” Each chapter gathers the data for five popular mechanisms using the approach and questionnaire found in Chapter 4. In this section we explore Java Database Connectivity, iBATIS, Hibernate, Java Persistence API, and pureQuery as representative examples of the various approaches to Java persistence outlined in Chapter 1. We wrap up this part and the book itself with a summary in Chapter 10 that compares the mechanisms side by side and enumerates some scenarios in which one or the other best applies.

It has been said that “the more things change, the more they stay the same.” But they still change. The questionnaire developed in Chapter 4, “Evaluating Your Options,” is crucial to the longevity of this book because it is relatively easy to apply the same yardstick to new persistence frameworks and APIs as they become available. So to further enhance the value of the book, we include a reference to a download site for the questionnaire and code examples associated with the evaluations that you can use as is or customize for use within your own company as you evaluate these or other mechanisms.



Because this is a developerWorks book, we make use of some special references called “Links to developerWorks” that appear with a special icon in the margin and in their own section at the end the chapter. These references can be linked to the IBM developerWorks site at www.ibm.com/developerWorks/. We have found this feature to be a very exciting one because you can follow along on the site while you are reading the book and get instant access to those and other related articles and online books. The net effect is to extend this book into the World Wide Web, further enhancing its longevity.

We’ve also included a “References” section at the end of the chapters, which lists additional resources cited throughout the chapter. These resources are cited by [Author] within the chapter text.

We hope you agree that this book is worth stealing; but please take it to the nearest check-out counter or click the Add to Cart button now. If this happens to be a friend’s copy, please put it back on your friend’s desk or bookshelf and buy your own—it is available for purchase at online retail web sites and traditional brick-and-mortar stores that sell technical books about Java or database technologies.

Chapter 8

Apache OpenJPA

In this chapter we discuss OpenJPA, an implementation of the Java Persistence API 1.0 specification. We will fill in our evaluation template and implement our common example with OpenJPA. We will discuss OpenJPA extensions to the specification and finish with a brief look forward at JPA 2.0.

Background

In Chapter 1, “A Brief History of Object-Relational Mapping,” we gave some history of the Java Persistence space. We discussed the development of the EJB spec from a persistence standpoint, how it grew in developer dissatisfaction over the years, and how open-source frameworks like Hibernate began to grow. We also discussed how this led to the creation of JPA.

As discussed in Chapter 1, by the time version 2 of the Enterprise Java Beans specification started to make its way into products, a counter-current was building in the Java community that began looking for other ways of doing persistence. When the EJB 3.0 committee began meeting, it became clear that revisiting persistence would need to be a key feature of the new specification. The opinion of the committee was that something significant needed to be changed—and as a result, the committee made the following decisions:

- The EJB 3.0 persistence model would need to be a POJO-based model and would have to address the issue of “disconnected” or detached data in a distributed environment.
- The specification would need to support both annotations and XML descriptors to define the mapping between objects and relational databases.
- The mapping would need to be complete—specifying not only the abstract persistence of a class, but also its mapping to relational tables and mappings of attributes to columns.

So the EJB committee combined the best ideas from several sources—TopLink, Hibernate, and Java Data Objects API—to create a new persistence architecture, which was released as a separate part of EJB 3.0 and dubbed the Java Persistence API. JPA represents the confluence of a number of different threads in the Java persistence arena and has since been adopted by all the major persistence vendors and various open-source projects.

In the preceding chapter, we evaluated Hibernate. You will find many features in JPA similar to Hibernate. Hibernate itself has a full-blown JPA implementation as part of its Entity Manager that you can read about for more details [Hibernate]. In this chapter, however, we are going to use Apache OpenJPA.

The Apache OpenJPA project is an Apache-licensed open-source implementation of the Java Persistence API. OpenJPA is focused on building a robust, high-performance, scalable implementation of the JPA specification. You can read more about the Apache Open JPA project at the website [OpenJPA 1].

The original source code contribution was provided by BEA (via their SolarMetric Kodo acquisition, discussed in the “History” section to follow). Several other companies and individuals are participating as committers, contributors, and users in the OpenJPA project, including IBM. The OpenJPA community continues to grow and prosper, with the expectation of graduating from incubation sometime in the near future.

Type of Framework

Much like Hibernate, OpenJPA is a full Domain Mapper, allowing you to map a whole set of objects to your database tables, and abstracting the SQL language from the developer.

History

OpenJPA started its life with another specification called JDO, as discussed in Chapter 1. SolarMetric was one of the first implementers of the JDO specification back in 2001, with a product called Kodo. As the JPA specification began to finalize, SolarMetric began making Kodo both a JPA and JDO implementation. In 2005, BEA purchased SolarMetric and contributed most of the JPA code to Apache as the project OpenJPA. As stated earlier, other vendors like IBM are part of the OpenJPA community. BEA continues to have the SolarMetric Kodo product based on OpenJPA. OpenJPA will be the core persistence engine of BEA WebLogic Server, IBM WebSphere, and the Apache Geronimo Application Server. In May 2007, OpenJPA graduated from the incubator to a top-level project and also passed Sun’s Technology Compatibility Kit compliant with the Java Persistence API. In September 2007, OpenJPA released its first GA version.

Architectural Overview

Standards Adherence

OpenJPA is an implementation of the JPA 1.0 Specification that is a subspecification under the SUN EJB 3.0 specification, developed under JSR 220. At the time of this writing, JPA 2.0

is being developed under its own JSR 317, and EJB 3.1 is being developed under JSR 318. The website is the definitive source for the JPA 2.0 specifications [JPA 2].

Platforms Required

Before Java EE 5.0 and EJB 3.0, the persistence layer of the Java EE platform required a full-blown Java EE Application Server. JPA changes this. OpenJPA applications can be written to run in both Java SE and Java EE environments. However, there are some differences in JPA applications hosted inside a Java EE environment as opposed to a Java SE environment. Rather than discuss the differences here, we will highlight them where they exist throughout the remainder of this chapter.

Other Dependencies

OpenJPA comes bundled with several other JARs needed to make it run. Like many Apache projects, it makes use of other Apache licensed packages:

- Several of the Apache Commons projects :`commons-lang`, `commons-logging`, `commons-pool`, and `commons-collections`. Please refer to the Apache Commons website for more information [Apache].
- Apache OpenJPA relies on the Serp project for Java “bytecode enhancement” to add persistence behavior to annotated Java files as a separate step. The Serp JAR comes bundled with OpenJPA. We discuss this step in the later section “Development Process for the Common Example.” You can read more about Serp at their website [Serp].
- A valid JDBC database driver.
- For Java EE applications, any JARs required to run OpenJPA in a target Java EE Server.

Vendors and Licenses

OpenJPA is distributed under an Apache License. As mentioned in Chapter 2, an Apache License is more liberal in what you can do with the source because you can change parts of the code, and you don’t need to distribute them back to the original authors. In addition, several commercial products ship an OpenJPA implementation, such as IBM WebSphere Application Server and BEA WebLogic Server.

It is worth noting that because JPA is a specification, there are other JPA implementations available beyond the OpenJPA version:

- Hibernate JPA. Refer to Chapter 7, “Hibernate Core,” or the Hibernate website [Hibernate] for license information.
- TopLink Essentials, which is an open-source implementation that Oracle built on top of TopLink. We discussed TopLink briefly in Chapter 1. See their website for more information [TopLink].

Available Literature

OpenJPA is very well documented on their website. In addition, there are several articles available. Table 8.1 shows some examples.

Table 8.1 OpenJPA Resources

Resource	Link	Description
<i>OpenJPA Manuals</i> [OpenJPA 2]	openjpa.apache.org/ documentation.html	Comprehensive OpenJPA Module
<i>Integrating OpenJPA with Application Servers</i> [OpenJPA 3]	openjpa.apache.org/ integration.html	List of other articles for integrating OpenJPA with other Application Server.
<i>Building EJB 3 Applications with WebSphere Application Server</i>	www.ibm.com/ developerworks/websphere/ techjournal/0712_barcia/ 0712_barcia.html	Tutorial on using OpenJPA inside the WebSphere EJB 3 Container
 <i>Leveraging OpenJPA with WebSphere Application Server</i> A.8.1	www-128.ibm.com/ developerworks/websphere/ techjournal/0612_barcia/ 0612_barcia.html	Tutorial on using OpenJPA inside WebSphere Application Server
 <i>Java Persistence with Hibernate</i> [Bauer] A.8.2	www.amazon.com/ Java-Persistence-Hibernate- Christian-Bauer/dp/ 1932394885/ref=pd_bbs_sr_ 1?ie=UTF8&s=books&qid= 1201195747&sr=1-1	Book on using Hibernate JPA API
<i>Enterprise JavaBeans, 5th Edition</i> [Monson-Haefel]	www.amazon.com/ Enterprise-JavaBeans-3-0-Bill- Burke/dp/059600978X/ref= pd_bbs_sr_2?ie=UTF8&s= books&qid=1201195725& sr=8-2	Comprehensive sourcebook on EJB 3
 <i>Migrating Legacy Hibernate Applications to OpenJPA and EJB 3</i> A.8.3	www.ibm.com/ developerworks/websphere/ techjournal/0708_vines/ 0708_vines.html	Techniques useful to migrate a Hibernate Core Application to OpenJPA

Besides OpenJPA, there are many other resources available on the JPA programming model.

Programming Model

OpenJPA is a persistence framework based around persisting common POJOs. It relies on Java Annotations and/or XML to add the persistence behavior. Both options are available. OpenJPA also provides a very rich Object Query Language, batch statement features, and other useful features. As a last resort, OpenJPA allows a developer to drop down to Native SQL.

The basic programming model is relatively straightforward, with an EntityManager (created using an EntityManagerFactory or injected by the environment such as an EJB 3 container) being used to establish a persistent session context in which Plain Old Java Objects (POJOs) can be moved in and out of a persistent context.

It is worth mentioning here that an object association with an EntityManager defines the state in which it is managed. Figure 8.1 illustrates the life cycle of a Java object with respect to persistence in OpenJPA.

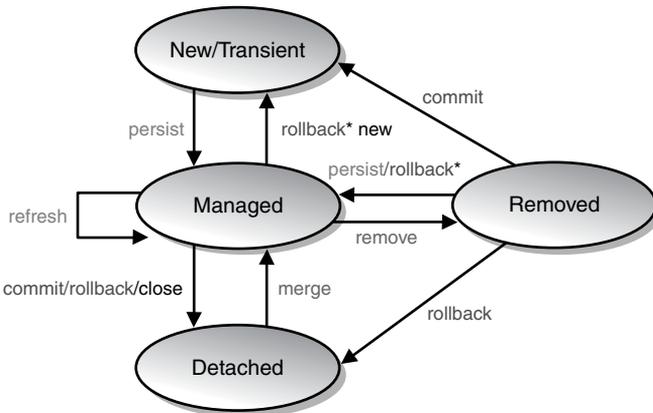


Figure 8.1 OpenJPA life cycle management.

This life cycle is interesting because it brings together the notion of an EntityManager and a POJO, resulting in four states to consider in the programming model:

- **New/Transient**—The Object is instantiated, but there is no relational data in the database.
- **Managed**—The Object is associated with the persistent manager, and therefore the instance has a database record and the Java instance is connected to its record. Executing `getters()` and `setters()` imply database operations.
- **Detached**—The EntityManager is closed but the instance is still around. It is just a Value Object at this point. Executing `getters()` and `setters()` do not imply database updates. You can move a detached instance back to a Managed state by calling `merge` on the EntityManager.

- **Removed**—This is an instance that is no longer in the database because it has been deleted. After the transaction is committed, the object is just like any other transient Java Object.

Within this context, we will consider the specific details of how to do initialization, make connections, create transactions, invoke CRUD methods, and so on.

Initialization

The heart of persisting POJOs relies on a special object called an `EntityManager`. The goal for a developer is to initialize an `EntityManager` with the proper mappings and database information necessary. JPA provides several ways to load an `EntityManager`, depending on the environment.

To initialize the framework, you need to create a file called `persistence.xml`, as shown in Figure 8.1, and define a persistence unit, as shown in Listing 8.1. The listing contains the information necessary for a Java SE Application to configure a persistence unit.

Listing 8.1 Java SE Persistence Unit

```
<persistence-unit name="pie-db-JAVA-SE">
  <provider>
    org.apache.openjpa.persistence.PersistenceProviderImpl
  </provider>
  <properties>
    <property name="openjpa.ConnectionURL"
      value="jdbc:derby://localhost:1527/PWTE"/>
    <property name="openjpa.ConnectionDriverName"
      value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="openjpa.jdbc.DBDictionary" value="derby"/>
    <property name="openjpa.Log"
      value="DefaultLevel=WARN,
      Runtime=INFO,
      Tool=INFO,
      SQL=TRACE"/>
    <property name="openjpa.jdbc.Schema" value="APP"/>
  </properties>
</persistence-unit>
```

The first thing you do is define your provider, which in this case is OpenJPA. Then you use the properties to set up the name of the Driver or DataSource implementation and any additional properties. A full list of properties can be found in the Open JPA Manual referenced earlier [OpenJPA 2].

In a Java EE environment, the configuration will be slightly different. Application Servers, like WebSphere Application Server, will often provide a default JPA implementation, and it is therefore not necessary to provide a JPA provider. Listing 8.2 shows an example of a `persistence.xml` file in a Java EE environment. In it, you can provide the JNDI name of a configured `DataSource`. The `DataSource` implementation is vendor specific. Some Application Server vendors allow for swapping the default implementation.

Listing 8.2 Java EE Persistence Unit

```
<persistence-unit name="pie-db-JAVA-EE">
  <jta-data-source>jdbc/orderds</jta-data-source>
  <properties>
    <property name="openjpa.jdbc.DBDictionary" value="derby"/>
    <property name="openjpa.jdbc.Schema" value="APP"/>
  </properties>
</persistence-unit>
```

Connections

After you configure the persistence unit, JPA will have the necessary information to instantiate a persistence context. A *persistence context* is a set of managed entity instances in which, for any persistent entity identity, there is a unique entity instance. Within the persistence context, the entity's association with the underlying persistence store is managed by the `EntityManager` (EM). If you are familiar with Hibernate, the `EntityManager` is similar to the Hibernate Session.

All the connections to the underlying database are encapsulated within the EM. So getting an instance of the `EntityManager` will get a connection for you as needed without any explicit coding on your part. However, getting an instance of the `EntityManager` varies between a Java SE and Java EE environment. In a Java SE environment, an *application-managed EM instance* is created by calling the `EntityManagerFactory`, and the lifetime of that instance is controlled by the application. For each application-managed EM instance, there are one or more corresponding *application-managed persistence contexts*, which are not linked with any transaction and are not propagated to other components. It is important to realize that you must open and close the Entity Manager yourself in an application coded to run in a Java SE environment.

The `Persistence` class is used as a bootstrap to get access to an `EntityManagerFactory` for a particular persistence unit configured in a Java SE environment. After you get access to the `EntityManagerFactory`, you can use that to get an instance of an `EntityManager` that can be used throughout your code to implement the persistence code. Listing 8.3 shows an example of this process. In this code, we illustrate the three steps just discussed. Notice that in this example, the `EntityManager` is scoped to each business method. This is one common pattern in a Java SE environment.

Listing 8.3 Look Up Entity Manager Factory

```

public class CustomerOrderServicesJavaSEImpl{
    protected EntityManagerFactory emf =
        Persistence.createEntityManagerFactory(
            "pie-db-JAVA-SE"
        );
    public Order openOrder(int customerId)throws Exception {
        EntityManager em = emf.createEntityManager();
        em = emf.createEntityManager();
        //Code
        em.close();
    }
    ...
}

```

It is worth noting that opening and closing an Entity Manager may be slower than keeping an instance around in some scenarios.

In a Java EE environment, the container can “inject” an EntityManagerFactory into a Java EE artifact, such as a Stateless Session Bean or an HttpServlet. Listing 8.4 shows an example of injecting an EntityManagerFactory into a Stateless Session Bean. Once injected, it is used the same way we illustrated earlier. Notice that you still must programmatically close the EntityManager because you used a factory to create the EntityManager. This is because you are still using an application-managed EntityManager.

Listing 8.4 Inject Entity Manager Factory

```

@Stateless
public class CustomerOrderServices {
    @PersistenceUnit (unitName = "pie-db-JAVA-EE")
    protected EntityManagerFactory emf;
    public Order openOrder(int customerId)throws Exception
    {
        EntityManager em = emf.createEntityManager();
        //Code
        em.close();
    }
    ...
}

```

In an EJB 3 environment, a *container-managed EM instance* is created by directing the container to inject one instance (either through direct injection or through JNDI lookup). The lifetime of that EM instance is controlled by the container; the instance matches the lifetime of the component into which it was injected. Container-managed Entity Managers will

also provide automatic propagation of transactions, connections, and other services. We will discuss this further after transactions are discussed.

An `EntityManager` can be injected directly into an EJB, using a Java annotation called `PersistenceContext`. Listing 8.5 shows an example of this. In this case, the developer does not have to worry about a factory. Furthermore, you delegate to the container the management of the `EntityManager` and the propagation of the proper persistence context from EJB component to EJB component. This is a more common pattern in Java EE. If you are using the `PersistenceContext` in an EJB 3 component, the EJB 3 container will automatically propagate the persistence context for a particular request across components. It is important to keep in mind that an EJB 3 Session Bean is a thread-safe component, and therefore, only one client request is being serviced by the EJB component at a time. This means the `EntityManager` can safely be used by the instance without fear of another thread accessing it. The container can also take advantage of this and pass the current persistence context which can contain an active transaction.

Listing 8.5 Inject EntityManager

```
@Stateless
public class CustomerOrderServices {
    @PersistenceContext (unitName = "pie-db-JAVA-EE") //Step 1
    protected EntityManager em;
```

To illustrate the object states shown in Figure 8.1 in context of the code to get access to the `EntityManager`, see Listing 8.6. This listing shows objects in various states and how they move from one state to another. The comments within the listing describe the action.

Listing 8.6 Object States

```
//Set up EM and Transient POJO instance
em = emf.createEntityManager();
Order newOrder = new Order();
newOrder.setStatus(Order.Status.OPEN);
newOrder.setTotal(new BigDecimal(0));

//Make POJO Managed by persisting it
em.persist(newOrder);
newOrder.setTotal(new BigDecimal(1));

//Make POJO Detached by closing
em.close();
newOrder.setTotal(new BigDecimal(2));
```

```
//Make POJO Managed by merging detached instance
em2 = emf.createEntityManager();
em2.merge(newOrder;)
```

As long as a POJO is associated with an EntityManager, updates to the database are implied and the instance is managed. Managed Entities are either (a) loaded by the EntityManager via a find method or query, or (b) associated with the EntityManager with a `persist` or `merge` operation. We discuss this more in later sections on the Create, Retrieve, Update, and Destroy operations.

Transactions

You can demarcate transactions in OpenJPA in the following ways:

- Using a standard programmatic API such as the JTA interfaces
- Using the special `javax.persistence.EntityTransaction` interface provided by JPA
- Using declarative transactions within an EJB 3 Container

We have covered JDBC and JTA transactions in previous chapters. Listing 8.7 shows an example of using the EntityTransaction interface. A developer can get an instance of an EntityTransaction by calling `getTransaction()` on the EntityManager. After they have an instance, you can call `begin`, `commit`, or `rollback`. This is often the norm when using OpenJPA in a Java SE environment or in the web container.

Listing 8.7 Entity Manager Transaction Demarcation

```
public Order openOrder(int customerId)
throws CustomerDoesNotExistException,
    OrderAlreadyOpenException,
    GeneralPersistenceException {
    EntityManager em = null;
    try {
        em = emf.createEntityManager();
        em.getTransaction().begin();
        //use em to manage objects
        em.getTransaction().commit();
        return newOrder;
    }
    catch(CustomerDoesNotExistException e){
        em.getTransaction().rollback();
        throw e;
    }
    //Handle other Exceptions not listed
    finally {
```

```
        if (em != null) em.close();
    }
}
```

If a developer uses an external API like JTA to demarcate transactions and you are using an application-managed EntityManager, you need to have your EntityManager instance “join” the transaction. Listing 8.8 illustrates this situation and shows the code needed to cause the join.

