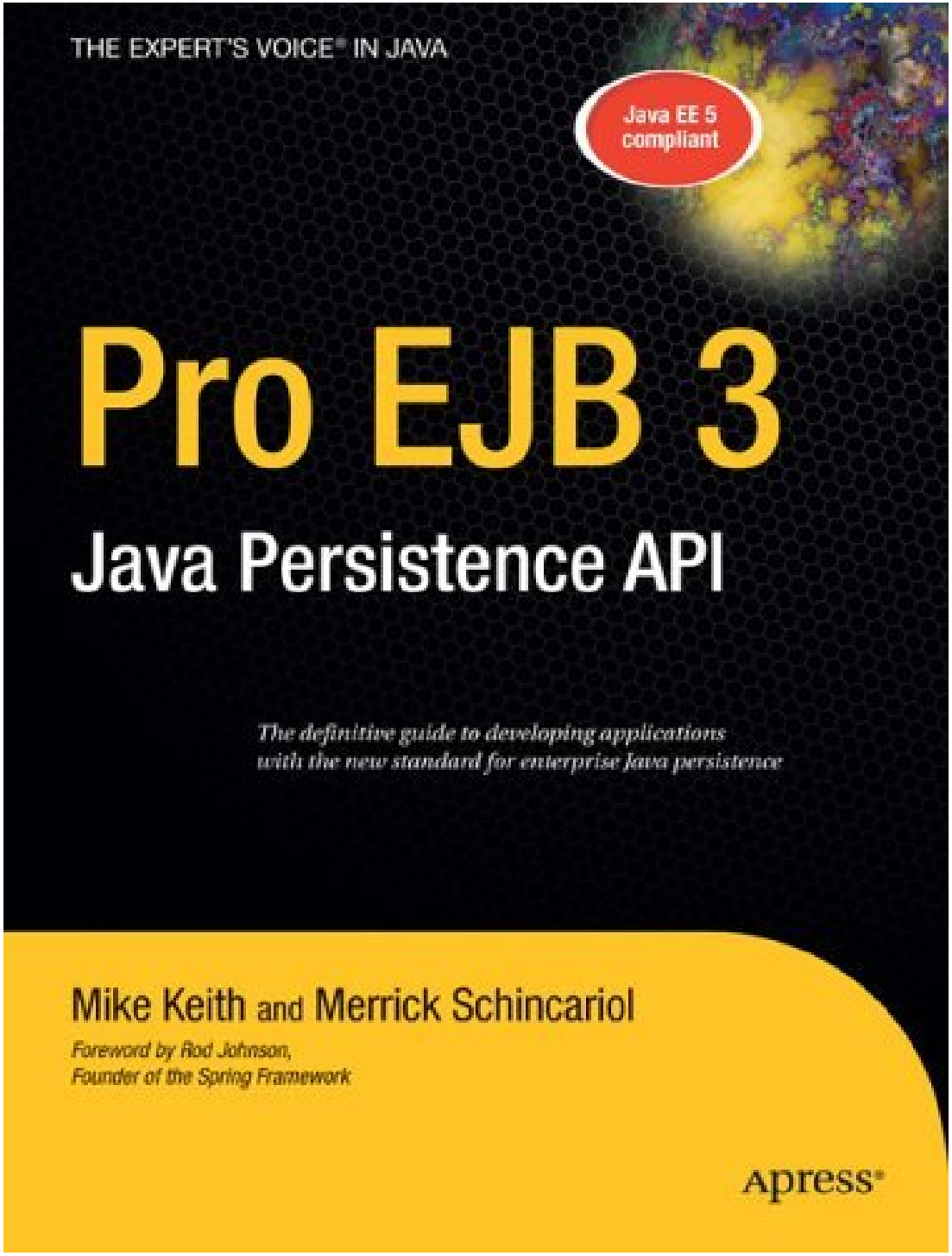

Pro EJB 3

Java Persistence API

Mike Keith

Merrick Schincariol

Apress®

Pro EJB 3: Java Persistence API

ISBN-13 (pbk): 978-1-59059-645-6

ISBN-10 (pbk): 1-59059-645-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Jason Haley, Huyen Nguyen, Shahid Shah
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Julie M. Smith
Copy Edit Manager: Nicole LeClerc
Copy Editor: Hastings Hart
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositors: Pat Christenson and Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

To the memory of my mother, who touched so many lives, and to my wife Darleen, who touches my life each and every day. —Mike

To my parents, for always supporting me in all of my adventures, and my wife Natalie, whose love and support kept me going to the end. —Merrick

Contents at a Glance

Contents

Foreword

I believe that the Java Persistence API (JPA) is the most important advance in the Java EE 5 platform revision. It offers a simple yet powerful standard for object-relational mapping (ORM). Leading persistence vendors have united to develop it, and developers should unite in adopting it.

Up to and including EJB 2.1, the persistence technology (entity beans) was the weakest part of the overall EJB specification—complex yet lacking in essential capabilities such as inheritance modelling, and unusable in isolation. JPA is now a fine model that can stand on its own and works in a range of environments. In future revisions, JPA's independence from its runtime environment is likely to become explicit, with JPA split into a separate specification from EJB. Thus JPA is relevant to *all* Java developers who work with relational databases, not merely those who work with the EJB component model overall.

Since the release of EJB 2.0 in 2001, enterprise Java has been revolutionized by a switch to a simpler, POJO-based programming model. This switch has been equally marked in the Fortune 500 as it has amongst developers of simple web applications. The focus is increasingly on the Java language itself rather than complex container contracts. Modern infrastructure can apply enterprise services to POJOs without their needing to be polluted by infrastructure concerns.

Unlike the former entity bean model, JPA offers the ability to persist POJOs. The JPA specification has been drawn from collective experience from more than 10 years, across products such as TopLink, Kodo, and Hibernate. POJO-based persistence offers many important benefits lacking in the former entity bean model: a simple, productive development experience; ease of testing; and the ability to build and persist true domain models capturing business concepts. The rise of POJO persistence will gradually change for the better how we design applications, enabling greater object orientation in place of the often procedural style traditionally associated with J2EE development.

Of course, ORM is not new, and many developers have been successfully working with ORM products for years. The importance of JPA lies in the fact that it offers users choices between implementations without subjecting them to a lowest common denominator. All leading ORM products will support the JPA in addition to their proprietary APIs (or other standards, such as JDO); users who choose to move to the JPA API will not be forced to switch persistence products but will gain greater choice in the future.

As leader of the Spring Framework open source project, I am particularly pleased that the leading JPA implementations are all open source. The Spring Framework 2.0 release integrates with JPA, and we aim to make it easy for Spring users to work with JPA in any environment. We are excited about working with the communities around the open source JPA implementations to help make the Spring/JPA experience still better.

Thus this is an important topic. Hopefully I have convinced you that you should read a book on JPA, but why should you read this one?

Mike Keith is ideally qualified to write such a book. He was co-lead of the EJB 3.0 specification, and he not only has intimate knowledge of the specification but also played a vital role in bringing it to completion. His experience with persistence engines dates back almost 15 years. He has been a key member of the TopLink team for five years, championing what would now

be called POJO persistence before it become fashionable, during the persistence Dark Ages of J2EE 1.3 and 1.4.

Most important, Mike and Merrick Schincariol have turned their experience into a clear, readable book providing a thorough introduction to JPA. While you won't be disappointed as you grow in your experience with JPA, and while you will find valuable content on more advanced questions and best practices, the book never feels bogged down in detail. The authors clearly convey the big picture of how you can use JPA effectively. The many code examples ensure that the discussion always comes back to practice, with examples that are directly relevant to developers.

I encourage you to browse for yourself and am confident you will my share my high opinion of *Pro EJB 3: Java Persistence API*.

Rod Johnson
Founder, Spring Framework
CEO, Interface21

About the Authors

MIKE KEITH is the co-specification lead of EJB 3.0 and a member of the Java EE 5 expert group. He holds a Master of Science degree in computing from Carleton University and has over 15 years of teaching, research, and practical experience in object persistence. He has implemented persistence systems for Fortune 100 corporations on a host of technologies, including relational and object databases, XML, directory services, and custom data formats. Since the fledgling EJB days he has worked on EJB implementations and integrations to multiple application servers. He has written various papers and articles and spoken at numerous conferences about EJB 3.0. He is currently employed at Oracle as a persistence architect.

MERRICK SCHINCARIOL is a senior engineer at Oracle and a reviewer of the EJB 3.0 specification. He has a Bachelor of Science degree in computer science from Lakehead University and has over seven years of experience in the industry. He spent some time consulting in the pre-Java enterprise and business intelligence fields before moving on to write Java and J2EE applications. His experience with large-scale systems and data warehouse design gave him a mature and practiced perspective on enterprise software, which later propelled him into doing EJB container implementation work. He was a lead engineer for Oracle's EJB 3.0 offering.

About the Technical Reviewers

JASON HALEY is a senior engineer at Oracle and a reviewer of the EJB 3.0 specification. He has a Bachelor of Science degree in computing from Carleton University and has been working with the EJB internals, and Java and J2EE applications for over seven years. He has done consulting and training but spent much of his time designing and implementing EJB container infrastructure both on the session and persistence sides. He has extensive experience with the inner workings of BEA WebLogic Server and Oracle Application Server, and has devised multiple session and persistence manager interfaces. He is a lead engineer for Oracle's EJB container.

HUYEN NGUYEN is quality assurance manager at Oracle in the Server Technologies group. He has a Bachelor of Applied Science degree in Systems Design Engineering from the University of Waterloo. In the field of object persistence, he has been working as an instructor and consultant for 11 years and a quality assurance manager for the past four years.

SHAHID N. SHAH is the Founder and CEO of Netspective Communications. Netspective is a software development firm that provides the tools and skills necessary for creating service-oriented systems using Java and .NET. He has recently built a large health-care informatics framework using EJB 3.0 and Java 5.0. Shahid has held the positions of VP of Technology, CTO, Chief Software Architect, and Lead Engineer at large enterprises for the past 15 years. Shahid's key technology expertise areas are service-oriented architectures, distributed object services, Java, J2EE, .NET, XML, UML, and object- and aspect-oriented software development. Shahid runs three successful blogs. At `http://shahid.shah.org` he writes about architecture issues, at `http://www.healthcareguy.com` he provides valuable insights on how to apply technology in health care, and at `http://www.hitsphere.com` he gives a glimpse of the health-care IT blogosphere as an aggregator. He can be reached at `shahid@shah.org`.

Acknowledgments

I want to thank all of the members of the expert group who contributed to the EJB 3.0 specification. Numerous conference calls, countless hours on the phone with Linda, Gavin, Patrick, and others, and bucket-loads of email produced a result that we all hope is worth the two years of our lives that we sacrificed for it.

I want to thank the four D's (Dennis Leung, Dan Lesage, Doug Clarke, and Donald Smith) for their support and friendship at various stages of the book. Thanks to Shahid for reviewing the early drafts of the chapters and to a host of other casual reviewers that looked at the occasional chapter. I especially owe huge thanks to three great friends: Jason for agreeing to tirelessly put all of the examples into code (and fix my bugs!), for reviewing, and even writing some of the early chapter drafts; Huyen for going above and beyond the call of duty by spending night after late night reviewing to meet a tight schedule; and of course Merrick for being great to work with and taking up the slack when I was out of commission or not able to keep up. Tony, Julie, Hastings, and Laura at Apress really helped out along the way and performed miracles getting this book to print. Last of all, and most important, my wife Darleen and my kids Cierra, Ariana, Jeremy, and Emma. I love them all without bounds. It is they who sacrificed the most for this book by being so very patient over months of having a distant husband and father slouched in a chair clacking away on a laptop.

Mike Keith

Writing a book involves many more people than will ever have their names printed on the cover. Over the last year I have been blessed with the support of many people, who offered advice, reviewed my work, and encouraged me along the way. First of all I'd like to thank Mike for giving me the opportunity to collaborate with him on this book. It was a true partnership from beginning to end. But it could not have been done without the loving support of my wife Natalie and the remarkable patience of my young son Anthony, who had to put up with a daddy who was often hidden away in his office writing. At Oracle, special thanks to Jason Haley for shouldering more than his fair share of senior engineering responsibilities while I worked on this project, and thanks for the support of Dennis Leung, Rob Campbell, and the entire EJB container team. At Apress, Julie Smith, Hastings Hart, and Laura Esterman pulled out all the stops to get things done on time. And finally, thanks to the many reviewers who looked over the drafts of this book. Huyen Nguyen and Jason Haley in particular were instrumental in helping us refine and make this book both accurate and readable.

Merrick Schincariol

Preface

The Java Persistence API has been a long time in coming, some might even say overdue. The arrival of an enterprise Java persistence standard based on a "POJO" development model fills a gap in the platform that has needed to be filled for a long time. The previous attempt missed the mark and advocated EJB entity beans that were awkward to develop and too heavy for many applications. It never reached the level of widespread adoption or general approval in many sectors of the industry. But in the absence of a standard, proprietary persistence products such as JBoss Hibernate and Oracle TopLink gained popularity in the industry and have been thriving. With the emergence of the Java Persistence API, developers can now create portable persistence code that will run on any compliant Java EE 5 server, as well as in a stand-alone JVM outside the server.

It could be argued that the result of waiting until the persistence market had matured was that a superior standard emerged based on product and user experience instead of theory and design. Contained in the Java Persistence API (abbreviated by some as "JPA") are the basic notions and interfaces that all persistence connoisseurs will recognize from their experience with existing products. This will make it easier for people who are already using these products to adopt the Java Persistence API while still allowing novice persistence developers to pick up the API and quickly learn it.

The specification was written for architects and developers alike, but it is still a specification. Few people enjoy sitting down with a tersely worded specification to find out how to use an API. It does not delve into the intricacies of applying the API, nor does it explain any of the peripheral issues that you may encounter during development. In this book we wanted to bring a more practical approach to the topic and highlight some of the usage patterns that we think are of value.

Our original intent was to write about the entire EJB 3.0 specification. We also wanted to produce a book that would fit in someone's laptop bag and not outweigh the laptop. It didn't take us long to realize that in order to provide adequate coverage of the topic and write a book that offered value to the average developer, we needed to focus on only half of the specification. Given our own persistence experience and the lack of existing outside knowledge of the Java Persistence API, the choice was an easy one.

Over the course of this book we will go into detail on all of the elements of the API. We will explain the concepts and illustrate their usage by providing practical examples of how to apply them in your own applications. We begin the journey with a quick tour of the API by creating a very simple application in the Java SE environment. We then move into the enterprise and provide an overview of the features in the EJB 3.0 and Java EE 5 standards that apply to persistence in enterprise applications.

Object-relational mapping is at the heart of storing object state in a relational database, and we go into detail on ORM technology and the mappings and features supported by the API. The `EntityManager` is the main interface used to interact with entities. Different aspects of using

the entity manager are explored, and we open the hood to expose the internals of the implementation to help you understand some of the important nuances. We also explore the queries that can be obtained from entity managers, and make distinctions between the different kinds of dynamic, static, or named queries that are available. We assess the query capabilities that may be accessed and present ideas about when the different kinds of queries should be used. Java Persistence Query Language is discussed in its entirety, with examples of all of its features.

Next we tackle some of the intermediate and advanced topics of ORM, such as inheritance, and show how to map different kinds of class hierarchies to a variety of data schemas. We also delve into the important subject of locking and explain how to best make use of locking strategies in your application. For those who like to use XML for metadata, we describe how XML mappings are specified and explain how they can be used to override annotations. The development life cycle is then completed by a discussion of how to configure the persistence unit and package it up in different categories of enterprise application components.

Much has been written about testing, and much is still being written. Another strength of the API is its ability to support unit testing and some of the other current testing methodologies and patterns that are being used today. We spend some time discussing some of the ways that you can test entities and the application logic that invokes them, both inside and outside the application server.

Finally, for those who are coming to the API from existing persistence systems, we devote some time to going over the migration issues. We offer some suggestions, through the use of some of the common design patterns, of ways to migrate different kinds of architected applications to use the Java Persistence API.

We hope that you enjoy both reading this book and learning how to use the Java Persistence API. We still couldn't fit everything that we wanted to fit in this book, but hopefully, like Indiana Jones, we "chose wisely" about what was important and what didn't need to be included. We welcome any suggestions for additional topics as well as any comments about the topics that we did include.

Who This Book Is For

We have written this book for everybody who wants to use persistence in enterprise and desktop applications. We do not assume that you have any experience with persistence products, although we do assume that you have some Java programming experience, as well as some exposure to the J2EE platform. Experience with the new Java EE 5 standard may be helpful but is certainly not required. Knowledge of previous versions of EJB is also not required.

A persistence API that maps objects and stores data in a relational database expects some amount of basic understanding of databases and SQL. In addition, since the API is implemented on top of Java Database Connectivity (JDBC) API that accesses the database, any knowledge of that API will also be an asset but is not absolutely needed.

About the Code Examples

Sometimes a single code example is worth more than the number of words that can fit in a chapter. We have tried to use inlined code examples when it is practical and when it suits the purpose. Although we tend to prefer learning from code rather than reading paragraphs of text, we find it frustrating when a code example goes on for pages, and by the time you reach the end

of it you have forgotten what it was you were trying to learn from it. We have attempted to alleviate the distraction of the nonrelevant code bits by using ellipses and eliding the parts that do not contribute to the point of the example. Our hope is that you will agree with us and think that it makes the examples more meaningful, not that the examples are only half-baked.

The API is somewhat flexible about the access modifier of persistent state in that it may be package, protected, or private. We have defined entity state to consistently be private in the examples to highlight how the state should be encapsulated within the entity. For the record, we are not dogmatic about state being private. We just happened to start out doing it that way and never bothered to change.

To ensure that the focus remains on understanding the technology and not puzzling over the sample domain, we have adopted the simplest and most prevalent domain model that we could think of, the tried and true Employee model. While being a bit on the dull side, we had to admit that it was ubiquitous to the point where it was a sure bet that virtually every developer on the planet would understand and be able to relate to it. It contains all of the necessary modeling variability (although admittedly we did have to stretch it a bit in some cases) that is needed to illustrate the concepts and practices of the API.

The examples that accompany the book have been implemented using the official Reference Implementation (RI) of the Java EE 5 application server and the Java Persistence API. The Java EE 5 RI is called "Glassfish" and is a fully featured open source application server that can be obtained and used under the Common Development and Distribution License (CDDL). The RI for the Java Persistence API is called "TopLink Essentials" and is an open source and freely available product derived from the commercially distributed Oracle TopLink enterprise data integration stack. GlassFish and TopLink Essentials can be obtained from the GlassFish project downloads link on java.net, but we recommend going to the persistence page at `http://glassfish.dev.java.net/javaee5/persistence`. TopLink Essentials can also be obtained from the TopLink page on the Oracle Technology Network at `http://www.oracle.com/technology/products/ias/toplink`.

The examples are available for download on the Apress website at `http://www.apress.com`. We recommend downloading them and poking around. The best way to learn the API is to try it out on your own, and taking an existing model and tweaking it is a great way to get started. The API is its own best selling point. Once you develop with it, you will see that it really does make persistence development much easier than it has ever been before!

Contacting Us

We can be contacted at `michael.keith@oracle.com` and `merrick.schincariol@oracle.com`.

CHAPTER 1

Introduction

The word *enterprise* is arguably one of the most overused terms in software development today. And yet, when someone states that they are developing an enterprise application, invariably a single word comes to mind: information. Enterprise applications are defined by their need to collect, transform and report on vast amounts of information. And, of course, that information does not simply exist in the ether. Storing and retrieving data is a multibillion dollar business, as evidenced by the burgeoning enterprise integration systems (EIS) and enterprise application integration (EAI) companies that have sprung up in recent years.

Many ways of persisting data have come and gone over the years, and no concept has had more staying power than the relational database. It turns out that the vast majority of the world's corporate data is now stored in relational databases. They are the starting point for every enterprise application with a lifespan that may continue long after the application has faded away.

Understanding the relational data is key to successful enterprise development. Developing applications to work well with database systems has become the primary business of software development. For Java in particular, part of its success can be attributed to the widespread adoption of the language for building enterprise database systems. From consumer web sites to automated gateways, Java applications are at the heart of enterprise data development.

Despite the success the Java platform has had in working with database systems, it still suffers from a problem. Moving data back and forth between a database system and the object model of a Java application is a lot harder than it needs to be. Java developers either seem to spend a lot of time converting row and column data into objects, or they find themselves tied to proprietary frameworks that try to hide the database from the developer.

Fortunately, a solution is finally at hand. Recently standardized and backed by both commercial and open source interests from across the spectrum, the Java Persistence API is set to have a major impact on the way we handle persistence within Java. For the first time, developers have a standard way of bridging the gap between object-oriented domain models and relational database systems.

Over the course of this book we will introduce the Java Persistence API and explore everything that it has to offer developers. Whether you are building client-server applications to collect form data in a Swing application or building a web site using the latest application framework, the Java Persistence API is a framework you can use to be more effective with persistence. One of its major strengths is that it can be slotted into whichever layer, tier, or framework that an application needs it to be in.

To set the stage for the Java Persistence API, this chapter first takes a step back to show where we've been and what problems we are trying to solve. From there we will look at the history of the specification and provide a high-level view of the value it brings to developers.

Java Support for Persistence

The Java platform is well supported for managing persistence to relational databases. From the earliest days of the platform, programming interfaces have existed to provide gateways into the database and even to abstract away much of the vendor-specific persistence requirements of business applications. In the next few sections we will look at the current set of Java standards for persistence and their role in enterprise applications.

JDBC

The second release of the Java platform ushered in the first major support for database persistence with the Java Database Connectivity specification, better known as JDBC. Offering a simple and portable abstraction of the proprietary client programming interfaces offered by database vendors, JDBC allows Java programs to fully interact with the database. This interaction is heavily reliant on SQL, offering developers the chance to write queries and data manipulation statements in the language of the database, but executed and processed using a simple Java programming model.

The irony of JDBC is that while the programming interfaces are portable, the SQL language is not. Despite many attempts to standardize the SQL language, it is still rare to write SQL of any complexity that will run unchanged on any two major database platforms. Even where the languages are the same, each database performs differently depending on the structure of the query, necessitating vendor-specific tuning in most cases.

There is also the issue of tight coupling between Java source and SQL text. Developers are constantly tempted by the lure of ready-to-run SQL queries either dynamically constructed at runtime or simply stored in variables or fields. This is a very effective programming model until the minute you realize that the application has to support a new database vendor and that it doesn't support the dialect of SQL you have been using.

Even with SQL text relegated to property files or other application metadata, there comes a point in working with JDBC where it not only feels wrong, but also becomes a cumbersome exercise to take tabular row and column data and continuously have to convert it back and forth into objects. The application has an object model—why does it have to be so hard to use with the database?

Enterprise JavaBeans

The first release of the Java 2 Enterprise Edition (J2EE) platform introduced a new solution for Java persistence in the form of the *entity bean*, part of the Enterprise JavaBean (EJB) family of components. Intended to fully insulate developers from dealing directly with persistence, it introduced an interface-based approach, where the concrete bean class is never directly used by client code. Instead, a specialized bean compiler generates an implementation of the bean interface that facilitates persistence, security, transaction management, and more, delegating only the business logic to the entity bean implementation. Entity beans are configured using a combination of standard and vendor-specific XML deployment descriptors which have become famous for their complexity and verbosity.

It's probably fair to say that entity beans were over-engineered for the problem they were trying to solve; yet ironically the first release of the technology lacked many features necessary to implement realistic business applications. Relationships between entities had to be managed by the application, requiring foreign key fields to be stored and managed on the bean class. The actual mapping of the entity bean to the database was done entirely using vendor-specific configurations, as was the definition of finders, the entity bean term for queries. Finally, entity beans were modeled as remote objects that used RMI and CORBA, introducing network overhead and restrictions that should never have been added to a persistent object to begin with. The entity bean seemed to have begun by solving the distributed persistent component problem that never existed to begin with, leaving behind the common case of locally accessed lightweight persistent objects.

The EJB 2.0 specification solved many of the problems identified in the early releases. The notion of container-managed entity beans was introduced, where bean classes became abstract and the server was responsible for generating a subclass to manage the persistent data. Local interfaces and container-managed relationships were introduced, allowing associations to be defined between entity beans and automatically kept consistent by the server. This release also saw the introduction of Enterprise JavaBeans Query Language (EJB QL), a query language designed to work with entities that could be portably compiled to any SQL dialect.