Listing 8.8 Join Transaction

```
@PersistenceUnit (unitName = "pie-db-JAVA-EE")
protected EntityManagerFactory emf;

public Order openOrder(int customerId) throws Exception {
    javax.transaction.UserTransaction tran =
        //code to lookup UserTransaction in JNDI

    EntityManager em = emf.createEntityManager();
    try {
        // Start JTA transaction
        tran.begin();

        // Have em explicitly join it
        em = emf.createEntityManager();
        em.joinTransaction();

        //Code in transaction scope ready to commit
        tran.commit();

        //Code outside of transaction scope
    }
    catch(Exception e) {
        tran.rollback();
        throw e;
    }
    finally {
        em.close();
    }
}
```

When OpenJPA is used in an EJB 3 container, OpenJPA will allow for transactions to be controlled by EJB transaction demarcation. Listing 8.9 shows an example of JPA being used

within an EJB 3 Session Bean. The EJB 3 method is marked with a Required transaction. In addition, the EntityManager is injected into the EJB 3 POJO. In this scenario, you get the benefit of having the container manage the EntityManager for you using the container-managed Entity Manager discussed in the preceding section.

Listing 8.9 EJB 3 Transaction Demarcation

```
@Stateless
public class CustomerOrderServicesImpl implements CustomerOrderServices
{
    @PersistenceContext(unitName="pie-db-JAVA-EE")
    protected EntityManager em;

    @TransactionAttribute(value=TransactionAttributeType.REQUIRED)
    public Order openOrder(int customerId)
    throws CustomerDoesNotExistException,
        OrderAlreadyOpenException,
        GeneralPersistenceException {
        // use em
    }
}
```

For each container-managed EM instance, there are one or more corresponding *container-managed persistence contexts* (PCs). At the time the PC is created, it is linked with the transaction currently in effect and propagated by the container, along with the transaction context to other called components within the same JVM.

The container-managed usage scenario is further subcategorized into transaction-scoped (lifetime is controlled by the transaction) and extended (lifetime is controlled by one or more stateful session bean instances). This means that in the transaction case, inside an EJB 3 container, the persistence context life cycle is governed by the transaction. You get automatic flushing of cache at the end of the transaction. JPA also provides an extended PersistenceContext that will be managed by some greater scope, such as a Stateful Session Bean or perhaps an Http Session. Listing 8.10 shows how you can inject a longer-lived Persistence Context.

Listing 8.10 Extended Persistence Context

```
@PersistenceContext(
    unitName="pie-db-JAVAAEE", type=PersistenceContextType.EXTENDED
)
protected EntityManager em;
```

In Chapter 5, “JDBC,” we discussed savepoints. Savepoints allow for fine-grained control over the transactional behavior of your application. The JPA specification does not allow for savepoints; however, some vendors, like OpenJPA, may have extensions. OpenJPA’s save-

point API allows you to set intermediate rollback points in your transaction. You can then choose to roll back changes made only after a specific savepoint, then commit or continue making new changes in the transaction. OpenJPA's `OpenJPAEntityManager` (subtype of JPA's `EntityManager`) supports these savepoint operations:

- `void setSavepoint(String name);`
- `void releaseSavepoint(String name);`
- `void rollbackToSavepoint(String name);`

Savepoints require some configuration, so refer to the OpenJPA documentation for more details.

Create

The `EntityManager` has most of the methods needed to persist Java objects, or entities. Persistence actions usually occur by passing instances of entities to and from the `EntityManager`. So for creating data, you would just create an instance of an entity and persist it using the `persist` method of the `EntityManager`. Listing 8.11 shows an example of this.

Listing 8.11 Persist Data

```
Order newOrder = new Order();
newOrder.setCustomer(customer);
newOrder.setStatus(Order.Status.OPEN);
newOrder.setTotal(new BigDecimal(0));
em.persist(newOrder);
```

This will correspond to an `INSERT` into the database. The `Order` object in the example is mapped to a table in the database. A POJO mapped to a database is called an Entity in JPA. We will discuss mappings in the “ORM Features Supported” section.

Retrieve

Reading data can be done several ways. The simplest read is to read an object by primary key. The `EntityManager` `find` method provides an easy way to do this. Listing 8.12 illustrates this. The `find` method takes the name of the class and a primary key value as a parameter. We discuss mapping primary keys in the later section, “ORM Features Supported.”

Listing 8.12 Finding Entity Instances

```
AbstractCustomer customer = em.find(AbstractCustomer.class, customerId);
```

Using the `find` methods, OpenJPA can load a whole Object Graph. Listing 8.13 shows an example of accessing the `Order` Object after loading the customer. If the `Order` Object is mapped as a relationship and the proper fetching strategies are set, OpenJPA will load more than the root object with the `find` method. Later in the chapter, fetching is discussed.

Listing 8.13 Related Entity Instances

```
AbstractCustomer customer = em.find(AbstractCustomer.class, customerId);
Order existingOpenOrder = customer.getOpenOrder();
```

JPA also comes with a rich query language called EJB-QL (sometimes called JPQL). You can issue queries against the object model. We will not provide a detailed tutorial on the query languages, and instead recommend that you read the reference guide [JPQL]; but the query language provides syntax to execute complex queries against related objects. Listing 8.14 shows an example of executing a query. As a developer, you can create a query using the `createQuery` against the `EntityManager`. Notice you can use `:name` to mark places where you want to use parameters. OpenJPA also supports using the `?` approach used by JDBC Prepared Statements. However, both approaches will translate to Prepared Statements.

Listing 8.14 Executing JPA Queries

```
Query query = em.createQuery(
    "select l from LineItem l
     where l.productId = :productId and l.orderId = :orderId "
);
query.setParameter("productId", productId);
query.setParameter("orderId", existingOpenOrder.getOrderId());
LineItem item = (LineItem) query.getSingleResult();
```

OpenJPA also supports the capability to externalize queries from the code using the “named query” concept, which allows you to associate a query to a name using an annotation or the XML mapping file. Listing 8.13 shows how you annotate a POJO with the `NamedQuery`. The rest of the annotations in the listing are explained in the later section, “ORM Features Supported.” After you define the `NamedQuery`, you can execute somewhere else in your code, as shown in the second part of Listing 8.15. It is worth mentioning that if you want to truly externalize the queries, you should use the XML deployment descriptor.

Listing 8.15 Executing JPA Queries

```
@Entity
@Table(name="LINE_ITEM")
@IdClass(LineItemId.class)
@NamedQuery(
    name="existing.lineitem.forproduct",
    query="
        select l from LineItem l
        where l.productId = :productId
        and l.orderId = :orderId"
)
```

```
public class LineItem {  
    ...  
    Query query = em.createNamedQuery("existing.lineitem.forproduct");  
    query.setParameter("productId", productId);  
    query.setParameter("orderId", existingOpenOrder.getOrderId());  
    LineItem item = (LineItem) query.getSingleResult();  
}
```

With EJB-QL, when you are querying for Objects, you can load related objects as well, depending on the mapping. You can also load objects using joins. OpenJPA also extends EJB-QL with some value adds.

For the majority of the cases, you should be able to get data you need. There are cases when you need to drop down to native SQL. This could be to call a Stored Procedure, to get an optimized SQL, or because you cannot get OpenJPA to generate the correct SQL needed for the use case. OpenJPA supports the notion of native queries. You can execute SQL and project onto a POJO. An example is shown in Listing 8.16.

Listing 8.16 Native SQL

```
Query query = em.createNativeQuery(  
    "SELECT * FROM LINE_ITEM", LineItem.class  
);  
List<LineItem> items = query.getResultList();
```

You can also have Native Named Queries if you want to externalize the SQL.

Update

OpenJPA supports updating existing data in a few ways. Figure 8.1 showed you the life cycle of a POJO with respect to the EntityManager. Any field updated on a POJO that is associated with the EntityManager implies a database update. Listing 8.17 shows an example of code finding an instance of LineItem and executing an update.

Listing 8.17 Updating Persistent Entities

```
LineItem existingLineItem = em.find(LineItem.class, lineItemId);  
existingLineItem.setQuantity(existingLineItem.getQuantity() + quantity);  
existingLineItem.setAmount(existingLineItem.getAmount().add(amount));
```

Any update to related objects also implies an update. In Listing 8.18, we show that after finding the customer Entity, you can traverse to the Order. Because the customer is still being managed by the EntityManager, so is the Order.

Listing 8.18 Updating Related Entities

```

AbstractCustomer customer = em.find(AbstractCustomer.class, customerId);
Order existingOpenOrder = customer.getOpenOrder();
BigDecimal amount = product.getPrice().multiply(new BigDecimal(quantity));
existingOpenOrder.setTotal(amount.add(existingOpenOrder.getTotal()));

```

OpenJPA also supports updating of detached Entities using the `merge` method. Listing 8.19 shows an example of a detached case. In the first part of the listing, you can see a fragment of Servlet code calling a service. The Service implementation is shown in the second part of the listing. A web request first reads the data using an HTTP GET, which gets access to the data and stores it in sessions. The service implementation uses a `find` to access the data and finish the request. Then an HTTP POST comes in to update the data in session. The Servlet `doPost` passes the detached instance into the `updateLineItem` method. The implementation of `updateLineItem` will attempt to merge the instance to the `EntityManager`.

Listing 8.19 Updating via merging

```

public void doGet(
    HttpServletRequest request, HttpServletResponse response)
{
    String customerId = populateFromRequest(request);
    LineItem li = customerService.getLineItem(li);
    writeToSession(li);
}

public void doPost(
    HttpServletRequest request, HttpServletResponse response)
{
    LineItem li = populateFromSession(request);
    customerService.updateLineItem(li);
}
...

public LineItem getLineItem(int liId) throws GeneralPersistenceException
{
    EntityManager em = //Get Entity Manager
    LineItem li = emf.find(LineItem.class,liId);
    return li;
}

```

```
public void updateLineItem(LineItem li) throws GeneralPersistenceException {
    EntityManager em = //Get Entity Manager
    em.merge(li);
}
```

OpenJPA allows you to use EJB-QL to issue updates as well, such as shown in Listing 8.20. This feature enables you to update many instances with one statement. You also avoid hydrating objects in certain scenarios where performance is important.

Listing 8.20 Updating via Query Language

```
Query query = em.createQuery(
    "UPDATE CUSTOMER c
    SET o.discount = :discount
    WHERE c.type = 'RESIDENTIAL'"
);
query.setParameter("discount", discount);
query.executeUpdate();
```

Delete

You can delete data several ways using OpenJPA. A managed instance can be removed by calling `remove` on the `EntityManager`, as shown in Listing 8.21. (Listing 8.23 shows an even better option.)

Listing 8.21 Deleting via EntityManager Remove

```
LineItem existingLineItem = em.find(LineItem.class, lineItemId);
if(existingLineItem != null){
    em.remove(existingLineItem);
}
```

You can configure OpenJPA to propagate deletes along an object graph. We will show you mapping relationships later; however, in Listing 8.22, you can see that `Order` has a relationship to a `Set` of `LineItem` instances. On the relationship, you can see that we have set the cascade to `REMOVE`. This means that when you delete an `Order`, all associated `LineItem` instances will be deleted as well.

Listing 8.22 Deleting via Cascading

```
@Entity
@Table(name="ORDERS")

public class Order implements Serializable {
```

```

...
@OneToMany(cascade=CascadeType.REMOVE, fetch=FetchType.EAGER )
@ElementJoinColumn(
    name="ORDER_ID",referencedColumnName="ORDER_ID"
)
protected Set<LineItem> lineitems;

```

Finally, much as with Updates, OpenJPA allows you to delete, using EJB-QL Queries. Listing 8.23 shows an example of deleting via a query. This approach is useful if you want to delete many rows with one network call.

Listing 8.23 Deleting via a Query

```

Query query = em.createQuery(
    "DELETE FROM LineItem l
    WHERE l.productId = :productId
    and l.orderId = :orderId"
);
query.setParameter("productId", productId);
query.setParameter("orderId", existingOpenOrder.getOrderId());
query.executeUpdate();

```

Stored Procedures

We already showed how you can use native queries in OpenJPA. Native queries can be used to call stored procedures as shown in Listing 8.24.

Listing 8.24 Using a Native Query to Call a Stored Procedure

```

Query query = em.createNativeQuery("CALL SHIP_ORDER(?)");
query.setParameter(1, orderId);
query.executeUpdate();

```

Batch Operations

As we showed in the Update and Delete sections, OpenJPA supports batching updates and deletes using EJB-QL. The Apache version of OpenJPA (1.0.0) currently does not support automatic statement batching for persistent operations. However, vendors that build on top of OpenJPA sometimes provide this function. The EJB 3 implementation of WebSphere Application Server provides an enhanced version of OpenJPA. They provide a configuration option for deferring update operations to commit time and batching them together. Other vendors may provide similar optimizations. The optimizations can make mass updates several orders of magnitude faster; see also the “Batch Operations” section in Chapter 5 for details of how this approach works.

Extending the Framework

When writing OpenJPA plug-ins or otherwise extending the OpenJPA runtime, however, you will use OpenJPA’s native APIs. OpenJPA allows you to extend the framework in various ways, including these:

- You can extend the default EntityManager or EntityManagerFactory. This is often done by Vendors offering enhanced JPA implementations on top of OpenJPA.
- You can extend the query engine. This is usually done to provide optimized solutions, such as integrating with cache technologies.
- Data Caches can be added to back the OpenJPA cache.
- Other areas support extending behavior, such as fetch and primary key generation strategies.

The OpenJPA implementation shows the specific interfaces and classes that need to be extended to provide your own extensions.

Error Handling

JPA Exceptions are unchecked. Figure 8.2 shows the JPA Exception Architecture. JPA uses standard exceptions where appropriate, most notably IllegalArgumentExceptions and IllegalStateExceptions. These exceptions can occur when you perform persistence actions without the proper setup—for example, sending an Entity to an EntityManager that is not managing it.

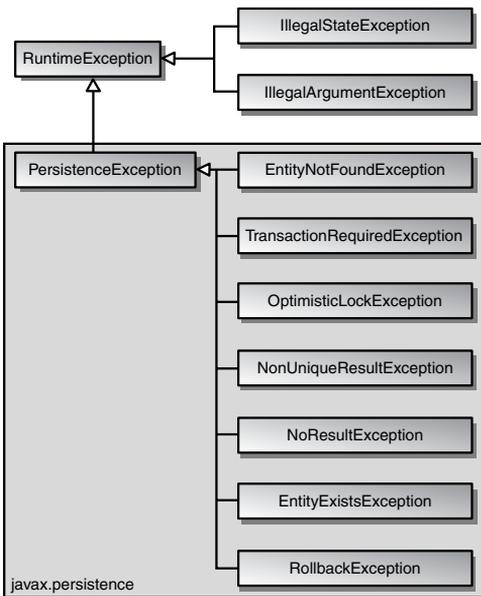


Figure 8.2 JPA exception class diagram.

The specification also provides a few JPA-specific exceptions in the `javax.persistence` package. Listing 8.25 shows an example of catching an `EntityNotFoundException`. Alternatively, because the exceptions are unchecked, you can choose to not catch it and handle it at a higher level.

Listing 8.25 Exception Example

```

{   try
    AbstractCustomer customer = em.find(
        AbstractCustomer.class, customerId
    );
    ...
}
catch(javax.persistence.EntityNotFoundException e)
{   throw new GeneralPersistenceException(e);
}

```

All exceptions thrown by OpenJPA implement `org.apache.openjpa.util.ExceptionInfo` to provide you with additional error information. Figure 8.3 shows the class diagram of the `ExceptionInfo` interface.

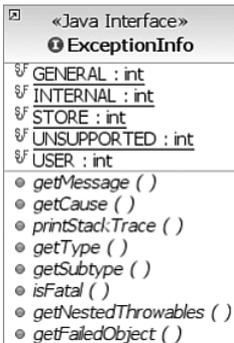


Figure 8.3 `ExceptionInfo` interface.

ORM Features Supported

We have shown you how a programmer would use Entities to perform the runtime persistence operations. We will now show you how to map Entities to a database using OpenJPA. OpenJPA allows mapping Entities via Java Annotations and XML. The choice is up to the developer and usually depends on whether you want to expose the underlying relational database details into the code; or in some cases, whether you have “legacy” Java objects that are not easy to change.

Objects

Throughout this book, we have been showing meet-in-the-middle Mapping. OpenJPA supports top-down, bottom-up, and meet-in-the-middle. It is worth mentioning that the JPA spec defines a common standard for top-down generation of database schemas. Listing 8.26 shows the minimum annotation needed to make a POJO a JPA Entity. Simply by marking a class with the `@Entity` annotation, you have a persistent capable class. If your database schema follows the JPA naming convention, or if you want to have the database schema generated, then the class can be used. In this case, OpenJPA will look for a table named `Customer`. The table will have two columns: `id` and `name`. The table names and column names will match whether they contain upper- or lowercase spellings.

Listing 8.26 Creating an Entity with Java Annotations

```
@Entity
public class Customer
{
    private String name;
    private int id;

    public getName(){return name;}
    public setName(String name){this.name=name;}

    public getId(){return id;}
    public setId(int id){this.id = id;}
}
```

As mentioned earlier, you can also use XML as an alternative to Java Annotations. Listing 8.27 shows an XML entity mapping for the same `Customer`. As noted previously, some developers prefer to externalize their database mappings to keep the Java code “pure.” Some other developers prefer to use Annotations together with XML. In some cases XML and Annotations can be used. In this case, XML will serve as an override for Annotations. This might not make sense when an entity is always mapped to a relational table; however, in other cases XML overrides can provide benefits (for example, enabling you to change a schema name or optimize a query without changing the Java code).

Listing 8.27 Creating an Entity with XML

```
<entity-mappings
    xmlns="java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
```

```

    version="1.0"
  >
    <entity class=" Customer" />
</entity-mappings>

```

OpenJPA does a lot of defaulting for you; as long as you match the naming conventions, things will fall into place. This includes naming conventions for relationships and other complex types. For the rest of the section, though, we will show explicit mappings to illustrate the important concepts. In addition, it is often the case that the database schema will not always match the Object model, and will require a meet-in-the-middle mapping—as occurs in our example throughout the book.

Mapping Entities to tables with different names is easy. Listing 8.28 shows the Order Entity mapped to a table called ORDERS. You use a simple annotation called `@Table`.

Listing 8.28 Mapping an Entity to a Table

```

@Entity
@Table(name="ORDERS")
public class Order implements Serializable {

```

Similarly, you can map Entities to a table using XML, as shown in Listing 8.29.

Listing 8.29 Mapping an Entity to a Table with XML

```

<entity-mappings xmlns="java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0"
>

  <entity class=" Order ">
    <table name="ORDERS"/>
  ...
</entity>

```

The `Table` annotation and XML both let you specify a schema name as well.

Inheritance

OpenJPA fully supports inheritance in persistent classes. It allows persistent classes to inherit from nonpersistent classes, persistent classes to inherit from other persistent classes, and nonpersistent classes to inherit from persistent classes. It is even possible to form inheritance hierarchies in which persistence skips generations. There are, however, a few important limitations:

- Persistent classes cannot inherit from certain natively implemented system classes such as `java.net.Socket` and `java.lang.Thread`.
- If a persistent class inherits from a nonpersistent class, the fields of the nonpersistent superclass cannot be persisted.
- All classes in an inheritance tree must use the same identity type. Identifiers will be covered in the next section.

OpenJPA supports three strategies for Inheritance:

1. Single Table
2. Joined
3. Table Per Class

We described each of these in detail in Chapter 3, “Designing Persistent Object Services.”

Single Table Strategy

All JPA providers have to provide an implementation of the Single Table and the Joined strategy. Table Per Class is optional. Single Table mandates all classes in the inheritance hierarchy map into one table. In our book example for OpenJPA, this is our default implementation.

The `AbstractCustomer` superclass contains a type discriminator. This discriminator will be used to create the correct type at runtime. All the fields are mapped into a single table. Listing 8.30 shows how this mapping would look in OpenJPA.