Despite the improvements introduced with EJB 2.0, there is one problem that could not be overcome by the EJB expert group: complexity. The specification assumed that development tools would insulate the developer from the challenge of configuring and managing the sheer number of artifacts that were required for each bean. Unfortunately, these tools took too long to materialize, and the development burden fell squarely on the shoulders of the developer even as the size and scope of EJB applications increased. Developers felt abandoned in a sea of complexity without the promised infrastructure to keep them afloat.

Java Data Objects

Due in part to some of the failures of the EJB persistence model, and some amount of frustration at not having a standardized persistence API that was satisfactory, another persistence specification effort was attempted. Java Data Objects (JDO) was inspired and supported primarily by the object-oriented database (OODB) community at the outset and probably at least partly because it did not garner the support that a specification needed to become adopted by the community. It required that vendors enhance the bytecode of the domain objects to produce class files that were binary-compatible across all vendors, and every compliant vendor had to be capable of both producing and consuming them. It also had a query language that was decidedly object-oriented in nature, which did not sit well with the relational database users, who as it turned out were the majority.

JDO reached the status of being an extension of the Java Development Kit (JDK) but never became an integrated part of the enterprise Java platform. It had a great many good features in it and was adopted by a small community of devoted and loyal users who stuck by it and tried to promote it. Unfortunately the major commercial vendors did not share the same view of how a persistence framework should be implemented. Few supported the specification, and as a result JDO spent most of its time in the persistence underground.

Some might argue that it was slightly ahead of its time and that its reputation for enhancement caused it to be unfairly stigmatized. This was probably true, and if it had been introduced three years later, it might have been much more accepted by a developer community that now thinks nothing of using frameworks that make extensive use of bytecode enhancement. Once the EJB 3.0 persistence movement was in motion, however, and the major vendors all signed up to be a part of the new enterprise persistence standard, the writing was on the wall for JDO. People soon complained to Sun that they now had two persistence specifications, one that was part of its enterprise platform and also worked in Java SE, and one that was standardized only for Java SE. Shortly thereafter Sun announced that JDO would be reduced to specification maintenance mode and that the Java Persistence API would draw from both JDO and the other persistence vendors and become the single supported standard going forward.

Why Another Standard?

Software developers knew what they wanted, but many could not find it in the existing standards, so they decided to look elsewhere. What they found was proprietary persistence frameworks, both in the commercial and open source domains. The products that implemented these technologies adopted a persistence model that did not intrude upon the domain objects. Persistence was nonintrusive to the business objects in that, unlike entity beans, they did not have to be aware of the technology that was persisting them. They did not have to implement any type of interface or extend a special class. The developer could simply develop the persistent object as with any other Java object, and then map it to a persistent store and use a persistence API to persist the object. Because the objects were regular Java objects, this persistence model came to be known as POJO (Plain Old Java Object) persistence.

The two most popular of these persistence APIs were TopLink in the commercial space and Hibernate in the open source community. These and other products grew to support all the major application servers and provided applications with all of the persistence features they needed. Application developers were quite satisfied to use a third-party product for their persistence needs.

As Hibernate, TopLink, and other persistence APIs became ensconced in applications and met the needs of the application perfectly well, the question was often asked, "Why bother updating the EJB standard to match what these products already did? Why not just continue to use these products as has already been done for years, or why not even just standardize on an open source product like Hibernate?" There are actually a great many reasons why this is not only infeasible but also unpalatable.

A standard goes far deeper than a product, and a single product (even a product as successful as Hibernate or TopLink) cannot embody a specification, even though it can implement one. At its very core, the intention of a specification is that it be implemented by different vendors and that it have different products offer standard interfaces and semantics that can be assumed by applications without coupling the application to any one product.

Binding a standard to an open source project like Hibernate would be problematic for the standard and probably even worse for the Hibernate project. Imagine a specification that was based on a specific version or checkpoint of the code base of an open source project, and how confusing that would be. Now imagine an open source software (OSS) project that could not change or could change only in discrete versions controlled by a special committee every two years, as opposed to the changes being decided by the project itself. Hibernate, and indeed any open source project, would likely be suffocated.

Standardization may not be valued by the consultant or the five-person software shop, but to a corporation it is huge. Software technologies are a big investment for most corporate IT shops, and when large sums of money are involved, risk must be measured. Using a standard technology reduces that risk substantially and allows the corporation to be able to switch vendors if the initial choice turns out not to have met the need.

Besides portability, the value of standardizing a technology is manifested in all sorts of other areas as well. Education, design patterns, and industry communication are just some of the many other benefits that standards bring to the table.

Object-Relational Mapping

"The domain model has a class. The database has a table. They look pretty similar. It should be simple to convert from one to the other automatically." This is a thought we've probably all had at one point or another while writing yet another Data Access Object to convert JDBC result sets into something object-oriented. The domain model looks similar enough to the relational model of the database that it seems to cry out for a way to make the two models talk to each other.

The science of bridging the gap between the object model and the relational model is known as *object-relational mapping,* often referred to as O-R mapping or simply ORM. The term comes from the idea that we are in some way mapping the concepts from one model onto another, with the goal of introducing a mediator to manage the automatic transformation of one to the other.

Before going into the specifics of object-relational mapping, let's define a brief manifesto of sorts for what the ideal solution *should* be:

- **Objects, not tables.** Applications should be written in terms of the domain model and not be bound to the relational model. It must be possible to operate on and query against the domain model without having to express it in the relational language of tables, columns, and foreign keys.

- **Convenience, not ignorance.** The task of mapping will be and should be done by someone familiar with relational technology. O-R mapping is not for someone who does not want to understand the mapping problems or have them hidden from their view. It is meant for those who have an understanding of the issues and know what they want but who just don't want to have to write thousands of lines of code that somebody has already written to solve the problem.

- **Unobtrusive, not transparent.** It is unreasonable to expect that persistence be transparent since an application always needs to have control of the objects that it is persisting and be aware of the entity life cycle. The persistence solution should not intrude on the domain model, however, and domain classes must not be required to extend classes or implement interfaces in order to be persistable.

- **Legacy data, new objects.** It is far more likely that an application will target an existing relational database schema instead of creating a new one. Support for legacy schemas is one of the most relevant use cases that will arise, and it is quite possible that such databases will outlive every one of us.

- **Enough, but not too much.** Enterprise applications have problems to solve, and they need features sufficient to solve those problems. What they don't like is being forced to eat a heavyweight persistence model that introduces large overhead because it is solving problems that many do not even agree are problems.
- **Local, but mobile.** A persistent representation of data does not need to be modeled as a full-fledged remote object. Distribution is something that exists as part of the application, not part of the persistence layer. The entities that contain the persistent state, however, must be able to travel to whichever layer needs them.

This would appear to be a somewhat demanding set of requirements, but it is one born of both practical experience and necessity. Enterprise applications have very specific persistence needs, and this shopping list of items is a fairly specific representation of the experience of the enterprise community.

The Impedance Mismatch

Advocates for object-relational mapping often describe the difference between the object model and the relational model as the *impedance mismatch* between the two. This is an apt description because the challenge of mapping one to the other lies not in the similarities between the two, but in the many concepts in both for which there is no logical equivalent in the other.

In the following sections we will present some basic object-oriented domain models and a variety of relational models to persist the same set of data. As you are about to see, the challenge in object-relational mapping is not so much the complexity of a single mapping but that there are so many possible mappings. The goal is not to explain how to get from one point to the other but to understand the roads that may have to be taken to arrive at an intended destination.

Class Representation

Let's begin this discussion with a simple class. Figure 1-1 shows an `Employee` class with four attributes: employee id, employee name, date they started, and current salary.

Figure 1-1. *The* `Employee` *class*

Now consider the relational model shown in Figure 1-2. The ideal representation of this class in the database corresponds to scenario (A). Each field in the class maps directly to a column in the table. The employee number becomes the primary key. With the exception of some slight naming differences, this is a straightforward mapping.

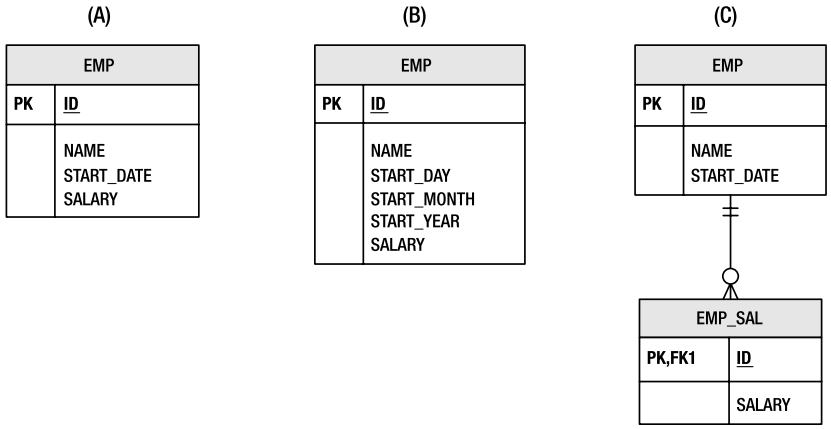

Figure 1-2. *Three scenarios for storing employee data*

In scenario (B), we see that the start date of the employee is actually stored as three separate columns, one each for the day, month, and year. Recall that the class used a `Date` object to represent this value. As database schemas are much harder to change, should the class be forced to adopt the same storage strategy in order to keep parity with the relational model? Also consider the inverse of the problem, where the class had used three fields and the table used a single date column. Even a single field becomes complex to map when the database and object model differ in representation.

Salary information is considered sensitive information, so it may be unwise to place the salary value directly in the `EMP` table, which may be used for a number of purposes. In scenario (C), the `EMP` table has been split so that the salary information is stored in a separate `EMP_SAL` table. This allows the database administrator to restrict SELECT access on salary information to only those users who genuinely require it. With such a mapping, even a single store operation for the `Employee` class now requires inserts or updates to two different tables.

Clearly, even storing the data from a single class in a database can be a challenging exercise. We concern ourselves with these scenarios because real database schemas in production systems were never designed with object models in mind. The rule of thumb in enterprise applications is that the needs of the database trump the wants of the application. It's up to the object model to adapt and find ways to work with the database schema without letting the physical design overpower the logical application model.

Relationships

Objects rarely exist in isolation. Just like relationships in a database, domain classes depend on and associate themselves with other domain classes. Consider the `Employee` class introduced in Figure 1-1. There are many domain concepts we could associate with an employee, but for now let's introduce the `Address` domain class, for which an `Employee` may have at most one instance. We say in this case that `Employee` has a one-to-one relationship with `Address`, represented in the Unified Modeling Language (UML) model by the `0..1` notation. Figure 1-3 demonstrates this relationship.

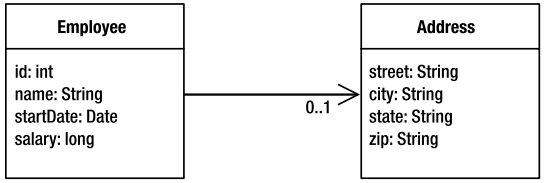

Figure 1-3. *The* `Employee` *and* `Address` *relationship*

We discussed different scenarios for representing the `Employee` state in the previous section, and likewise there are several approaches to representing a relationship in a database schema. Figure 1-4 demonstrates three different scenarios for a one-to-one relationship between an employee and an address.

The building block for relationships in the database is the foreign key. Each scenario involves foreign key relationships between the various tables, but in order for there to be a foreign key relationship, the target table must have a primary key. And so before we even get to associate employees and addresses with each other we have a problem. The domain class `Address` does not have an identifier, yet the table that it would be stored in must have one if it is to be part of relationships. We could construct a primary key out of all of the columns in the `ADDRESS` table, but this is considered bad practice. Therefore the `ID` column is introduced and the object relational mapping will have to adapt in some way.

In scenario (A) of Figure 1-4 we have the ideal mapping of this relationship. The `EMP` table has a foreign key to the `ADDRESS` table stored in the `ADDRESS_ID` column. If the domain class holds onto an instance of the `Address` class, then the primary key value for the address can be set during store operations.

And yet consider scenario (B), which is only slightly different yet suddenly much more complex. In our domain model, `Address` did not hold onto the `Employee` instance that owned it, and yet the employee primary key must be stored in the `ADDRESS` table. The object-relational mapping must either account for this mismatch between domain class and table or a reference back to the employee will have to be added for every address.

To make matters worse, scenario (C) introduces a join table to relate the `EMP` and `ADDRESS` tables. Instead of storing the foreign keys directly in one of the domain tables, the join table instead holds onto the pair of keys. Every database operation involving the two tables must now traverse the join table and keep it consistent. We could introduce an `EmployeeAddress` association class into our domain model to compensate, but that defeats the logical representation we are trying to achieve.

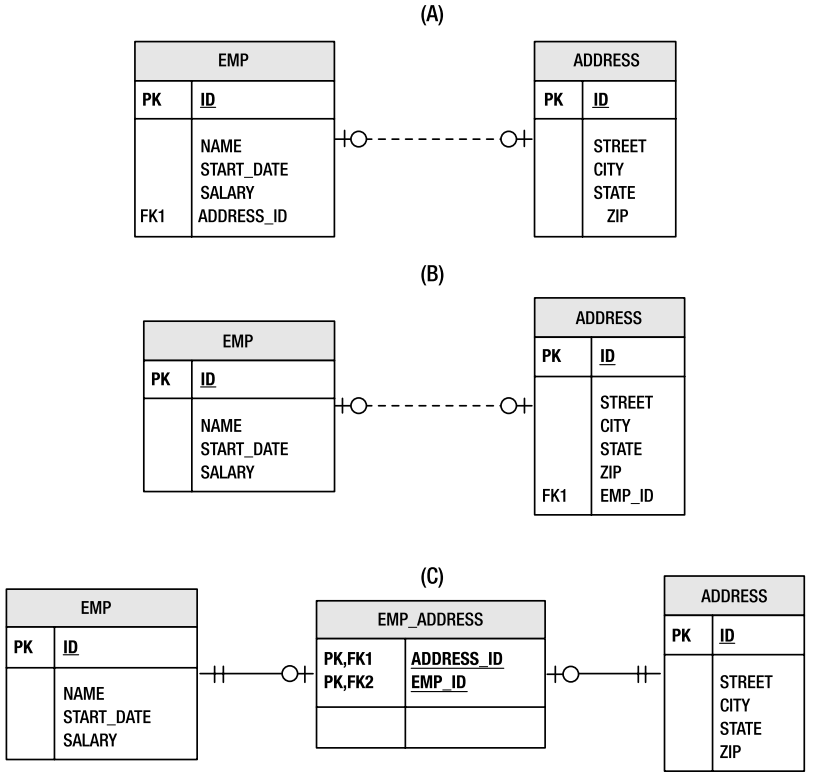

Figure 1-4. *Three scenarios for relating employee and address data*

Relationships present a challenge in any object-relational mapping solution. In this introduction we have covered only one-to-one relationships, and yet we have been faced with the need for primary keys not in the object model and the possibility of having to introduce extra relationships into the model or even association classes to compensate for the database schema.

Inheritance

A defining element of an object-oriented domain model is the opportunity to introduce generalized relationships between like classes. Inheritance is the natural way to express these relationships and allows for polymorphism in the application. Let's revisit the `Employee` class shown in Figure 1-1 and imagine a company that needs to distinguish between full-time and part-time employees. Part-time employees work off of an hourly rate, while full-time employees are assigned a salary. This is a good opportunity for inheritance, moving wage information to `PartTimeEmployee` and `FullTimeEmployee` subclasses. Figure 1-5 shows this arrangement.

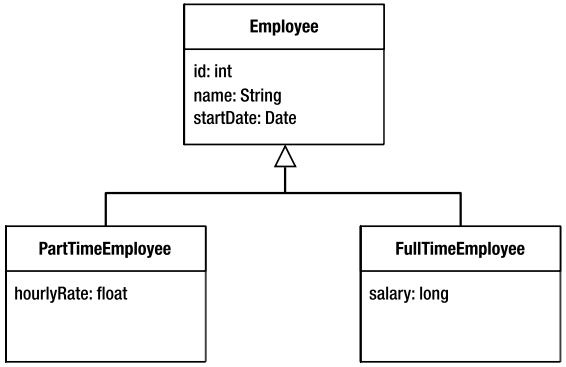

Figure 1-5. *Inheritance relationships between full-time and part-time employees*

Inheritance presents a genuine problem for object-relational mapping. We are no longer dealing with a situation where there is a natural mapping from a class to a table. Consider the relational models shown in Figure 1-6. Once again we demonstrate three different strategies for persisting the same set of data.

Arguably the easiest solution for someone mapping an inheritance structure to a database would be to put all of the data necessary for each class (including parent classes) into separate tables. This strategy is demonstrated by scenario (A) in Figure 1-6. Note that there is no relationship between the tables. This means that queries against these tables are now much more complicated if the user needs to operate on both full-time and part-time employees in a single step.

An efficient but denormalized alternative is to place all of the data required for every class in the model in a single table. That makes it very easy to query, but note the structure of the table shown in scenario (B) of Figure 1-6. There is a new column, `TYPE`, which does not exist in any part of the domain model. The `TYPE` column indicates whether or not the employee is part-time or full-time. This information must now be interpreted by an object-relational mapping solution to know what kind of domain class to instantiate for any given row in the table.

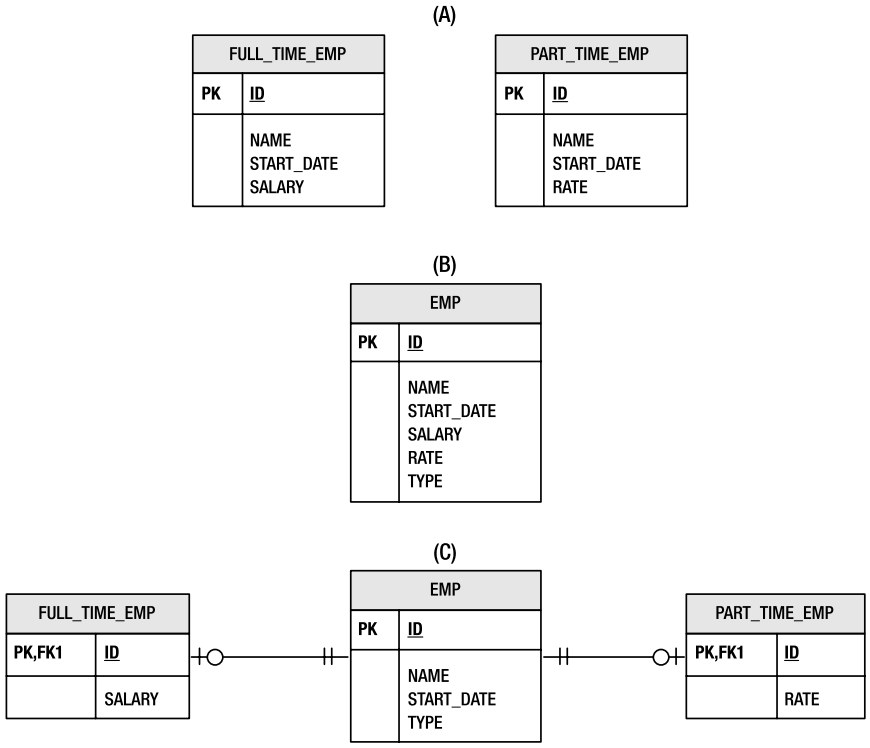

Figure 1-6. *Inheritance strategies in a relational model*

Scenario (C) takes this one step further, this time normalizing the data into separate tables each for full-time and part-time employees. Unlike scenario (A), however, these tables are related by a common `EMP` table that stores all of the data common to both employee types. It might seem like an excessive solution for a single column of extra data, but a real schema with many columns specific to each type of employee would likely use this type of table structure. It presents the data in a logical form and also simplifies querying by allowing the tables to be joined together. Unfortunately, what works well for the database does not necessarily work well for an object model mapped to such a schema. Even without associations to other classes, the object-relational mapping of the domain class must now take joins between multiple tables into account.

When you start to consider abstract superclasses or parent classes with no persistent form, inheritance rapidly becomes a complex issue in object-relational mapping. Not only is there a challenge with storage of the class data, but the complex table relationships are difficult to query efficiently.

The Java Persistence API

The Java Persistence API is a lightweight, POJO-based framework for Java persistence. Although object-relational mapping is a major component of the API, it also offers solutions to the architectural challenges of integrating persistence into scalable enterprise applications. In the following sections we will look at the evolution of the specification and provide an overview of the major aspects of this technology.

History of the Specification

The Java Persistence API is remarkable not only for what it offers developers but also for the way in which it came to be. The following sections outline the prehistory of object-relational persistence solutions and the genesis of the Java Persistence API as part of EJB 3.0.

The Early Years

It may come as a surprise to learn that object-relational mapping solutions have been around for a long time, longer even than the Java language itself. Products such as Oracle TopLink originally got their start in the Smalltalk world before making the switch to Java. Perhaps one of the greatest ironies in the history of Java persistence solutions is that one of the first implementations of entity beans, which have long been criticized for their complexity, was actually demonstrated by mapping the bean class and persisting it using TopLink.

Commercial object-relational mapping products like TopLink have been available since the earliest days of the Java language. They were successful, but the techniques were never standardized for the Java platform. An approach similar to object-relational mapping was standardized in the form of JDO, but as we mentioned previously, that standard failed to gain any significant market penetration.

It was actually the popularity of open source object-relational mapping solutions such as Hibernate that led to some surprising changes in the direction of persistence in the Java platform and brought about a convergence towards object-relational mapping as the preferred solution.

EJB 3.0

After years of complaints about the complexity of building enterprise applications with Java, "ease of development" was adopted as the theme for the Java EE 5.0 platform release. The members of the EJB 3.0 expert group were charged with finding ways to make Enterprise JavaBeans easier and more productive to use.

In the case of session beans and message-driven beans, solutions for usability issues were largely cosmetic in scope. By simply removing some of the more onerous implementation requirements and letting components look more like plain Java objects, the goal was largely achieved early on.

In the case of entity beans, however, a more serious problem faced the expert group. If the definition of "ease of use" is to keep implementation interfaces and descriptors out of application code and to embrace the natural object model of the Java language, how do you make coarse-grained, interface-driven, container-managed entity beans look and feel like a domain model?

The conclusion reached by the expert group was nothing short of remarkable: *start over*. Leave entity beans alone and introduce a new model for persistence. And start over we did, but not from scratch. The Java Persistence API was born out of recognition of the demands of practitioners and the existing proprietary solutions that they were using to solve their problems. To ignore that experience would have been folly.

The expert group reached out to the leading vendors of object-relational mapping solutions and invited them to come forward and standardize the best practices represented by their products. Hibernate and TopLink were the first to sign on with the existing EJB vendors, followed later by the JDO vendors.

Years of industry experience coupled with a mission to simplify development combined to produce the first specification to truly embrace the new programming models offered by the Java SE 5 platform. The use of annotations in particular resulted in a new way of using persistence in applications that had never been seen before.

The resulting EJB 3.0 specification ended up being divided into three distinct pieces and split across three separate documents. The first includes the existing EJB 2.1 APIs and the traditional contracts from the perspectives of the container, the bean provider, and the client. This content was incremented by the additional Java EE injection features as well as the new EJB 3.0 interceptor specifications and lifecycle callback changes. This is the heavy document that describes the "old school" of EJB development plus some of the new features that have been made available to the old API.

The second document describes a simplified API that people can use to develop new session and message-driven components against. It is essentially an overview of the ease-of-use features that were introduced for EJB components by EJB 3.0. It outlines the basic ideas of how to define and annotate beans, use them without home interfaces, add callback methods and interceptors, and apply these new features.

The third document is the Java Persistence API, a stand-alone specification that describes the persistence model in both the Java SE and Java EE environments, and the subject of this book. In the next iteration the Java Persistence API will become a separate specification in the Java EE platform, distinct from the Enterprise JavaBeans specification.

Overview

The model of the Java Persistence API is simple and elegant, powerful and flexible. It is natural to use, and easy to learn, especially if you have used any of the existing persistence products on the market today on which the API was based. The main operational API that an application will be exposed to is contained within only a few classes.

POJO Persistence

Perhaps the most important aspect of the Java Persistence API is the fact that the objects are POJOs, meaning that there is nothing special about any object that is made persistent. In fact, any existing application object can be made persistent without so much as changing a single line of code. Object-relational mapping with the Java Persistence API is entirely metadata-driven. It can be done either by adding annotations to the code or using externally defined

XML. The objects that are persisted are lightweight in memory and as light as the user happens to define and map them in the database.

Non-intrusiveness

The persistence API exists as a separate layer from the persistent objects. The persistence API is called by the application business logic and is passed the persistence objects and instructed to operate upon them. So even though the application must be aware of the persistence API, since it has to call into it, the persistent objects themselves need not be aware. This is noteworthy because some people are under the misconception that transparent persistence means that objects magically get persisted, the way that object databases of yesteryear used to do when a transaction got committed. This is an incorrect notion and even more irrational when you think about querying. You need to have some way of retrieving the objects from the data store. This requires a separate API object and, in fact, even object databases used separate `Extent` objects to issue queries. Applications absolutely need to manage their persistent objects in very explicit ways, and they require a designated API to do it. Because the API does not intrude upon the persistent objects themselves, we call this *non-intrusive persistence*.

Object Queries

A powerful query framework offers the ability to query across entities and their relationships without having to use concrete foreign keys or database columns. Queries are expressed in Java Persistence Query Language, a query language that is derived from EJB QL and modeled after SQL for its familiarity, but it is not tied to the database schema. Queries use a schema abstraction that is based on the state of an entity as opposed to the columns in which the entity is stored. Creating a query does not require knowledge of the database mapping information and typically returns results that are in the form of entities.

A query may be defined statically in metadata or created dynamically by passing query criteria when constructing it. It is also possible to escape to SQL if a special query requirement exists that cannot be met by the SQL generation from the persistence framework. These queries can all return results that are entities and are valuable abstractions that enable querying across the Java domain model instead of across database tables.

Mobile Entities

Client/server and web applications and other distributed architectures are clearly the most popular types of applications in a connected world. To acknowledge this fact meant acknowledging that persistent entities must be mobile in the network. Objects must be able to be moved from one virtual machine to another and then back again, and must still be usable by the application.

The detachment model provides a way of reconciling any newly relocated or deserialized instance that may have changed state along the way, with the instance or state that was left behind. Objects that leave the persistence layer are called detached, and a key feature of the persistence model is the ability to reattach such detached entities upon their return.

Simple Configuration

There are a great number of persistence features that the specification has to offer and which we will explain in the chapters of this book. All of the features are configurable through the use of Java SE 5 annotations, or XML, or a combination of the two. Annotations offer ease of use that is unparalleled in the history of Java metadata. They are convenient to write and painless to read, and they make it possible for beginners to get an application going quickly and easily. Configuration may also be done in XML for those who like XML or are more comfortable with it. Of greater significance than the metadata language is the fact that the Java Persistence API 3.0 makes heavy use of defaults. This means that no matter which method is chosen, the amount of metadata that will be required just to get running is the absolute minimum. In some cases, if the defaults are good enough almost no metadata will be required at all.

Integration and Testability

Multitier applications hosted on an application server have become the de facto standard for application architectures. Testing on an application server is a challenge that few relish. It can bring pain and hardship, and it is often prohibitive to unit testing and white box testing.

This is solved by defining the API to work outside as well as inside the application server. While it is not as common a use case, those applications that do run on two tiers (the application talking directly to the database tier) can use the persistence API without the existence of an application server at all. The more common scenario is for unit tests and automated testing frameworks that can be run easily and conveniently in Java SE environments.

The Java Persistence API really has introduced a new era in standardized integrated persistence. When running inside a container, all of the benefits of container support and superior ease of use apply, but the same application may also be configured to run outside the container as well.

Summary

In this chapter we presented an introduction to the Java Persistence API. We began with an overview of current standards for persistence, looking at JDBC, EJB, and JDO. In each case, we looked at the conditions forced on us by these frameworks and developed a view of what a better solution could be.

In the Object-Relational Mapping section we introduced the primary problem facing developers trying to use object-oriented domain models in concert with a relational database: the impedance mismatch. To demonstrate the complexity bridging the gap, we presented three small object models and nine different ways to represent the same information.

We concluded the chapter with a brief look at the Java Persistence API. We looked at the history of the specification and the vendors who came together to create it. We then looked at the role it plays in enterprise application development and introduced the feature set offered by the specification.

In the next chapter we will get our feet wet with the Java Persistence API, taking a whirlwind tour of the API basics and building a simple application in the process.

CHAPTER 2

Getting Started

From the outset, one of the main goals when creating the Java Persistence API was to ensure that it is simple to use and easy to understand. Although the problem domain cannot be trivialized or watered down, the technology that enables one to deal with it can be straightforward and intuitive. In this chapter we will show how effortless it is to develop and use entities.

We will start this chapter off by describing the basic characteristics of entities. We'll define what an entity is and how to create, read, update, and delete them. We'll also introduce entity managers and how they are obtained and used. Then we'll take a quick look at queries and how to specify and execute a query using the `EntityManager` and `Query` objects. The chapter will conclude by showing a simple working application that runs in a standard Java SE 5 environment and that demonstrates all of the example code in action.

Entity Overview

The entity is not a new thing. In fact, entities have been around longer than many programming languages and certainly longer than Java. They were first introduced by Peter Chen in his seminal paper on entity-relationship modeling.[1] He described entities as things that have attributes and relationships. The expectation was that the attributes were going to be persisted in a relational database, as were the relationships.

Even now, the definition still holds true. An *entity* is essentially a noun, or a grouping of state associated together as a single unit. It may participate in relationships to any number of other entities in a number of standard ways. In the object-oriented paradigm, we would add behavior to it and call it an object. In the Java Persistence API, any application-defined object can be an entity, so the important question might be, What are the characteristics of an object that has been turned into an entity?

Persistability

The first and most basic characteristic of entities is that they are *persistable*. This generally just means that they can be made persistent. More specifically it means that their state can be represented in a data store and can be accessed at a later time, perhaps well after the end of the process that created it.

1. Peter C. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems* 1, no. 1 (1976): 9–36.

We could call them persistent objects, and many people do, but it is not technically correct. Strictly speaking, a persistent object becomes persistent the moment it is instantiated. If a persistent object exists, then by definition it is already persistent.

An entity is persistable because it *can* be created in a persistent store. The difference is that it is not automatically persisted and that in order for it to have a persistent representation the application must actively invoke an API method to initiate the process. This is an important distinction because it leaves control over persistence firmly in the hands of the application. It offers the application the flexibility to manipulate data and perform business logic on the entity, and then only when the application decides that it is the right time to persist the entity, actually causing it to be persistent. The lesson is that entities may be manipulated without necessarily having persistent repercussions, and it is the application that decides whether or not they do.

Identity

Like any other Java object, an entity has an object identity, but when it exists in the data store it also has a *persistent identity*. Persistent identity, or an *identifier*, is the key that uniquely identifies an entity instance and distinguishes it from all of the other instances of the same entity type. An entity has a persistent identity when there exists a representation of it in the data store, that is, a row in a database table. If it is not in the database then even though the in-memory entity may have its identity set in a field, it does not have a persistent identity. The entity identifier, then, is equivalent to the primary key in the database table that stores the entity state.

Transactionality

Entities are what we might call *quasi-transactional*. They are normally only created, updated, and deleted within a transaction,[2] and a transaction is required for the changes to be committed in the database. Changes made to the database either succeed or fail atomically, so the persistent view of an entity should indeed be transactional.

In memory it is a slightly different story in the sense that entities may be changed without the changes ever being persisted. Even when enlisted in a transaction, they may be left in an undefined or inconsistent state in the event of a rollback or transaction failure. The in-memory entities are simple Java objects that obey all of the rules and constraints that are applied by the Java virtual machine to other Java objects.

Granularity

Finally, we can also learn something about what entities are by describing what they are *not*. They are not primitives, primitive wrappers, or built-in objects. These are no more than scalars and do not have any designated semantic meaning to an application. A string, for example is too fine-grained an object to be an entity because it does not have any domain-specific connotation. Rather, a string is well-suited and very often used as a type for an entity attribute and given meaning according to the entity attribute that it is typing.

2. In most cases this is a requirement, but in certain configurations the transaction may not be present until later.

Entities are fine-grained objects that have a set of aggregated state that is normally stored in a single place, such as a row in a table, and typically have relationships to other entities. In the most general sense they are business domain objects that have specific meaning to the application that accesses them.

While it is certainly true that entities may be defined in exaggerated ways to be as fine-grained as storing a single string or coarse-grained enough to contain 500 columns' worth of data, the suggested granularity of an entity is definitely on the smaller end of the spectrum. Ideally, entities should be designed and defined as fairly lightweight objects of equal or smaller size than that of the average Java object.

Entity Metadata

Associated with every entity is metadata in some amount, possibly small, that describes it. This metadata enables the persistence layer to recognize, interpret, and properly manage the entity from the time it is loaded through to its runtime invocation.

The metadata that is actually required for each entity is minimal, rendering entities easy to define and use. However, like any sophisticated technology with its share of switches, levers, and buttons, there is also the possibility to specify much, much more metadata than is required. It may be extensive amounts, depending upon the application requirements, and may be used to customize every detail of the entity configuration or state mappings.

Entity metadata may be specified in one of two ways—annotations or XML. Each is equally valid, but the one that you use will depend upon your development preferences or process.

Annotations

Annotation metadata is a language feature that allows structured and typed metadata to be attached to the source code. It was introduced as part of Java SE 5 and is a key part of the EJB 3.0 and Java EE 5 specifications.[3] Although annotations are not required by the Java Persistence API, they are a convenient way to learn and use the API. Because annotations co-locate the metadata with the program artifacts, it is not necessary to escape to an additional file and additional language (XML) just to specify the metadata.

Annotations are used throughout both the examples and the accompanying explanations in this book. All of the API annotations that are shown and described, except for Chapter 3, which talks about Java EE annotations, are defined in the `javax.persistence` package. Example code snippets can be assumed to have an implicit import of the form `import javax.persistence.*;`.

XML

For those who prefer to use the traditional XML descriptors, this option is still available. It should be a fairly straightforward process to switch to using XML descriptors after having learned and understood the annotations since the XML has in large part been patterned after the annotations. Chapter 10 describes how to use XML to specify or override entity mapping metadata.

3. The Java EE 5 platform specification and all of its sub-specifications require the use of Java SE 5.

ANNOTATIONS

Java annotations are specially defined types that may annotate (be attached to or placed in front of) Java programming elements including classes, methods, fields, and variables. When they annotate a program element, the compiler reads the information contained in them and may retain it in the class files or dispose of it according to what was specified in the annotation type definition. When retained in the class files the elements contained in the annotation may be queried at runtime through a reflection-based API. A running program can in this way obtain the metadata that exists on a Java program element. An example of a custom annotation type definition that could be used to indicate classes that should be validated (whatever validate means to the application or tool that is processing it) is:

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Validate {
    boolean flag;
}
```

This annotation definition is in fact itself annotated by `@Target` and `@Retention` built-in annotations that determine what kinds of program elements the annotation may annotate and at what point the annotation metadata should be discarded from the class. The annotation defined above may annotate any type and will not be discarded from the class (that is, it will be retained in the class file even at runtime). This annotation may, for example, annotate any given class definition. An example usage of this annotation could be:

```
@Validate(flag=true)
public class MyClass {
  ...
}
```

An application that looks at all classes in the system for this annotation will be able to determine that `MyClass` should be validated and perform that validation whenever it makes sense. The semantic meaning of `@Validate` is completely up to the component that defines the annotation type and the one that reads and processes the annotation.

Configuration by Exception

The notion of *configuration by exception* means that the persistence engine defines defaults that apply to the majority of applications and that users need to supply values only when they want to override the default value. In other words, having to supply a configuration value is the exception to the rule, not a requirement.

Configuration by exception is ingrained in the Java Persistence API and is a strong contributing factor to its usability. The majority of configuration values have defaults, rendering the metadata that does have to be specified more relevant and concise.

The extensive use of defaults and the ease of use that it brings to configuration comes with a price, however. When defaults are embedded into the API and do not have to be specified, then they are not visible or obvious to users. This *can* make it possible for users to be unaware of the complexity of developing persistence applications, making it slightly more difficult to debug or to change the behavior when it becomes necessary.

Defaults are not meant to shield users from the often complex issues surrounding persistence. They are meant to allow a developer to get started easily and quickly with something that will work and then iteratively improve and implement additional functionality as the complexity of their application increases. Even though the defaults may be what you want to have happen most of the time, it is still fairly important for developers to be familiar with the default values that are being applied. For example, if a table name default is being assumed, then it is important to know what table the runtime is expecting, or if schema generation is used, what table will be generated.

For each of the annotations we will also discuss the default value so that it is clear what will be applied if the annotation is not specified. We recommend that you remember these defaults as you learn them. After all, a default value is still part of the configuration of the application; it was just really easy to configure!

Creating an Entity

Regular Java classes are easily transformed into entities simply by annotating them. In fact, by adding a couple of annotations, virtually any class with a no-arg constructor can become an entity.

Let's start by creating a regular Java class for an employee. Listing 2-1 shows a simple `Employee` class.

Listing 2-1. `Employee` *Class*

```
public class Employee {
    private int id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(int id) { this.id = id; }

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

You may notice that this class resembles a JavaBean-style class with three properties: `id`, `name`, and `salary`. Each of these properties is represented by a pair of accessor methods to get and set the property and is backed by a member field. Properties or member fields are the units of state within the entity that we want to persist.

To turn `Employee` into an entity we first need to annotate the class with `@Entity`. This is primarily just a marker annotation to indicate to the persistence engine that the class is an entity.

The second annotation that we need to add is `@Id`. This annotates the particular field or property that holds the persistent identity of the entity (the primary key) and is needed so the provider knows which field or property to use as the unique identifying key in the table.

Adding these two annotations to our Employee class, we end up with pretty much the same class that we had before, except that now it is an entity. Listing 2-2 shows the entity class.

Listing 2-2. Employee *Entity*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(int id) { this.id = id; }

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

When we say that the @Id annotation is placed on the field or property, we mean that the user can choose to annotate either the declared field, or the getter method[4] of a JavaBean-style property. Either field or property strategy is allowed, depending upon the needs and tastes of the entity developer, but whichever strategy is chosen, it must be followed for all persistent state annotations in the entity. We have chosen in this example to annotate the field because it is simpler; in general, this will be the easiest and most direct approach. We will learn more about the details of annotating persistent state using field or property access in subsequent chapters.

Automatic State Mapping

The fields in the entity are automatically made persistable by virtue of their existence in the entity. Default mapping and loading configuration values apply to these fields and enable them to be persisted when the object is persisted. Given the questions that were brought up in the last chapter, one might be led to ask, "How did the fields get mapped, and where do they get persisted to?"

To find the answer we must first take a quick detour to dig inside the @Entity annotation and look at an element called name that uniquely identifies the entity. The entity name may be explicitly specified for any entity by using this name element in the annotation, as in @Entity(name="Emp"). In practice this is seldom specified because it gets defaulted to be the unqualified name of the entity class. This is almost always both reasonable and adequate.

Now we can get back to the question about where the data gets stored. It turns out that the default name of the table used to store any given entity of a particular entity type is the name

4. Annotations on setter methods will just be ignored.

of the entity. If we have specified the name of the entity, then that will be the default table name, but if we have not, then the default value of the entity name will be used. We just stated that the default entity name was the unqualified name of the entity class, so that is effectively the answer to the question of which table gets used. In our Employee example all entities of type Employee will get stored in a table called EMPLOYEE.

Each of the fields or properties has individual state in it and needs to be directed to a particular column in the table. We know to go to the EMPLOYEE table, but we don't know which column to use for any given field or property. When no columns are explicitly specified, then the default column is used for a field or property, which is just the name of the field or property itself. So our employee id will get stored in the ID column, the name in the NAME column, and the salary in the SALARY column of the EMPLOYEE table.

Of course these values can all be overridden to match an existing schema. We will discuss how to override them when we get to Chapter 4 and discuss mapping in more detail.

Entity Manager

In the Entity Overview section, it was stated that a specific API call needs to be invoked before an entity actually gets persisted to the database. In fact, separate API calls are needed to perform many of the operations on entities. This API is implemented by the entity manager and encapsulated almost entirely within a single interface called EntityManager. When all is said and done, it is to an entity manager that the real work of persistence is delegated. Until an entity manager is used to actually create, read, or write an entity, the entity is nothing more than a regular (non-persistent) Java object.

When an entity manager obtains a reference to an entity, either by having it explicitly passed in or because it was read from the database, that object is said to be *managed* by the entity manager. The set of managed entity instances within an entity manager at any given time is called its *persistence context*. Only one Java instance with the same persistent identity may exist in a persistence context at any time. For example, if an Employee with a persistent identity (or id) of 158 exists in the persistence context, then no other object with its id set to 158 may exist within that same persistence context.

Entity managers are configured to be able to persist or manage specific types of objects, read and write to a given database, and be implemented by a particular *persistence provider* (or *provider* for short). It is the provider that supplies the backing implementation engine for the entire Java Persistence API, from the EntityManager through to Query implementation and SQL generation.

All entity managers come from factories of type EntityManagerFactory. The configuration for an entity manager is bound to the EntityManagerFactory that created it, but it is defined separately as a *persistence unit*. A persistence unit dictates either implicitly or explicitly the settings and entity classes used by all entity managers obtained from the unique EntityManagerFactory instance bound to that persistence unit. There is, therefore, a one-to-one correspondence between a persistence unit and its concrete EntityManagerFactory.

Persistence units are named to allow differentiation of one EntityManagerFactory from another. This gives the application control over which configuration or persistence unit is to be used for operating on a particular entity.

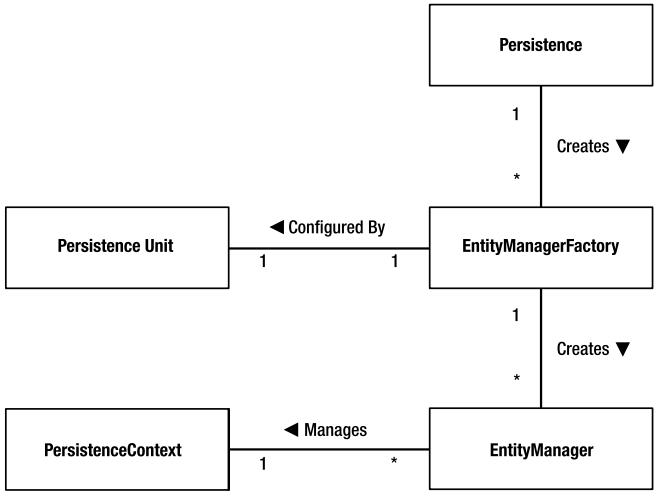

Figure 2-1. *Relationships between Java Persistence API concepts*

Figure 2-1 shows that for each persistence unit there is an `EntityManagerFactory` and that many entity managers can be created from a single `EntityManagerFactory`. The part that may come as a surprise is that many entity managers can point to the same persistence context. We have talked only about an entity manager and its persistence context, but later on we will see that this is indeed the case and that there may be multiple references to different entity managers which all point to the same group of managed entities.

Obtaining an Entity Manager

An entity manager is always obtained from an `EntityManagerFactory`. The factory from which it was obtained determines the configuration parameters that govern its operation. While there are shortcuts that veil the factory from the user view when running in a Java EE application server environment, in the Java SE environment we can use a simple bootstrap class called `Persistence`. The static `createEntityManagerFactory()` method in the `Persistence` class returns the `EntityManagerFactory` for the specified persistence unit name. The following example demonstrates creating an `EntityManagerFactory` for the persistence unit named "EmployeeService":

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("EmployeeService");
```

The name of the specified persistence unit "EmployeeService" passed into the `createEntityManagerFactory()` method identifies the given persistence unit configuration that determines such things as the connection parameters that entity managers generated from this factory will use when connecting to the database.

Now that we have a factory, we can easily obtain an entity manager from it. The following example demonstrates creating an entity manager from the factory that we acquired in the previous example:

```
EntityManager em = emf.createEntityManager();
```

With this entity manager, we are now in a position to start working with persistent entities.

Persisting an Entity

Persisting an entity is the operation of taking a transient entity, or one that does not yet have any persistent representation in the database, and storing its state so that it can be retrieved later. This is really the basis of persistence—creating state that may outlive the process that created it. We are going to start by using the entity manager to persist an instance of `Employee`. Here is a code example that does just that:

```
Employee emp = new Employee(158);
em.persist(emp);
```

The first line in this code segment is simply creating an `Employee` instance that we want to persist. If we ignore the sad fact that we seem to be employing a nameless individual and paying them nothing (we are setting only the id, not the name or salary) the instantiated `Employee` is just a regular Java object.

The next line obtains an entity manager and uses it to persist the entity. Calling `persist()` is all that is required to initiate it being persisted in the database. If the entity manager encounters a problem doing this, then it will throw an unchecked `PersistenceException`; otherwise the employee will be stored in the database. When the `persist()` call returns, `emp` will be a managed entity within the entity manager's persistence context.

Listing 2-3 shows how to incorporate this into a simple method that creates a new employee and persists it to the database.

Listing 2-3. *Method for Creating an Employee*

```
public Employee createEmployee(int id, String name, long salary) {
    Employee emp = new Employee(id);
    emp.setName(name);
    emp.setSalary(salary);
    em.persist(emp);
    return emp;
}
```

This method assumes the existence of an entity manager in the `em` field of the instance and uses it to persist the `Employee`. Note that we do not need to worry about the failure case in this example. It will result in a runtime `PersistenceException` being thrown, which will get propagated up to the caller.

Finding an Entity

Once an entity is in the database, then the next thing one typically wants to do is find it again. In this section we will show how an entity can be found using the entity manager. There is really only one line that we need to show:

```
Employee emp = em.find(Employee.class, 158);
```

We are passing in the class of the entity that is being sought (in this example we are looking for an instance of `Employee`) and the id or primary key that identifies the particular entity (in our case we want to find the entity that we just created). This is all the information needed by the entity manager to find the instance in the database, and when the call completes, the employee that gets returned will be a managed entity, meaning that it will exist in the current persistence context associated with the entity manager.

PARAMETERIZED TYPES

Another of the principal features included in Java SE 5 was the introduction of generics. The abstraction of Java types allowed them to be parameterized and used generically by a class or method. Such classes or methods that make use of type parameterization are called generic types or generic methods. An example of a generic class is one that defines a parameterized type variable in its definition. It could then use that type in the signature of its methods just as does the following generic class:

```
public class Holder<T> {
    T contents;
    public void setContents(T obj) { contents = obj; }
    public T getContents() { return contents; }
}
```

This `Holder` class is parameterized by the `T` type variable making it possible to create an instance that can hold a given type. Why is this better than simply using `Object` everywhere where `T` is used? The reason is because once the type is supplied and the `Holder` is instantiated to be of a given type, then only instances of that type will be allowed to be stored. This makes any given `Holder` instance strongly typed for the type of our choice. For example, we can do the following:

```
Holder<String> stringHolder = new Holder<String>();
stringHolder.setContents("MyOwnString");
Holder<Integer> intHolder = new Holder<Integer>();
intHolder.setContents(100);
String s = stringHolder.getContents();
stringHolder.setContents(101); // compile error
```

We have a `Holder` that stores `String` objects or anything we want, but once we define it then we get the strong compile-time type checking that frees us from having to type-check at runtime. `ClassCastExceptions` can be a thing of the past (well, almost!). As an added bonus, we don't have to cast. The `getContents()` generic method returns precisely the type that was passed to `Holder` as the type parameter, so the compiler can type-check and safely assign as needed.

You may have noticed that there is no cast required to make the return result an `Employee` object, even though the `find()` method call can be used for any type of entity. Those who have used Java SE 5 will recognize that this is just because the return type of the `find()` method is parameterized to return the same class that was passed in, so if `Employee` was passed as the entity class, then it will also be the return type.

What happens if the object has been deleted or if we supplied the wrong id by accident? In the event that the object was not found, then the `find()` call simply returns `null`. We would need to ensure that a null check is performed before the next time the `emp` variable is used.

The code for a method that looks up and returns the `Employee` with a given id is now trivial and shown in Listing 2-4.

Listing 2-4. *Method for Finding an Employee*

```
public Employee findEmployee(int id) {
    return em.find(Employee.class, id);
}
```

In the case where no employee exists for the id that is passed in, then the method will return `null`, since that is what `find()` will return.

Removing an Entity

Removal of an entity from the database is not as common a thing as some might think. Many applications simply never delete objects, or if they do they just flag the data as being out of date or no longer valid and then just keep it out of sight of clients. We are not talking about that kind of application-level logical removal, where the data is not actually even removed from the database. We are talking about something that results in a `DELETE` statement being made across one or more tables.

In order to remove an entity, the entity itself must be managed, meaning that it is present in the persistence context. This means that the calling application should have already loaded or accessed the entity and is now issuing a command to remove it. This is not normally a problem given that most often the application will have caused it to become managed as part of the process of determining that this was the object that it wanted to remove.

A simple example for removing an employee is:

```
Employee emp = em.find(Employee.class, 158);
em.remove(emp);
```

In this example we are first finding the entity using the `find()` call, which returns a managed instance of `Employee`, and then removing the entity using the `remove()` call on the entity manager. Of course, we learned in the previous section that if the entity was not found then the `find()` method will return null. We would get a `java.lang.IllegalArgumentException` if it turned out that we passed null into the `remove()` call because we forgot to include a null check before calling `remove()`.

In our application method for removing an employee, we can fix the problem by checking for the existence of the employee before we issue the `remove()` call, as shown in Listing 2-5.

Listing 2-5. *Method for Removing an Employee*

```
public void removeEmployee(int id) {
    Employee emp = em.find(Employee.class, id);
    if (emp != null) {
        em.remove(emp);
    }
}
```

This method will ensure that the employee with the given id is removed from the database. It will return successfully whether the employee exists or not.

Updating an Entity

An entity may be updated in a few different ways, but for now we will illustrate the most common and simple case. This is the case where we have a managed entity and want to make changes to it. If we do not have a reference to the managed entity, then we must first obtain one using `find()` and then perform our modifying operations on the managed entity. This code adds $1,000 to the salary of the employee with id 158:

```
Employee emp = em.find(Employee.class, 158);
emp.setSalary(emp.getSalary() + 1000);
```

Note the difference between this operation and the others. In this case we are not calling into the entity manager to modify the object but directly on the object itself. For this reason it is important that the entity be a managed instance, otherwise the persistence provider will have no means of detecting the change, and no changes will be made to the persistent representation of the employee.

Our method to raise the salary of a given employee will take the id and amount of the raise, find the employee, and change the salary to the adjusted one. Listing 2-6 demonstrates this approach.

Listing 2-6. *Method for Updating an Employee*

```
public Employee raiseEmployeeSalary(int id, long raise) {
    Employee emp = em.find(Employee.class, id);
    if (emp != null) {
        emp.setSalary(emp.getSalary() + raise);
    }
    return emp;
}
```

If we can't find the employee, then we return `null` so the caller will know that no change could be made. We indicate success by returning the updated employee.

Transactions

The keen reader may have noticed something in the code to this point that was inconsistent with earlier statements made about transactionality when working with entities. There were no

transactions in any of the above examples, even though we said that changes to entities must be made persistent using a transaction.

In all the examples except the one that only called `find()`, we assume that a transaction enclosed each method. The `find()` call is not a mutating operation, so it may be called any time, with or without a transaction.

Once again, the key is the environment in which the code is being executed. The typical situation when running inside the Java EE container environment is that the standard Java Transaction API (JTA) is used. The transaction model when running in the container is to assume the application will ensure that a transactional context is present when one is required. If a transaction is not present, then either the modifying operation will throw an exception or the change will simply never be persisted to the data store. We will come back to discussing transactions in the Java EE environment in more detail in Chapter 3.

In our example in this chapter, though, we are not running in Java EE. We are in a Java SE environment, and the transaction service that should be used in Java SE is the `EntityTransaction` service. When executing in Java SE we either need to begin and to commit the transaction in the operational methods, or we need to begin and to commit the transaction before and after calling an operational method. In either case, a transaction is started by calling `getTransaction()` on the entity manager to get the `EntityTransaction` and then invoking `begin()` on it. Likewise, to commit the transaction the `commit()` call is invoked on the `EntityTransaction` obtained from the entity manager. For example, starting and committing before and after the method would produce code that creates an employee the way it is done in Listing 2-7.

Listing 2-7. *Beginning and Committing an* `EntityTransaction`

```
em.getTransaction().begin();
createEmployee(158, "John Doe", 45000);
em.getTransaction().commit();
```

Further detail about resource-level transactions and the `EntityTransaction` API are contained in Chapter 5.

Queries

In general, given that most developers have used a relational database at some point or other in their lives, most of us pretty much know what a database query is. In the Java Persistence API, a query is similar to a database query, except that instead of using Structured Query Language (SQL) to specify the query criteria, we are querying over entities and using a language called Java Persistence Query Language (which we will abbreviate as JPQL).

A query is implemented in code as a `Query` object. `Query` objects are constructed using the `EntityManager` as a factory. The `EntityManager` interface includes a variety of API calls that return a new `Query` object. As a first class object, this query can in turn be customized according to the needs of the application.

A query can be defined either *statically* or *dynamically*. A static query is defined in either annotation or XML metadata, and it must include both the query criteria as well as a user-assigned name. This kind of query is also called a *named query*, and it is later looked up by its name at the time it is executed.

A dynamic query can be issued at runtime by supplying only the JPQL query criteria. These may be a little more expensive to execute because the persistence provider cannot do any

query preparation beforehand, but they are nevertheless very simple to use and can be issued in response to program logic or even user logic.

Following is an example showing how to create a query and then execute it to obtain all of the employees in the database. Of course this may not be a very good query to execute if the database is large and contains hundreds of thousands of employees, but it is nevertheless a legitimate example. The simple query is as follows:

```
Query query = em.createQuery("SELECT e FROM Employee e");
Collection emps = query.getResultList();
```

We create a `Query` object by issuing the `createQuery()` call on the `EntityManager` and passing in the JPQL string that specifies the query criteria. The JPQL string refers not to an `EMPLOYEE` database table but the `Employee` entity, so this query is selecting all `Employee` objects without filtering them any further. We will be diving into queries in Chapter 6 and JPQL in Chapters 6 and 7. You will see that you can be far more discretionary about which objects you want to be returned.

To execute the query we simply invoke `getResultList()` on it. This returns a `List` (a sub-interface of `Collection`) containing the `Employee` objects that matched the query criteria. Note that a `List<Employee>` is not returned. Unfortunately this is not possible, since no class is passed into the call, so no parameterization of the type is able to occur. The return type is inferred by the persistence provider as it processes the JPQL string. We could cast the result to a `Collection<Employee>`, however, to make a neater return type for the caller. Doing so, we can easily create a method that returns all of the employees, as shown in Listing 2-8.

Listing 2-8. *Method for Issuing a Query*

```
public Collection<Employee> findAllEmployees() {
    Query query = em.createQuery("SELECT e FROM Employee e");
    return (Collection<Employee>) query.getResultList();
}
```

This example shows how simple queries are to create, execute, and process, but what this example does not show is how powerful they are. In Chapter 6 we will examine many other extremely useful and interesting ways of defining and using queries in an application.

Putting It All Together

We can now take all of the methods that we have created and combine them into a class. The class will act like a service class, which we will call `EmployeeService`, and will allow us to perform operations on employees. The code should be pretty familiar by now. Listing 2-9 shows the complete implementation.

Listing 2-9. *Service Class for Operating on* `Employee` *Entities*

```
import javax.persistence.*;
import java.util.Collection;
```



```
public class EmployeeService {
    protected EntityManager em;

    public EmployeeService(EntityManager em) {
        this.em = em;
    }

    public Employee createEmployee(int id, String name, long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
    }

    public void removeEmployee(int id) {
        Employee emp = findEmployee(id);
        if (emp != null) {
            em.remove(emp);
        }
    }

    public Employee raiseEmployeeSalary(int id, long raise) {
        Employee emp = em.find(Employee.class, id);
        if (emp != null) {
            emp.setSalary(emp.getSalary() + raise);
        }
        return emp;
    }

    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);
    }

    public Collection<Employee> findAllEmployees() {
        Query query = em.createQuery("SELECT e FROM Employee e");
        return (Collection<Employee>) query.getResultList();
    }
}
```

This is a simple yet fully functional class that can be used to issue the typical CRUD (create, read, update, and delete) operations on `Employee` entities. This class requires that an entity manager is created and passed into it by the caller and also that any required transactions are begun and committed by the caller. This may seem strange at first, but decoupling the transaction logic from the operation logic makes this class more portable to the Java EE environment. We will revisit this example in the next chapter, where we focus on Java EE applications.

A simple main program that uses this service and performs all of the required entity manager creation and transaction management is shown in Listing 2-10.

Listing 2-10. *Using* `EmployeeService`

```
import javax.persistence.*;
import java.util.Collection;

public class EmployeeTest {

    public static void main(String[] args) {
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();
        EmployeeService service = new EmployeeService(em);

        //  create and persist an employee
        em.getTransaction().begin();
        Employee emp = service.createEmployee(158, "John Doe", 45000);
        em.getTransaction().commit();
        System.out.println("Persisted " + emp);

        // find a specific employee
        emp = service.findEmployee(158);
        System.out.println("Found " + emp);

        // find all employees
        Collection<Employee> emps = service.findAllEmployees();
        for (Employee e : emps)
            System.out.println("Found employee: " + e);

        // update the employee
        em.getTransaction().begin();
        emp = service.raiseEmployeeSalary(158, 1000);
        em.getTransaction().commit();
        System.out.println("Updated " + emp);

        // remove an employee
        em.getTransaction().begin();
        service.removeEmployee(158);
        em.getTransaction().commit();
        System.out.println("Removed Employee 158");

        // close the EM and EMF when done
        em.close();
        emf.close();
    }
}
```


Packaging It Up

Now that we know the basic building blocks of the Java Persistence API, we are ready to organize the pieces into an application that runs in Java SE. The only thing left to discuss is how to put it together so that it runs.

Persistence Unit

The configuration that describes the persistence unit is defined in an XML file called `persistence.xml`. Each persistence unit is named, so when a referencing application wants to specify the configuration for an entity it need only reference the name of the persistence unit that defines that configuration. A single `persistence.xml` file may contain one or more named persistence unit configurations, but each persistence unit is separate and distinct from the others, and they can be logically thought of as being in separate `persistence.xml` files.

Many of the persistence unit elements in the `persistence.xml` file apply to persistence units that are deployed within the Java EE container. The only ones that we need to specify for our example are `name`, `transaction-type`, `class`, and `properties`. There are a number of other elements that can be specified in the persistence unit configuration in the `persistence.xml` file, but these will be discussed in more detail in Chapter 11. Listing 2-11 shows the relevant part of the `persistence.xml` file for this example.

Listing 2-11. *Elements in the* `persistence.xml` *File*

```
<persistence>
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
        <class>examples.model.Employee</class>
        <properties>
            <property name="toplink.jdbc.driver"
                      value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="toplink.jdbc.url"
                      value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="toplink.jdbc.user" value="APP"/>
            <property name="toplink.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

The `name` element indicates the name of our persistence unit and is the string that we specify when we create the `EntityManagerFactory`. We have used "EmployeeService" as the name. The `transaction-type` element indicates that our persistence unit uses resource level `EntityTransaction` instead of JTA transactions. The `class` element lists the entity that is part of the persistence unit. Multiple `class` elements may be specified when there is more than one entity. These would not normally be needed when deploying in a Java EE container, but they are needed for portable execution when running in Java SE. We only have a single `Employee` entity.

The last part that we use is a list of properties that are vendor-specific. The login parameters to a database must be specified when running in a Java SE environment, so these properties exist

to tell the provider what to connect to. Other provider properties, such as logging options, are also useful.

Persistence Archive

The persistence artifacts are packaged in what we will loosely call a *persistence archive*. This is really just a JAR-formatted file that contains the `persistence.xml` file in the `META-INF` directory and normally the entity class files.

Since we are running as a simple Java SE application, all we have to do is put the application JAR, the persistence provider JARs, and the Java Persistence API JAR on the classpath when the program is executed.

Summary

In this chapter we discussed just enough of the basics of the Java Persistence API to develop and run a simple application in a Java SE runtime.

We started out discussing the entity, how to define one, and how to turn an existing Java class into one. We discussed entity managers and how they are obtained and constructed in the Java SE environment.

The next step was to instantiate an entity instance and use the entity manager to persist it in the database. After we inserted some new entities, we were able to retrieve them again and then remove them. We also made some updates and ensured that the changes were written back to the database.

We talked about the resource-local transaction API and how to use it. We then went over some of the different types of queries and how to define and execute them. Finally, we aggregated all of these techniques and combined them into a simple application that we can execute in isolation from an enterprise environment.

In the next chapter, we will look at the impact of the Java EE environment when developing enterprise applications using the Java Persistence API.

CHAPTER 3

Enterprise Applications

No technology exists in a vacuum, and the Java Persistence API is no different than any other in this regard. Although the fat-client style of application demonstrated in the previous chapter is a viable use of the Java Persistence API, the vast majority of enterprise Java applications are deployed to a Java EE application server. Therefore it is essential to understand the components that make up a Java EE application and the role of the Java Persistence API in this environment.

We will begin with an overview of the major Java EE technologies relevant to persistence. As part of this overview, we will also detour into the EJB component model, demonstrating the new syntax for stateless, stateful, and message-driven beans. Even if you have experience with previous versions of these components, you may find this section helpful to get up to speed with the changes in EJB 3.0 and Java EE 5. As part of the ease-of-development initiative for Java EE 5, EJBs have undergone a major revision and have become considerably easier to implement in the process.

Although this chapter is not a complete or detailed exploration of Java EE, it will hopefully serve as a sufficient overview to the new simplified programming interfaces. We will introduce features only briefly and spend the bulk of the chapter focusing on the elements relevant to developing applications that use persistence.

Next we will look at the other application server technologies that are going to have a major impact on applications using the Java Persistence API: transactions and dependency management. Transactions, of course, are a fundamental element of any enterprise application that needs to ensure data integrity. The new dependency-management facilities of Java EE 5 are also key to understanding how the entity manager is acquired by enterprise components and how these components can be linked together.

Finally, we will demonstrate how to use the Java EE components described in this chapter, with a focus on how persistence integrates into each component technology. We will also revisit the Java SE application from the previous chapter and retarget it to the Java EE 5 platform.

Application Component Models

The word *component* has taken on many meanings in software development, so let's begin with a definition. A component is a self-contained, reusable software unit that can be integrated into an application. Clients interact with components via a well-defined contract. In Java, the simplest form of software component is the JavaBean, commonly referred to as just a *bean*. Beans are components implemented in terms of a single class whose contract is defined

by the naming patterns of the methods on the bean. The JavaBean naming patterns are so common now that it is easy to forget that they were originally intended to give user-interface builders a standard way of dealing with third-party components.

In the enterprise space, components focus more on implementing business services, with the contract of the component defined in terms of the business operations that may be carried out by that component. The standard component model for Java EE is the Enterprise JavaBeans model, which defines ways to package, deploy, and interact with self-contained business services. The EJB's type determines the contract required to interact with it. Session beans use standard Java interfaces to define the set of business methods that may be invoked, while message-driven bean behavior is determined by the type and format of the messages the bean is designed to receive.

Choosing whether or not to use a component model in your application is largely a personal preference. With some exceptions, most of the container services available to session beans are also available to servlets. As a result, many web applications today sidestep EJBs entirely, going directly from servlets to the database. Using components requires organizing the application into layers, with business services living in the component model and presentation services layered on top of it.

Historically, one of the challenges in adopting components in Java EE was the complexity of implementing them. With that problem largely solved, we are left with the benefits that a well-defined set of business services brings to an application:

- **Loose coupling.** Using components to implement services encourages loose coupling between layers of an application. The implementation of a component may change without any impact to the clients or other components that depend on it.
- **Dependency management.** Dependencies for a component may be declared in metadata and automatically resolved by the container.
- **Life cycle management.** The life cycle of components is well defined and managed by the application server. Component implementations can participate in life cycle operations to acquire and release resources or perform other initialization and shutdown behavior.
- **Declarative container services.** Business methods for components are intercepted by the application server in order to apply services such as concurrency, transaction management, security, and remoting.
- **Portability.** Components that comply to Java EE standards and that are deployed to standards-based servers can be more easily ported from one compliant server to another.
- **Scalability and reliability.** Application servers are designed to ensure that components are managed efficiently with an eye to scalability. Depending on the component type and server configuration, business operations implemented using components can retry failed method calls or even fail over to another server in a cluster.

One of the themes you will encounter as you read this book is the tendency for example code to be written often in terms of session beans. This is intentional. Not only are session beans easy to write and a good way to organize application logic, but they are also a natural fit for interacting with the Java Persistence API. In fact, as web application frameworks continue to push application code further away from the servlet, the ability for session beans to seamlessly integrate and acquire the services of other components makes them more valuable today than ever before.

Session Beans

Session beans are a component technology designed to encapsulate business services. The operations supported by the service are defined using a regular Java interface, referred to as the *business interface* of the session bean, that clients use to interact with the bean. The bean implementation is little more than a regular Java class which implements the business interface. And yet, by virtue of being part of the Enterprise JavaBeans component model, the bean has access to a wide array of container services that it can leverage to implement the business service. The significance of the name *session bean* has to do with the way in which clients access and interact with them. Once a client acquires a reference to a session bean from the server, it starts a session with that bean and may invoke business operations on it.

There are two types of session bean, *stateless* and *stateful*. Interaction with a stateless session bean begins at the start of a business method call and ends when the method call completes. There is no state that carries over from one business operation to the other. An interaction with stateful session beans becomes more of a conversation that begins from the moment the client acquires a reference to the session bean and ends when the client explicitly releases it back to the server. Business operations on a stateful session bean may maintain state on the bean instance across calls. We will provide more detail on the implementation considerations of this difference in interaction style as we describe each type of session bean.

Clients never interact directly with a session bean instance. The client references and invokes an implementation of the business interface provided by the server. This implementation class acts as a proxy to the underlying bean implementation. This decoupling of client from bean allows the server to intercept method calls in order to provide the services required by the bean, such as transaction management. It also allows the server to optimize and reuse instances of the session bean class as necessary.

Stateless Session Beans

As we mentioned, a stateless session bean sets out to accomplish the goals of an operation entirely within the lifetime of a single method. Stateless beans may implement many business operations, but each method cannot assume that any other was invoked before it.

This might sound like a limitation of the stateless bean, but it is by far the most common form of business service implementation. Unlike stateful session beans, which are good for accumulating state during a conversation (such as the shopping cart of a retail application), stateless session beans are designed to carry out independent operations very efficiently. Stateless session beans may scale to large numbers of clients with minimal impact to overall server resources.

Defining a Stateless Session Bean

A session bean is defined in two parts:

- One or more business interfaces that define what methods a client may invoke on the bean
- A class that implements these interfaces, called the bean class, which is marked with the `@Stateless` annotation

Most session beans have only a single business interface, but there is no restriction on the number of interfaces that a session bean may expose to its clients. When the server encounters the `@Stateless` annotation, it knows to treat the bean class as a session bean. It will configure the bean in the EJB container and make it available for use by other components in the application. The `@Stateless` annotation and other annotations described in this chapter are defined in the `javax.ejb` and `javax.annotation` packages.

Let's look at a complete implementation of a stateless session bean. Listing 3-1 shows the business interface that will be supported by this session bean. In this example, the service consists of a single method, `sayHello()`, which accepts a `String` argument corresponding to a person's name and returns a `String` response. There is no annotation or parent interface to indicate that this is a business interface. When implemented by the session bean, it will be automatically treated as a local business interface, meaning that it is accessible only to clients within the same application server. A second type of business interface for remote clients is discussed later in the section Remote Business Interfaces. To emphasize that an interface is a local business interface, the `@Local` annotation may be optionally added to the interface.

Listing 3-1. *The Business Interface for a Session Bean*

```
public interface HelloService {
    public String sayHello(String name);
}
```

Now let's consider the implementation, which is shown in Listing 3-2. This is a regular Java class that implements the `HelloService` business interface. The only thing unique about this class is the `@Stateless` annotation that marks it as a stateless session bean. The business method is implemented without any special constraints or requirements. This is a regular class that just happens to be an EJB.

Listing 3-2. *The Bean Class Implementing the* `HelloService` *Interface*

```
@Stateless
public class HelloServiceBean implements HelloService {
    public String sayHello(String name) {
        return "Hello, "  + name;
    }
}
```


There are only a couple of caveats about the stateless session bean class definition. The first is that it needs a no-arg constructor, but the compiler normally generates this automatically when no other constructors are supplied. The second is that static fields should not be used, primarily because of bean redeployment issues.

Many EJB containers create a pool of stateless session bean instances and then select an arbitrary instance to service each client request. Therefore there is no guarantee that the same state will be used between calls, and hence it cannot be relied on. Any state placed on the bean class should be restricted to factory classes that are inherently stateless, such as `DataSource`.

Lifecycle Callbacks

Unlike a regular Java class used in application code, the server manages the life cycle of a stateless session bean. This impacts the implementation of a bean in two ways.

First, the server decides when to create and remove bean instances. The application has no control over when or even how many instances of a particular stateless session bean are created or how long they will stay around.

Second, the server has to initialize services for the bean after it is constructed but before the business logic of the bean is invoked. Likewise, the bean may have to acquire a resource such as a JDBC data source before business methods can be used. However, in order for the bean to acquire a resource, the server must first have completed initializing its services for the bean. This limits the usefulness of the constructor for the class since the bean won't have access to any resources until server initialization has completed.

To allow both the server and the bean to achieve their initialization requirements, EJBs support lifecycle callback methods that are invoked by the server at various points in the bean's life cycle. For stateless session beans there are two lifecycle callbacks, PostConstruct and PreDestroy. The server will invoke the PostConstruct callback as soon as it has completed initializing all of the container services for the bean. In effect, this replaces the constructor as the location for initialization logic since it is only here that container services are guaranteed to be available. The server invokes the PreDestroy callback immediately before the server releases the bean instance to be garbage-collected. Any resources acquired during PostConstruct that require explicit shutdown should be released during PreDestroy.

Listing 3-3 shows a stateless session bean that acquires a reference to a `java.util.logging.Logger` instance during the PostConstruct callback. A bean may have at most one PostConstruct callback method[1] that is identified by the `@PostConstruct` marker annotation. Likewise, the PreDestroy callback is identified by the `@PreDestroy` annotation.

1. In inheritance situations, additional callback methods from parent classes may also be invoked.

Listing 3-3. *Using the PostConstruct Callback to Acquire a Logger*

```
@Stateless
public class LoggerBean implements Logger {
    private java.util.logging.Logger logger;

    @PostConstruct
    public void init() {
        logger = Logger.getLogger("notification");
    }

    public void logMessage(String message) {
        logger.info(message);
    }
}
```

Remote Business Interfaces

So far we have only discussed session beans that use a local business interface. Local in this case means that a dependency on the session bean may be declared only by Java EE components that are running together in the same application server instance. It is not possible to use a session bean with a local interface from a remote client, for example.

To accommodate remote clients, session beans may mark their business interface with the `@Remote` annotation to declare that it should be useable remotely. Listing 3-4 demonstrates this syntax for a remote version of the `HelloService` interface shown in Listing 3-1. Marking an interface as being remote is equivalent to having it extend the `java.rmi.Remote` interface. The reference to the bean that gets acquired by a client is no longer a local reference on the server but a Remote Method Invocation (RMI) stub that will invoke operations on the session bean from across the network. No special support is required on the bean class to use remote interfaces.

Listing 3-4. *A Remote Business Interface*

```
@Remote
public interface HelloServiceRemote {
    public String sayHello(String name);
}
```

Making an interface remote has consequences both in terms of performance and how arguments to business methods are handled. Remote business interfaces may be used locally within a running server, but doing so may still result in network overhead if the method call is routed through the RMI layer. Arguments to methods on remote interfaces are also *passed by value* instead of *passed by reference*. This means that the argument is serialized even when the client is local to the session bean. Local interfaces for local clients are generally a better approach. Local interfaces preserve the semantics of regular Java method calls and avoid the costs associated with networking and RMI.

■ **Tip** Many application servers provide options to improve the performance of remote interfaces when used locally within an application server. This may include the ability to disable serialization of method arguments or may go so far as to sidestep RMI entirely. Use caution when relying on these features in application code, as they are not portable across different application servers.

Stateful Session Beans

In our introduction to session beans we described the difference between stateless and stateful beans as being based on the interaction style between client and server. In the case of stateless session beans, that interaction started and ended with a single method call. Sometimes clients need to issue multiple requests to a service and have each request be able to access or consider the results of previous requests. Stateful session beans are designed to handle this scenario by providing a dedicated service to a client that starts when the client obtains a reference to the bean and ends only when the client chooses to end the conversation.