Listing 8.30 Superclass Mapping with a Single Table

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    protected int customerId;

    ....
}
```

The subclasses would then specify a discriminator value. Listing 8.31 shows the mapping for `ResidentialCustomer` (each subclass would have a similar mapping). Notice that you do not have to specify a table mapping because the superclass handles the mapping.

Listing 8.31 Subclass Mapping with a Single Table

```
@Entity
@DiscriminatorValue("RESIDENTAL")
public class ResidentialCustomer
extends AbstractCustomer
```

```
implements Serializable {
    protected short householdSize;
    protected boolean frequentCustomer;
}
```

In Listing 8.32, the same mapping is shown in XML. We show it here just to highlight that mappings can be done in XML.

Listing 8.32 XML Mapping for Single Table

```
<entity class=" org.pwte.example.domain.AbstractCustomer ">
    <table name="CUSTOMER" />
    <inheritance strategy="SINGLE_TABLE"/>
    ...
</entity>
<entity class=" org.pwte.example.domain.ResidentialCustomer ">
    ...
</entity>
```

Single table inheritance mapping is the most performant of all inheritance models because it does not require a join to retrieve the persistent data necessary to populate the class hierarchy of a single instance from the database (it still may require a join to retrieve related objects, of course). Similarly, persisting or updating a single persistent instance can often be accomplished with a single `INSERT` or `UPDATE` statement. Finally, relations to any other class within a single table inheritance hierarchy are just as efficient as relations to a base class.

However, the larger the inheritance model gets, the “wider” the mapped table gets—in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequences on the database size, because a wide or deep inheritance hierarchy will result in tables with many mostly empty columns. In addition, changes to any class in the hierarchy would result in changes to the table. This issue is significant because after systems are deployed, it is generally very difficult to change database schemas of existing tables.

Joined Strategy

The Joined Strategy is really the table per class strategy we discussed in Chapter 3. Here, every class in the inheritance chain gets its own table. Each subclass’s primary key would also be a foreign key to the primary key table.

Listing 8.33 shows the `AbstractCustomer` with the Inheritance strategy of `Joined`. Notice it is mapped to an `ABSTRACT_CUSTOMER` table.

Listing 8.33 Mapping Superclass with the Joined Strategy

```
@Entity
@Inheritance(strategy=JOINED)
```

```

@Table(name = "ABSTRACT_CUSTOMER")
public abstract class AbstractCustomer implements Serializable {

    @Id
    protected int customerId;
    protected String name;
    protected String type;

```

In Listing 8.34, you will notice that the `Residential` subclass is mapped to its own table. Notice the use of the `@PrimaryKeyJoinColumn` annotation to link the primary key to the superclass. The `Business Customer` class will be similar.

Listing 8.34 Mapping Subclass with the Joined Strategy

```

@Entity
@Table(name = "RESIDENTIAL_CUSTOMER")
@PrimaryKeyJoinColumn(name="CUSTOMER_ID",
referencedColumnName="CUSTOMER_ID")

public class ResidentialCustomer
extends AbstractCustomer implements Serializable {
    protected short householdSize;
    protected boolean frequentCustomer;

```

The joined strategy has the following advantages:

- Using joined subclass tables results in the most *normalized* database schema, meaning the schema with the least spurious or redundant data.
- As more subclasses are added to the data model over time, the only schema modification that needs to be made is the addition of corresponding subclass tables in the database (rather than having to change the structure of existing tables).

Relations to a base class using this strategy can be loaded through standard joins and can use standard foreign keys, as opposed to the machinations required to load polymorphic relations to Table-per-Class base types, described next. The joined strategy is often the slowest of the inheritance models, unless provisions are made to “lazily load” levels of the hierarchy. Retrieving any subclass can require one or more database joins, and storing subclasses can require multiple `INSERT` or `UPDATE` statements.

Table-per-Class Strategy

The Table-per-Class strategy is what we defined as the Concrete Table Inheritance Strategy in Chapter 3. In this model, each concrete subclass will have its own table, and the superclass information is repeated in each of the tables. Listing 8.35 shows how you would configure the `AbstractCustomer` superclass with the `TABLE_PER_CLASS` strategy option. All that is needed is setting the inheritance type because the class is Abstract Class.

Listing 8.35 Mapping Superclass with Table-per-Class

```

@Entity
@Inheritance(strategy=TABLE_PER_CLASS)
public abstract class AbstractCustomer implements Serializable {

    @Id
    protected int customerId;
    protected String name;
    protected String type;

```

Listing 8.36 shows both subclasses, each mapping to its corresponding table.

Listing 8.36 Mapping Subclass with Table-per-Class

```

@Entity
@Table(name = "RESIDENTIAL_CUSTOMER")
public class ResidentialCustomer
extends AbstractCustomer implements Serializable {
    protected short householdSize;
    protected boolean frequentCustomer;

    ...

@Entity
@Table(name = "BUSINESS_CUSTOMER")
public class BusinessCustomer
extends AbstractCustomer implements Serializable {
    protected boolean volumeDiscount;
    protected boolean businessPartner;

```

As mentioned in Chapter 3, you need a way to manage the primary keys across the various concrete classes. Some databases support the notion of a sequence to generate keys across tables.

The Table-per-Class strategy is very efficient when operating on instances of a known class. Under these conditions, the strategy never requires joining to superclass or subclass tables. Reads, joins, inserts, updates, and deletes are all efficient in the absence of polymorphic behavior. Also, as in the joined strategy, adding new classes to the hierarchy does not require modifying existing class tables, as is required in the Single-Table strategy.

Polymorphic relations to nonleaf classes in a Table-per-Class hierarchy have many limitations. When the concrete subclass is not known, the related object could be in any of the subclass tables, making joins through the relation impossible. This ambiguity also affects identity lookups and queries; these operations require multiple SQL `SELECTs` (one for each possible subclass), or a complex `UNION`.

Table-per-Class inheritance mapping has the following limitations:

- You cannot traverse polymorphic relations to nonleaf classes in a Table-per-Class inheritance hierarchy in queries.
- You cannot map a one-sided polymorphic relation to a nonleaf class in a Table-per-Class inheritance hierarchy using an inverse foreign key.
- You cannot use an order column in a polymorphic relation to a nonleaf class in a Table-per-Class inheritance hierarchy mapped with an inverse foreign key.
- Table-per-Class hierarchies impose limitations on eager fetching. We will discuss fetching later in the section on “Tuning Options.”

A more serious issue with the Table-per-Class strategy is what happens when a non-leaf (superclass) in the hierarchy is changed. In this case, every concrete class that inherits from that class—either directly or indirectly—must change. Therefore, you should only use the Table-per-Class strategy when the hierarchy is relatively stable.

Keys

Any Entity instance is uniquely identified by an ID in JPA. The ID property is then mapped to the primary key. You can mark a field on your entity as an ID using the `@Id` annotation. Listing 8.37 shows an example.

Listing 8.37 ID Field

```
@Entity
public class Product implements Serializable {
    private static final long serialVersionUID = 2435504714077372968L;

    @Id
    protected int productId;

    protected BigDecimal price;
    protected String description;

    ...
}
```

In some cases, primary keys need to be generated at Entity Creation Time. OpenJPA supports a number of mechanisms to do this.

JPA includes the `GeneratedValue` annotation for this purpose. It has the following properties:

- **GenerationType.AUTO**—The default. Assign the field a generated value, leaving the details to the JPA vendor.
- **GenerationType.IDENTITY**—The database will assign an identity value on insert.
- **GenerationType.SEQUENCE**—Use a datastore sequence to generate a field value.
- **GenerationType.TABLE**—Use a sequence table to generate a field value.

OpenJPA also offers two additional generator strategies for non-numeric fields, which you can access by setting `strategy` to `AUTO` (the default), and setting the `generator` string to one of the following:

- **uuid-string**—OpenJPA will generate a 128-bit UUID unique within the network, represented as a 16-character string. For more information on UUIDs, see the IETF UUID draft specification at www1.ics.uci.edu/~ejw/authoring/uuid-guid/.
- **uuid-hex**—Same as `uuid-string`, but represents the UUID as a 32-character hexadecimal string.

Listing 8.38 shows an example of using the Identity mapping. In this case, OpenJPA will defer to the database's implementation of generating an identity. The Sequence and Identity columns require that your database support these features.

Listing 8.38 Generating an ID Using the Identity Strategy

```
@Entity
@Table(name="ORDERS")
public class Order implements Serializable {
    private static final long serialVersionUID = 7779370942277849463L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="ORDER_ID")
    protected int orderId;
    protected BigDecimal total;
```

For the Table Generator Strategy, you must define a table in the database. (The OpenJPA documentation contains details on the schema for this table.) Listing 8.39 shows an example of using the Table Strategy. In this case, you define a specific generator that points to a table. Then you point your generated strategy to the table. The Sequence would work in a very similar fashion. Sequence and table generators usually work better when you have to define an ID across several tables. For example, when using the Table-per-Concrete method of mapping Inheritance, all of your subclasses may need to share a common sequence or generator to ensure data integrity across instances.

Listing 8.39 Generating an Identity with the Table Strategy

```
@Entity
@Table(name="CUSTOMER")
public class Customer {

    @Id
    @GeneratedValue(
```

```

        strategy=GenerationType.TABLE, generator="AuthorGen"
    )
    @TableGenerator(
        name="AuthorGen", table="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID"
    )
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    ...
}

```

The JPA specification requires you to declare one or more identity fields in your persistent classes. OpenJPA fully supports this form of object identity, called application identity. OpenJPA, however, also supports datastore identity. In datastore identity, you do not declare any primary key fields. OpenJPA manages the identity of your persistent objects for you through a surrogate key in the database.

You can control how your JPA datastore identity value is generated through OpenJPA's `org.apache.openjpa.persistence.DataStoreId` class annotation. This annotation has `strategy` and `generator` properties that mirror the same-named properties on the standard `javax.persistence.GeneratedValue` annotation just described.

To retrieve the identity value of a datastore identity entity, use the `OpenJPAEntityManager.getObjectId(Object entity)` method. Listing 8.40 shows an example of using this method.

Listing 8.40 Using an Application Managed Identity in OpenJPA

```

import org.apache.openjpa.persistence.*;

@Entity
@DataStoreId
public class LineItem {

    ... no @Id fields declared ...
}

```

If you choose to use application identity, you may want to take advantage of OpenJPA's application identity tool. The application identity tool generates Java code implementing the identity class for any persistent type using application identity. The code satisfies all the requirements the specification places on identity classes. You can use it as-is, or simply use it as a starting point, editing it to meet your needs. Refer to the OpenJPA documentation for more details.

When your entity has multiple identity fields, at least one of which is a relation to another entity, you must use an identity class. You cannot use an embedded identity object. Identity class fields corresponding to entity identity fields should be of the same type as the related entity's identity.

Your identity class must meet the following criteria:

- The class must be public.
- The class must be serializable.
- The class must have a public no-args constructor.
- The names of the nonstatic fields or properties of the class must be the same as the names of the identity fields or properties of the corresponding entity class, and the types must be identical.
- The `equals` and `hashCode` methods of the class must use the values of all fields or properties corresponding to identity fields or properties in the entity class.
- If the class is an inner class, it must be `static`.
- All entity classes related by inheritance must use the same identity class, or else each entity class must have its own identity class whose inheritance hierarchy mirrors the inheritance hierarchy of the owning entity classes.

Listing 8.41 shows an example of an ID class that can be used as an Identity. This class can be used then in the find operation as input.

Listing 8.41 An ID class in OpenJPA

```
@Embeddable
public class LineItemId implements Serializable{
    private static final long serialVersionUID =
        2160402020032769707L;

    private int orderId;
    private int productId;

    //getters and setters
    @Override
    public int hashCode() {
        //calculate hash
    }
    @Override
    public boolean equals(Object obj) {
        //implement equals
    }
}
```

After you do this, you can use the `@IdClass` annotation to specify the class, as shown in Listing 8.42. The ID fields on the Entity must match that on the ID class.

Listing 8.42 IdClass Annotation Usage

```

@Entity
@Table(name="LINE_ITEM")
@IdClass(LineItemId.class)
@NamedQuery(
    name="existing.lineitem.forproduct",
    query="select l from LineItem l
          where l.productId = :productId
          and l.orderId = :orderId"
)
public class LineItem implements Serializable {

    @Id
    @Column(name="ORDER_ID")
    private int orderId;

    @Id
    @Column(name="PRODUCT_ID")
    private int productId;

```

Alternatively, you can use the `@EmbeddedId` annotation and have the ID class as a member of the entity. This is assuming the ID class is annotated as embeddable, as in Listing 8.40. We will discuss embeddable classes later in the chapter. Listing 8.43 shows an alternative implementation of the `LineItem` Entity with the `EmbeddedId`.

Listing 8.43 Embeddable ID

```

@Entity
@Table(name="LINE_ITEM")
public class LineItem implements Serializable {

    @EmbeddedId
    private LineItemId lineItemId;

```

The JPA specification limits identity fields to simple types. OpenJPA, however, also allows `ManyToOne` and `OneToOne` relations to be identity fields. To identify a relation field as an identity field, simply annotate it with both the `@ManyToOne` or `@OneToOne` relation annotation and the `@Id` identity annotation. Listing 8.44 shows an example of how this looks.

Listing 8.44 Entities as ID

```

@Entity
@IdClass(LineItemId.class)

```

```
public class LineItem {  
  
    @Id  
    private int lineItemId;  
  
    @Id  
    @ManyToOne  
    private Order order;  
  
    ...  
}
```

OpenJPA allows you to specify an ID through XML mapping as well. For details, we again refer you to its extensive documentation [OpenJPA 2].

Attributes

A field on an Entity is a primitive Java type. By default, all fields on an Entity are persistent. Therefore, all you have to do is declare a POJO as an entity and define its Ids. Listing 8.45 shows an example of a persistent Product class.

Listing 8.45 Persistent Fields

```
@Entity  
public class Product implements Serializable {  
    @Id  
    protected int productId;  
  
    protected BigDecimal price;  
    protected String description;  
    //getters and setters  
}
```

Alternatively, you can use the `@Basic` annotation to denote that a field is persistent. That `@Basic` annotation is usually not used because fields are persistent by default; however, if you need to override the default handling, you can use it. You will see an example later. To map the field to a column, you can use the `@Column` annotation. An example is shown in Listing 8.46. Notice that you do not have to specify a column annotation for attributes whose name matches that of the column. OpenJPA will map each field to the column with the same name.

Listing 8.46 Mapping Columns

```
public class LineItem implements Serializable {

    @Id
    @Column(name="ORDER_ID")
    private int orderId;

    @Id
    @Column(name="PRODUCT_ID")
    private int productId;

    ...
}
```

You can express the same mappings using XML, as shown in Listing 8.47.

Listing 8.47 XML Mapping for Attributes

```
<entity-mappings>

    <entity class="Order">
        <attributes>
            <id name="orderId">
                <column name="ORDER_ID"/>
            </id>
            <basic name="total"/>
            <basic name="tax">
                <column name="TAX_FIELD">
            </basic>
        </attributes>
    </entity>

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

Fields that you do not want to persist can be marked as transient using the `@Transient` annotation. An example is shown in Listing 8.48. It is left up to the developer to populate this field. We will discuss derived fields later.

Listing 8.48 Declaring a Field to be Transient

```
@Entity
@Table(name="ORDERS")
public class Order implements Serializable {

    @Id
    protected int orderId;
    protected BigDecimal total;
    protected BigDecimal tax;
    @Transient
    protected BigDecimal totalAndTax;
```

The JPA specification defines default mappings between Java Types and Database types. It will handle most basic conversions between Strings and VARCHAR and even Strings to number types. The `@Column` annotation and XML equivalent have additional attributes to customize the mapping.

- **String name**—The column name. Defaults to the field name.
- **String columnDefinition**—The database-specific column type name. This property is used only by vendors that support creating tables from your mapping metadata. During table creation, the vendor will use the value of the `columnDefinition` as the declared column type. If no `columnDefinition` is given, the vendor will choose an appropriate default based on the field type combined with the column's length, precision, and scale.
- **int length**—The column length. This property is typically used only during table creation, though some vendors might use it to validate data before flushing. CHAR and VARCHAR columns typically default to a length of 255; other column types use the database default.
- **int precision**—The precision of a numeric column. This property is often used in conjunction with `scale` to form the proper column type name during table creation.
- **int scale**—The number of decimal digits a numeric column can hold. This property is often used in conjunction with `precision` to form the proper column type name during table creation.
- **boolean nullable**—Whether the column can store null values. Vendors may use this property both for table creation and at runtime; however, it is never required. Defaults to `true`.
- **boolean insertable**—By setting this property to `false`, you can omit the column from SQL INSERT statements. Defaults to `true`.
- **boolean updatable**—By setting this property to `false`, you can omit the column from SQL UPDATE statements. Defaults to `true`.
- **String table**—Sometimes you will need to map fields to tables other than the primary table. This property allows you to specify that the column resides in a secondary table. We will see how to map fields to secondary tables later in the chapter.

The JPA specification also helps deal with more complicated mapping like dates. The `@Temporal` annotation allows you to map a Java Date field to the appropriate database date type. Listing 8.49 shows an example of mapping a Java Date to a `TIMESTAMP` using the `@Temporal` annotation.

Listing 8.49 Declaring a Date Field to be Temporal

```
@Entity
@Table(name="ORDERS")
public class Order implements Serializable {

    @Id
    protected int orderId;
    protected BigDecimal total;
    protected BigDecimal tax;
    @Transient
    protected BigDecimal totalAndTax;
    @Temporal(TemporalType.TIMESTAMP)
    protected java.util.Date orderCreated;
```

OpenJPA also supports mapping fields to CLOB or BLOB columns using the `@Lob` annotation. Listing 8.50 shows an example of using `@Lob` to map a JPEG of the picture into a database column.

Listing 8.50 Declaring a Lob Mapping Type

```
@Entity
public class Product implements Serializable {
    @Id
    protected int productId;
    protected String name;
    protected String description;
    @Lob
    protected JPEG picture;
```

In Java 5, you can use Enumerations. You can mark your mapping to say which value (ordinal or String) you want to persist. Listing 8.51 shows how you can mark the status enumeration to be treated as the String value in the mapping.

Listing 8.51 Declaring Enumerated Field Mapping Types

```
@Entity
@Table(name="ORDERS")
public class Order implements Serializable {
```

```

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="ORDER_ID")
protected int orderId;
protected BigDecimal total;

public static enum Status { OPEN, SUBMITTED, CLOSED }
@Enumerated(EnumType.STRING)
protected Status status;

```

The default is `Ordinal`, but you can change the default value using OpenJPA-specific settings.

JPA supports customizing the fetch option of a field as well. You can fetch any field eagerly (loaded when the object is loaded) versus lazy (loaded when a field is accessed on the managed object). Eager is the default setting. Listing 8.52 shows an example of mapping the JPEG field with a fetch pattern of lazy.

Listing 8.52 Declaring the Fetch style

```

@Entity
public class Product implements Serializable {

    @Id
    protected int productId;
    protected String name;
    protected String description;

    @Lob
    @Basic(fetch=FetchType.LAZY)
    protected JPEG picture;

```

OpenJPA provides an `@ExternalValues` annotation for extending the default mapping. In addition, you can create custom types. See the OpenJPA documentation for more details.

Contained Objects

JPA supports contained objects, which are called embedded objects. For example, suppose there is an `Address` object associated with a `Customer` object in our domain model, but the database has only the `Customer` table with the address fields in it. Figure 8.4 shows this mapping.

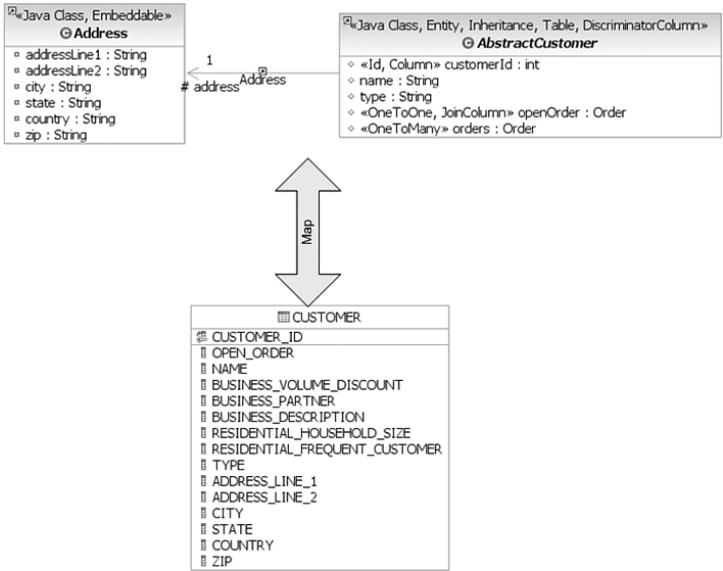


Figure 8.4 Component mapping.

When creating the Address class, you would annotate it as `@Embeddable`. You can add column mappings on that class as well. Listing 8.53 shows the Address class.

Listing 8.53 Marking an Object as Embeddable

```

@Embeddable
public class Address implements Serializable{
    @Column(name="ADDRESS_LINE_1")
    private String addressLine1;

    @Column(name="ADDRESS_LINE_2")
    private String addressLine2;
    private String city;
    private String state;
    private String country;
    private String zip;
}
    
```

Then you can declare an address instance as a member of the class and mark it as `Embedded`, as shown in Listing 8.54.

Listing 8.54 Embedding an Embeddable Object

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    @Column(name="CUSTOMER_ID")
    protected int customerId;
    protected String name;
    protected String type;

    ...
    @Embedded
    protected Address address;
```

The `Embedded` annotation also allows you to override the mapping in case you want to embed the `Address` into another object where the column names in the associated relational table are likely different. You would use special override annotations to do this; refer to the OpenJPA specification for more details.

OpenJPA supports the reverse scenario as well—in which you may have a single object but multiple tables. Suppose that this time your domain model has a `Customer` object with the address information declared as properties rather than as a separate `Address` object, but in the relational database there were a `CUSTOMER` table and an `ADDRESS` table. Figure 8.5 shows the mapping.

In this case, the `ADDRESS` table's primary key is also a foreign key to the `CUSTOMER` table. The mapping for the `Customer` object will look as shown in Listing 8.55. In JPA, you use the `@SecondaryTable` annotation to denote the address table. Then you add the table attribute to each `@Column` annotation you want mapped to the secondary table. You can have several secondary tables allowing you to map several tables that share a primary key to a single object.

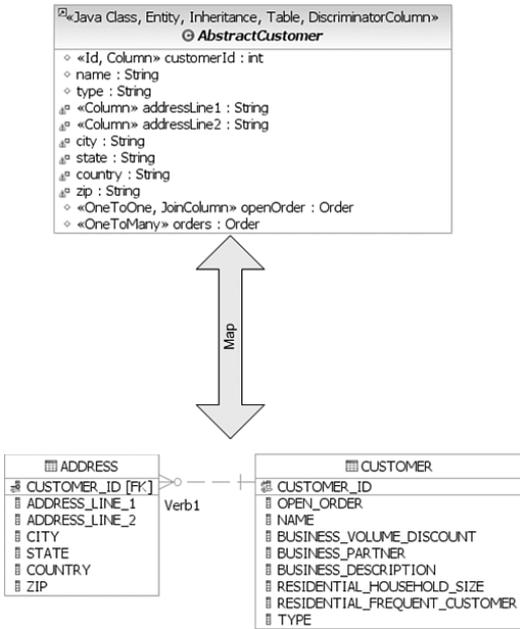


Figure 8.5 Secondary table.

Listing 8.55 Secondary Table Annotation

```

@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@SecondaryTable(name="ADDRESS")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    @Column(name="CUSTOMER_ID")
    protected int customerId;
    protected String name;
    protected String type;

    @Column(name="ADDRESS_LINE_1", table="ADDRESS")
    private String addressLine1;

    @Column(name="ADDRESS_LINE_2", table="ADDRESS")
    
```

```
private String addressLine2;

...

```

Relationships

OpenJPA supports mapping one-to-one, one-to-many, many-to-one, and many-to-many relationships. It allows relationships to be either unmanaged (unidirectional) or managed (bidirectional). The annotations or XML match the type of relationship: `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`. Without any database mappings, the default mapping will use a naming convention for foreign keys. However, you can specify the keys by using the `@JoinColumn`, as shown in Listing 8.56. This listing shows an example of a One-to-One relationship between Customers. You can also define the fetch behavior much like you can with a field, and the behavior for operations against the root object. Cascading operations were shown earlier in the chapter.

Listing 8.56 One-to-One Relationship

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    @Column(name="CUSTOMER_ID")
    protected int customerId;
    protected String name;
    protected String type;

    @OneToOne(
        fetch=FetchType.EAGER,
        cascade = {CascadeType.MERGE,
            CascadeType.REFRESH},
        optional=true
    )
    @JoinColumn(name="OPEN_ORDER", referencedColumnName = "ORDER_ID")
    protected Order openOrder;
}

```

In our example, the Customer also has a one-to-many relationship with all the Orders, as shown in Listing 8.57. You will notice that no Join Column is specified. Instead, the `mappedBy` attribute is used to define a bidirectional relationship.

Listing 8.57 One-to-Many Relationship

```

@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    @Column(name="CUSTOMER_ID")
    protected int customerId;
    protected String name;
    protected String type;

    @OneToOne(
        fetch=FetchType.EAGER,
        cascade = {CascadeType.MERGE,
            CascadeType.REFRESH},
        optional=true
    )
    @JoinColumn(name="OPEN_ORDER", referencedColumnName = "ORDER_ID")
    protected Order openOrder;

    @OneToMany(mappedBy="customer", fetch=FetchType.LAZY)
    protected Set<Order> orders;

```

Listing 8.58 shows the other side of the relationship. The Order has a reference to Customer and it has a many-to-one relationship. Notice it defines all the metadata for the bidirectional relationship.

Listing 8.58 Many-to-One Relationship

```

@Entity
@Table(name="ORDERS")
public class Order implements Serializable {
    private static final long serialVersionUID = 7779370942277849463L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="ORDER_ID")
    protected int orderId;
    protected BigDecimal total;

```

```

public static enum Status { OPEN, SUBMITTED, CLOSED }
@Enumerated(EnumType.STRING)
protected Status status;

@ManyToOne
@JoinColumn(
    name="CUSTOMER_ID", referencedColumnName="CUSTOMER_ID"
)
protected AbstractCustomer customer;

```

Listing 8.59 shows Order's one-to-many relationship to LineItems. This is a unidirectional relationship because LineItem does not have a reference to Order. In this case, notice the use of a special `@ElementJoinColumn`. This is an OpenJPA extension that allows you to define the metadata for a one-to-many one-way mapping.

Listing 8.59 One-to-Many Relationship

```

@Entity
@Table(name="ORDERS")
public class Order implements Serializable {

    // Removing code that is the same as in Listing 8.58

    @OneToMany(cascade=CascadeType.REMOVE, fetch=FetchType.EAGER )
    @ElementJoinColumn(
        name="ORDER_ID", referencedColumnName="ORDER_ID"
    )
    protected Set<LineItem> lineitems;

```

Another way to map relationships in OpenJPA is through a join table, as described in Chapter 3. This is actually used for many-to-many relationships, but it can also be used for one-to-many and many-to-one relationships. As a matter of fact, join tables are the only implementation of unidirectional one-to-many relationships that the JPA specification demands for compliance. In Chapter 3 we show a `PRODUCT_CATEORGY` table that defines both keys for the product and category tables. A Product can belong to different categories, and a category groups many products.

Listing 8.60 shows a bidirectional relationship between products and categories.

Listing 8.60 Many-to-Many Relationship

```

@Entity
@NamedQuery(name="product.all",query="select p from Product p")
public class Product implements Serializable {

```

```

@Id
@Column(name="PRODUCT_ID")
protected int productId;

protected BigDecimal price;
protected String description;

```

@ManyToMany

```

@JoinTable(name="PRODUCT_CATEGORY",
    joinColumns={@JoinColumn(name="PRODUCT_ID")},
    inverseJoinColumns={@JoinColumn(name="CAT_ID")}
)
protected Collection<Category> categories;

```

...

```

@Entity
public class Category implements Serializable {

```

```

@Id
protected int CAT_ID;
protected String name;

```

@ManyToMany(mappedBy="categories")

```

protected Collection<Product> products;

```

```

public int getCAT_ID() {
    return CAT_ID;
}

```

...

Collection-based Entities can also be ordered using the `@OrderBy` annotation, as shown in Listing 8.61.

Listing 8.61 Order Constraint

```

@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

```

```

@Id
@Column(name="CUSTOMER_ID")
protected int customerId;
protected String name;
protected String type;

@OrderBy("status")
protected Collection orders;

```

Constraints

OpenJPA allows you to define constraints on various mappings as well as work with database constraints. Some constraints have special annotations, whereas others are attributes on another annotation. Listing 8.62 shows an example of a unique constraint that declares the field as suitable for a key.

Listing 8.62 Unique Constraint

```

@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    @Column(name="CUSTOMER_ID")
    protected int customerId;
    protected String name;
    protected String type;

    @Unique
    protected String ssID;

```

Refer once again to the associated OpenJPA documentation for a list of other constraints supported [OpenJPA 2].

Derived Attributes

Earlier, we showed that we can make a field transient with annotations in the code or XML mapping file. Often, you need to calculate transient fields based on other persistent fields. To do this properly, you need to know when data is loaded and persisted to manage the state of the object. Although Entities are just POJOs in OpenJPA, you can define Entity Listener methods on the POJO, or attach an `EntityListener` class to the POJO. Listing 8.63 shows an example of marking a method with the `@PostLoad` annotation, which causes the method to be invoked after the data is loaded.

Listing 8.63 Life Cycle Methods

```
@Entity
@Table(name="ORDERS")
public class Order implements Serializable {

    @Id
    protected int orderId;
    protected BigDecimal total;
    protected BigDecimal tax;
    @Transient protected BigDecimal totalAndTax; //Not persisted

    @PostLoad
    protected void calculateTotal() {
        totalAndTax =
            total+tax;
    }
}
```

As you can see from Listing 8.63, the `totalAndTax` field will not be set until the object is loaded and the persistent fields have been set.

JPA also supports the following callbacks for life cycle events and their corresponding method markers:

- **PrePersist**—Methods marked with this annotation will be invoked before an object is persisted. This could be used for assigning primary key values to persistent objects. This is equivalent to the XML element tag `pre-persist`.
- **PostPersist**—Methods marked with this annotation will be invoked after an object has transitioned to the persistent state. You might want to use such methods to update a screen after a new row is added. This is equivalent to the XML element tag `post-persist`.
- **PostLoad**—Methods marked with this annotation will be invoked after all eagerly fetched fields of your class have been loaded from the datastore. No other persistent fields can be accessed in this method. This is equivalent to the XML element tag `post-load`.
- **PreUpdate**—This is the complement to `PostLoad`. While methods marked with `PostLoad` are most often used to initialize nonpersistent values from persistent data, methods annotated with `PreUpdate` are normally used to set persistent fields with information cached in nonpersistent data.
- **PostUpdate**—Methods marked with this annotation will be invoked after changes to a given instance have been stored to the datastore. This is useful for clearing stale data cached at the application layer. This is equivalent to the XML element tag `post-update`.

- **PreRemove**—Methods marked with this annotation will be invoked before an object transactions to the deleted state. Access to persistent fields is valid within this method. You might use this method to cascade the deletion to related objects based on complex criteria, or to perform other cleanup. This is equivalent to the XML element tag `pre-remove`.
- **PostRemove**—Methods marked with this annotation will be invoked after an object has been marked as to be deleted. This is equivalent to the XML element tag `post-remove`.

You can also externalize the callback methods to a different class using the `@EntityListener` annotation on the class. Refer to the OpenJPA documentation for more details.

Tuning Options

Query Optimizations

Some vendors allow an option to compile the associated SQL at build time, which can greatly improve performance. Also, it is possible to create named queries that enable specifying the SQL in the external configuration file. The SQL can then be tuned separately from the code.

Caching

The JPA specification defines two levels of caching: one that is scoped to the Entity Manager and another that is scoped to the Persistent Unit. The Entity Manager level cache is associated with a current transaction unless you are using the Extended Context—in which you can scope the cache to the life cycle of a Stateful Session Bean. We discussed this life cycle earlier in the “Programming Model” section. The JPA Entity Manager has a flush method you can invoke to clear the cache and push changes to the database.

The Entity Manager cache is there is for keeping data around during a multi-request flow, often called a “conversation;” however, the persistence unit cache is meant to really be a performance booster for read-only or read-mostly data. OpenJPA provides a data-level cache at the persistence unit level. OpenJPA provides a Single JVM cache provider. This may be ideal for read-only data that is initialized during the startup of an application. To use it, you can configure a cache provider on the persistence unit. Listing 8.64 shows an example of configuring the single JVM cache.

Listing 8.64 Caching

```
<persistence-unit name="pie-db-JAVA-SE">
  <provider>
    org.apache.openjpa.persistence.PersistenceProviderImpl
  </provider>
```

```

<properties>
  <property name="openjpa.MaxFetchDepth" value="5"/>
  <property name="openjpa.jdbc.MappingDefaults"
    value="StoreEnumOrdinal=false"/>
  <property name="openjpa.ConnectionURL"
    value="jdbc:derby://localhost:1527/PWTE"/>
  <property name="openjpa.ConnectionDriverName"
    value="org.apache.derby.jdbc.ClientDriver"/>
  <property name="openjpa.jdbc.DBDictionary" value="derby"/>
  <property name="openjpa.jdbc.Schema" value="APP"/>
  <property name="openjpa.DataCache" value="true"/>
</properties>
</persistence-unit>

```

After you configure the cache, you can configure whichever entity you need to cache. For example, in Listing 8.65 we cache the product data. In the listing, the timeout attribute of the annotation specifies when the provider should invalidate the cache from when the data was first created.

Listing 8.65 Life Cycle Methods

```

import org.apache.openjpa.persistence.DataCache;

@Entity
@NamedQuery(name="product.all",query="select p from Product p")
@DataCache(timeout=6000000)
public class Product implements Serializable {

    @Id
    @Column(name="PRODUCT_ID")
    protected int productId;

```

A single JVM often will not work for read-mostly cases or distributed cache facilities meant to deal with large volumes of data in a distributed fashion to alleviate database contention. There are some distributed cache technologies that specialize in these situations. IBM WebSphere Extended Data Grid [DataGrid] also allows you to work with distributed cache technologies as a second-level cache to OpenJPA, as well as supporting other patterns for using JPA with cached data.

OpenJPA also supports a QueryCache to cache the results of queries so that they can be reused without going back to the database. See the OpenJPA documentation for more details.



Loading Related Objects

OpenJPA supports both eager and lazy loading. We showed in the “Attributes” and “Relationships” sections that you can mark a field or relationship with a fetch *type*. In addition, OpenJPA supports fetch *groups* that allow you to link objects that are logically linked together in your application. If you have a class with two lazy fields, but you know in your application that when you load one, you will most likely load the other, you can configure a fetch group. Listing 8.66 shows how to load Customer and LineItem objects together whenever the other is accessed in the context of a lazily loaded Order.

Listing 8.66 Fetch Groups

```
@Entity
@FetchGroups({
    @FetchGroup(name="detail", attributes={
        @FetchAttribute(name="lineItems"),
        @FetchAttribute(name="customers")
    })
})
class Order
```

This `FetchGroups` feature can be very important to minimize the number of round trips to the database at the same time that you minimize the initial load time of an object.

Locking

OpenJPA provides both configuring and an API option for affecting the desired locking level. For more details on database locking, see the paper titled “*Locking Strategies for Database Access.*”

In addition, because OpenJPA supports disconnected patterns, you can use the `@Version` annotation to map a particular version column to a database. Listing 8.67 shows how simple this task can be.

Listing 8.67 Version Number

```
@Entity
public class Order {

    @Id private String orderId;
    @Version private int version;
```

Remember that when an object becomes disassociated with its `EntityManager`, it becomes disconnected. When the object is reattached, OpenJPA will check whether the version number has changed in the database before performing the updates. When the JPA runtime detects an attempt to concurrently modify the same record, it throws an exception to the



A.8.5

transaction attempting to commit last. This prevents overwriting the previous commit with stale data.

A version field is not always required, but without one, concurrent threads or processes might succeed in making conflicting changes to the same record at the same time.

Development Process of the Common Example

Now that we have gone through OpenJPA in detail, we will show you the steps required to develop a complete application. To run it yourself, follow the directions in Appendix A, “Setting Up the Common Example.” Keep in mind that Chapter 2, “High-Level Requirements and Persistence,” and Chapter 3, “Designing Persistent Object Services,” describe the requirements and design of our example. This section focuses on the details related to developing OpenJPA applications after you understand the requirements and settle on a design.

Defining the Object

When developing an OpenJPA application, you define your objects by coding Java Classes with annotations, and then coding XML mapping files. The following listings show our domain model implemented in Java with OpenJPA annotations being used to specify the mapping metadata. We only show subsets of the classes to illustrate the mapping. You can examine all the code by downloading the sample as shown in Appendix A. Listing 8.68 lists the `AbstractCustomer` superclass. It is mapped using a Single Table Strategy. Besides the defaulted primitive value fields, we explicitly map the open order field as a one-to-one relationship to the `ORDERS` table because a Customer object may have at most one open order, and we map the orders field as a one-to-many relationship with respect to the `ORDERS` table because a Customer might refer to more than one in any state, including `open`. The orders field is declared a bidirectional relationship, and therefore additional details are defined on the Order side. We also define an Eager fetching strategy, because we want to fetch the open order record whenever the customer is accessed. We use a `Lazy` option to load the orders collection only when we specifically access the history.

Listing 8.68 Abstract Customer

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@Table(name = "CUSTOMER")
@DiscriminatorColumn(name="TYPE", discriminatorType = STRING)
public abstract class AbstractCustomer implements Serializable {

    @Id
    @Column(name="CUSTOMER_ID")
    protected int customerId;
```

```

protected String name;
protected String type;

@OneToOne(
    fetch=FetchType.EAGER,
    cascade = {CascadeType.MERGE,CascadeType.REFRESH},
    optional=true
)
@JoinColumn(name="OPEN_ORDER", referencedColumnName = "ORDER_ID")
protected Order openOrder;
@OneToMany(mappedBy="customer", fetch=FetchType.LAZY)
protected Set<Order> orders;

... //Getters and Setters...
}

```

Listing 8.69 shows the `ResidentialCustomer` subclass. Notice we used the `DiscriminatorValue` to determine the type. We described this in the Inheritance section of the template.

Listing 8.69 Residential Customer

```

@Entity
@DiscriminatorValue("RESIDENTIAL")
public class ResidentialCustomer
extends AbstractCustomer implements Serializable {
    @Column(name="RESIDENTIAL_HOUSEHOLD_SIZE")
    protected short householdSize;
    @Column(name="RESIDENTIAL_FREQUENT_CUSTOMER")
    protected boolean frequentCustomer;
    //Getters and Setters...
}

```

Listing 8.70 shows the `BusinessCustomer` subclass, which is similar to the `ResidentialCustomer` subclass.

Listing 8.70 Business Customer

```

@Entity
@DiscriminatorValue("BUSINESS")
public class BusinessCustomer extends AbstractCustomer implements Serializable {
    @Column(name="BUSINESS_VOLUME_DISCOUNT")

```

```

protected boolean volumeDiscount;
@Column(name="BUSINESS_PARTNER")
protected boolean businessPartner;
@Column(name="BUSINESS_DESCRIPTION")
protected String description;
//Getters and Setters...
}

```

Listing 8.71 shows the Order object. We elected to generate the Order ID using the Identity Strategy because this primary key is bound to a single table. We also map a relationship back to the `AbstractCustomer` using `ManyToOne`. The detail of this bidirectional relationship is defined on the Order object.

The Order object also has a Set of `LineItems` for the Order that implements a unidirectional relationship between the Order and `LineItem` classes. We have no requirement to navigate from a `LineItem` to an Order; defining a single-sided relationship will make the underlying SQL optimal.

Listing 8.71 Order Object

```

@Entity
@Table(name="ORDERS")
public class Order implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="ORDER_ID")
    protected int orderId;
    protected BigDecimal total;
    public static enum Status { OPEN, SUBMITTED, CLOSED }
    @Enumerated(EnumType.STRING)
    protected Status status;
    @ManyToOne
    @JoinColumn(
        name="CUSTOMER_ID", referencedColumnName = "CUSTOMER_ID"
    )
    protected AbstractCustomer customer;

    @OneToMany(cascade=CascadeType.REMOVE, fetch=FetchType.EAGER )
    @ElementJoinColumn(name="ORDER_ID",referencedColumnName="ORDER_ID"
    )
    protected Set<LineItem> lineitems;