The quintessential example of the stateful session bean is the shopping cart of an e-commerce application. The client obtains a reference to the shopping cart, starting the conversation. Over the span of the user session, the client adds or removes items from the shopping cart, which maintains state specific to the client. Then, when the session is complete, the client completes the purchase, causing the shopping cart to be removed.

This is not unlike using a non-managed Java object in application code. We create an instance, invoke operations on the object that accumulate state, and then dispose of the object when we no longer need it. The only difference with the stateful session bean is that the server manages the actual object instance and the client interacts with that instance indirectly through the business interface of the bean.

Stateful session beans offer a superset of the functionality available in stateless session beans. The features that we covered for stateless session beans such as remote interfaces apply equally to stateful session beans.

Defining a Stateful Session Bean

Now that we have established the use case for a stateful session bean, let's look at how to define one. Similar to the stateless session bean, a stateful session bean is comprised of one or more business interfaces implemented by a single bean class. A sample local business interface for a shopping cart bean is demonstrated in Listing 3-5.

Listing 3-5. *Business Interface for a Shopping Cart*

```
public interface ShoppingCart {
    public void addItem(String id, int quantity);
    public void removeItem(String id, int quantity);
    public Map<String,Integer> getItems();
    public void checkout(int paymentId);
    public void cancel();
}
```

Listing 3-6 shows the bean class that implements the ShoppingCart interface. The bean class has been marked with the @Stateful annotation to indicate to the server that the class is a stateful session bean.

Listing 3-6. *Implementing a Shopping Cart Using a Stateful Session Bean*

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
    private HashMap<String,Integer> items = new HashMap<String,Integer>();

    public void addItem(String item, int quantity) {
        Integer orderQuantity = items.get(item);
        if (orderQuantity == null) {
            orderQuantity = 0;
        }
        orderQuantity += quantity;
        items.put(item, orderQuantity);
    }

    // ...

    @Remove
    public void checkout(int paymentId) {
        // store items to database
        // ...
    }

    @Remove
    public void cancel() {
    }
}
```

There are two things different in this bean compared to the stateless session beans we have been dealing with so far.

The first difference is that the bean class has state fields that are modified by the business methods of the bean. This is allowed because the client that uses the bean effectively has access to a private instance of the session bean on which to make changes.

The second difference is that there are methods marked with the @Remove annotation. These are the methods that the client will use to end the conversation with the bean. After one of these methods has been called, the server will destroy the bean instance, and the client reference will throw an exception if any further attempt is made to invoke business methods. Every stateful session bean must define at least one method marked with the @Remove annotation, even if the method doesn't do anything other than serve as an end to the conversation. In Listing 3-6, the checkout() method is called if the user completes the shopping transaction, while cancel() is called if the user decides not to proceed. The session bean is removed in either case.

Lifecycle Callbacks

Like the stateless session bean, the stateful session bean also supports lifecycle callbacks in order to facilitate bean initialization and cleanup. It also supports two additional callbacks to allow the bean to gracefully handle passivation and activation of the bean instance. Passivation is the process by which the server serializes the bean instance so that it can either be stored offline to free up resources or so that it can be replicated to another server in a cluster. Activation is the process of deserializing a passivated session bean instance and making it active in the server once again. Because stateful session beans hold state on behalf of a client and are not removed until the client invokes one of the remove methods on the bean, the server cannot destroy a bean instance to free up resources. Passivation allows the server to reclaim resources while preserving session state.

Before a bean is passivated, the server will invoke the PrePassivate callback. The bean uses this callback to prepare the bean for serialization, usually by closing any live connections to other server resources. The PrePassivate method is identified by the @PrePassivate marker annotation. After a bean has been activated, the server will invoke the PostActivate callback. With the serialized instance restored, the bean must then reacquire any connections to other resources that the business methods of the bean may be depending on. The PostActivate method is identified by the @PostActivate marker annotation. Listing 3-7 shows a session bean that makes full use of the lifecycle callbacks in order to maintain a JDBC connection. Note that only the JDBC Connection is explicitly managed. As a resource connection factory, the server automatically saves and restores the data source during passivation and activation.

Listing 3-7. *Using Lifecycle Callbacks on a Stateful Session Bean*

```
@Stateful
public class OrderBrowserBean implements OrderBrowser {
    DataSource ds;
    Connection conn;

    @PostConstruct
    public void init() {
        // acquire the data source
        // ...

        acquireConnection();
    }

    @PrePassivate
    public void passivate() { releaseConnection(); }

    @PostActivate
    public void activate() { acquireConnection(); }

    @PreDestroy
    public void shutdown() { releaseConnection(); }
```

```
    private void acquireConnection() {
        try {
            conn = ds.getConnection();
        } catch (SQLException e) {
            throw new EJBException(e);
        }
    }

    private void releaseConnection() {
        try {
            conn.close();
        } catch (SQLException e) {
        }
        conn = null;
    }

    public Collection<Order> listOrders() {
        // ...
    }
}
```

Message-Driven Beans

So far we have been looking at components that are synchronous in nature. The client invokes a method through the business interface, and the server completes that method invocation before returning control to the client. For the majority of services, this is the most natural approach. There are cases, however, where it is not necessary for the client to wait for a response from the server. We would like the client to be able to issue a request and continue while the server processes the request asynchronously.

The *message-driven bean* (MDB) is the EJB component for asynchronous messaging. Clients issue requests to the MDB using a messaging system such as Java Message Service (JMS). These requests are queued and eventually delivered to the MDB by the server. The server invokes the business interface of the MDB whenever it receives a message sent from a client. Whereas the component contract of a session bean is defined by its business interface, the component contract of an MDB is defined by the structure of the messages it is designed to receive.

Defining a Message-Driven Bean

When defining a session bean, the developer creates a business interface, and the bean class implements it. In the case of message-driven beans, the bean class implements an interface specific to the messaging system the MDB is based on. The most common case is JMS, but other messaging systems are possible with the Java Connector Architecture. For JMS message-driven beans, the business interface is `javax.jms.MessageListener`, which defines a single method, `onMessage()`.

Listing 3-8 shows the basic structure of a message-driven bean. The `@MessageDriven` annotation marks the class as an MDB. The activation configuration properties, defined using the `@ActivationConfigProperty` annotations, tell the server the type of messaging system and

any configuration details required by that system. In this case the MDB will be invoked only if the JMS message has a property named `RECIPIENT` where the value is `ReportProcessor`. Whenever the server receives a message, it invokes the `onMessage()` method with the message as the argument. Because there is no synchronous connection with a client, the `onMessage()` method does not return anything. However, the MDB can use session beans, data sources, or even other JMS resources to process and carry out an action based on the message.

Listing 3-8. *Defining a JMS Message-Driven Bean*

```
@MessageDriven(
  activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType",
                              propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="messageSelector",
                              propertyValue="RECIPIENT='ReportProcessor'")
})
public class ReportProcessorBean implements javax.jms.MessageListener {
    public void onMessage(javax.jms.Message message) {
        // ...
    }
}
```

Servlets

Servlets are a component technology designed to serve the needs of web developers who need to respond to HTTP requests and generate dynamic content in return. Servlets are the oldest and most popular technology introduced as part of the Java EE platform. They are the foundation for technologies such as JavaServer Pages (JSP) and the backbone of web frameworks such as Apache Struts and JavaServer Faces (JSF).

Although it is quite likely that readers will have some experience with servlets, it is worth describing the impact that web application models have had on enterprise application development. The web, due to its reliance on the HTTP protocol, is inherently a stateless medium. Much like the stateless session beans we described earlier, a client makes a request, the server triggers the appropriate service method in the servlet, and content is generated and returned to the client. Each request is entirely independent from the last.

This presents a challenge, because many web applications involve some kind of conversation between the client and the server in which the previous actions of the user influence the results returned on subsequent pages. To maintain that conversational state, many early applications attempted to dynamically embed context information into URLs. Unfortunately not only does this technique not scale very well, it requires a dynamic element to all content generation that makes it difficult for non-developers to write content for a web application.

Servlets solve the problem of conversational state with the *session*. Not to be confused with the session bean, the HTTP session is a map of data associated with a session id. When the application requests that a session be created, the server generates a new id and returns an `HTTPSession` object that the application can use to store key/value pairs of data. It then uses techniques such as browser cookies to link the session id with the client, tying the two together

into a conversation. For web applications, the client is largely ignorant of the conversational state that is tracked by the server.

Using the HTTP session effectively is an important element of servlet development. Listing 3-9 demonstrates the steps required to request a session and store conversational data in it. In this example, assuming that the user has logged in, the servlet stores the user id in the session, making it available for use in all subsequent requests by the same client. The `getSession()` call on the `HttpServletRequest` object will either return the active session or create a new one if one does not exist. Once obtained, the session acts like a map, with key/value pairs set and retrieved with the `setAttribute()` and `getAttribute()` methods respectively. As we see later in this chapter, the servlet session, which stores unstructured data, is sometimes paired with a stateful session bean in order to manage session information with the benefit of a well-defined business interface.

Listing 3-9. *Maintaining Conversational State with a Servlet*

```
public class LoginServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String userId = request.getParameter("user");
        HttpSession session = request.getSession();
        session.setAttribute("user", userId);

        // ...
    }
}
```

The rise of application frameworks targeted to the web has also changed the way in which we develop web applications. Application code written in servlets is rapidly being replaced with application code further abstracted from the base model using frameworks like JavaServer Faces. When working in an environment such as this, basic application persistence issues such as where to acquire and store the entity manager and how to effectively use transactions quickly become more challenging.

Although we will explore some of these issues, persistence in the context of a framework such as JavaServer Faces is beyond the scope of this book. As a general solution, we recommend adopting a session bean component model in which to focus persistence operations. Session beans are easily accessible from anywhere within a Java EE application, making them perfect neutral ground for business services. The ability to exchange entities inside and outside of the session bean model means that the results of persistence operations will be directly usable in web frameworks without having to tightly couple your presentation code to the persistence API.

Dependency Management

The business logic of a Java EE component is not always self-contained. More often than not, the implementation depends on other resources hosted by the application server. This may

include server resources such as a JDBC data source or JMS message queue, or application-defined resources such as a session bean or entity manager for a specific persistence unit.

To manage these dependencies, Java EE components support the notion of *references* to resources that are defined in metadata for the component. A reference is a named link to a resource that may be resolved dynamically at runtime from within application code or resolved automatically by the container when the component instance is created. We'll cover each of these scenarios shortly.

A reference consists of two parts: a name and a target. The name is used by application code to resolve the reference dynamically while the server uses target information to find the resource the application is looking for. The type of resource to be located determines the type of information required to match the target. Each resource reference requires a different set of information specific to the resource type it refers to.

A reference is declared using one of the resource reference annotations: `@Resource`, `@EJB`, `@PersistenceContext`, or `@PersistenceUnit`. These annotations may be placed on a class, field, or setter method. The choice of location determines the default name of the reference and whether or not the server resolves the reference automatically.

Dependency Lookup

The first strategy for resolving dependencies in application code that we will discuss is called *dependency lookup*. This is the traditional form of dependency management in Java EE, where the application code is responsible for using the Java Naming and Directory Interface (JNDI) to look up a named reference.

All of the resource annotations support an element called `name` that defines the name of the reference. When the resource annotation is placed on the class definition, this element is mandatory. If the resource annotation is placed on a field or a setter method, the server will generate a default name. When using dependency lookup, annotations are typically placed at the class level, and the name is explicitly specified. Placing a resource reference on a field or setter method has other effects besides generating a default name that we will discuss in the next section.

The role of the name is to provide a way for the client to resolve the reference dynamically. Every Java EE application server supports JNDI, and each component has its own locally scoped JNDI naming context called the environment naming context. The name of the reference is bound into the environment naming context, and when it is looked up using the JNDI API, the server resolves the reference and returns the target of the reference.

Consider the DeptServiceBean session bean shown in Listing 3-10. It has declared a dependency on a session bean using the @EJB annotation and given it the name "audit". The beanInterface element of the @EJB annotation references the business interface of the session bean that the client is interested in. In the PostConstruct callback, the audit bean is looked up and stored in the audit field. The Context and InitialContext interfaces are both defined by the JNDI API. The lookup() method of the Context interface is the primary way to retrieve objects from a JNDI context. To find the reference named "audit", the application looks up the name "java:comp/env/audit" and casts the result to the AuditService business interface. The prefix "java:comp/env/" that was added to the reference name indicates to the server that the environment naming context should be searched to find the reference. If the name is incorrectly specified, an exception will be thrown when the lookup fails.

Listing 3-10. *Looking Up an EJB Dependency*

```
@Stateless
@EJB(name="audit", beanInterface=AuditService.class)
public class DeptServiceBean implements DeptService {
    private AuditService audit;

    @PostConstruct
    public void init() {
        try {
            Context ctx = new InitialContext();
            audit = (AuditService) ctx.lookup("java:comp/env/audit");
        } catch (NamingException e) {
            throw new EJBException(e);
        }
    }

    // ...
}
```

Using the JNDI API to look up resource references from the environment naming context is supported by all Java EE components. It is, however, a somewhat cumbersome method of finding a resource due to the exception-handling requirements of JNDI. EJBs also support an alternative syntax using the lookup() method of the EJBContext interface. The EJBContext interface (and subinterfaces such as SessionContext and MessageDrivenContext) is available to any EJB and provides the bean with access to runtime services such as the timer service. Listing 3-11 shows the same example as Listing 3-10 using the lookup() method. The SessionContext instance in this example is provided via a setter method. We will revisit this example later in the section called Referencing Server Resources to see how it is invoked.

Listing 3-11. *Using the* EJBContext lookup() *Method*

```
@Stateless
@EJB(name="audit", beanInterface=AuditService.class)
public class DeptServiceBean implements DeptService {
    SessionContext context;
    AuditService audit;

    public void setSessionContext(SessionContext context) {
        this.context = context;
    }

    @PostConstruct
    public void init() {
        audit = (AuditService) context.lookup("audit");
    }

    // ...
}
```

The EJBContext lookup() method has two advantages over the JNDI API. The first is that the argument to the method is the name exactly as it was specified in the resource reference. The second is that only runtime exceptions are thrown from the lookup() method so the checked exception handling of the JNDI API can be avoided. Behind the scenes the exact same sequence of JNDI API calls from Listing 3-10 is being made, but the JNDI exceptions are handled automatically.

Dependency Injection

When a resource annotation is placed on a field or setter method, two things occur. First, a resource reference is declared just as if it had been placed on the bean class (similar to the example in Listing 3-10), and the name for that resource will be bound into the environment naming context when the component is created. Second, the server does the lookup automatically on your behalf and sets the result into the instantiated class.

The process of automatically looking up a resource and setting it into the class is called *dependency injection* because the server is said to inject the resolved dependency into the class. This technique, one of several commonly referred to as *inversion of control*, removes the burden of manually looking up resources from the JNDI environment context.

Dependency injection is rapidly being considered a best practice for application development, not only because it reduces the need for JNDI lookups (and the associated Service Locator[2] pattern), but also because it simplifies testing. Without any JNDI API code in the class that has dependencies on the application server runtime environment, the bean class may be instantiated directly in a unit test. The developer can then manually supply the required dependencies and test the functionality of the class in question instead of worrying about how to work around the JNDI APIs.

2. Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*. Upper Saddle River, N.J.: Prentice Hall PTR, 2003.

Field Injection

The first form of dependency injection is called *field injection*. Injecting a dependency into a field means that after the server looks up the dependency in the environment naming context, it assigns the result directly into the annotated field of the class. Listing 3-12 revisits the example from Listing 3-10 and demonstrates the `@EJB` annotation, this time by injecting the result into the `audit` field. All of the directory interface code we demonstrated before is gone, and the business methods of the bean can assume that the audit field holds a reference to the `AuditService` bean.

Listing 3-12. *Using Field Injection*

```
@Stateless
public class DeptServiceBean implements DeptService {
    @EJB AuditService audit;

    // ...
}
```

Field injection is certainly the easiest to implement, and the examples in this book use this form exclusively to conserve space. The only thing to consider with field injection is that if you are planning on unit testing, then you need either to add a setter method or to make the field accessible to your unit tests in order to manually satisfy the dependency. Private fields, though legal, require unpleasant hacks if there is no accessible way to set their value. Consider package scope for field injection if you want to unit test without having to add a setter.

We mentioned in the previous section that a name is automatically generated for the reference when a resource annotation is placed on a field or setter method. For completeness, we will describe the format of this name, but it is unlikely that you will find many opportunities to use it. The generated name is the fully qualified class name followed by a forward slash and then the name of the field or property. This means that if the `AuditService` bean is located in the `persistence.session` package, then the injected EJB referenced in Listing 3-12 would be accessible in the environment naming context under the name "persistence.session.AuditService/audit". Specifying the `name` element for the resource annotation will override this default value.

Setter Injection

The second form of dependency injection is called *setter injection* and involves annotating a setter method instead of a class field. When the server resolves the reference, it will invoke the annotated setter method with the result of the lookup. Listing 3-13 revisits Listing 3-10 for the last time to demonstrate using setter injection.

Listing 3-13. *Using Setter Injection*

```
@Stateless
public class DeptServiceBean implements DeptService {
    private AuditService audit;

    @EJB
    public void setAuditService(AuditService audit) {
        this.audit = audit;
    }

    // ...
}
```

This style of injection allows for private fields yet also works well with unit testing. Each test can simply instantiate the bean class and manually perform the dependency injection by invoking the setter method, usually by providing an implementation of the required resource that is tailored to the test.

Declaring Dependencies

The following sections describe the resource annotations defined by the Java EE and EJB specifications. Each annotation has a `name` element for optionally specifying the reference name for the dependency. Other elements on the annotations are specific to the type of resource that needs to be acquired.

Referencing a Persistence Context

In the previous chapter we demonstrated how to create an entity manager for a persistence context using an `EntityManagerFactory` returned from the `Persistence` class. In the Java EE environment, the `@PersistenceContext` annotation may be used to declare a dependency on a persistence context and have the entity manager for that persistence context acquired automatically.

Listing 3-14 demonstrates using the `@PersistenceContext` annotation to acquire an entity manager through dependency injection. The `unitName` element specifies the name of the persistence unit on which the persistence context will be based.

■ **Tip** If the `unitName` element is omitted, it is vendor-specific how the unit name for the persistence context is determined. Some vendors may provide a default value if there is only one persistence unit for an application, while others may require that the unit name be specified in a vendor-specific configuration file.

Listing 3-14. *Injecting an* `EntityManager` *Instance*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    // ...
}
```

After the warnings about using a state field in a stateless session bean, you may be wondering how this code is legal. After all, entity managers must maintain their own state in order to be able to manage a specific persistence context. The good news is that the specification was designed with Java EE integration in mind, so what actually gets injected in Listing 3-14 is not an entity manager instance like the ones we used in the previous chapter. The value injected into the bean is a container-managed proxy that acquires and releases persistence contexts on behalf of the application code. This is a powerful feature of the Java Persistence API in Java EE and one we will cover extensively in Chapter 5. For now it is safe to assume that the injected value will "do the right thing." It does not have to be disposed of and works automatically with the transaction management of the application server.

Referencing a Persistence Unit

The `EntityManagerFactory` for a persistence unit may be referenced using the `@PersistenceUnit` annotation. Like the `@PersistenceContext` annotation, the `unitName` element identifies the persistence unit for the `EntityManagerFactory` instance we wish to access. If the persistent unit name is not specified in the annotation, then it is vendor-specific how the name is determined.

Listing 3-15 demonstrates injection of an `EntityManagerFactory` instance into a stateful session bean. The bean then creates an `EntityManager` instance from the factory during the PostConstruct lifecycle callback. An injected `EntityManagerFactory` instance may be safely stored on any component instance. It is thread-safe and does not need to be disposed of when the bean instance is removed.

Listing 3-15. *Injecting an* `EntityManagerFactory` *Instance*

```
@Stateful
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceUnit(unitName="EmployeeService")
    private EntityManagerFactory emf;
    private EntityManager em;

    @PostConstuct
    public void init() {
        em = emf.createEntityManager();
    }

    // ...
}
```


The `EntityManagerFactory` for a persistence unit is not used very often in the Java EE environment since injected entity managers are easier to acquire and use. As we will see in Chapter 5, there are important differences between the entity managers returned from the factory and the ones provided by the server in response to the `@PersistenceContext` annotation.

Referencing Enterprise JavaBeans

When a component needs to access an EJB, it declares a reference to that bean with the `@EJB` annotation. The target of this reference type is typically a session bean. Message-driven beans have no client interface, so they cannot be accessed directly and cannot be injected. We have already demonstrated the `beanInterface` element for specifying the business interface of the session bean that the client is interested in. The server will search through all deployed session beans to find the one that implements the requested business interface.

In the rare case that two session beans implement the same business interface or if the client needs to access a session bean located in a different EJB jar, then the `beanName` element may also be specified to identify the session bean by its name. The name of a session bean defaults to the unqualified class name of the bean class, or it may be set explicitly by using the `name` element of the `@Stateless` and `@Stateful` annotations. Listing 3-16 revisits the example shown in Listing 3-12, this time specifying the `beanName` element on the injected value. Sharing the same business interface across multiple bean implementations is not recommended. The `beanName` element should almost never be required.

Listing 3-16. *Qualifying an EJB Reference Using the Bean Name*

```
@Stateless
public class DeptServiceBean implements DeptService {
    @EJB(beanName="AuditServiceBean")
    AuditService audit;

    // ...
}
```

Referencing Server Resources

The `@Resource` annotation is the catchall reference for all resource types that don't correspond to one of the types we have described so far. It is used to define references to resource factories, message destinations, data sources, and other server resources. The `@Resource` annotation is also the simplest to define, as the only additional element is `resourceType`, which allows you to specify the type of resource if the server can't figure it out automatically. For example, if the field you are injecting into is of type Object, then there is no way for the server to know that you wanted a data source instead. The `resourceType` element can be set to `javax.sql.DataSource` to make the need explicit.

One of the features of the `@Resource` annotation is that it is used to acquire logical resources specific to the component type. This includes `EJBContext` implementations as well as services such as the EJB timer service. Without defining it as such, we used setter injection to acquire the `EJBContext` instance in Listing 3-11. To make that example complete, the `@Resource` annotation would be placed on the `setSessionContext()` method. Listing 3-17 revisits the example from Listing 3-11, this time demonstrating field injection to acquire a `SessionContext` instance.

Listing 3-17. *Injecting a* `SessionContext` *instance*

```
@Stateless
@EJB(name="audit", beanInterface=AuditService.class)
public class DeptServiceBean implements DeptService {
    @Resource SessionContext context;
    AuditService audit;

    @PostConstruct
    public void init() {
        audit = (AuditService) context.lookup("audit");
    }

    // ...
}
```

Transaction Management

More than any other type of enterprise application, applications that use persistence require careful attention to issues of transaction management. When transactions start, when they end, and how the entity manager participates in container-managed transactions are all essential topics for developers using the Java Persistence API. In the following sections we will lay out the foundation for transactions in Java EE and then revisit this topic in detail again in Chapter 5 as we look at the entity manager and how it participates in transactions. Advanced transaction topics are beyond the scope of this book. We recommend *Java Transaction Processing*[3] for an in-depth discussion on using and implementing transactions in Java.

Transaction Review

A transaction is an abstraction that is used to group together a series of operations. Once grouped together, the set of operations is treated as a single unit, and all of the operations must succeed or none of them can succeed. The consequence of only some of the operations being successful would produce an inconsistent view of the data that would be harmful or undesirable to the application. The term used to describe whether the operations succeed together or not at all is called *atomicity* and is arguably the most important of the four basic properties that

3. Little, Mark, Jon Maron, and Greg Pavlik. *Java Transaction Processing: Design and Implementation*. Upper Saddle River, N.J.: Prentice Hall PTR, 2004.

are used to characterize how transactions behave. Understanding these four properties is fundamental to understanding transactions. The following list summarizes these properties:

- **Atomicity:** All of the operations in a transaction are successful or none of them are. The success of every individual operation is tied to the success of the entire group.
- **Consistency:** The resulting state at the end of the transaction adheres to a set of rules that define acceptability of the data. The data in the entire system is legal or valid with respect to the rest of the data in the system.
- **Isolation:** Changes made within a transaction are visible only to the transaction that is making the changes. Once a transaction commits the changes they are atomically visible to other transactions.
- **Durability:** The changes made within a transaction endure beyond the completion of the transaction.

A transaction that meets all of these requirements is said to be an ACID transaction (the familiar ACID term being obtained by combining the first letter of each of the four properties).

Not all transactions are ACID transactions, and those that are often offer some flexibility in the fulfillment of the ACID properties. For example, the isolation level is a common setting that can be configured to provide either looser or tighter degrees of isolation than what was described earlier. These are typically done for reasons of either increased performance or, on the other side of the spectrum, if an application has more stringent data consistency requirements. The transactions that we discuss in the context of Java EE are normally of the ACID variety.

Enterprise Transactions in Java

Transactions actually exist at different levels within the enterprise application server. The lowest and most basic transaction is at the level of the resource, which in our discussion is assumed to be a relational database fronted by a `DataSource` interface. This is called a *resource-local transaction* and is equivalent to a database transaction. These types of transactions are manipulated by interacting directly with the JDBC `DataSource` that is obtained from the application server. Resource-local transactions are used much more infrequently than container transactions.

The broader *container transaction* uses the Java Transaction API (JTA) that is available in every compliant Java EE application server. This is the typical transaction that is used for enterprise applications and may involve or *enlist* a number of resources including data sources as well as other types of transactional resources. Resources defined using Java Connector Architecture (J2C) components may also be enlisted in the container transaction.

Containers typically add their own layer on top of the JDBC `DataSource` to perform functions such as connection management and pooling that make more efficient use of the resources and provide a seamless integration with the transaction-management system. This is also necessary because it is the responsibility of the container to perform the commit or rollback operation on the data source when the container transaction completes.

Because container transactions use JTA and because they may span multiple resources, they are also called *JTA transactions* or *global transactions*. The container transaction is a central aspect of programming within Java EE application servers.

Transaction Demarcation

Every transaction has a beginning and an end. Beginning a transaction will allow subsequent operations to become a part of the same transaction until the transaction has completed. Transactions may be completed in one of two ways. They may be committed, causing all of the changes to be persisted to the data store, or rolled back, indicating that the changes should be discarded. The act of causing a transaction to either begin or complete is termed *transaction demarcation*. This is a critical part of writing enterprise applications, since doing transaction demarcation incorrectly is one of the most common sources of performance degradation.

Resource-local transactions are always demarcated explicitly by the application, while container transactions may either be demarcated automatically by the container or by using a JTA interface that supports application-controlled demarcation. In the first case, when the container takes over the responsibility of transaction demarcation, we call it container-managed transaction management, but when the application is responsible for demarcation we call it bean-managed transaction management.

EJBs may use either container-managed transactions or bean-managed transactions. Servlets are limited to the somewhat poorly named bean-managed transaction. The default transaction management style for an EJB component is container-managed. To configure an EJB to have its transactions demarcated one way or the other, the `@TransactionManagement` annotation should be specified on the session or message-driven bean class. The TransactionManagementType enumerated type defines `BEAN` for bean-managed transactions and `CONTAINER` for container-managed transactions. Listing 3-18 demonstrates how to enabled bean-managed transactions using this approach.

Listing 3-18. *Changing the Transaction Management Type of a Bean*

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ProjectServiceBean implements ProjectService {
    // methods in this class manually control transaction demarcation
}
```

Since the default transaction management for a bean is container-managed, this annotation needs to be specified only if bean-managed transactions are desired.

Container-Managed Transactions

The most common way to demarcate transactions is to use container-managed transactions (CMTs). This spares the application the effort and code to begin and commit transactions explicitly.

Transaction requirements are determined by metadata on session and message-driven beans and are configurable at the granularity of method execution. For example, a session bean may declare that whenever any specific method on that bean gets invoked, then the container must ensure that a transaction is started before the method begins. The container would also be responsible for committing the transaction after the completion of the method.

It is quite common for one bean to invoke another bean from one or more of its methods. In this case a transaction that may have been started by the calling method would not have been committed, because the calling method will not be completed until its call to the second

bean has completed. This leads to the requirement to have settings for defining how the container should behave when a method is invoked within a specific transactional context.

For example, if a transaction is already in progress when a method is called, then the container may be expected to just make use of that transaction, whereas it may be directed to start a new one if no transaction is active. These settings are called *transaction attributes*, and they determine exactly what the container-managed transactional behavior is.

The defined transaction attributes choices are

- `MANDATORY`: If this attribute is specified for a method, then a transaction is expected to have already been started and be active when the method is called. If no transaction is active, then an exception is thrown. This attribute is seldom used but can be a development tool to catch transaction demarcation errors in cases where it is expected that a transaction should already have been started.
- `REQUIRED`: This is the most common case where a method is expected to be in a transaction. The container provides a guarantee that a transaction is active for the method. If one is already active, then that one is used, but if one does not exist, then a new transaction is created for the method execution.
- `REQUIRES_NEW`: This is used when the method always needs to be in its own transaction, that is, the method should be committed or rolled back independent of methods further up the call stack. It should be used with caution, as it can lead to excessive transaction overhead.
- `SUPPORTS`: Methods marked with Supports are not dependent upon a transaction but will tolerate running inside one if it exists. This is an indicator that no transactional resources are accessed in the method.
- `NOT_SUPPORTED`: A method marked to not support transactions will cause the container to suspend the current transaction if one is active when the method is called. It implies that the method does not perform transactional operations but may fail in other ways that could undesirably affect the outcome of a transaction. This is not a commonly used attribute.
- `NEVER`: A method marked to never support transactions will cause the container to throw an exception if a transaction is active when the method is called. This attribute is very seldom used but can be a development tool to catch transaction demarcation errors in cases when it is expected that transactions should already have been completed.

Any time the container starts a transaction for a method, the container is assumed to also attempt to commit the transaction at the end of the method. Each time the current transaction must be suspended, then the container is responsible for resuming the suspended transaction at the conclusion of the method.

The transaction attribute for a method may be indicated by annotating a session or message-driven bean class, or one of its methods that is part of the business interface, with the `@TransactionAttribute` annotation. This annotation requires a single argument of the enumerated type `TransactionAttributeType`, the values of which are defined in the preceding list. Annotating the bean class will cause the transaction attribute to apply to all of the business methods in the class, while annotating a method applies the attribute only to the method. If both class-level and method-level annotations exist, then the method-level annotation takes

precedence. In the absence of class-level or method-level `@TransactionAttribute` annotations, the default attribute of `REQUIRED` will be applied.

Listing 3-19 shows how the `addItem()` method from the shopping cart bean in Listing 3-6 might use a transaction attribute. No transaction management setting was supplied, so container-managed transactions will be used. No attribute was specified on the class, so the default behavior of `REQUIRED` will apply to all of the methods of the class. The exception is that the `addItem()` method has declared a transaction attribute of `SUPPORTS`, which overrides the `REQUIRED` setting. Whenever a call to add an item is made, then that item will be added to the cart, but if no transaction was active then none will need to be started.

Listing 3-19. *Specifying a Transaction Attribute*

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public void addItem(String item, Integer quantity) {
        verifyItem(item, quantity);
        // ...
    }

    // ...
}
```

Furthermore, before the `addItem()` method adds the item to the cart, it does some validation in a private method called `verifyItem()` that is not shown in the example. When this method is invoked from `verifyItem()`, it will run in whatever transactional context `addItem()` was invoked.

Any bean wanting to cause a container-managed transaction to roll back may do so by invoking the `setRollbackOnly()` method on the `EJBContext` object. While this will not cause the immediate rollback of the transaction, it is an indication to the container that the transaction should be rolled back when the time comes. Note that entity managers will also cause the current transaction to be set to roll back when an exception is thrown during an entity manager invocation or when the transaction completes.

Bean-Managed Transactions

The other way of demarcating transactions is to use bean-managed transactions (BMT). Declaring that a bean is using bean-managed transactions means that the bean class is assuming the responsibility to begin and commit the transactions whenever it deems it's necessary. With this responsibility, however, comes the expectation that the bean class will get it right. Beans that use BMT must ensure that any time a transaction has been started, it must also be completed before returning from the method that started it. Failure to do so will result in the container rolling back the transaction automatically and an exception being thrown.

One penalty of transactions being managed by the application instead of by the container is that they do not get propagated to methods called on another BMT bean. For example, if Bean A begins a transaction and then calls Bean B, which is using bean-managed transactions, then the transaction will not get propagated to the method in Bean B. Any time a transaction is active when a BMT method is invoked, the active transaction will be suspended until control returns to the calling method.

BMT is not generally recommended for use in EJBs because it adds complexity to the application and requires the application to do work that the server can already do for it. It is necessary, though, when transactions must be initiated from the web tier, since it is the only supported way that non-EJB components can use container transactions.

UserTransaction

In order to be able to manually begin and commit container transactions, the application must have an interface that supports it. The `UserTransaction` interface is the designated object in the JTA that applications can hold on to and invoke to manage transaction boundaries. An instance of `UserTransaction` is not actually the current transaction instance but is a sort of proxy that provides the transaction API and represents the current transaction. A `UserTransaction` instance may be injected into BMT components by using the `@Resource` annotation. When using dependency lookup, it is found in the environment naming context using the reserved name "java:comp/UserTransaction". The `UserTransaction` interface is shown in Listing 3-20.

Listing 3-20. *The* `UserTransaction` *Interface*

```
public interface javax.transaction.UserTransaction {
    public abstract void begin();
    public abstract void commit();
    public abstract int getStatus();
    public abstract void rollback();
    public abstract void setRollbackOnly();
    public abstract void setTransactionTimeout(int seconds);
}
```

Each JTA transaction is associated with an execution thread, so it follows that no more than one transaction can be active at any given time. So if one transaction is active, the user cannot start another one in the same thread until the first one has committed or rolled back. Alternatively, the transaction may time out, causing the transaction to roll back.

We discussed earlier that in certain CMT conditions the container will suspend the current transaction. From the previous API you can see that there is no `UserTransaction` method for suspending a transaction. Only the container can do this using an internal transaction management API. In this way multiple transactions can be associated with a single thread, even though only one can ever be active at a time.

Rollbacks may occur in several different scenarios. The `setRollbackOnly()` method indicates that the current transaction may not be committed, leaving rollback as the only possible outcome. The transaction may be rolled back immediately by calling the `rollback()` method. Alternately, a time limit for the transaction may be set with the `setTransactionTimeout()` method, causing the transaction to roll back when the limit is reached. The only catch with transaction timeouts is that the time limit must be set before the transaction starts and it cannot be changed once the transaction is in progress.

In JTA every thread has a transactional status that can be accessed through the `getStatus()` call. The return value of this method is one of the constants defined on the `java.transaction.Status` interface. If no transaction is active, for example, then the value

returned by `getStatus()` will be the `STATUS_NO_TRANSACTION`. Likewise if `setRollbackOnly()` has been called on the current transaction, then the status will be `STATUS_MARKED_ROLLBACK` until the transaction has begun rolling back.

Listing 3-21 shows a fragment from a servlet using the `ShoppingCart` bean in order to demonstrate using `UserTransaction` to invoke multiple EJB methods within a single transaction. The `doPost()` method uses the `UserTransaction` instance injected with the `@Resource` annotation to start and commit a transaction. Note the `try ... finally` block required around the transaction operations in order to ensure that the transaction is correctly cleaned up in the event of a failure.

Listing 3-21. *Using the* `UserTransaction` *Interface*

```
public class ProjectServlet extends HttpServlet {
    @Resource UserTransaction tx;
    @EJB ProjectService bean;

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // ...

        try {
            tx.begin();
            try {
                bean.assignEmployeeToProject(projectId, empId);
                bean.updateProjectStatistics();
            } finally {
                tx.commit();
            }
        } catch (Exception e) {
            // handle exceptions from UserTransaction methods
            // ...
        }

        // ...
    }
}
```

Using Java EE Components

Now that we have described how to define Java EE components and make use of services such as transaction management that are provided by the application server, we can demonstrate how to put these components to work. Once again we must caution that this is not an exhaustive overview of these technologies but is provided to put the upcoming persistence examples in context and preview the new features in Java EE 5 for developers who may be new to the platform.

Using a Stateless Session Bean

A client of a stateless session bean is any Java EE component that can declare a dependency on the bean. This includes other session beans, message-driven beans, and servlets. Two-tier access from a remote client is also possible if the bean defines a remote business interface.

Consider the servlet shown in Listing 3-22, which uses the EJB from Listing 3-2 to obtain a message and then generates a simple HTML page. As we discussed earlier in the section on Dependency Management, the `@EJB` annotation causes the `HelloService` bean to be automatically injected into the servlet. Therefore when the `doGet()` method is invoked, methods on the business interface can be invoked without any extra steps.

Listing 3-22. *A Servlet That Uses a Session Bean*

```
public class HelloServlet extends HttpServlet {
    @EJB HelloService bean;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        String name = request.getParameter("name");
        String message = bean.sayHello(name);

        PrintWriter out = response.getWriter();
        out.println("<html>" +
                    "<head><title>Hello</title></head>" +
                    "<body><p>" + message + "</p></body>" +
                    "</html>");
    }
}
```

The use of annotations to manage dependencies is an important change in Java EE 5 that significantly reduces the complexity of weaving together components within applications. In the case of session beans that depend on other session beans, note that it is always safe to declare a reference to a stateless session bean and store it in a field on the bean. The bean reference in the case of a stateless session bean is itself a stateless and thread-safe object.

Using a Stateful Session Bean

There are a few basic things to keep in mind when working with stateful session beans:

1. When a client obtains a reference to a stateful session bean, a private instance of that bean is created for the client. In other words, there is one bean instance per client reference.
2. The bean does not go away until the client invokes a method annotated with `@Remove`. If the client forgets or is unable to end the conversation with the bean, it will hang around until the server can determine that it is safe to remove it.
3. A reference to a stateful session bean cannot be shared between threads.

A consequence of these rules is that clients need to plan carefully on when they need to start the session and when it can be ended. It also means that using the `@EJB` annotation to inject a stateful session bean is not a good solution. Servlets, stateless session beans, and message-driven beans are all stateless components. As we stated before in the description of stateless session beans, that means that any state placed on a stateless component must also be stateless as well. A stateful session bean reference is itself stateful because it references a private instance of the bean managed by the server. If `@EJB` were used to inject a stateful session bean into a stateless session bean where the server had pooled 100 bean instances, then there would be 100 stateful session bean instances created as well. The only time it is ever safe to inject a stateful session bean is into another stateful session bean.

Dependency lookup is the preferred method for acquiring a stateful session bean instance for a stateless client. The `EJBContext lookup()` method is the easiest way to accomplish this, but JNDI will be required if the client is a servlet. Listing 3-23 demonstrates a typical pattern for servlets using stateful session beans. A reference is declared to the bean, it is looked up lazily when needed, and the result is bound to the HTTP session. The stateful session bean and HTTP session have similar life cycles, making them good candidates to work together.

Listing 3-23. *Creating and Using a Stateful Session Bean*

```
@EJB(name="cart", beanInterface=ShoppingCart.class)
public class ShoppingCartServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        ShoppingCart cart = (ShoppingCart) session.getAttribute("cart");
        if (cart == null) {
            try {
                Context ctx = new InitialContext();
                cart = (ShoppingCart) ctx.lookup("java:comp/env/cart");
                session.setAttribute("cart", cart);
            } catch (NamingException e) {
                throw new ServletException(e);
            }
        }

        if (request.getParameter("action").equals("add")) {
            String itemId = request.getParameter("item");
            String quantity = request.getParameter("quantity");
            cart.addItem(itemId, Integer.parseInt(quantity));
        }
```



```
        if (request.getParameter("action").equals("cancel")) {
            cart.cancel();
            session.removeAttribute("cart");
        }

        // ...
    }
}
```

When the server receives a request to look up a stateful session bean, it asks the EJB container to create a new instance of the bean, which is then assigned a unique identifier. The reference to the bean that is returned keeps track of this identifier and uses it when communicating with the server to ensure that the right bean instance is used to invoke each business method.

Using a Message-Driven Bean

As an asynchronous component, clients of a message-driven bean can't directly invoke business operations. Instead they send messages, which are then delivered to the MDB by the messaging system being used. The client needs to know only the format of the message that the MDB is expecting and the messaging destination where the message must be sent. Listing 3-24 demonstrates sending a message to the MDB we defined in Listing 3-8. The `ReportProcessor` MDB expects an employee id as its message format. Therefore the session bean client in this example creates a text message with the employee id and sends it through the JMS API. The same criteria that was specified on the MDB to filter the messages is also specified here on the client.

Listing 3-24. *Sending a Message to an MDB*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @Resource Queue destinationQueue;
    @Resource QueueConnectionFactory factory;

    public void generateReport() {
        try {
            QueueConnection connection = factory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false, 0);
            QueueSender sender = session.createSender(destinationQueue);

            Message message = session.createTextMessage("12345");
            message.setStringProperty("RECIPIENT", "ReportProcessor");

            sender.send(message);
```

```
            sender.close();
            session.close();
            connection.close();
        } catch (JMSException e) {
            // ...
        }
    }

    // ...
}
```

Adding the Entity Manager

Using stateless session beans as components to manage persistence operations is the preferred strategy for Java EE applications. Clients gain the benefit of working with a session façade that presents a business interface that is decoupled from the specifics of the implementation. The bean is able to leverage the dependency-management capabilities of the server to access the entity manager and can make use of services such as container-managed transactions to precisely specify the transaction requirements of each business operation. Finally, the POJO nature of entities allows them to be easily returned from and passed as arguments to a session bean method.

Leveraging the stateless session bean for persistence is largely a case of injecting an entity manager. Listing 3-25 demonstrates a typical session bean that injects an entity manager and uses it to implement its business operations.

Listing 3-25. *Using the Entity Manager with a Stateless Session Bean*

```
@Stateless
public class DepartmentServiceBean {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public void addEmployeeToDepartment(int empId, int deptId) {
        Employee emp = em.find(Employee.class, empId);
        Department dept = em.find(Deptartment.class, deptId);
        dept.getEmployees().add(emp);
        emp.setDept(dept);
    }

    // ...
}
```

Stateful session beans are also well suited to managing persistence operations within an application component model. The ability to store state on the session bean means that query criteria or other conversational state can be constructed across multiple method calls before being acted upon. The results of entity manager operations may also be cached on the bean instance in some situations.

Listing 3-26 revisits the shopping cart bean from Listing 3-6. In this example `Order` and `Item` are entities representing a sales transaction. The order is built up incrementally over the life of the session and then persisted to the database using the injected entity manager when payment has been confirmed.

Listing 3-26. *Using the Entity Manager with a Stateful Session Bean*

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
    @PersistenceContext(unitName="order")
    private EntityManager em;
    private Order order = new Order();

    public void addItem(Item item, int quantity) {
        order.addItem(item, quantity);
    }

    // ...

    @Remove
    public void checkout(int paymentId) {
        order.setPaymentId(paymentId);
        em.persist(order);
    }
}
```

From the perspective of using the entity manager with message-driven beans, the main question is whether or not the MDB should use the Java Persistence API directly or delegate to another component such as a session bean. A common pattern in many applications is to treat the MDB as an asynchronous façade for session beans in situations where the business logic does not produce results that are customer-facing, that is, where the results of the business operation are stored in a database or propagated to another messaging system. This is largely an issue of personal taste, as message-driven beans fully support injecting the entity manager and can leverage container-managed transactions.

Putting It All Together

Now that we have discussed the application component model and services available as part of a Java EE application server, we can revisit the `EmployeeService` example from the previous chapter and port it to the Java EE environment. Along the way, we'll provide example code to show how the components fit together and how they relate back to the Java SE example.

Defining the Component

To begin, let's consider the definition of the `EmployeeService` class from Listing 2-9 in the previous chapter. The goal of this class is to provide business operations related to the maintenance of employee data. In doing so, it encapsulates all of the persistence operations. To

introduce this class into the Java EE environment, we must first decide how it should be represented. The service pattern exhibited by the class suggests the session bean as the ideal component. Since the business methods of the bean have no dependency on each other, we can further decide that a stateless session bean is suitable. In fact, this bean demonstrates a very typical design pattern called a Session Façade,[4] in which a stateless session bean is used to shield clients from dealing with a particular persistence API. Our first step is to extract a business interface from the original bean. Listing 3-27 shows the `EmployeeService` business interface.

Listing 3-27. *The* `EmployeeService` *Business Interface*

```
public interface EmployeeService {
    public Employee createEmployee(int id, String name, long salary);
    public void removeEmployee(int id);
    public Employee changeEmployeeSalary(int id, long newSalary);
    public Employee findEmployee(int id);
    public Collection<Employee> findAllEmployees();
}
```

In the Java SE example, the `EmployeeService` class must create and maintain its own entity manager instance. We can replace this logic with dependency injection to acquire the entity manager automatically. Having decided on a stateless session bean and dependency injection, the converted stateless session bean is demonstrated in Listing 3-28. With the exception of how the entity manager is acquired, the business methods are identical. This is an important feature of the Java Persistence API, as the same `EntityManager` interface can be used both inside and outside of the application server.

Listing 3-28. *The* `EmployeeService` *Session Bean*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    protected EntityManager em;

    public EntityManager getEntityManager() {
        return em;
    }

    public Employee createEmployee(int id, String name, long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        getEntityManager().persist(emp);
        return emp;
    }
```

4. Alur et al., *Core J2EE Patterns*.