    //getters and setters
}

```

Listing 8.72 shows the `LineItem` class. You will notice that the `LineItem` class is a composite key. This choice was made to illustrate how to map to certain legacy schemas. ORM technologies tend to work better with generated keys. The `LineItem` also has a one-to-one relationship to the `Product`. This is also a unidirectional relationship because there is no requirement to navigate from a `Product` instance to the `LineItem` objects that reference it. `Product` instances also usually are cached because the product catalog changes infrequently.

Listing 8.72 `LineItem`

```

@Entity
@Table(name="LINE_ITEM")
@IdClass(LineItemId.class)
@NamedQuery(name="existing.lineitem.forproduct",
    query="select l from LineItem l
        where l.productId = :productId and l.orderId = :orderId"
)

public class LineItem implements Serializable {
    @Id
    @Column(name="ORDER_ID")
    private int orderId;

    @Id
    @Column(name="PRODUCT_ID")
    private int productId;
    protected long quantity;
    protected BigDecimal amount;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn({
        @JoinColumn(name="PRODUCT_ID",
            referencedColumnName = "PRODUCT_ID"
        )
    })
    protected Product product;

    //getters and setters
}

```

Listing 8.73 shows the composite primary key for the `LineItem` Entity.

Listing 8.73 Line Item ID

```
@Embeddable
public class LineItemId implements Serializable{
    private int orderId;
    private int productId;
    //getters and setters
    @Override
    public int hashCode() {
        //unique hashcode
    }
    @Override
    public boolean equals(Object obj) {
        //equals
    }
}
```

Listing 8.74 shows the Product. The Product is a simple class that is mapped to the PRODUCT table. It has no relationships. (Real products usually have many more details and belong to categories and such.) We have added a `NamedQuery` to the Product annotations to specify a query that retrieves the list of products. As noted previously, we also cache the Product. This example should not be considered complete and is for demonstration purposes only; products in real enterprise systems are usually indexed, categorized, and related to extensive catalog and inventory management systems.

Listing 8.74 Product

```
@Entity
@NamedQuery(name="product.all",query="select p from Product p")
@DataCache(timeout=6000000)
public class Product implements Serializable {
    @Id
    @Column(name="PRODUCT_ID")
    protected int productId;
    protected BigDecimal price;
    protected String description;
    //getters and setters
}
```

We did not employ the option to provide XML mapping files. OpenJPA is meant to reduce the number of artifacts one has to develop. We could have chosen to minimize the annotations and specify the mapping inside XML descriptors. For example, in a real application, we recommend externalizing the cache period and the named query so that the code need not change to modify these parameters.

Key Point

It is a best practice to externalize named queries and cache configurations in EJB XML deployment descriptors and not to place them in the source.

Implementing the Services

This section shows the implementation of the Service. For OpenJPA, we provided both an EJB 3 version of the service and a Java SE version. Because EJB 3 Session Beans are Java classes, the Java SE version just extends the proper EJB 3 class and bootstraps the EntityManager as described earlier. Figure 8.6 shows all the exceptions for our services. See the downloadable sample or refer to Chapter 3 for details.

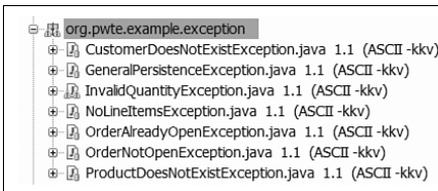


Figure 8.6 Exceptions from the common example.

We have two implementations of our service: one for Java SE using OpenJPA and another using IBM JPA (built on OpenJPA) for EJB 3. The interface for the service was introduced in Chapter 3, so refer to the details there. For the EJB 3 version, the service interface has an extra `@Local` annotation to denote a Local EJB. The service implementations vary slightly from a bootstrapping standpoint with Java SE.

Specifically, the Java SE version of the application looks up the EntityManager using the EntityManagerFactory, as illustrated earlier in Figure 8.3. The EJB 3 version is illustrated earlier in Figure 8.5.

The `loadCustomer` operation uses the EntityManager `find` method to look up the customer. Because the mapping file defined the proper eager loading, it will load the customer record, an open order if it exists, and any line items for that order. The Inheritance is also automatic; based on the type, it will return the correct subclass. All this happens with one call. Listing 8.75 shows the `loadCustomer` method as implemented in the EJB 3 version. The Java SE version uses transaction demarcation. You can examine the downloadable source to see the difference. Appendix A shows how to load the code into your Eclipse-based development environment.

Listing 8.75 The loadCustomer Implementation

```
public AbstractCustomer loadCustomer(int customerId)
throws ExistException, GeneralPersistenceException {
    AbstractCustomer customer = em.find(
        AbstractCustomer.class, customerId
    );
    return customer;
}
```

The `openOrder` operation creates an order Java Object and persists it. It then sets the new order onto the customer instance. The transaction is implied because of the nature of EJBs. The `openOrder` routine will check if an order is open and throw an exception if it is. Listing 8.76 shows the implementation.

Listing 8.76 The openOrder Implementation

```
public Order openOrder(int customerId)
throws CustomerDoesNotExistException, OrderAlreadyOpenException,
    GeneralPersistenceException{
    AbstractCustomer customer = loadCustomer(customerId);
    Order existingOpenOrder = customer.getOpenOrder();
    if(existingOpenOrder != null) {
        throw new OrderAlreadyOpenException();
    }
    Order newOrder = new Order();
    newOrder.setCustomer(customer);
    newOrder.setStatus(Order.Status.OPEN);
    newOrder.setTotal(new BigDecimal(0));
    em.persist(newOrder);
    customer.setOpenOrder(newOrder);
    return newOrder;
}
```

The implementation for the `addLineItem` operation first checks to see whether the Product exists using the Entity Manager `find` method as seen in the other examples. It then queries to check whether a Line Item already exists, and updates the quantity if it does. Otherwise, it creates a new instance. Again, the transaction is implied due to the EJB 3 method. Listing 8.77 shows the implementation of `addLineItem`.

Listing 8.77 The addLineItem Implementation

```
public LineItem addLineItem(
    int customerId,
```

```
int productId,
long quantity)
throws CustomerDoesNotExistException,
    OrderNotOpenException,
    ProductDoesNotExistException,
    GeneralPersistenceException,
    InvalidQuantityException {
    Product product = em.find(Product.class,productId);
    if(quantity <= 0 ) throw new InvalidQuantityException();
    if(product == null) throw new ProductDoesNotExistException();
    AbstractCustomer customer = loadCustomer(customerId);
    Order existingOpenOrder = customer.getOpenOrder();
    if(existingOpenOrder == null) {
        throw new OrderNotOpenException();
    }
    BigDecimal amount = product.getPrice().multiply(
        new BigDecimal(quantity)
    );
    existingOpenOrder.setTotal(
        amount.add(existingOpenOrder.getTotal())
    );
    LineItemId lineItemId = new LineItemId();
    lineItemId.setProductId(productId);
    lineItemId.setOrderId(existingOpenOrder.getOrderId());
    LineItem existingLineItem = em.find(LineItem.class,lineItemId);
    if(existingLineItem == null) {
        LineItem lineItem = new LineItem();
        lineItem.setOrderId(existingOpenOrder.getOrderId());
        lineItem.setProductId(product.getProductId());
        lineItem.setAmount(amount);
        lineItem.setProduct(product);
        lineItem.setQuantity(quantity);
        em.persist(lineItem);
        return lineItem;
    }
    else {
        existingLineItem.setQuantity(
            existingLineItem.getQuantity() + quantity
        );
        existingLineItem.setAmount(
            existingLineItem.getAmount().add(amount)
        );
    }
}
```

```

        return existingLineItem;
    }
}

```

The `removeLineItem` operation simply deletes the `LineItem` by finding the record and removing it. Listing 8.78 shows the implementation.

Listing 8.78 The `removeLineItem` Implementation

```

public void removeLineItem(
    int customerId,
    int productId
)
throws CustomerDoesNotExistException,
    OrderNotOpenException,
    ProductDoesNotExistException,
    NoLineItemsException,
    GeneralPersistenceException {
    Product product = em.find(Product.class,productId);
    if(product == null) throw new ProductDoesNotExistException();
    AbstractCustomer customer = loadCustomer(customerId);
    Order existingOpenOrder = customer.getOpenOrder();
    if(existingOpenOrder == null ||
        existingOpenOrder.getStatus() != Order.Status.OPEN)
        throw new OrderNotOpenException();
    LineItemId lineItemId = new LineItemId();
    lineItemId.setProductId(productId);
    lineItemId.setOrderId(existingOpenOrder.getOrderId());
    LineItem existingLineItem = em.find(LineItem.class,lineItemId);
    if(existingLineItem != null) {
        em.remove(existingLineItem);
    }
    else {
        throw new NoLineItemsException();
    }
}

```

The `submitOrder` operation is almost as simple—it changes the status of the order and removes it from the `openOrder` property of the abstract customer class. Listing 8.79 shows the `submitOrder` implementation.

Listing 8.79 The submitOrder Implementation