```
    public void removeEmployee(int id) {
        Employee emp = findEmployee(id);
        if (emp != null) {
            getEntityManager().remove(emp);
        }
    }

    public Employee changeEmployeeSalary(int id, long newSalary) {
        Employee emp = findEmployee(id);
        if (emp != null) {
            emp.setSalary(newSalary);
        }
        return emp;
    }

    public Employee findEmployee(int id) {
        return getEntityManager().find(Employee.class, id);
    }

    public Collection<Employee> findAllEmployees() {
        Query query = getEntityManager().createQuery("SELECT e FROM Employee e");
        return (Collection<Employee>) query.getResultList();
    }
}
```

Defining the User Interface

The next question to consider is how the bean will be accessed. A web interface is the standard presentation method for modern enterprise applications. To demonstrate how this stateless session bean might be used by a servlet, consider Listing 3-29. The request parameters are interpreted to determine the action, which is then carried out by invoking methods on the injected `EmployeeService` bean. Although only the first action is described, you can see how this could easily be extended to handle each of the operations defined on the `EmployeeService` business interface.

Listing 3-29. *Using the* `EmployeeService` *Session Bean from a Servlet*

```
public class EmployeeServlet extends HttpServlet {
    @EJB EmployeeService bean;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response) {
        String action = request.getParameter("action");
```

```
        if (action.equals("create")) {
            String id = request.getParameter("id");
            String name = request.getParameter("name");
            String salary = request.getParameter("salary");
            bean.createEmployee(Integer.parseInt(id), name,
                                Long.parseLong(salary));
        }

        // ...
    }
}
```

Packaging It Up

In the Java EE environment, many properties required in the `persistence.xml` file for Java SE may be omitted. In Listing 3-30 we see the `persistence.xml` file from Listing 2-11 in the previous chapter converted for deployment as part of a Java EE application. Instead of JDBC properties for creating a connection, we now declare that the entity manager should use the data source name "jdbc/EmployeeDS". The `transaction-type` attribute has also been removed to allow the persistence unit to default to JTA. The application server will automatically find entity classes, so even the list of classes has been removed. This example represents the ideal minimum Java EE configuration.

Since the business logic that uses this persistence unit is implemented in a stateless session bean, the `persistence.xml` file would typically be located in the `META-INF` directory of the corresponding EJB JAR. We will fully describe the `persistence.xml` file and its placement within a Java EE application later in Chapter 11.

Listing 3-30. *Defining a Persistence Unit in Java EE*

```
<persistence>
    <persistence-unit name="EmployeeService">
        <jta-data-source>jdbc/EmployeeDS</jta-data-source>
    </persistence-unit>
</persistence>
```

Summary

It would be impossible to provide details on all of the features of the Java EE platform in a single chapter. However, is it likewise difficult to put the Java Persistence API in context without understanding the application server environment in which it will be used. Therefore over the course of this chapter we have attempted to introduce the technologies that are of the most relevance to the developer using persistence in enterprise applications.

We began with an introduction to software component models and introduced the Enterprise JavaBeans model for enterprise components. We argued that the use of components is more important than ever before and identified some of the benefits that come from leveraging this approach.

In the section on session beans, we introduced the fundamentals and then looked in detail at both stateless and stateful session beans. We learned about the difference in interaction style between the two session types and looked at the syntax for declaring beans. We also looked at the difference between local and remote business interfaces.

We next looked at dependency management in Java EE application servers. We discussed the reference annotation types and how to declare them. We also looked at the difference between dependency lookup and dependency injection. In the case of injection we looked at the difference between field and setter injection. Finally, we explored each of the resource types demonstrating how to acquire server and Java Persistence API resources.

In the section on transaction management, we looked at the Java Transaction API and its role in building datacentric applications. We then looked at the difference between bean-managed transactions and container-managed transactions for EJBs. We documented the different types of transaction attributes for CMT beans and showed how to manually control bean-managed transactions.

Finally, we concluded the chapter by exploring how to use Java EE components in applications and how they can leverage the Java Persistence API. We also discussed an end-to-end example of the Java Persistence API in the Java EE environment, converting the example application introduced in the previous chapter from a command-line Java SE application to a web-based application running on an application server.

Now that we have introduced the Java Persistence API in both the Java SE and Java EE environments, it's time to dive into the specification in detail. In the next chapter we begin this journey with the central focus of the Java Persistence API, object-relational mapping.

CHAPTER 4

Object-Relational Mapping

The largest part of an API that persists objects to a relational database ends up being the object-relational mapping component. The topic of object-relational mapping usually includes everything from how the object state is mapped to the database columns to how to issue queries across the objects. We are focusing this chapter primarily on how to define and map entity state to the database, emphasizing the simple manner in which it may be done.

In this chapter we will introduce the basics of mapping fields to database columns and then go on to show how to map and automatically generate entity identifiers. We will go into some detail about different kinds of relationships and show examples that demonstrate how they are mapped from the domain model to the data model.

Persistence Annotations

We have shown in previous chapters how annotations have been used extensively both in the EJB 3.0 and Java Persistence API specifications. We are going to discuss in significant detail persistence and mapping metadata, and since we use annotations to explain the concepts, it is worth reviewing a few things about the annotations before we get started.

Persistence annotations may be applied at three different levels: at the class, method, and field levels. To annotate any of these, the annotation must be placed in front of the code definition of the artifact being annotated. In some cases we will put them on the same line just before the class, method, or field, and in other cases we will put them on the line above. It is based completely upon the preferences of the person applying the annotations, and if you have not already noticed, we think it makes sense to do one thing in some cases and the other in other cases. It depends on how long the annotation is and what the most readable format seems to be.

The Java Persistence API annotations were designed to provide maximum readability, be easy to specify, and be flexible enough to allow different combinations of metadata. Most annotations are specified as siblings instead of being nested inside each other. As with all trade-offs, the piper must be paid however, and the cost of flexibility is that many possible permutations of top-level metadata will be syntactically correct but semantically invalid. The compiler will be of no use, but the provider runtime will often do some basic checking for some improper annotation groupings. The nature of annotations, however, is that when they are unexpected, then they will often just not get noticed at all. This is worth remembering when attempting to understand behavior that may not match what you thought you specified in the annotations. It could be that one or more of the annotations are just being ignored.

The mapping annotations can be categorized as being in one or the other of the two categories: logical annotations and physical annotations. The annotations in the logical group are those that describe the entity model from an object modeling view. They are tightly bound to the domain model and are the sort of metadata that you might want to specify in UML or any other object modeling language or framework. The physical annotations relate to the concrete data model in the database. They deal with tables, columns, constraints, and other database-level artifacts that the object model might never otherwise be aware of.

We will make use of both types of annotations throughout the examples and to demonstrate the mapping metadata. Understanding and being able to distinguish between these two levels of metadata will better qualify you to make decisions about where to declare metadata, and where to use annotations and XML. As we will see in Chapter 10, there are XML equivalents to all of the mapping annotations described in this chapter, giving you the freedom to use the approach that best suits your development needs.

Accessing Entity State

The mapped state of an entity must be accessible to the provider at runtime, so that when it comes time to write the data out, it can be obtained from the entity instance and stored in the database. Similarly, when the state is loaded from the database, the provider runtime must be able to insert it into a new entity instance. The way the state is accessed in the entity is called the *access mode*.

In Chapter 2 we saw briefly that there were two different ways that could be used to specify persistent entity state: we could either annotate the fields or annotate the JavaBean-style properties. The mechanism that we use to designate the persistent state is the same as the access mode that the provider uses to access that state. If we annotate fields, then the provider will get and set the fields of the entity using reflection. If the annotations are set on the getter methods of properties, then those getter and setter methods will be invoked by the provider to access and set the state.

■ **Tip** Some vendors may support annotating a mixture of fields and properties on the same entity class. Be careful of relying upon this kind of behavior, as it could cause your entity class definitions to be non-portable.

Field Access

Annotating the fields of the entity will cause the provider to use field access to get and set the state of the entity. Getter and setter methods may or may not be present, but if they are present, they are ignored by the provider. All fields must be declared as either `protected`, `package`, or `private`. Public fields are disallowed because it would open up the state fields to access by any unprotected class in the VM. Doing so is not just an obviously bad practice but could also defeat the provider implementation. Other classes must use the methods of an entity in order to access its persistent state.

The example in Listing 4-1 shows the `Employee` entity being mapped using field access. The `@Id` annotation indicates not only that the `id` field is the persistent identifier or primary key for the entity but also that field access should be assumed. The `name` and `salary` fields are then defaulted to being persistent, and they get mapped to columns of the same name.

Listing 4-1. *Using Field Access*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

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

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

    public long getSalary() { return salary; }
    public void setSalary(long salary) { this. salary = salary; }
}
```

Property Access

When property access mode is used, the same contract as for JavaBeans applies, and there must be getter and setter methods for the persistent properties. The type of the property is determined by the return type of the getter method and must be the same as the type of the single parameter passed into the setter method. Both methods must be either `public` or `protected` visibility. The mapping annotations for a property must be on the getter method.

In Listing 4-2 the `Employee` class has an `@Id` annotation on the `getId()` getter method so the provider will use property access to get and set the state of the entity. The `name` and `salary` properties will be made persistent by virtue of the getter and setter methods that exist for them and will be mapped to `NAME` and `SALARY` columns respectively. Note that the `salary` property is backed by the `wage` field, which does not share the same name. This goes unnoticed by the provider, since by specifying property access, we are telling the provider to ignore the entity fields and use only the getter and setter methods for naming.

Listing 4-2. *Using Property Access*

```
@Entity
public class Employee {
    private int id;
    private String name;
    private long wage;
```

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

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

    public long getSalary() { return wage; }
    public void setSalary(long salary) { this.wage = salary; }
}
```

Mapping to a Table

We saw in Chapter 2 that mapping an entity to a table in the simplest case does not need any mapping annotations at all. Only the `@Entity` and `@Id` annotations need to be specified to create and map an entity to a database table.

In those cases the default table name, which is just the unqualified name of the entity class, was perfectly suitable. If it happens that the default table name is not the name that we like, or if a suitable table that contains the state already exists in our database with a different name, then we must specify the name of the table. We do this by annotating the entity class with the `@Table` annotation and by including the name of the table using the `name` element. Many databases have terse names for tables. Listing 4-3 shows an entity that is mapped to a table that has a name different than its class name.

Listing 4-3. *Overriding the Default Table Name*

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

■ **Tip** Default names are not specified to be either uppercase or lowercase. Most databases are not case-sensitive, so it won't generally matter whether a vendor uses the case of the entity name or converts it to uppercase.

The `@Table` annotation provides the ability to not only name the table that the entity state is being stored in but also to name a database schema or catalog. The schema name is commonly used to differentiate one set of tables from another and is indicated by using the `schema` element. Listing 4-4 shows an `Employee` entity that is mapped to the `EMP` table in the `HR` schema.

Listing 4-4. *Setting a Schema*

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }
```


When specified, the schema name will be prepended to the table name when the persistence runtime goes to the database to access the table. In this case the `HR` schema will be prepended to the `EMP` table each time the table is accessed.

■ **Tip** Some vendors may allow the schema to be included in the name element of the table without having to specify the schema element—for example, `@Table(name="HR.EMP")`. Support for inlining the name of the schema with the table name is non-standard.

Some databases support the notion of a catalog. For these databases, the `catalog` element of the `@Table` annotation may be specified. Listing 4-5 shows a catalog being explicitly set for the `EMP` table.

Listing 4-5. *Setting a Catalog*

```
@Entity
@Table(name="EMP", catalog="HR")
public class Employee { ... }
```

Mapping Simple Types

Simple Java types are mapped as part of the immediate state of an entity in its fields or properties. The list of persistable types is quite lengthy and includes pretty much every type that you would want to persist. They are

- **Primitive Java types:** `byte`, `int`, `short`, `long`, `boolean`, `char`, `float`, `double`
- **Wrapper classes of primitive Java types:** `Byte`, `Integer`, `Short`, `Long`, `Boolean`, `Character`, `Float`, `Double`
- **Byte and character array types:** `byte[]`, `Byte[]`, `char[]`, `Character[]`
- **Large numeric types:** `java.math.BigInteger`, `java.math.BigDecimal`
- **Strings:** `java.lang.String`
- **Java temporal types:** `java.util.Date`, `java.util.Calendar`
- **JDBC temporal types:** `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`
- **Enumerated types:** Any system or user-defined enumerated type
- **Serializable objects:** Any system or user-defined serializable type

Sometimes the type of the database column being mapped to is not exactly the same as the Java type. In almost all cases the provider runtime can convert the type returned by JDBC into the correct Java type of the attribute. If the type from the JDBC layer cannot be converted to the Java type of the field or property, then an exception will normally be thrown, although it is not guaranteed.

Tip When the persistent type does not match the JDBC type, some providers may choose to take proprietary action or make a best guess to convert between the two. In other cases, the JDBC driver may be performing the conversion on its own.

When persisting a field or property, the provider looks at the type and ensures that it is one of the persistable types listedearlier. If it is in the list, the provider will persist it using the appropriate JDBC type and pass it through to the JDBC driver. At that point, if the field or property is not serializable, the result is unspecified. The provider may choose to throw an exception or just try and pass the object through to JDBC.

An optional `@Basic` annotation may be placed on a field or property to explicitly mark it as being persistent. This annotation is mostly for documentation purposes and is not required for the field or property to be persistent. Because of the annotation, we call mappings of simple types *basic mappings*.

Now that we have seen how we can persist either fields or properties and how they are virtually equivalent in terms of persistence, we will just call them *attributes*. An attribute is a field or property of a class, and we will use the term *attribute* from now on to avoid having to continually refer to fields or properties in specific terms when one or the other may apply.

Column Mappings

Where the persistent attributes can be thought of as being logical mappings that indicate that a given attribute is persistent, the physical annotation that is the companion annotation to the basic mapping is the `@Column` annotation. Specifying `@Column` on the attribute indicates specific characteristics of the physical database column that the object model is less concerned about. In fact, the object model might never even need to know to which column it is mapped, and the column name and physical mapping metadata may be located in a separate XML file.

A number of annotation elements may be specified as part of `@Column`, but most of them apply only to schema generation and will be covered in Chapter 9. The only one that is of consequence is the `name` element, which is just a string that specifies the name of the column that the attribute has been mapped to. This is used when the default column name is not appropriate or does not apply to the schema being used. We can think of the `name` element of the `@Column` annotation as a means of overriding the default column name that would have otherwise been applied.

The example in Listing 4-6 shows how we can override the default column name for an attribute.

Listing 4-6. *Mapping Attributes to Columns*

```
@Entity
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    @Column(name="SAL")
    private long salary;
    @Column(name="COMM")
    private String comments;
    // ...
}
```

To put these annotations in context, let's look at the full table mapping represented by this entity. The first thing that we notice is that no `@Table` annotation exists on the class, so the default table name of "EMPLOYEE" will be applied to it.

The next thing we see is that `@Column` can be used with `@Id` mappings as well as with basic mappings. The `id` field is being overridden to map to the `EMP_ID` column instead of the default `ID` column. The name field is not annotated with `@Column`, so the default column name `NAME` would be used to store and retrieve the employee name. The `salary` and `comments` fields, however, are annotated to map to the `SAL` and `COMM` columns, respectively. The `Employee` entity is therefore mapped to the table that is shown in Figure 4-1.

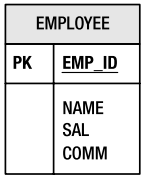

Figure 4-1. `EMPLOYEE` *entity table*

Lazy Fetching

On occasion, we know that certain portions of an entity will be seldom accessed. In these situations we can optimize the performance when retrieving the entity by fetching only the data that we expect to be frequently accessed. We would like the remainder of the data to be fetched only when or if it is required. There are many names for this kind of feature, including lazy loading, deferred loading, lazy fetching, on-demand fetching, just-in-time reading, indirection, and others. They all mean pretty much the same thing, which is just that some data may not be loaded when the object is initially read from the database but will be fetched only when it is referenced or accessed.

The *fetch type* of a basic mapping can be configured to be lazily or eagerly loaded by specifying the `fetch` element in the corresponding `@Basic` annotation. The `FetchType` enumerated type defines the values for this element, which may be either `EAGER` or `LAZY`. Setting the fetch type of a basic mapping to `LAZY` means that the provider may defer loading the state for that attribute until it is referenced. The default is to eagerly load all basic mappings. Listing 4-7 shows an example of overriding a basic mapping to be lazily loaded.

Listing 4-7. *Lazy Field Loading*

```
@Entity
public class Employee {
    // ...
    @Basic(fetch=FetchType.LAZY)
    @Column(name="COMM")
    private String comments;
    // ...
}
```

We are assuming in this example that applications will seldom access the comments in an employee record, so we mark it as being lazily fetched. Note that in this case the `@Basic` annotation is not only present for documentation purposes but also required in order to specify the fetch type for the field. Configuring the `comments` field to be fetched lazily will allow an `Employee` instance returned from a query to have the `comments` field empty. The application does not have to do anything special to get it, however. By simply accessing the `comments` field, it will be transparently read and filled in by the provider if it was not already loaded.

Before you use this feature you should be aware of a few pertinent points about lazy attribute fetching. First and foremost, the directive to lazily fetch an attribute is meant only to be a hint to the persistence provider to help the application achieve better performance. The provider is not required to respect the request, since the behavior of the entity is not compromised if the provider goes ahead and loads the attribute. The converse is not true, though, since specifying that an attribute be eagerly fetched may be critical to being able to access the entity state once the entity is detached from the persistence context. We will discuss detachment more in Chapter 5 and explore the connection between lazy loading and detachment.

Second, on the surface it may appear that this is a good idea for certain attributes of an entity, but in practice it is almost never a good idea to lazily fetch simple types. The reason is that there is little to be gained in only returning part of a database row unless you are certain that the state will not be accessed in the entity later on. The only times when lazy loading of a basic mapping should be considered are when either there are many columns in a table (for example, dozens or hundreds) or when the columns are large (for example, very large character strings or byte strings). It could take significant resources to load the data, and not loading it could save quite a lot of effort, time, and resources. Unless either of these two cases is true, then in the majority of cases this will cause lazily fetching a subset of object attributes to end up being more expensive than eagerly fetching them.

Lazy fetching is quite relevant when it comes to relationship mappings, though, so we will be discussing this topic more later in the chapter.

Large Objects

A common database term for a character or byte-based object that can be very large (up to the gigabyte range) is *large object*, or *LOB* for short. Database columns that can store these types of large objects require special JDBC calls to be accessed from Java. To signal to the provider that it should use the LOB methods when passing and retrieving this data to and from the JDBC driver, an additional annotation must be added to the basic mapping. The `@Lob` annotation acts as the marker annotation to fulfill this purpose and may appear in conjunction with the `@Basic` annotation, or it may appear when `@Basic` is absent and implicitly assumed to be on the mapping.

Since a LOB mapping is just a specialized kind of basic mapping, it can also be accompanied by an `@Column` annotation when the name of the LOB column needs to be overridden from the assumed default name.

LOBs come in two flavors in the database: character large objects, called *CLOB*s, and binary large objects, or *BLOB*s. As their names imply, a CLOB column holds a large character sequence, and a BLOB column can store a large byte sequence. The Java types mapped to BLOB columns are `byte[]`, `Byte[]`, and `Serializable` types, while `char[]`, `Character[]`, and `String` objects are mapped to CLOB columns. The provider is responsible for making this distinction based upon the type of the attribute being mapped.

An example of mapping an image to a BLOB column is shown in Listing 4-8. Here, the `PIC` column is assumed to be a BLOB column to store the employee picture that is in the `picture` field. We have also marked this field to be loaded lazily, a common practice applied to LOBs that do not get referenced often.

Listing 4-8. *Mapping a BLOB Column*

```
@Entity
public class Employee {
    @Id
    private int id;
    @Basic(fetch=FetchType.LAZY)
    @Lob @Column(name="PIC")
    private byte[] picture;
    // ...
}
```

Enumerated Types

Another of the simple types that may be treated specially is the enumerated type. The values of an enumerated type are constants that can be handled differently depending upon the application needs.

As with enumerated types in other languages, the values of an enumerated type in Java have an implicit ordinal assignment that is determined by the order in which they were declared. This ordinal cannot be modified at runtime and can be used to represent and store the values of the enumerated type in the database. Interpreting the values as ordinals is the default way that providers will map enumerated types to the database, and the provider will assume that the database column is an integer type.

Consider the following enumerated type:

```
public enum EmployeeType {
    FULL_TIME_EMPLOYEE,
    PART_TIME_EMPLOYEE,
    CONTRACT_EMPLOYEE
}
```

The ordinals assigned to the values of this enumerated type at compile time would be 0 for FULL_TIME_EMPLOYEE, 1 for PART_TIME_EMPLOYEE, and 2 for CONTRACT_EMPLOYEE. In Listing 4-9 we define a persistent field of this type.

Listing 4-9. *Mapping an Enumerated Type Using Ordinals*

```
@Entity
public class Employee {
    @Id private int id;
    private EmployeeType type;
    // ...
}
```

We can see that mapping EmployeeType is trivially easy to the point where we don't actually have to do anything at all. The defaults are applied, and everything will just work. The type field will get mapped to an integer TYPE column, and all full-time employees will have an ordinal of 0 assigned to them. Similarly the other employees will have their types stored in the TYPE column accordingly.

If an enumerated type changes, however, then we have a problem. The persisted ordinal data in the database will no longer apply to the correct value. For example, if the company benefits policy changed and we started giving additional benefits to part-time employees who worked over 20 hours a week, then we would want to differentiate between the two types of part-time employees. By adding a PART_TIME_BENEFITS_EMPLOYEE value after PART_TIME_EMPLOYEE, we would be causing a new ordinal assignment to occur, where our new value would get assigned the ordinal of 2 and CONTRACT_EMPLOYEE would get 3. This would have the effect of causing all of the contract employees on record to suddenly become part-time employees with benefits, clearly not the result that we were hoping for.

We could go through the database and adjust all of the Employee entities to have their correct type, but if the employee type is used elsewhere, then we would need to make sure that they were all fixed as well. This is not a good maintenance situation to be in.

A better solution would be to store the name of the value as a string instead of storing the ordinal. This would isolate us from any changes in declaration and allow us to add new types without having to worry about the existing data. We can do this by adding an @Enumerated annotation on the attribute and specifying a value of STRING.

The @Enumerated annotation actually allows an EnumType to be specified, and the EnumType is itself an enumerated type that defines values of ORDINAL and STRING. While it is somewhat ironic that an enumerated type is being used to indicate how the provider should represent enumerated types, it is wholly appropriate. Since the default value of @Enumerated is ORDINAL, specifying @Enumerated(ORDINAL) is useful only when you want to make this mapping explicit.

In Listing 4-10 we are storing strings for the enumerated values. Now the TYPE column must be a string-based type, and all of the full-time employees will have the string "FULL_TIME_EMPLOYEE" stored in their corresponding TYPE column.

Listing 4-10. *Mapping an Enumerated Type Using Strings*

```
@Entity
public class Employee {
    @Id
    private int id;
    @Enumerated(EnumType.STRING)
    private EmployeeType type;
    // ...
}
```

Note that using strings will solve the problem of inserting additional values in the middle of the enumerated type, but it will leave the data vulnerable to changes in the names of the values. For instance, if we wanted to change PART_TIME_EMPLOYEE to PT_EMPLOYEE, then we would be in trouble. This is a less likely problem, though, because changing the names of an enumerated type would cause all of the code that uses the enumerated type to have to change also. This would be a bigger bother than reassigning values in a database column.

In general, storing the ordinal is going to be the best and most efficient way to store enumerated types as long as the likelihood of additional values inserted in the middle is not high. New values could still be added on the end of the type without any negative consequences.

One final note about enumerated types is that they are defined quite flexibly in Java. In fact, it is even possible to have values that contain state. There is currently no support within the Java Persistence API for mapping state contained within enumerated values.

Temporal Types

Temporal types are the set of time-based types that may be used in persistent state mappings. The list of supported temporal types includes the three java.sql types java.sql.Date, java.sql.Time, and java.sql.Timestamp, and it includes the two java.util types java.util.Date and java.util.Calendar.

The java.sql types are completely hassle-free. They act just like any other simple mapping type and do not need any special consideration. The two java.util types need additional metadata, however, to indicate which of the JDBC java.sql types to use when communicating

with the JDBC driver. This is done by annotating them with the `@Temporal` annotation and specifying the JDBC type as a value of the `TemporalType` enumerated type. There are three enumerated values of `DATE`, `TIME`, and `TIMESTAMP` to represent each of the `java.sql` types.

Listing 4-11 shows how `java.util.Date` and `java.util.Calendar` may be mapped to date columns in the database.

Listing 4-11. *Mapping Temporal Types*

```
@Entity
public class Employee {
    @Id
    private int id;
    @Temporal(TemporalType.DATE)
    private Calendar dob;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}
```

Like the other varieties of basic mappings, the `@Column` annotation may be used to override the default column name.

Transient State

Attributes that are part of a persistent entity but not intended to be persistent can either be modified with the `transient` modifier in Java or be annotated with the `@Transient` annotation. If either of these is specified, then the provider runtime will not apply its default mapping rules to the attribute it was specified on.

Transient fields are used for various reasons. One might be when you want to cache some in-memory state that you don't want to have to recompute, rediscover, or reinitialize. For example, in Listing 4-12 we are using a transient field to save the correct locale-specific word for "Employee" so that we print it correctly wherever it is being displayed. We have used the `transient` modifier instead of the `@Transient` annotation so that if the `Employee` gets serialized from one VM to another then the translated name will get reinitialized to correspond to the locale of the new VM. In cases where the non-persistent value should be retained across serialization, the annotation should be used instead of the modifier.

Listing 4-12. *Using a Transient Field*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    transient private String translatedName;
    // ...
```



```
    public String toString() {
        if (translatedName == null) {
            translatedName =
                ResourceBundle.getBundle("EmpResources").getString("Employee");
        }
        return translatedName + ": " id + " " + name;
    }
}
```

Mapping the Primary Key

Every entity that is mapped to a relational database must have a mapping to a primary key in the table. We have already learned the basics of how the `@Id` annotation indicates the identifier of the entity. In this section we explore simple identifiers and primary keys in a little more depth and learn how we can let the persistence provider generate unique identifier values for us.

Except for its special significance in designating the mapping to the primary key column, an id mapping is almost the same as the basic mapping. Another difference is that id mappings are generally restricted to the following types:

- **Primitive Java types:** `byte`, `int`, `short`, `long`, `char`
- **Wrapper classes of primitive Java types:** `Byte`, `Integer`, `Short`, `Long`, `Character`
- **Arrays of primitive or wrapper types:** `Byte`, `Integer`, `Short`, `Long`, `Character`
- **Strings:** `java.lang.String`
- **Large numeric types:** `java.math.BigInteger`
- **Temporal types:** `java.util.Date`, `java.sql.Date`

Floating point types like `float` and `double` are permitted, as well as the `Float` and `Double` wrapper classes and `java.math.BigDecimal`, but these are discouraged because of the nature of rounding error and the untrustworthiness of the `equals()` operator when applied to them. Using floating types for primary keys is a risky endeavor and definitely not recommended.

Just as with basic mappings, the `@Column` annotation may be used to override the column name that the id attribute is mapped to. The same defaulting rules apply to id mappings as apply to basic mappings, which is that the name of the column is assumed to be the same as the name of the attribute.

Identifier Generation

Sometimes applications do not want to be bothered with trying to define and ensure uniqueness in some aspect of their domain model and are content to let the identifier values be automatically generated for them. This is called id generation and is specified by the `@GeneratedValue` annotation.

When id generation is enabled, the persistence provider will generate an identifier value for every entity instance of that type. Once the identifier value is obtained, the provider will insert it into the newly persisted entity; however, depending upon the way it is generated, it may not actually be present in the object until the entity has been inserted in the database. In

other words, the application cannot rely upon being able to access the identifier until after either a flush has occurred or the transaction has completed.

Applications can choose one of four different id generation strategies by specifying a strategy in the `strategy` element. The value may be any one of `AUTO`, `TABLE`, `SEQUENCE`, or `IDENTITY` enumerated values of the `GenerationType` enumerated type.

Table and sequence generators may be specifically defined and then reused by multiple entity classes. These generators are named and are globally accessible to all of the entities in the persistence unit.

Automatic Id Generation

If an application does not care what kind of generation is used by the provider but wants generation to occur, then it can specify a strategy of `AUTO`. This means that the provider will use whatever strategy it wants to generate identifiers. Listing 4-13 shows an example of using automatic id generation. This will cause an identifier value to be created by the provider and inserted into the `id` field of each `Employee` entity that gets persisted.

Listing 4-13. *Using Auto Id Generation*

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    // ...
}
```

There is a catch to using `AUTO`, though. The provider gets to pick its own strategy to store the identifiers, but it needs to have some kind of persistent resource in order to do so. For example, if it chooses a table-based strategy, then it needs to create a table; if it chooses a sequence-based strategy, then it needs to create a sequence. The provider can't always rely upon the database connection that it obtains from the server to have permissions to create a table in the database. This is normally a privileged operation that is often restricted to the DBA. There will need to be some kind of creation phase or schema generation to cause the resource to be created before the `AUTO` strategy is able to function.