```

public void submit(int customerId)
throws CustomerDoesNotExistException, OrderNotOpenException,
       NoLineItemsException,
       GeneralPersistenceException {
    AbstractCustomer customer = loadCustomer(customerId);
    Order existingOpenOrder = customer.getOpenOrder();
    if(existingOpenOrder == null ||
        existingOpenOrder.getStatus() != Order.Status.OPEN)
        throw new OrderNotOpenException();
    if(existingOpenOrder.getLineitems() == null ||
        existingOpenOrder.getLineitems().size() <= 0 )
        throw new NoLineItemsException();
    existingOpenOrder.setStatus(Order.Status.SUBMITTED);
    customer.setOpenOrder(null);
}

```

Packaging the Components

The environment you deploy to will affect how the application is packaged. A Java SE environment may require you to copy files, or package the code into a JAR. You most likely have to write some deployment scripts. Figure 8.7 shows the Java Project in Eclipse. It is a plain Java Project with the `persistence.xml` defined in the `meta-inf` directory. The `persistence.xml` is necessary to obtain the connection.

**Figure 8.7** Packaging.

In a Java EE application, you usually have to package an application in an EAR file. Figure 8.8 shows the layout of the EAR file, which is made up of other files, such as EJB-JAR files for EJBs or WAR files for web applications. See the Java EE specification for details. Notice that we can use the same Java SE JAR as an EJB 3 module as well.

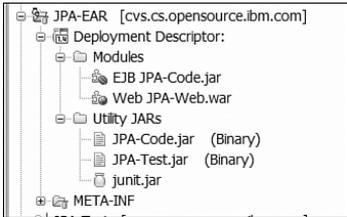


Figure 8.8 Java EE EAR.

The `persistence.xml` file is packaged in the `meta-inf` directory and contains our persistence units. We illustrated the format earlier. As with the operation implementations, you can examine the downloadable source for more details.

Unit Testing

Depending on the methodology, you may have coded your unit test before or after implementing the service operations. Agile methods usually push a test-driven approach and encourage coding test cases first. Regardless, our OpenJPA example contains the unit test to run the application. Figure 8.9 shows the Unit Test project for the common example.

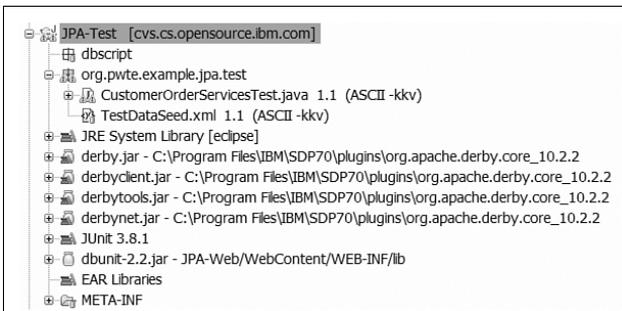


Figure 8.9 Unit Test package.

Chapter 3 explains the aspects of the unit test that are the same regardless of the persistence mechanism used (as it should be). The only difference with OpenJPA is that we also provide the option to look up the Session Bean in the Java EE case and use `JUnitEE` to run it. There is no JPA-specific information within the unit test, but there is EJB information. As long as you have the Java EE API JARs, this unit test can run in both a Java SE environment and a standard Java EE environment. Examine the downloadable source for details and Appendix A for instructions on how to run them.

Deploying to Production

Deploying to production involves setting the proper configuration values for the target environment. Other than the common testing needed to move an application to production, there are no additional considerations other than those already specified in this section. In a Java SE environment, you have to run the bytecode enhancer yourself. In a Java EE container, like WebSphere Application Server, the bytecode enhancer is run automatically when the installation tools are run, so it will not be an explicit step.

Summary

OpenJPA is a very capable persistence technology. In addition to providing much function, it is based on a standard. Products like WebSphere Application Server provide enhanced versions of OpenJPA, and caching technologies like ObjectGrid can provide enterprise-quality applications. In this chapter, we showed you how to code against the OpenJPA APIs, map entities, and configure the system for your applications. The OpenJPA documentation can provide you with many more details. In addition, we showed you how to implement the services from the common example.

The JPA 2.0 specification was being written at the time of this writing. Some extensions from OpenJPA, Hibernate, and others may eventually be included in the specification. The Specification group is looking at the following items:

- Expanded object/relational mapping functionality, including greater flexibility in combining existing mapping options, support for collections of embedded objects, multiple levels of embedded objects, ordered lists, combinations of access types, and so forth.
- Additions to the Java Persistence query language.
- An API for “criteria” queries. Hibernate contains a Criteria API we covered in the previous chapter.
- Standardization of sets of “hints” for query configuration and for Entity Manager configuration.
- Standardization of additional metadata to support DDL generation and “Java2DB” mapping.
- Expanded pluggability contracts to support efficient passivation and replication of extended persistence contexts in Java EE environments.
- Standardization of additional contracts for entity detachment and merge, and persistence context management.
- Support for validation.

You should keep an eye on the progress of this specification and see which of these features actually get added to the standard.



Links to developerWorks

A.8.1 *Building EJB 3.0 applications with WebSphere Application Server*

This article shows you how to create EJB 3 style applications with IBM's JPA provider, which is built on OpenJPA.

www.ibm.com/developerworks/websphere/techjournal/0712_barcia/0712_barcia.html 2007

A.8.2 *Leveraging OpenJPA with WebSphere Application Server*

This article discusses configuring Apache OpenJPA with WebSphere Application Server version 6.1 when the EJB 3 Feature Pack is present.

www.ibm.com/developerworks/websphere/techjournal/0612_barcia/0612_barcia.html

A.8.3 *Migrating legacy Hibernate Applications to OpenJPA and EJB 3*

This article discusses best practices for migrating a Hibernate Core Application to OpenJPA

www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html

A.8.4 *Build Grid-Ready Apps with ObjectGrid*

Tutorial on building application for IBM ObjectGrid (renamed WebSphere XD DataGrid)

www.ibm.com/developerworks/edu/wes-dw-wes-objectgrid.html

A.8.5 *Locking Strategies for Database Access*

Excellent paper on database locking.

www.ibm.com/developerworks/websphere/techjournal/0603_ilechko/0603_ilechko.html

References

[Apache] *Apache Commons*. commons.apache.org/

[Bauer] Bauer, Christian and King, Gavin. *Java Persistence with Hibernate*. Manning Publications Company 2007

[DataGrid] *WebSphere Extended Deployment Data Grid*. www-306.ibm.com/software/web-servers/appserv/extend/datagrid/

[Hibernate] *Java Persistence with Hibernate*. hibernate.org/397.html

[JPA 2] *JSR 317: Java Persistence 2.0*. jcp.org/en/jsr/detail?id=317

[JPQL] *JPQL Language Reference*. openjpa.apache.org/builds/1.0.1/apache-openjpa-1.0.1/docs/manual/jpa_langref.html

[Monson-Haefel] Burke, Bill and Monson-Haefel, Richard. *Enterprise JavaBeans 5th Edition*. O'Reilly 2006

[OpenJPA 1] *Apache Open JPA Home Page*. openjpa.apache.org/

[OpenJPA 2] *Open JPA Documentation*. openjpa.apache.org/documentation.html

[OpenJPA 3] *Open JPA Integration Index*. openjpa.apache.org/integration.html

[Serp] *Serp Homepage*. serp.sourceforge.net/

[TopLink] *TopLink JPA*. www.oracle.com/technology/products/ias/toplink/jpa/index.html

Index

Symbols

- { } (braces), 125
- \${...} syntax, 324
- 1NF (First Normal Form), 59
- 2NF (Second Normal Form), 59
- 3NF (Third Normal Form), 59

A

- AbstractCustomer class
 - AbstractCustomer.hbm.xml mapping file, 235-237
 - Hibernate, 234
 - iBATIS, 180
 - JDBC (Java Database Connectivity), 135
 - OpenJPA, 297-298
 - pureQuery, 343
- AbstractCustomer.hbm.xml mapping file, 235-237
- ACID properties, 37
- addLineItem service
 - Hibernate, 242-244
 - iBATIS, 190-192
 - JDBC (Java Database Connectivity), 139-140
 - OpenJPA, 303-305
 - pureQuery, 348-349
- administration, 31-32
- aggregations, 56
- agile approaches, 20-21, 365-367
- Analyst roles, xxx
- annotations. *See specific annotations*
- Apache Derby
 - Apache Derby Nature installation, 378
 - downloading, 374
 - overview, 372
- Apache iBATIS. *See iBATIS*
- Apache Network Server, starting, 379
- Apache OpenJPA. *See OpenJPA*
- AppendTimestampRowHandler, 162
- application-managed EM instances, 255
- application-managed persistence contexts, 255
- Architect roles, xxx
- Architectural Overview section (evaluation questionnaire), 97-98
- architecture
 - comparison of persistence mechanisms, 358
 - Hibernate
 - dependencies, 201
 - licenses, 201
 - overview, 200
 - platforms, 201

- references and further reading, 201-202, 248
 - standards, 200-201
 - vendors, 201
- iBATIS
 - dependencies, 147-148
 - licenses, 148
 - overview, 146
 - platforms, 147
 - resources, 148-149
 - vendors, 148
- JDBC (Java Database Connectivity)
 - dependencies, 113
 - drivers, 112
 - JDBC 1.0, 111
 - JDBC 2.0, 111
 - JDBC 3.0, 111
 - JDBC 4.0, 111
 - licenses, 113
 - online resources, 113-114
 - overview, 110-111
 - vendors, 113
- OpenJPA
 - dependencies, 251
 - licenses, 251
 - platforms, 251
 - references and further reading, 251-252, 309-310
 - standards, 250
 - vendors, 251
- pureQuery
 - dependencies, 316
 - layered approach, 314
 - licenses, 316
 - life cycle, 313-314
 - platforms, 316
 - references and further reading, 316-317, 356
 - standards, 315-316
 - vendors, 316
- associations, 67-71
- atomic transactions, 37
- attributes
 - comparison of persistence mechanisms, 361
 - Hibernate, 223-224, 228
 - iBATIS, 169-172, 176-178
 - JDBC (Java Database Connectivity), 130-133
 - OpenJPA, 280-284, 292-294
 - pureQuery, 336-338
- availability
 - of education and mentors, 30
 - of skilled practitioners, 29-30
- B**
- Background section (evaluation questionnaire), 97
- backward compatibility (iBATIS), 147
- @Basic annotation, 280
- batch operations
 - batch applications, 37-38
 - batch fetching, 231
 - comparison of persistence mechanisms, 360
 - Hibernate, 215
 - iBATIS, 160
 - JDBC (Java Database Connectivity), 125
 - OpenJPA, 266
 - pureQuery, 331
- batch statement, 160
- BatchException, 163
- BCNF (Boyce-Codd normal form), 59
- Beck, Kent, 373
- Begin, Clinton, 146
- bidirectional relationships, 55
- bootstrapping, 240
- Borland Delphi, 5
- bottom-up object-relational mapping, 81-82
- Boyce-Codd normal form (BCNF), 59
- braces ({}), 125
- build versus buy, 32
- Building EJB 3 Applications with WebSphere Application Server (tutorial), 252

Building RESTful Services for your Web
Application with Project Zero
(tutorial), 317

Business Executives, 20

business processes, 34-36

BusinessCustomer class

 Hibernate, 234

 JDBC (Java Database Connectivity), 136

 OpenJPA, 298-299

 pureQuery, 344

buy versus build, 32

bytecode enhancement, 389-390

C

CacheController interface, 179

CachedRowSet objects, 123

caching, 40-41

 comparison of persistence
 mechanisms, 361

 distributed caching, 179

 Hibernate, 229

 iBATIS

 distributed caching, 179

 single-JVM caching, 178-179

 OpenJPA, 294-295

 OSCache, 148

 pureQuery, 340

 single-JVM caching, 178-179

 tuning options, 133

@Call annotation, 330

Callable Statements, 118

cascading, deleting data via, 265

CD/CD (Community-Driven Commercial
Development), 315

Centers of Excellence (COE), 30

checklist, evaluation, 90-91

Class Table Inheritance pattern, 73-74

classes. *See specific classes*

clear() method, 215

close() method, 117

Cloudscape®, 372

Codd, Edgar Frank, 59

COE (Centers of Excellence), 30

@Column annotation, 280-282, 336

commitTransaction() method, 153

common example (customer order
management application), 53

AbstractCustomer class

 AbstractCustomer.hbm.xml mapping
 file, 235-237

 Hibernate, 234

 iBATIS, 180

 JDBC (Java Database Connectivity), 135

 OpenJPA, 297-298

 pureQuery, 343

addLineItem service

 Hibernate, 242-244

 iBATIS, 190-192

 JDBC (Java Database
 Connectivity), 139-140

 pureQuery, 348-349

benefits of, 53

bootstrapping, 240

BusinessCustomer class

 addLineItem service, 303-305

 component packaging, 306-307

 deployment, 308

 Hibernate, 234

 JDBC (Java Database Connectivity), 136

 LineItem class, 300

 LineItemId class, 300-301

 loadCustomer service, 302-303

 OpenJPA, 298-299

 openOrder service, 303

 Order object, 299

 Product class, 301

 pureQuery, 344

 removeLineItem service, 305

 submitOrder service, 305-306

 unit testing, 307

component packaging

 Hibernate, 246-247

 iBATIS, 194-195

 JDBC (Java Database Connectivity), 142

 pureQuery, 351-352

configuration

- Apache Derby Nature installation, 378
- Apache Network Server, starting, 379
- database script, 379-380
- DbUnit resolution, 377
- EJB3 application, running, 393-400
- Hibernate application, running, 385-386
- Hibernate dependencies, resolving, 384-385
- iBATIS dependencies, resolving, 381-382
- iBATIS JUnit, 383
- Java SE applications, importing, 375-376
- JDBC unit test, 380
- OpenJPA application, running, 391
- OpenJPA bytecode enhancement, 389-390
- OpenJPA dependencies, resolving, 387-388
- overview, 371
- prerequisites, 373-375
- Project Zero application, running, 391-392
- references and further reading, 401
- troubleshooting, 400
- Customer.xml file, 181-183
- CustomerLoad.xml file, 185-187
- database constraints, 58
- database normalization, 59-60
- database schema, 57-58
- deployment
 - Hibernate, 247
 - iBATIS, 195
 - JDBC (Java Database Connectivity), 142
 - pureQuery, 354
- domain model, 53-56
 - customer inheritance hierarchy, 54
 - illustration, 54
 - relationship between Customer and Order, 55-56

- relationship between Order and LineItem, 56
- relationship between Order and Product, 56-57
- enterprise application tiers, 60-61
- Java interface, 61-62
- JDBC development process, 136
- LineItem class
 - Hibernate, 237
 - JDBC (Java Database Connectivity), 136
 - LineItem.hbm.xml mapping file, 237-238
 - LineItem.xml file, 184
 - pureQuery, 345
- loadCustomer service
 - Hibernate, 240
 - iBATIS, 188
 - JDBC (Java Database Connectivity), 137-138
 - pureQuery, 346-347
- openOrder service
 - Hibernate, 241-242
 - iBATIS, 189-190
 - JDBC (Java Database Connectivity), 138-139
 - pureQuery, 347-348
- Order class
 - JDBC (Java Database Connectivity), 136
 - Order.xml file, 183
 - pureQuery, 344-345
- overview, 53
- Product class
 - Hibernate, 238
 - JDBC (Java Database Connectivity), 137
 - Product.hbm.xml mapping file, 238-239
 - Product.xml file, 184
 - pureQuery, 345
- removeLineItem service
 - Hibernate, 244-245
 - iBATIS, 192-193
 - JDBC (Java Database Connectivity), 140-141
 - pureQuery, 349-350

- ResidentialCustomer class
 - Hibernate, 235
 - iBATIS, 181
 - JDBC (Java Database Connectivity), 136
 - OpenJPA, 298
 - pureQuery, 343
- service usage patterns, 62
- source code, downloading, 373
- submit() function, 245
- submitOrder service
 - iBATIS, 193-194
 - JDBC (Java Database Connectivity), 141-142
 - pureQuery, 350-351
- supporting technologies
 - Apache Derby, 372
 - DbUnit, 373
 - Eclipse, 372-373
 - JUnit, 373
 - overview, 372
- unit testing
 - Hibernate, 247
 - iBATIS, 195
 - JDBC (Java Database Connectivity), 142
 - pureQuery, 353
- Community-Driven Commercial Development (CD/CD), 315
- community-driven development, 26
- component maintainability, 41-42
- component packaging
 - comparison of persistence mechanisms, 362
 - Hibernate, 246-247
 - iBATIS, 194-195
 - JDBC (Java Database Connectivity), 142
 - OpenJPA, 306-307
 - pureQuery, 351-352
- composition, 71
- Concrete Table Inheritance pattern, 74-75
- configuration
 - common example. *See* common example
 - DataSources, 151
 - Session Factories, 203
 - “Configuring Data Access with Project Zero” (article), 317
- ConnectionPoolDataSource class, 117
- connections
 - comparison of persistence mechanisms, 359
 - Hibernate, 205
 - iBATIS, 151
 - JDBC (Java Database Connectivity), 7
 - obtaining with DataSources, 116
 - obtaining with DriverManager class, 115-116
 - pooling, 117-118
 - OpenJPA
 - EntityManager (EM), 255
 - EntityManagerFactory, 255-256
 - object states, 257-258
 - pureQuery
 - Data API, 320
 - Project Zero, 321
- consistency, 37
- constraints
 - common example database, 58
 - comparison of persistence mechanisms, 361
 - Hibernate, 227-228
 - iBATIS, 175-176
 - JDBC (Java Database Connectivity), 132
 - OpenJPA
 - order constraints, 291
 - unique constraints, 292
 - pureQuery, 338
- contained objects
 - comparison of persistence mechanisms, 361
 - Hibernate, 224-226
 - iBATIS, 172
 - JDBC (Java Database Connectivity), 131
 - OpenJPA, 284-288
 - overview, 72
 - pureQuery, 336
- container-managed EM instances, 256
- container-managed persistence contexts, 260
- containment. *See* contained objects

context, importance in evaluations, 88
 costs, TCO (total cost of ownership), 22-24
 create() method, 321
 createNativeQuery() method, 263
 createOrder() method, 323
 createQuery() method, 262
 creating data

- comparison of persistence mechanisms, 359
- Hibernate, 207-208
- iBATIS, 153-155
- JDBC (Java Database Connectivity), 121
- OpenJPA, 261
- pureQuery
 - Data API, 323-324
 - Project Zero, 324-325

 Criteria mechanism (Hibernate), 210
 curly braces ({}), 125
 custom cache, 229
 custom ResultHandlers, 331-332
 custom RowHandlers, 332
 customer order management application. *See*
 common example
 Customer.xml file, 181-183
 CustomerDoesNotExistException, 66
 CustomerLoad.xml file, 185-187
 CustomerOrderServicesTest object, 63-64
 Customers (common example), 55

- loading. *See* loadCustomer service
- relationship with Orders, 55-56

D

Data API (pureQuery)

- batch operations, 331
- connections, 320
- creating data, 323-324
- deleting data, 329
- exceptions, 333
- initialization, 318-319
- reading data, 326
- stored procedures, 330
- transactions, 321
- updating data, 327-328

Data Governance, 31-32
 data types

- iBATIS type mapping, 169-170
- JDBC (Java Database Connectivity) type mappings, 130-131

 database script, 379-380
 databases

- Apache Derby, 372-374
- common example
 - database constraints, 58
 - database normalization, 59-60
 - database schema, 57-58
- creating data
 - comparison of persistence mechanisms, 359
 - Hibernate, 207-208
 - iBATIS, 153-155
 - OpenJPA, 261
 - Project Zero, 324-325
 - pureQuery, 323-324
- database script, 379-380
- deleting data
 - comparison of persistence mechanisms, 359
 - Data API, 329
 - Hibernate, 212-214
 - iBATIS, 158-159
 - OpenJPA, 265-266
 - Project Zero, 329-330
- JDBC (Java Database Connectivity). *See*
 JDBC
- locking. *See* locking
- reading data
 - comparison of persistence mechanisms, 359
 - Data API, 326
 - Hibernate, 208-211
 - iBATIS, 155-157
 - OpenJPA, 261-263
 - Project Zero, 326-327
- updating data
 - comparison of persistence mechanisms, 359
 - Data API, 327-328

- Hibernate, 211-212
 - iBATIS, 157
 - OpenJPA, 263-265
 - Project Zero, 328-329
- DataRuntimeException, 333
- DataSources
- iBATIS configuration, 151
 - Hibernate configuration, 204
- DbKit (NeXT), 5
- DBTools.h++, 5
- DbUnit
- overview, 373
 - resolving for projects, 377
- decisions, documenting, 367-368
- delete statement, 124, 158-159
- delete() method, 212
- deleting data
- comparison of persistence mechanisms, 359
 - Hibernate, 212-214
 - iBATIS, 158-159
 - JDBC (Java Database Connectivity), 124
 - OpenJPA
 - remove() method, 265
 - via cascading, 265
 - via EJB-QL, 266
 - pureQuery
 - Data API, 329
 - Project Zero, 329-330
- Delphi, 5
- dependencies
- comparison of persistence mechanisms, 358
 - DI (dependency injection), 26
 - hardware/software, 24
 - Hibernate, 201, 384-385
 - iBATIS, 147-148, 381-382
 - JDBC (Java Database Connectivity), 113
 - OpenJPA, 251, 387-388
 - pureQuery, 316
- deployment
- Hibernate, 247
 - iBATIS, 195
- JDBC (Java Database Connectivity), 142
- OpenJPA, 308
- pureQuery, 354
- Derby
- Apache Derby Nature installation, 378
 - downloading, 374
 - Hibernate configuration file, 203-204
 - overview, 372
- derived attributes
- comparison of persistence mechanisms, 361
 - Hibernate, 228
 - iBATIS, 176-178
 - JDBC (Java Database Connectivity), 132-133
 - OpenJPA, 292-294
 - pureQuery, 338
- design
- agile approaches, 365-367
 - associations, 67-71
 - design models, 51-52
 - developerWorks links, 85
 - domain models
 - changing, 52
 - common example domain model, 53-56
 - compared to design models, 51-52
 - Domain Model pattern, 48-49
 - stakeholder involvement in, 51
 - UML (Unified Modeling Language), 49-50
 - when to use, 52
 - object-relational impedance mismatch, 66-67
 - object-relational mapping. *See* object-relational mapping
 - overview, 47
 - pattern languages, 48
 - references and further reading, 85
 - RUP (Rational Unified Process), 365-367
 - summary, 84
 - test first design, 53
- Detached state, 203

“Detect and Fix SQL Problems Inside Java Program” (article), 317

Developer roles, xxx

developerWorks

design links, 85

development links, 369

Hibernate links, 248

iBATIS links, 196-197

JDBC (Java Database Connectivity) links, 144

OpenJPA links, 309

pureQuery links, 355

development

community-driven development, 26

comparison of persistence

mechanisms, 362

tools, 31-32

Development Process section (evaluation questionnaire), 103-104

DI (dependency injection), 26

diagrams, entity-relationship (E-R), 57-58. *See also* domain models

dirty reads, 39

discriminator result mapping, 166-167

distributed caching, 179

Distributed Façade pattern, 83

documenting decisions, 367-368

domain models

changing, 52

common example domain model, 53-57

customer inheritance hierarchy, 54

illustration, 54

relationship between Customer and Order, 55-56

relationship between Order and LineItem, 56

relationship between Order and Product, 56-57

compared to design models, 51-52

Domain Model pattern, 48-49

stakeholder involvement in, 51

UML (Unified Modeling Language), 49-50

when to use, 52

downloading

Apache Derby, 374

common example source code, 373

Eclipse, 374

Milestone 1 Project Zero Eclipse Plug-in, 375

DriverManager class, 115-116

drivers (JDBC)

DriverManager class, 115-116

loading, 115

table, 112

durability, 37

dynamic SQL, 338-340

E

E-R (entity-relationship) diagrams, 57-58

eager fetching, 230

Eclipse

downloading, 374

Milestone 1 Project Zero Eclipse Plug-in, 375

overview, 372-373

education, availability of, 30

efficiency

caching, 40-41

isolation levels, 38-39

overview, 38

EHCACHE, 229

EJB (Enterprise Java Beans)

EJB 1.0, 8

EJB 2.0, 9-10

EJB 3

running, 393-400

transaction demarcation, 260

EJB-QL, 262-263

deleting data via, 266

updating via, 265

entity EJB components, 42-43

portability, 42-43

EM (EntityManager)

application-managed EM instances, 255

container-managed EM instances, 256

find() method, 261

- injecting, 257
 - persist() method, 261
 - remove() method, 265
 - transaction demarcation, 258-259
 - @Embeddable annotation, 285
 - embeddable IDs (OpenJPA), 279
 - @Embedded annotation, 286
 - encapsulation, 72-73
 - End User roles, xxx
 - end-to-end application architecture, xxx
 - endTransaction() method, 153
 - enterprise application tiers, 60-61
 - Enterprise Java Beans. *See* EJB
 - Enterprise JavaBeans, 5th Edition*, 252
 - Enterprise Object Framework (EOF), 5
 - enterprise quality solutions, 44-45
 - enterprise requirements as evaluation standard
 - amount of detail, 96
 - overview, 93
 - questions to ask, 95-96
 - relationship of project types to enterprise requirements, 94
 - entity EJB components, 42-43
 - Entity Manager (Hibernate), 199
 - entity-relationship (E-R) diagrams, 57-58
 - @EntityListener annotation, 294
 - EntityManager. *See* EM
 - EntityManagerFactory, 255-256
 - enumerations, 283
 - EOF (Enterprise Object Framework), 5
 - error handling
 - comparison of persistence mechanisms, 360
 - Hibernate, 215
 - iBATIS, 163
 - JDBC (Java Database Connectivity), 126
 - OpenJPA, 267-268
 - pureQuery, 333
 - evaluating persistence mechanisms
 - architectural overview, 358
 - background, 358
 - checklist of features, 90-91
 - context, 88
 - developerWorks links, 106
 - development process for common example, 362
 - documenting decisions, 367-368
 - enterprise requirements
 - amount of detail, 96
 - overview, 93
 - questions to ask, 95-96
 - relationship of project types to enterprise requirements, 94
 - evaluation questionnaire
 - architectural overview, 97-98
 - background, 97
 - development process, 103-104
 - ORM features supported, 100-102
 - overview, 97
 - programming model, 98-100
 - recording answers to, 105
 - tuning options, 102-103
 - when to use, 104-105
 - exhaustive approach, 87-88
 - independent standards, 89-90
 - ORM features supported, 360-361
 - overview, 87
 - practical standards, 91-92
 - programming models, 359-360
 - recording evaluations, 105
 - top ten key points, 105-106
 - trade-offs, 363-364
 - tuning options, 361-362
- evaluation questionnaire
 - architectural overview, 97-98
 - background, 97
 - development process, 103-104
 - ORM features supported, 100-102
 - overview, 97
 - programming model, 98-100
 - recording answers to, 105
 - tuning options, 102-103
 - when to use, 104-105
 - exceptions
 - comparison of persistence mechanisms, 360
 - Hibernate, 215

- iBatis framework, 163
- JDBC (Java Database Connectivity)
 - framework, 126
 - OpenJPA, 267-268
 - pureQuery, 333
- exclusive write locks, 39
- executeBatch() method, 160
- executing JPA queries, 262
- Executive roles, xxix
- exhaustive comparisons of persistence
 - frameworks, 87-88
- extended persistence contexts, 260
- extending frameworks
 - comparison of persistence
 - mechanisms, 360
 - Hibernate, 215
 - iBatis, 160-162
 - JDBC (Java Database Connectivity), 126
 - open source projects, 364-365
 - OpenJPA, 267
 - pureQuery, 331-332
- external transactions, 152
- @ExternalValues annotation, 284

F

- FetchGroups feature (OpenJPA), 296
- files. *See specific files*
- find() method, 261
- finding entity instances, 261
- First Normal Form (1NF), 59
- flush() method, 215
- Foreign-Key Mappings, 83
- four-way join example, 134
- Fowler, Martin, 48
- function-centric agile approach, 20-21
- functions. *See specific functions*

G

- Gamma, Erich, 373
- General Public License (GPL), 28-29
- GeneratedValue annotation, 275
- GenerationType.AUTO property
 - (GeneratedValue annotation), 275

- GenerationType.IDENTITY property
 - (GeneratedValue annotation), 275
- GenerationType.SEQUENCE property
 - (GeneratedValue annotation), 275
- GenerationType.TABLE property
 - (GeneratedValue annotation), 275
- get() method, 208
- getConnection() method, 117
- getErrorCode() method, 126
- getMessage() method, 126
- getPooledConnection() method, 117
- getSQLState() method, 126
- Getting Started with JDBC 4 Using Apache
 - Derby (tutorial), 114
- getTransaction() method, 258
- GPL (General Public License), 28-29
- grid-based caches, 41
- Groovy, 315, 318, 343. *See also* Project Zero
- GStrings, 324-325, 328
- Guid generator (Hibernate), 222

H

- handleRow() function, 176
- hardware dependencies, 24
- Hegel, Friedrich, 4
- Hibernate, 10-11
 - Annotations, 199
 - architecture
 - dependencies, 201
 - licenses, 201
 - overview, 200
 - platforms, 201
 - references and further reading,
 - 201-202, 248
 - standards, 200-201
 - vendors, 201
 - batch operations, 215
 - connections, 205
 - Core, 199
 - creating data, 207-208
 - Criteria mechanism, 210
 - definition, 95
 - deleting data, 212-214

- dependencies, resolving, 384-385
 - developerWorks links, 248
 - development process for common example
 - AbstractCustomer class, 234
 - AbstractCustomer.hbm.xml mapping file, 235-237
 - addLineItem service, 242-244
 - bootstrapping, 240
 - BusinessCustomer class, 234
 - component packaging, 246-247
 - deployment, 247
 - LineItem class, 237
 - LineItem.hbm.xml mapping file, 237-238
 - loadCustomer service, 240
 - openOrder service, 241-242
 - overview, 233
 - Product class, 238
 - Product.hbm.xml mapping file, 238-239
 - removeLineItem service, 244-245
 - ResidentialCustomer class, 235
 - submit() function, 245
 - unit testing, 247
 - Entity Manager, 199
 - error handling, 215
 - extending framework, 215
 - Hibernate JPA, 251
 - Hibernate Query Language (HQL), 208-209
 - Hibernate Reference Guide, 202
 - Hibernate Wiki, 202
 - history, 200
 - initialization, 203-205
 - configuration file for DataSource, 204
 - configuration file for Derby, 203-204
 - configuration file for JTA, 206
 - hibernate.cfg.xml file, 204
 - HibernateUtil class, 204
 - Session Factory setup, 203
 - ORM features supported, 215
 - attributes, 223-224
 - constraints, 227-228
 - contained objects, 224-226
 - derived attributes, 228
 - inheritance, 217-219
 - keys, 219-223
 - objects, 216-217
 - relationships, 226-227
 - overview, 199-200
 - POJO states, 203
 - reading data, 208-211
 - running, 385-386
 - stored procedures, 214-215
 - strengths and weaknesses, 363
 - transactions
 - JTA (Java Transaction API), 206-207
 - Transaction API, 205-206
 - tuning options
 - caching, 229
 - loading related objects, 229-232
 - locking, 232-233
 - query optimization, 229
 - type of framework, 200
 - updating data, 211-212
- Hibernate JPA, 251
 - Hibernate Query Language (HQL), 208-209
 - Hibernate Reference Guide, 202
 - “Hibernate simplifies inheritance mapping” (article), 202
 - Hibernate Wiki, 202
 - hibernate.cfg.xml file, 203-204, 246
 - HibernateException, 215
 - HibernateUtil class, 204
 - high-level requirements
 - availability of education and mentors, 30
 - availability of skilled practitioners, 29-30
 - build versus buy, 32
 - developerWorks links, 45-46
 - development and administration tools, 31-32
 - function-centric agile approach, 20-21
 - hardware and software dependencies, 24
 - intellectual property considerations, 28-29
 - licenses, 28
 - measurable software quality characteristics
 - efficiency, 38-41
 - functionality, 34-36

- interoperability, 44-45
- maintainability, 41-42
- overview, 33
- portability, 42-44
- reliability, 36-37
- usability, 37-38
- open-source and community-driven
 - activities, 26
 - overview, 19
 - phase-centric waterfall approach, 20-21
 - references and further reading, 46
 - standards supported, 25
 - TCO (total cost of ownership), 22-24
 - technology evaluation workshops, 21-22
 - vendors and support, 27
- Hilo generator (Hibernate), 222
- history
 - of Hibernate, 200
 - of iBATIS, 146
 - of JDBC (Java Database Connectivity), 110
 - of object-relational mapping
 - Delphi, 5
 - EJB (Enterprise Java Beans), 8-10
 - Hibernate, 10-11
 - iBATIS, 11
 - IBM ObjectExtender, 6
 - information as a service, 14
 - JDBC (Java Database Connectivity), 7-8
 - JDO (Java Data Objects), 13
 - JPA (Java Persistence Architecture), 13
 - NeXT DbKit, 5
 - object-relational impedance
 - mismatch, 4, 66-67
 - ODMG (Object Data Management Group), 12-13
 - overview, 3-4
 - ProjectZero, 15
 - pureQuery, 15
 - references and further reading, 17
 - Rogue Wave DBTools.h++, 5
 - timeline, 15-16
 - TopLink for Java, 8
 - TopLink for Smalltalk, 6
 - VisualAge Persistence Builder, 8-9
 - of OpenJPA, 250
 - of pureQuery, 312-313
 - HQL (Hibernate Query Language), 208-209

I

 - iBATIS, 11
 - architecture
 - dependencies, 147-148
 - licenses, 148
 - overview, 146
 - platforms, 147
 - resources, 148-149
 - vendors, 148
 - batch operations, 160
 - CacheController interface, 179
 - connections, 151
 - creating data, 153-155
 - definition, 90
 - deleting data, 158-159
 - dependencies, resolving, 381-382
 - developerWorks links, 196-197
 - development process for common
 - example
 - AbstractCustomer class, 180
 - addLineItem service, 190-192
 - component packaging, 194-195
 - Customer.xml file, 181-183
 - CustomerLoad.xml file, 185-187
 - deployment, 195
 - LineItem.xml file, 184
 - loadCustomer service, 188
 - openOrder service, 189-190
 - Order.xml file, 183
 - Product.xml file, 184
 - removeLineItem service, 192-193
 - ResidentialCustomer class, 181
 - submitOrder service, 193-194
 - unit testing, 195
 - error handling, 163
 - extending framework, 160-162
 - history, 146
 - iBATIS IN ACTION, 149
 - iBATIS Tutorial, 149
 - initialization, 150-151

- JUnit, running, 383
- ORM features supported
 - attributes, 169-172
 - constraints, 175-176
 - contained objects, 172
 - derived attributes, 176-178
 - discriminator result mapping, 166-167
 - inheritance, 164-166
 - keys, 167-169
 - objects, 163-164
 - overview, 163
 - relationships, 173-175
 - type mapping, 169-170
- overview, 145
- reading data, 155-157
- references and further reading, 197
- stored procedures, 159
- strengths and weaknesses, 363
- transactions, 152-153
- tuning options
 - distributed caching, 179
 - loading related objects, 179-180
 - locking, 180
 - query optimization, 178
 - single-JVM caching, 178-179
- type of framework, 145
- updating data, 157
- iBatis Tutorial, 149
- IBM
 - ObjectExtender, 6
 - ProjectZero, 15
 - pureQuery. *See* pureQuery
- IBM Software Services for WebSphere (ISSW), xxx, 21
- @Id annotation, 335-337
- Identity generator (Hibernate), 222
- identity of objects, 77-78
- IDs (OpenJPA)
 - application managed identities, 277
 - embeddable IDs, 279
 - entities as IDs, 279
 - generating using Identity strategy, 276
 - generating using Table strategy, 276
- ID class, 278
- ID fields, 275
- IdClass annotation usage, 279
- importing Java SE applications, 375-376
- Improve persistence with Apache Derby and iBatis (tutorial), 149
- Increment generator (Hibernate), 222
- independent standards, establishing, 89-90
- information as a service, 14
- Informix® Software, Inc., 372
- inheritance
 - Class Table Inheritance pattern, 73-74
 - comparison of persistence mechanisms, 360
 - Concrete Table Inheritance pattern, 74-75
 - customer inheritance hierarchy (common example), 54
 - Hibernate, 217-219
 - iBatis, 164-166
 - Java code for new table inheritance, 165
 - Java code for superclass table inheritance, 164
 - SQLMap for subclass table inheritance, 166
 - SQLMap for superclass table inheritance, 164
 - SQLMaps for new table inheritance, 166
 - JDBC (Java Database Connectivity), 128-129
 - OpenJPA, 270-271
 - Joined strategy, 272-273
 - Single Table strategy, 271-272
 - Table-per-Class strategy, 273-275
 - overview, 73
 - pureQuery, 333-335
 - root-leaf inheritance, 128-129
 - Single Table Inheritance pattern, 75
 - union inheritance, 128
- initialization
 - comparison of persistence mechanisms, 359

- Hibernate, 203-205
 - configuration file for DataSource, 204
 - configuration file for Derby, 203-204
 - configuration file for JTA, 206
 - hibernate.cfg.xml file, 204
 - HibernateUtil class, 204
 - Session Factory setup, 203
 - iBATIS, 150-151
 - JDBC (Java Database Connectivity), 115-116
 - OpenJPA
 - Java EE persistence unit, 255
 - Java SE persistence unit, 254
 - pureQuery
 - Data API, 318-319
 - Project Zero, 319-320
 - injecting
 - EntityManager, 257
 - EntityManagerFactory, 256
 - insert statement, 121, 154
 - installing Apache Derby Nature, 378
 - Integrating OpenJPA with Application Servers*, 252
 - intellectual property considerations, 28-29
 - interfaces
 - CacheController, 179
 - Savepoint, 120
 - TypeHandlerCallback, 160
 - interoperability, 44-45
 - inTransaction() method, 322
 - isolation levels, 37-39, 119-120, 134, 153
 - ISSW (IBM Software Services for WebSphere), xxx, 21
 - Ivy, 352
- J**
- Java
 - EJB (Enterprise Java Beans)
 - EJB 1.0, 8
 - EJB 2.0, 9-10
 - EJB 3, 260, 393-400
 - EJB-QL, 262-266
 - entity EJB components, 42-43
 - portability, 42-43
 - Hibernate. *See* Hibernate
 - iBATIS, 11
 - Java 5.0 development kit, 374
 - Java Annotations, creating entities
 - with, 269
 - Java interface for common example, 61-62
 - Java SE applications, importing, 375-376
 - JDBC (Java Database Connectivity). *See* JDBC
 - JDO (Java Data Objects), 13
 - JET (Java Emitter Template) engine, 366
 - JNDI (Java Naming and Directory Interface API), 8, 151
 - JPA (Java Persistence Architecture), 13
 - JSE (Java Platform Standard Edition), 24, 110
 - JTA (Java Transaction API), 8
 - TopLink for Java, 8
 - VisualAge Persistence Builder, 8-9
 - Java Persistence with Hibernate*, 202, 252
 - JBoss TreeCache, 229
 - JDBC (Java Database Connectivity), 109
 - architecture
 - dependencies, 113
 - drivers, 112
 - JDBC 1.0, 7-8, 111
 - JDBC 2.0, 7-8, 111
 - JDBC 3.0, 111
 - JDBC 4.0, 111
 - licenses, 113
 - online resources, 113-114
 - overview, 110-111
 - vendors, 113
 - batch operations, 125
 - CachedRowSet objects, 123
 - connections, 318
 - obtaining with DataSources, 116
 - obtaining with DriverManager
 - class, 115-116
 - pooling, 117-118
 - creating data, 121
 - DataSources, 319
 - deleting data, 124
 - developerWorks links, 144

- development process for common example
 - AbstractCustomer class, 135
 - addLineItem service, 139-140
 - BusinessCustomer class, 136
 - component packaging, 142
 - deployment, 142
 - LineItem class, 136
 - loadCustomer service, 137-138
 - object definitions, 136
 - openOrder service, 138-139
 - Order class, 136
 - Product class, 137
 - removeLineItem service, 140-141
 - ResidentialCustomer class, 136
 - submitOrder service, 141-142
 - unit testing, 142
- drivers
 - DriverManager class, 115-116
 - loading, 115
 - table of, 112
- error handling, 126
- extending framework, 126
- history, 110
- initialization, 115-116
- JDBC Database Access Tutorial, 114
- JDBC Technotes website, 114
- JDBC unit test, running, 380
- ORM features supported
 - attributes, 130-131
 - constraints, 132
 - contained objects, 131
 - derived attributes, 132-133
 - inheritance, 128-129
 - keys, 129-130
 - objects, 127-128
 - overview, 127
 - relationships, 131-132
 - type mappings, 130-131
- overview, 109
- programming model components, 114-116
- reading data, 121-123
- references and further reading, 144
- ResultSets, 121-123
- Statements
 - Callable Statements, 118
 - PreparedStatement objects, 117
 - Statement objects, 117
- stored procedures, 124-125
- strengths and weaknesses, 363
- transactions
 - isolation levels, 119-120
 - managing explicitly, 118-119
 - Savepoint interface, 120
- tuning options
 - caching, 133
 - loading related objects, 133-134
 - locking, 134
 - query optimization, 133
 - type of framework, 109-110
 - updating data, 123-124
- JDBC API Tutorial and Reference, Third Edition*, 114
- JDBC Technotes website, 114
- JDO (Java Data Objects), 13
- JET (Java Emitter Template) engine, 366
- JNDI (Java Naming and Directory Interface API), 8, 151
- Jobs, Steve, 5
- @Join annotation, 337, 343
- join fetching, 231
- join transactions, 259
- Joined strategy (inheritance), 272-273
- Joines, Stacy, 53
- JoinResultHandler, 337-338
- joins
 - four-way join example, 134
 - join fetching, 231
 - join transactions, 259
- JPA (Java Persistence Architecture), 13
- JPetStore, 146
- JPQL, 262
- JSE (Java Platform Standard Edition), 24, 110
- JSONResultHandler, 331-332
- JTA (Java Transaction API), 8, 206-207
- JUnit, 373, 383

K

keys

- comparison of persistence mechanisms, 360
- Hibernate, 219-223
 - key generation schemes, 222-223
 - LineItem class, 220-221
 - LineItemId class, 220-221
 - simple integer key mapping, 219
- iBATIS, 167-169
- JDBC (Java Database Connectivity), 129-130
- OpenJPA, 275-280
 - application managed identity, 277
 - embeddable IDs, 279
 - entities as IDs, 279
 - GeneratedValue annotation, 275
 - generating IDs with Identity strategy, 276
 - generating IDs with Table strategy, 276
 - ID class, 278
 - ID field, 275
 - IdClass annotation usage, 279
 - pureQuery, 335-336

King, Gavin, 10

Kodo, 250

L

- layered approach (pureQuery), 314
- lazy fetching, 230
- lazy loading, 179-180
- legacy JDK support, 147
- Leveraging OpenJPA with WebSphere Application Server (tutorial), 252
- licenses, 28
 - comparison of persistence mechanisms, 358
 - Hibernate, 201
 - iBATIS, 148
 - JDBC (Java Database Connectivity), 113
 - OpenJPA, 251
 - pureQuery, 316

life cycle

OpenJPA, 253-254

pureQuery, 313-314

LineItem class, 220-221

Hibernate, 237

JDBC (Java Database Connectivity), 136

OpenJPA, 300

pureQuery, 345

LineItem.hbm.xml mapping file, 237-238

LineItem.xml file, 184

LineItemId class, 220-221, 300-301

LineItemRowHandler, 177

LineItems, 56

load() method, 208

loadCustomer service

calling, 65

Hibernate, 240

iBATIS, 188

JDBC (Java Database Connectivity), 137-138

OpenJPA, 302-303

pureQuery, 346-347

test case, 64-66

loading

customers. *See* loadCustomer service

JDBC drivers, 115

related objects, 133-134

comparison of persistence

mechanisms, 361

Hibernate, 229-232

iBATIS, 179-180

OpenJPA, 296

pureQuery, 340

@Lob annotation, 283

local transactions, 152

locking, 39

comparison of persistence

mechanisms, 362

Hibernate, 232-233

iBATIS, 180

JDBC (Java Database Connectivity), 134

OpenJPA, 296-297

pureQuery, 341

Locking Strategies for Database Access
(paper), 296
logging, 148
looking up EntityManagerFactory, 255-256

M

mailing lists, 202
maintainability, 41-42
Manager roles, xxix
many-to-many relationships (OpenJPA),
290-291
many-to-one relationships
 Hibernate, 226
 OpenJPA, 289
mapping
 entities to tables, 270
 objects. *See* object-relational mapping
max() function, 169
measurable software quality characteristics.
 See quality characteristics
meet-in-the-middle object-relational
 mapping, 82
mentors, 30
merge() method, 264
merging, updating via, 264-265
Metadata Mapping, 83
methods. *See* *specific methods*
Milestone 1 Project Zero Eclipse Plug-in, 375
models, domain. *See* domain models

N

native generator (Hibernate), 223
navigation, 78-79
NestedSQLException, 163
New/Transient state, 253
NeXT DbKit, 5
nextVal() function, 169
NFs (normal forms), 59
NodeletException, 163
normal forms (NFs), 59
normalization, 59-60
NOT NULL constraints, 58

O

O notation, 88
Object Data Management Group (ODMG),
12-13
The Object People, 6
object-relational impedance mismatch,
4, 66-67
object-relational mapping
 associations, 67-70
 bottom-up approach, 81-82
 composition, 71
 containment, 72
 Distributed Façade pattern, 83
 encapsulation, 72-73
 Foreign-Key Mappings, 83
 history
 Delphi, 5
 EJB (Enterprise Java Beans), 8-10
 Hibernate, 10-11
 iBATIS, 11
 IBM ObjectExtender, 6
 information as a service, 14
 JDBC (Java Database Connectivity), 7-8
 JDO (Java Data Objects), 13
 JPA (Java Persistence Architecture), 13
 NeXT DbKit, 5
 object-relational impedance
 mismatch, 4, 66-67
 ODMG (Object Data Management
 Group), 12-13
 overview, 3-4
 ProjectZero, 15
 pureQuery, 15
 references and further reading, 17
 Rogue Wave DBTools.h++, 5
 timeline, 15-16
 TopLink for Java, 8
 TopLink for Smalltalk, 6
 VisualAge Persistence Builder, 8-9

- inheritance
 - Class Table Inheritance pattern, 73-74
 - Concrete Table Inheritance pattern, 74-75
 - overview, 73
 - Single Table Inheritance pattern, 75
- meet-in-the-middle approach, 82
- Metadata Mapping, 83
- object identity, 77-78
- object navigation, 78-79
- polymorphism, 76
- top-down approach, 80-81
- Unit of Work pattern, 83
- Object Technology International, 6
- ObjectExtender, 6
- objects. *See specific objects*
- Occam's Razor, 96
- ODMG (Object Data Management Group), 12-13
- one-to-many relationships
 - iBATIS, 174-175
 - OpenJPA, 289-290
- one-to-N relationships, 226
- one-to-one relationships
 - Hibernate, 226
 - iBATIS, 173-174
 - OpenJPA, 288
- open source projects, extending, 364-365
- open-source software, 26
- opening orders. *See* openOrder service
- OpenJPA
 - architecture
 - dependencies, 251
 - licenses, 251
 - platforms, 251
 - references and further reading, 251-252, 309-310
 - standards, 250
 - vendors, 251
 - batch operations, 266
 - bytecode enhancement, 389-390
 - connections
 - EntityManager (EM), 255
 - EntityManagerFactory, 255-256
 - object states, 257-258
 - creating data, 261
 - deleting data
 - remove() method, 265
 - via cascading, 265
 - via EJB-QL, 266
 - dependencies, resolving, 387-388
 - developerWorks links, 309
 - development process for common example
 - AbstractCustomer class, 297-298
 - addLineItem service, 303-305
 - BusinessCustomer class, 298-299
 - component packaging, 306-307
 - deployment, 308
 - LineItem class, 300
 - LineItemId class, 300-301
 - loadCustomer service, 302-303
 - openOrder service, 303
 - Order object, 299
 - overview, 297
 - Product class, 301
 - removeLineItem service, 305
 - ResidentialCustomer class, 298
 - submitOrder service, 305-306
 - unit testing, 307
 - error handling, 267-268
 - extending framework of, 267
 - history, 250
 - initialization
 - Java EE persistence unit, 255
 - Java SE persistence unit, 254
 - life cycle management, 253-254
 - ORM features supported
 - attributes, 280-284
 - constraints, 292
 - contained objects, 284-288
 - derived attributes, 292-294
 - inheritance, 270-275

- keys, 275-280
- objects, 269-270
- overview, 268
- relationships, 288-292
- overview, 249-250
- reading data, 261-263
- running, 391
- stored procedures, 266
- strengths and weaknesses, 363
- transactions, 258-261
 - EJB 3 transaction demarcation, 260
 - Entity Manager transaction demarcation, 258-259
 - extended persistence contexts, 260
 - join transactions, 259
 - savepoints, 260-261
- tuning options
 - caching, 294-295
 - loading related objects, 296
 - locking, 296-297
 - query optimizations, 294
- type of framework, 250
- updating data, 263-265
 - persistent entities, 263
 - related entities, 263
 - updating via EJB-QL, 265
 - updating via merging, 264-265
- OpenJPA Manuals, 252
- openOrder service
 - Hibernate, 241-242
 - iBATIS, 189-190
 - JDBC (Java Database Connectivity), 138-139
 - OpenJPA, 303
 - pureQuery, 347-348
- OpenSymphony cache (OSCache), 148, 229
- Operators roles, xxx
- optimistic locking, 232-233
- optimization. *See* performance tuning
- Order class
 - JDBC (Java Database Connectivity), 136
 - iBATIS, 163
 - OpenJPA, 299
 - pureQuery, 344-345
- order constraints, 291
- Order.xml file, 183
- Orders (common example), 55
 - opening. *See* openOrder service, 241
 - relationship with Customers, 55-56
 - relationship with LineItem, 56
 - relationship with Product, 56-57
 - submitting, 245
- ORM
 - common example. *See* common example
 - comparison of persistence mechanism support, 360-361
 - domain models
 - changing, 52
 - common example domain model, 53-56
 - compared to design models, 51-52
 - Domain Model pattern, 48-49
 - stakeholder involvement in, 51
 - UML (Unified Modeling Language), 49-50
 - when to use, 52
 - Hibernate support for ORM, 215
 - attributes, 223-224
 - constraints, 227-228
 - contained objects, 224-226
 - derived attributes, 228
 - inheritance, 217-219
 - keys, 219-223
 - objects, 216-217
 - relationships, 226-227
 - iBATIS support for ORM
 - attributes, 169-172
 - constraints, 175-176
 - contained objects, 172
 - derived attributes, 176-178
 - discriminator result mapping, 166-167
 - inheritance, 164-166
 - keys, 167-169
 - objects, 163-164
 - overview, 163
 - relationships, 173-175
 - JDBC support for ORM
 - attributes, 130-131

- constraints, 132
 - contained objects, 131
 - derived attributes, 132-133
 - inheritance, 128-129
 - keys, 129-130
 - objects, 127-128
 - overview, 127
 - relationships, 131-132
 - OpenJPA support for ORM
 - attributes, 280-284
 - constraints, 292
 - contained objects, 284-288
 - derived attributes, 292-294
 - inheritance, 270-275
 - keys, 275-280
 - objects, 269-270
 - overview, 268
 - relationships, 288-292
 - pattern languages, 48
 - pureQuery support for ORM
 - attributes, 336
 - constraints, 338
 - contained objects, 336
 - derived attributes, 338
 - inheritance, 333-335
 - keys, 335-336
 - objects, 333
 - relationships, 337-338
 - ORM Features Supported section (evaluation questionnaire), 100-102
 - OSCache, 148
 - “Overview of pureQuery Tools” (article), 316
- P**
- packaging
 - comparison of persistence mechanisms, 362
 - Hibernate, 246-247
 - iBATIS, 194-195
 - JDBC (Java Database Connectivity), 142
 - OpenJPA, 306-307
 - pureQuery, 351-352
 - parameterMap statement, 154
 - pattern languages, 48
 - Pattern Languages of Program Design*, 48
 - patterns
 - Class Table Inheritance pattern, 73-74
 - Concrete Table Inheritance pattern, 74-75
 - Distributed Façade pattern, 83
 - Domain Model pattern, 48-49
 - Domain Model pattern. *See* domain models
 - pattern languages, 48
 - Single Table Inheritance pattern, 75
 - Unit of Work pattern, 83
 - Patterns: Elements of Reusable Object-Oriented Software*, 210
 - Patterns of Enterprise Application Architecture*, 48
 - PCs (persistence contexts), 255, 260
 - performance tuning
 - comparison of persistence mechanisms, 361-362
 - Hibernate
 - caching, 229
 - loading related objects, 229-232
 - locking, 232-233
 - query optimization, 229
 - iBATIS
 - distributed caching, 179
 - loading related objects, 179-180
 - locking, 180
 - query optimization, 178
 - single-JVM caching, 178-179
 - JDBC (Java Database Connectivity)
 - caching, 133
 - loading related objects, 133-134
 - locking, 134
 - query optimization, 133
 - OpenJPA
 - caching, 294-295
 - locking, 296-297
 - query optimizations, 294
 - pureQuery
 - caching, 340
 - loading related objects, 340

- locking, 341
- overview, 338
- query optimizations, 338-340
- Perpetual Beta, 41
- persist() method, 261
- persistence contexts, 255, 260
- persistence mechanisms
 - evaluating
 - architectural overview, 358
 - background, 358
 - checklist of features, 90-91
 - context, 88
 - developerWorks links, 106
 - development process for common example, 362
 - documenting decisions, 367-368
 - enterprise requirements, 93-96
 - evaluation questionnaire, 97-105
 - exhaustive approach, 87-88
 - independent standards, 89-90
 - ORM features supported, 360-361
 - overview, 87
 - practical standards, 91-92
 - programming models, 359-360
 - recording evaluations, 105
 - top ten key points, 105-106
 - trade-offs, 363-364
 - tuning options, 361-362
- Hibernate. *See* Hibernate
- history
 - Delphi, 5
 - EJB (Enterprise Java Bean), 8-10
 - Hibernate, 10-11
 - iBATIS, 11
 - IBM ObjectExtender, 6
 - information as a service, 14
 - JDBC (Java Database Connectivity), 7-8
 - JDO (Java Data Objects), 13
 - JPA (Java Persistence Architecture), 13
 - NeXT DbKit, 5
 - object-relational impedance
 - mismatch, 4, 66-67
 - ODMG (Object Data Management Group), 12-13
 - overview, 3-4
 - ProjectZero, 15
 - pureQuery, 15
 - references and further reading, 17
 - Rogue Wave DBTools.h++, 5
 - timeline, 15-16
 - TopLink for Java, 8
 - TopLink for Smalltalk, 6
 - VisualAge Persistence Builder, 8-9
 - iBATIS. *See* iBATIS
 - JDBC (Java Database Connectivity). *See* JDBC
- persistence.xml file, 254
- Persistent state, 203
- phantom reads, 39
- phase-centric waterfall approach, 20-21
- Plain Old Java Objects (POJOs), 203, 253
- platforms
 - Hibernate, 201
 - iBATIS, 147
 - OpenJPA, 251
 - portability
 - entity EJB components, 42-43
 - requirements for persistence frameworks, 43-44
 - session EJB components, 42
 - pureQuery, 316
- plug-ins, Milestone 1 Project Zero Eclipse Plug-in, 375
- POJOs (Plain Old Java Objects), 203, 253
- polymorphism, 76
- PooledConnection class, 117
- pooling connections (JDBC), 117-118
- portability
 - entity EJB components, 42-43
 - requirements for persistence frameworks, 43-44
 - session EJB components, 42
- @PostLoad annotation, 292-293
- @PostPersist annotation, 293
- @PostRemove annotation, 294
- @PostUpdate annotation, 293
- practical standards, establishing, 91-92
- PreparedStatement objects, 117

- @PrePersist annotation, 293
- @PreRemove annotation, 294
- @PreUpdate annotation, 293
- procedures, stored. *See* stored procedures
- processes, 34-36
- Product class
 - Hibernate, 238
 - JDBC (Java Database Connectivity), 137
 - OpenJPA, 301
 - pureQuery, 345
 - relationship with Orders, 56-57
- Product.hbm.xml mapping file, 238-239
- Product.xml file, 184
- Programming Model section (evaluation questionnaire), 98-100
- programming models, 359-360
- Project Zero. *See also* pureQuery
 - batch operations, 331
 - caching, 340
 - connections, 321
 - creating data, 324-325
 - deleting data, 329-330
 - deployment, 354
 - goals, 315-316
 - initialization, 319-320
 - keys, 336
 - module layout, 351-352
 - overview, 15, 315
 - reading data, 326-327
 - running, 391-392
 - stored procedures, 330
 - transactions, 321-322
 - unit testing, 353
 - updating data, 328-329
 - Zero Resource Model, 354-355
- pureQuery. *See also* Project Zero
 - application development patterns, 317-318
 - architecture
 - dependencies, 316
 - layered approach, 314
 - licenses, 316
 - life cycle, 313-314
 - platforms, 316
 - references and further reading, 316-317, 356
 - standards, 315-316
 - vendors, 316
 - batch operations, 331
 - connections
 - Data API, 320
 - Project Zero, 321
 - creating data
 - Data API, 323-324
 - Project Zero, 324-325
 - deleting data
 - Data API, 329
 - Project Zero, 329-330
 - developerWorks links, 355
 - development process for common example
 - AbstractCustomer class, 343
 - addLineItem service, 348-349
 - BusinessCustomer class, 344
 - component packaging, 351-352
 - deployment, 354
 - LineItem class, 345
 - loadCustomer service, 346-347
 - openOrder service, 347-348
 - Order class, 344-345
 - overview, 341-343
 - Product object, 345
 - removeLineItem service, 349-350
 - ResidentialCustomer class, 343
 - submitOrder service, 350-351
 - unit testing, 353
 - error handling, 333
 - extending framework, 331-332
 - history, 312-313
 - initialization
 - Data API, 318-319
 - Project Zero, 319-320
 - ORM features supported
 - attributes, 336
 - constraints, 338
 - contained objects, 336
 - derived attributes, 338
 - inheritance, 333-335

- keys, 335-336
- objects, 333
- relationships, 337-338
- overview, 15, 311-312
- reading data
 - Data API, 326
 - Project Zero, 326-327
- stored procedures, 330
- strengths and weaknesses, 363
- transactions
 - Data API, 321
 - Project Zero, 321-322
- tuning options
 - caching, 340
 - loading related objects, 340
 - locking, 341
 - overview, 338
 - query optimizations, 338-340
- type of framework, 312
- updating data
 - Data API, 327-328
 - Project Zero, 328-329

PWTE.zip file, 373

Q

- quality characteristics (software)
 - efficiency
 - caching, 40-41
 - isolation levels, 38-39
 - overview, 38
 - functionality, 34-36
 - interoperability, 44-45
 - maintainability, 41-42
 - overview, 33
 - portability
 - entity EJB components, 42-43
 - requirements for persistence frameworks, 43-44
 - session EJB components, 42
 - reliability, 36-37
 - usability, 37-38
- queries
 - HQL (Hibernate Query Language), 209
 - OpenJPA, 262-263

- optimization
 - comparison of persistence mechanisms, 361
 - Hibernate, 229
 - iBATIS, 178
 - OpenJPA, 294
 - pureQuery, 338-340
 - SQL. *See* SQL
- queryArray() method, 326
- queryFirst() method, 326
- queryForObject method, 156
- queryList() method, 326
- questionnaire. *See* evaluation questionnaire

R

- Rational Unified Process (RUP), 365-367
- read committed isolation level, 39
- read locks, 39
- read uncommitted isolation level, 39
- reading data
 - comparison of persistence mechanisms, 359
 - dirty reads, 39
 - Hibernate, 208-211
 - iBATIS, 155-157
 - JDBC (Java Database Connectivity), 121-123
 - OpenJPA, 261-263
 - phantom reads, 39
 - pureQuery
 - Data API, 326
 - Project Zero, 326-327
 - repeatable reads, 39
- recording evaluations, 105
- related entities, updating, 263
- related objects, loading, 133-134
 - comparison of persistence mechanisms, 361
 - Hibernate, 229-232
 - iBATIS, 179-180
 - OpenJPA, 296
 - pureQuery, 340

relationships

- aggregations, 56
 - bidirectional relationships, 55
 - common example
 - relationship between Customer and Order, 55-56
 - relationship between Order and LineItem, 56
 - relationship between Order and Product, 56-57
 - comparison of persistence mechanisms, 361
 - Hibernate, 226-227
 - iBATIS, 173-175
 - JDBC (Java Database Connectivity), 131-132
 - OpenJPA, 288-292
 - many-to-many relationships, 290-291
 - many-to-one relationships, 289
 - one-to-many relationships, 289-290
 - one-to-one relationships, 288
 - pureQuery, 337-338
 - unidirectional relationships, 56
- releaseSavepoint() method, 120
- reliability, 36-37
- remove() method, 265
- removeLineItem service
 - Hibernate, 244-245
 - iBATIS, 192-193
 - JDBC (Java Database Connectivity), 140-141
 - OpenJPA, 305
 - pureQuery, 349-350
- removeLineItem() method, 329-330
- repeatable read isolation level, 39
- repeatable reads, 39
- requests, 36-37
- ResidentialCustomer class
 - Hibernate, 235
 - iBATIS, 181
 - JDBC (Java Database Connectivity), 136
 - OpenJPA, 298
 - pureQuery, 343

ResidentialCustomer object, 153-154

resolving

- DbUnit, 377
 - Hibernate dependencies, 384-385
 - iBATIS dependencies, 381-382
 - OpenJPA dependencies, 387-388
- resources (JDBC), 113-114

RESTful applications in an SOA: Part 2 (tutorial), 317

ResultHandlers

- custom ResultHandlers, 331-332
- JoinResultHandler, 337-338

ResultSets, 7, 121-123

retrieving data. *See* reading data

Rogue Wave DBTools.h++, 5

rollback() method, 120

root-leaf inheritance, 128-129

row handlers

- AppendTimestampRowHandler, 162
- custom RowHandlers, 332
- LineItemRowHandler, 177

running

database script, 379-380

EJB3, 393-400

Hibernate, 385-386

iBATIS JUnit, 383

OpenJPA, 391

Project Zero, 391-392

runtime resource efficiency

caching, 40-41

isolation levels, 38-39

overview, 38

RuntimeSQLException, 163

RUP (Rational Unified Process), 365-367

S

save() method, 206

Savepoint interface, 120

savepoints, 260-261

schema (common example database), 57-58

scripts, database, 379-380

SDN (Sun Developer Network), 113

Second Normal Form (2NF), 59

@SecondaryTable annotation, 286-288

- @Select annotation, 326
- select fetching, 231
- select generator (Hibernate), 223
- select statement, 121, 156
- seqhilo generator (Hibernate), 223
- sequence generator (Hibernate), 222
- serializable isolation level, 39
- servers, Apache Network Server, 379
- Service Oriented Architecture (SOA), 315
- services. *See also specific services*
 - bootstrapping, 240
 - enterprise application tiers, 60-61
 - information as a service, 14
 - Java interface, 61-62
 - usage patterns, 62
- session EJB components, 42
- Session Factories, 203
- setParameter() method, 161
- setQuantity() method, 211
- setSavePoint() method, 120
- setTransactionIsolation() method, 134
- setUp() method, 64
- setup. *See* configuration
- Single Table Inheritance pattern, 75, 271-272
- single-JVM caching, 178-179
- skilled practitioners, availability of, 29-30
- Smalltalk TopLink, 6
- SOA (Service Oriented Architecture), 315
- software
 - dependencies, 24
 - measurable software quality characteristics
 - efficiency, 38-41
 - functionality, 34-36
 - interoperability, 44-45
 - maintainability, 41-42
 - overview, 33
 - portability, 42-44
 - reliability, 36-37
 - usability, 37-38
 - open-source software, 26
- SolarMetric, 250
- SQL (Standard Query Language)
 - batch statements, 125, 160
 - delete statement, 124, 158-159
 - dynamic versus static, 338-340
 - insert statement, 121, 154
 - parameterMap statement, 154
 - select statement, 121, 156
 - SQLException object, 126
 - update statement, 123-124, 157
- sql-map-config file, 194-195
- SQLException object, 126
- SqlMap
 - active customer retrieve SqlMap XML, 156
 - customer retrieve SqlMap XML, 155
 - Customer.xml file, 181-183
 - CustomerLoad.xml file, 185-187
 - delete LineItem SqlMap XML, 158
 - LineItem.xml file, 184
 - order create SqlMap XML, 154
 - Order.xml file, 183
 - Product.xml file, 184
 - SQLMap for subclass table inheritance, 166
 - SQLMap for superclass table inheritance, 164
 - SQLMap for new table inheritance, 166
 - swap order stored procedure SqlMap XML, 159
 - update LineItem SqlMap XML, 157
- SqlMapConfig.xml file, 150
- SQLException, 163
- stakeholders
 - Architect, xxx
 - Business Executives, 20
 - Developer, xxx
 - End User, xxx
 - Executive, xxix
 - involvement in domain models, 51
 - Managers, xxix
 - Operators, xxx
 - Technical Leaders, 20
 - Tester, xxx
- Standard Query Language. *See* SQL
- standards
 - adherence to, 25
 - Hibernate, 200-201
 - independent standards, establishing, 89-90
 - JDBC (Java Database Connectivity), 111

- practical standards, establishing, 91-92
- pureQuery, 315-316
- startBatch() method, 160
- starting Apache Network Server, 379
- startTransaction() method, 152-153
- Statement objects, 117
- statements, 7
 - batch, 125, 160
 - Callable Statements, 118
 - delete, 124, 158-159
 - insert, 121, 154
 - parameterMap, 154
 - PreparedStatement objects, 117
 - select, 121, 156
 - Statement objects, 117
 - update, 123-124, 157
- states, 257-258
 - Detached, 203
 - New/Transient, 253
 - Persistent, 203
 - Transient, 203
- static SQL, 338-340
- StatusEnumTypeHandler, 171-172
- stored procedures
 - comparison of persistence mechanisms, 360
 - Hibernate, 214-215
 - iBATIS, 159
 - JDBC (Java Database Connectivity), 124-125
 - OpenJPA, 266
 - pureQuery, 330
- submit() function, 245
- submitOrder service
 - iBATIS, 193-194
 - JDBC (Java Database Connectivity), 141-142
 - OpenJPA, 305-306
 - pureQuery, 350-351
- subselect fetching, 231
- Sun Developer Network (SDN), 113
- supportsTransactionIsolationLevel() method, 119

- supportsTransactions() method, 119
- swap_customer_order stored procedure, 124
- SwarmCache, 229

T

- @Table annotation, 333
- Table-per-Class strategy (inheritance), 273-275
- tables, mapping entities to, 270
- TCO (total cost of ownership), 22-24
- Technical Leaders, 20
- technology evaluation workshops, 21-22
- @Temporal annotation, 283
- test first design, 53
- Test-Driven Development By Example*, 373
- Tester roles, xxx
- testing. *See* unit testing
- testLoadCustomer() method, 64-66
- Third Normal Form (3NF), 59
- top-down object-relational mapping, 80-81
- TopLink Essentials, 251
- TopLink for Java, 8
- TopLink for Smalltalk, 6
- TopLink JPA, 87
- toString() method, 324
- total cost of ownership (TCO), 22-24
- trade-offs of persistence mechanisms, 363-364
- Transaction API, 205-206
- Transaction Script approach, 5
- TransactionException, 163
- transactions, 118-120
 - ACID properties, 37
 - comparison of persistence mechanisms, 359
 - Hibernate, 205-207
 - JTA (Java Transaction API), 206-207
 - Transaction API, 205-206
 - iBATIS, 119-120, 152-153
 - isolation levels, 134, 153
 - managing explicitly, 118-119
 - OpenJPA
 - EJB 3 transaction demarcation, 260
 - Entity Manager transaction demarcation, 258-259
 - extended persistence contexts, 260

- join transactions, 259
 - savepoints, 260-261
 - pureQuery
 - Data API, 321
 - Project Zero, 321-322
 - reliability, 36-37
 - Savepoint interface, 120
 - TRANSACTION_NONE constant, 120
 - TRANSACTION_READ_COMMITTED
 - constant, 120
 - TRANSACTION_READ_UNCOMMITTED
 - constant, 120
 - TRANSACTION_REPEATABLE_READ
 - constant, 120
 - @Transient annotation, 281
 - Transient state, 203
 - troubleshooting common example setup, 400
 - tuning. *See* performance tuning
 - Tuning Options section (evaluation questionnaire), 102-103
 - Type 1 JDBC drivers, 112
 - Type 2 JDBC drivers, 112
 - Type 3 JDBC drivers, 112
 - Type 4 JDBC drivers, 112
 - TYPE_FORWARD_ONLY ResultSet, 122
 - type handlers
 - AppendTimestampRowHandler, 162
 - LineItemRowHandler, 177
 - StatusEnumTypeHandler, 171-172
 - TypeHandlerCallback interface, 160
 - YesNoBoolTypeHandler, 161
 - TypeHandlerCallback interface, 160
 - TYPE_SCROLL_INSENSITIVE ResultSet, 122
 - TYPE_SCROLL_SENSITIVE ResultSet, 122
- U**
- UML (Unified Modeling Language), 34, 49-50
 - “Understand the DB2 UDB JDBC Universal Driver” (article), 114
 - unidirectional relationships, 56
 - Unified Modeling Language (UML), 34, 49-50
 - union inheritance, 128
 - unique constraints, 292
 - Unit of Work pattern, 83
 - unit testing
 - CustomerOrderServicesTest object, 63-64
 - Hibernate, 247
 - iBATIS, 195
 - JDBC (Java Database Connectivity), 142
 - JDBC unit test, 380
 - loading customer test case, 64-66
 - OpenJPA, 307
 - overview, 62-63
 - pureQuery, 353
 - test case setup method, 64
 - @Update annotation, 323, 329
 - update statement, 123-124, 157
 - updateLineItem() method, 264
 - updateMany() method, 331
 - updating data
 - comparison of persistence mechanisms, 359
 - Hibernate, 211-212
 - iBATIS, 157
 - JDBC (Java Database Connectivity), 123-124
 - OpenJPA
 - persistent entities, 263
 - related entities, 263
 - updating via EJB-QL, 265
 - updating via merging, 264-265
 - pureQuery
 - Data API, 327-328
 - Project Zero, 328-329
 - usability, 37-38
 - use cases, 34
 - “Use Project Zero’s data access APIs to build a simple wiki” (article), 317
 - user sessions, 37-38
 - Using Hibernate to Persist Your Java Objects to IBM DB2 Universal Database*, 202
 - Using Spring and Hibernate with WebSphere Application Server*, 202
 - uuid.hex generator (Hibernate), 223
 - uuid-string generator string, 276

V

- valueOf() method, 161
- VAP (VisualAge Persistence Builder), 8-9
- vendors, 27, 358
 - Hibernate, 201
 - iBATIS, 148
 - JDBC (Java Database Connectivity), 113
 - OpenJPA, 251
 - pureQuery, 316
- @Version annotation, 296
- versions of JDBC (Java Database Connectivity), 111
- VisualAge Persistence Builder, 8-9

W-X-Y-Z

- waterfall approach, 20-21
- Web Extended SOA, 315
- write locks, 39

- XML, creating entities with, 269-270

- YesNoBoolTypeHandler, 161

- Zero Resource Model, 354-355
- Zero. *See* Project Zero
- zero.config file, 352-353