The `AUTO` mode is really a generation strategy for development or prototyping. It works well as a means of getting you up and running more quickly when the database schema is being generated. In any other situation it would be better to use one of the other generation strategies discussed in the later sections.

Id Generation Using a Table

The most flexible and portable way to generate identifiers is to use a database table. Not only will it port to different databases, but it also allows for storing multiple different identifier sequences for different entities within the same table. The easiest way to use a table to generate identifiers is to simply specify the generation strategy to be `TABLE` in the `strategy` element:

```
@Id GeneratedValue(strategy=GenerationType.TABLE)
private int id;
```


Since the generation strategy is indicated but no generator has been specified, the provider will assume a table of its own choosing. If schema generation is used, then it will be created, but if not, then the default table assumed by the provider must be known and must exist in the database.

A more explicit approach would be to actually specify the table that is to be used for id storage. This is done by defining a table generator that, contrary to what its name implies, does not actually generate tables. Rather, it is an identifier generator that uses a table to store them. We can define one by using a `@TableGenerator` annotation and then refer to it by name in the `@GeneratedValue` annotation:

```
@TableGenerator(name="Emp_Gen")
@Id @GeneratedValue(generator="Emp_Gen")
private int id;
```

Although we are showing the `@TableGenerator` annotating the identifier attribute, it can actually be defined on any attribute or class. Regardless of where it is defined, it will be available to the entire persistence unit. A good practice would be to define it locally on the id attribute if only one class is using it but to define it in XML, as described in Chapter 10, if it will be used for multiple classes.

The `name` element globally names the generator, which then allows us to reference it in `@GeneratedValue`. This is functionally equivalent to the previous example where we simply said that we wanted to use table generation but did not specify the generator. Now we are specifying the name of the generator but not supplying any of the generator details, leaving them to be defaulted by the provider.

A further qualifying approach would be to specify the table details, as in the following:

```
@TableGenerator(name="Emp_Gen",
    table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL")
```

We have included some additional elements after the name of the generator. Following the name are three elements, `table`, `pkColumnName`, and `valueColumnName`, which define the actual table that stores the identifiers for "Emp_Gen". The `table` element just indicates the name of the table.

Every table that is used for id generation should have two columns. It could have more than two columns, but only two will be used. The first column is of a string type and is used to identify the particular generator sequence. It is the primary key for all of the generators in the table. The name of this column is specified by `pkColumnName`. The second column is of an integer type and stores the actual id sequence that is being generated. The value stored in this column is the last identifier that was allocated in the sequence. The name of this column is specified by `valueColumnName`. In our case our table is named "ID_GEN". The name of the primary key column of the table, or the column that stores the generator names, is named "GEN_NAME", and the column that stores the id sequence values is named "GEN_VAL".

Each defined generator represents a row in the table. The name of the generator becomes the value stored in the `pkColumnName` column for that row and is used by the provider to look up the generator to obtain its last allocated value.

In our example we named our generator "Emp_Gen" so our table would look like the one in Figure 4-2.

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0

Figure 4-2. *Table for identifier generation*

We can see that the last allocated `Employee` identifier is 0, which tells us that no identifiers have been generated yet. An `initialValue` element representing the last allocated identifier may be specified as part of the generator definition, but the default setting of 0 will suffice in almost every case. This setting is used only during schema generation when the table is created. During subsequent executions, the provider will read the contents of the value column to determine the next identifier to give out.

To avoid updating the row for every single identifier that gets requested, an allocation size is used. This will cause the provider to pre-allocate a block of identifiers and then give out identifiers from memory as requested until the block is used up. Once this block is used up, the next request for an identifier triggers another block of identifiers to be pre-allocated, and the identifier value is incremented by the allocation size. By default, the allocation size is set to 50. This value can be overridden to be larger or smaller through the use of the `allocationSize` element when defining the generator.

■ **Tip** The provider may allocate identifiers within the same transaction as the entity being persisted or in a separate transaction. It is not specified, but you should check your provider documentation to see how it can avoid the risk of deadlock when concurrent threads are creating entities and locking resources.

Shown in Listing 4-14 is an example of defining a second generator to be used for `Address` entities but that uses the same `ID_GEN` table to store the identifier sequence. In this case we are actually explicitly dictating the value we are storing in the identifier table's primary key column by specifying the `pkColumnvalue` element. This element allows the name of the generator to be different from the column value, although doing so is rarely needed. The example shows an `Address` id generator named "Address_Gen" but then defines the value stored in the table for `Address` id generation as "Addr_Gen". The generator also sets the initial value to 10000 and the allocation size to 100.

Listing 4-14. *Using Table Id Generation*

```
@TableGenerator(name="Address_Gen",
    table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL",
    pkColumnValue="Addr_Gen",
    initialValue=10000,
    allocationSize=100)
```



```
@Id @GeneratedValue(generator="Address_Gen")
private int id;
```

If both "Emp_Gen" and "Address_Gen" generators were defined, then on application startup the `ID_GEN` table would look like Figure 4-3. As the application allocates identifiers, the values stored in the `GEN_VAL` column will increase.

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0
Addr_Gen	10000

Figure 4-3. *Table for generating* `Address` *and* `Employee` *identifiers*

If schema generation has not been run, then the table must exist in the database and be configured to be in this state when the application starts up for the first time. The following SQL could be applied to create and initialize this table:

```
CREATE TABLE id_gen (
    gen_name VARCHAR(80),
    gen_val INTEGER,
    CONSTRAINT pk_id_gen
        PRIMARY KEY (gen_name)
);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Emp_Gen', 0);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Addr_Gen', 10000);
```

Id Generation Using a Database Sequence

Many databases support an internal mechanism for id generation called sequences. A database sequence may be used to generate identifiers when the underlying database supports them.

As we saw with table generators, if it is known that a database sequence should be used for generating identifiers and we are not concerned that it be any particular sequence, then specifying the generator type alone should be sufficient:

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
private int id;
```

In this case no generator is named, so the provider will use a default sequence object of its own choosing. Note that if multiple sequence generators are defined but not named, then it is not specified whether they use the same default sequence or different ones. The only difference between using one sequence for multiple entity types and using one for each entity would be the ordering of the sequence numbers and possible contention on the sequence. The safer route would be to define a named sequence generator and refer to it in the `@GeneratedValue` annotation:

```
@SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
@Id @GeneratedValue(generator="Emp_Gen")
private int getId;
```

Unless schema generation is enabled, this would require that the sequence be defined and already exist. The SQL to create such a sequence would be:

```
CREATE SEQUENCE Emp_Seq
    MINVALUE 1
    START WITH 1
    INCREMENT BY 50
```

The initial value and allocation size can also be used in sequence generators and would need to be reflected in the SQL to create the sequence. We can see that the default allocation size is 50, just as it is with table generators. If schema generation is not being used and the sequence is being manually created, then the INCREMENT BY clause would need to be configured to match the setting or default value of the allocation size.

Id Generation Using Database Identity

Some databases support a primary key identity column sometimes referred to as an autonumber column. Whenever a row is inserted into the table, then the identity column will get a unique identifier assigned to it. This can be used to generate the identifiers for objects, but once again is available only when the underlying database supports it.

To indicate that `IDENTITY` generation should occur, the `@GeneratedValue` annotation should specify a generation strategy of `IDENTITY`. This will indicate to the provider that it must reread the inserted row from the table after an INSERT has occurred. This will allow it to obtain the newly generated identifier from the database and put it into the in-memory entity that was just persisted:

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

There is no generator annotation for `IDENTITY` since it must be defined as part of the database schema for the primary key column itself. Identity generation obviously may not be shared across multiple entity types.

Another difference, hinted at earlier, between using `IDENTITY` and other id generation strategies is that the identifier will not be accessible until after the insert has occurred. While no guarantee is made as to the accessibility of the identifier before the transaction has completed, it is at least possible for other types of generation to eagerly allocate the identifier, but when using identity, it is the action of inserting that causes the identifier to be generated. It would be impossible for the identifier to be available before the entity is inserted into the database, and because insertion of entities is most often deferred until commit time, the identifier would not be available until after the transaction has been committed.

Relationships

If entities contained only simple persistent state then the business of object-relational mapping would be a trivial one indeed. Most entities need to be able to reference, or have relationships with, other entities. This is what produces the domain model graphs that are common in business applications.

In the following sections we will explore the different kinds of relationships that can exist and show how to define and map them using Java Persistence API mapping metadata.

Relationship Concepts

Before we go off and start mapping relationships we should really take a quick tour through some of the basic relationship concepts and terminology. Having a firm grasp on these concepts will make it easier to understand the remainder of the relationship mapping sections.

Roles

There is an old adage that says every story has three sides: yours, mine, and the truth. Relationships are kind of the same in that there are three different perspectives. The first is the view from one side of the relationship, the second is from the other side, and the third is from a global perspective that knows about both sides. The "sides" are called *roles*. In every relationship there are two entities that are related to one another, and each entity is said to play a role in the relationship.

Relationships are everywhere, so examples are not hard to come by. An employee has a relationship to the department that he or she works in. The `Employee` entity plays the role of working in the department, while the `Department` entity plays the role of having an employee working in it.

Of course the role a given entity is playing differs according to the relationship, and an entity may be participating in many different relationships with many different entities. We can conclude, therefore, that any entity may be playing a number of different roles in any given model. If we think of an `Employee` entity, we realize that it does in fact play other roles in other relationships, such as the role of working for a manager in its relationship with another `Employee` entity, working on a project in its relationship with the `Project` entity, and so forth.

Unlike EJB 2.1, where the roles all had to be enumerated in metadata for every relationship, the Java Persistence API does not have metadata requirements to declare the role an entity is playing. Nevertheless, roles are still helpful as a means of understanding the nature and structure of relationships.

Directionality

In order to have relationships at all, there has to be a way to create, remove, and maintain them. The basic way this is done is by an entity having a relationship attribute that refers to its related entity in a way that identifies it as playing the other role of the relationship. It is often the case that the other entity in turn has an attribute that points back to the original entity. When each entity points to the other, the relationship is *bidirectional*. If only one entity has a pointer to the other, the relationship is said to be *unidirectional*.

A relationship from an `Employee` to the `Project` that they work on would be bidirectional. The `Employee` should know its `Project`, and the `Project` should point to the `Employee` working on it. A UML model of this relationship is shown in Figure 4-4. The arrows going in both directions indicate the bidirectionality of the relationship.

An `Employee` and its `Address` would likely be modeled as a unidirectional relationship because the `Address` is not expected to ever need to know its resident. If it did, of course, then it would need to become a bidirectional relationship. Figure 4-5 shows this relationship. Because the relationship is unidirectional the arrow points from the `Employee` to the `Address`.

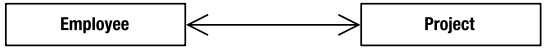

Figure 4-4. `Employee` *and* `Project` *in a bidirectional relationship*

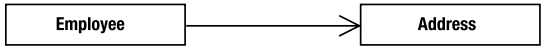

Figure 4-5. `Employee` *in a unidirectional relationship with* `Address`

As we will see later in the chapter, although they both share the same concept of directionality, the object and data models each see it a little differently because of the paradigm difference. In some cases, unidirectional relationships in the object model can pose a problem in the database model.

Source and Target

Even though we can use the directionality of a relationship to help describe and explain a model, when it comes to actually discussing it in concrete terms, it makes sense to think of every bidirectional relationship as a pair of unidirectional relationships.

So instead of having a single bidirectional relationship of an `Employee` working on a `Project`, we would have one unidirectional "project" relationship where the `Employee` points to the `Project` they work on, and another unidirectional "worker" relationship where the `Project` points to the `Employee` that works on it. Each of these relationships has an entity that is the *source* or referring role, and the side that is the *target* or referred-to role. The beauty of this is that we can use the same terms no matter which relationship we are talking about and no matter what the roles are in the relationship. Figure 4-6 shows how the two relationships have source and target entities, and how from each relationship perspective the source and target entities are different.

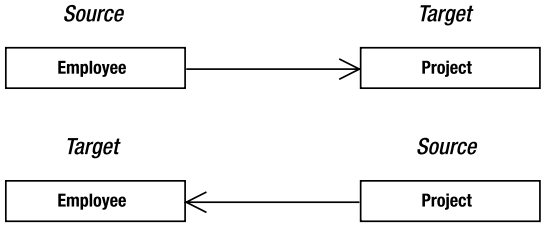

Figure 4-6. *Unidirectional relationships between* `Employee` *and* `Project`

Cardinality

It isn't very often that a project has only a single employee working on it. We would like to be able to capture the aspect of how many entities exist on each side of the same relationship instance. This is called the *cardinality* of the relationship. Each role in a relationship will have its own cardinality, which indicates whether there can be only one instance of the entity or many instances.

In our employee and department example, we might first say that one employee works in one department, so the cardinality of both sides would be *one*. But chances are that more than one employee works in the department. Because of this we would make the relationship have a *many* cardinality on the `Employee` or source side, meaning that many `Employee` instances could each point to the same `Department`. The target or `Department` side would keep its cardinality of one. Figure 4-7 shows this many-to-one relationship. The "many" side is marked with a "*".

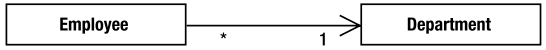

Figure 4-7. *Unidirectional many-to-one relationship*

In our `Employee` and `Project` example, we have a bidirectional relationship, or two relationship directions. If an employee can work on multiple projects and a project can have multiple employees working on it, then we would end up with cardinalities of "many" on the sources and targets of both directions. Figure 4-8 shows the UML diagram of this relationship.

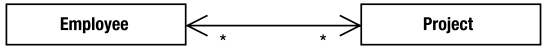

Figure 4-8. *Bidirectional many-to-many relationship*

A picture is worth a thousand words, and describing these relationships in text is quite a lot harder than simply showing a picture. In words, though, this picture indicates the following:

- Each employee can work on a number of projects
- Many employees can work on the same project
- Each project can have a number of employees working on it
- Many projects can have the same employee working on them

Implicit in this model is the fact that there can be sharing of `Employee` and `Project` instances across multiple relationship instances.

Ordinality

A role may be further specified by determining whether or not it may be present at all. This is called the *ordinality* and serves to show whether the target entity needs to be specified when the source entity is created. Because the ordinality is really just a Boolean value, we also refer to it as the *optionality* of the relationship.

In cardinality terms, ordinality would be indicated by the cardinality being a range instead of a simple value, and the range would begin with 0 or 1 depending upon the ordinality. It is simpler, though, to merely state that the relationship is either optional or mandatory. If optional, then the target may not be present, but if mandatory, then a source entity without a reference to its associated target entity is in an invalid state.

Mappings Overview

Now that we know enough theory and have the conceptual background to be able to discuss relationships, we can go on to explaining and using relationship mappings.

Each one of the mappings is named for the cardinality of the source and target roles. As we saw in the previous sections, we can view a bidirectional relationship as a pair of two unidirectional mappings. Each of these mappings is really a unidirectional relationship mapping, and if we take the cardinalities of the source and target of the relationship and combine them together in that order, permuting them with the two possible values of "one" and "many", we end up with the following names given to the mappings:

1. Many-to-one
2. One-to-one
3. One-to-many
4. Many-to-many

These mapping names are also the names of the annotations that are used to indicate the relationship types on the attributes that are being mapped. They are the basis for the logical relationship annotations, and they contribute to the object modeling aspects of the entity.

Like basic mappings, relationship mappings may be applied to either fields or properties of the entity.

Single-Valued Associations

An association from an entity instance to another entity instance (where the cardinality of the target is "one") is called a single-valued association. The many-to-one and one-to-one relationship mappings fall into this category because the source entity refers to at most one target entity. We will discuss these relationships and some of their variants first.

Many-to-One Mappings

In our cardinality discussion of the `Employee` and `Department` relationship (shown in Figure 4-7) we first thought of an employee working in a department, so we just assumed that it was a one-to-one relationship. However, when we realized that more than one employee works in the same department, we changed it to a many-to-one relationship mapping. It turns out that many-to-one is the most common mapping and is the one that is normally used when creating an association to an entity.

In Figure 4-9 we show a many-to-one relationship between `Employee` and `Department`. `Employee` is the "many" side and the source of the relationship, and `Department` is the "one" side and the target. Once again, because the arrow points in only one direction, from `Employee` to `Department`, the relationship is unidirectional. Note that in UML, the source class has an implicit attribute of the target class type if it can be navigated to. For example, `Employee` has an attribute called `department` that will contain a reference to a single `Department` instance.

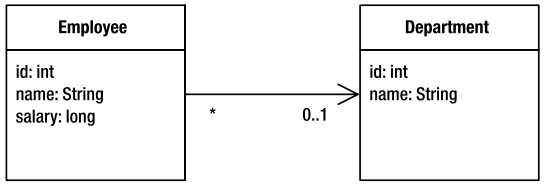

Figure 4-9. *Many-to-one relationship from* `Employee` *to* `Department`

A many-to-one mapping is defined by annotating the attribute in the source entity (the attribute that refers to the target entity) with the `@ManyToOne` annotation. In Listing 4-15 we can see how the `@ManyToOne` annotation is used to map this relationship. The `department` field in `Employee` is the source attribute that is annotated.

Listing 4-15. *Many-to-One Relationship from* `Employee` *to* `Department`

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    private Department department;
    // ...
}
```

We have included only the bits of the class that are relevant to our discussion, but we see from the previous example that the code was rather anticlimactic. A single annotation was all that was required to map the relationship, and it turned out to be quite dull, really.

The same kinds of attribute flexibility and modifier requirements that were described for basic mappings also apply to relationship mappings. The annotation may be present on either the field or property, depending upon the strategy used for the entity.

Using Join Columns

In the database, a relationship mapping means that one table has a reference to another table. The database term for a column that refers to a key (usually the primary key) in another table is a *foreign key* column. In the Java Persistence API we call them *join columns*, and the `@JoinColumn` annotation is the primary annotation used to configure these types of columns.

Consider the `EMPLOYEE` and `DEPARTMENT` tables shown in Figure 4-10 that correspond to the `Employee` and `Department` entities. The `EMPLOYEE` table has a foreign key column named `DEPT_ID` that references the `DEPARTMENT` table. From the perspective of the entity relationship, `DEPT_ID` is the join column that associates the `Employee` and `Department` entities.

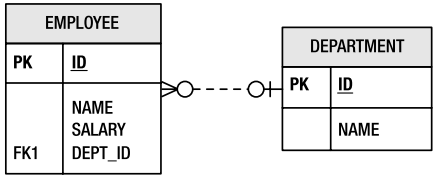

Figure 4-10. `EMPLOYEE` *and* `DEPARTMENT` *tables*

In almost every relationship, independent of source and target sides, one of the two sides is going to have the join column in its table. That side is called the *owning side* or the *owner* of the relationship. The side that does not have the join column is called the non-owning or *inverse* side.

Ownership is important for mapping because the physical annotations that define the mappings to the columns in the database (for example, `@JoinColumn`) are always defined on the owning side of the relationship. If they are not there, then the values are defaulted from the perspective of the attribute on the owning side.

Many-to-one mappings are always on the owning side of a relationship, so if there is a `@JoinColumn` to be found in the relationship that has a many-to-one side, then that is where it will be located. To specify the name of the join column, the `name` element is used. For example, the `@JoinColumn(name="DEPT_ID")` annotation means that the `DEPT_ID` column in the source entity table is the foreign key to the target entity table, whatever the target entity of the relationship happens to be.

If no `@JoinColumn` annotation accompanies the many-to-one mapping, then a default column name will be assumed. The name that is used as the default is formed from a combination of both the source and target entities. It is the name of the relationship attribute in the source entity, which is `department` in our example, plus an underscore character ("_"), plus the name of the primary key column of the target entity. So if the `Department` entity were mapped to a table that had a primary key column named `ID`, then the join column in the `EMPLOYEE` table would be assumed to be named `DEPARTMENT_ID`. If this is not actually the name of the column, then the `@JoinColumn` annotation must be defined to override the default.

Going back to Figure 4-10, the foreign key column is named `DEPT_ID` instead of the defaulted `DEPARTMENT_ID` column name. Listing 4-16 shows the `@JoinColumn` annotation being used to override the join column name to be `DEPT_ID`.

Listing 4-16. *Many-to-One Relationship Overriding the Join Column*

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;
    // ...
}
```


Annotations allow us to specify `@JoinColumn` on either the same line as `@ManyToOne` or on a separate line, above or below it. By convention the logical mapping should appear first and then the physical mapping. This makes the object model clear since the physical part is less important to the object model.

One-to-One Mappings

If it really was the case that only one employee could work in a department, then we would be back to the one-to-one association again. A more realistic example of a one-to-one association, however, would be an employee who has a parking space. Assuming that every employee got his or her own parking space, then we would create a one-to-one relationship from `Employee` to `ParkingSpace`. Figure 4-11 shows this relationship.

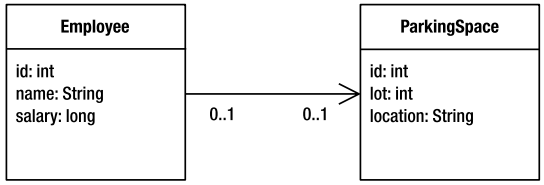

Figure 4-11. *One-to-one relationship from* `Employee` *to* `ParkingSpace`

We define the mapping in a similar way to the way we define a many-to-one mapping, except that we use the `@OneToOne` annotation on the `parkingSpace` attribute instead of a `@ManyToOne` annotation. Just as with a many-to-one mapping, the one-to-one mapping has a join column in the database and needs to override the name of the column in an `@JoinColumn` annotation when the default name does not apply. The default name is composed the same way as for many-to-one mappings using the name of the source attribute and the target primary key column name.

Figure 4-12 shows the tables mapped by the `Employee` and `ParkingSpace` entities. The foreign key column in the `EMPLOYEE` table is named `PSPACE_ID` and refers to the `PARKING_SPACE` table.

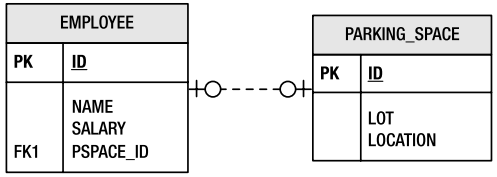

Figure 4-12. `EMPLOYEE` *and* `PARKING_SPACE` *tables*

As it turns out, one-to-one mappings are almost the same as many-to-one mappings except for the fact that only one instance of the source entity is able to refer to the same target entity instance. In other words, the target entity instance is not shared amongst the source entity instances. In the database, this equates to having a uniqueness constraint on the source foreign

key column (that is, the foreign key column in the source entity table). If there were more than one foreign key value that was the same, then it would contravene the rule that no more than one source entity instance can refer to the same target entity instance.

In Listing 4-17 we see the mapping for this relationship. The `@JoinColumn` annotation has been used to override the default join column name of `PARKINGSPACE_ID` to be `PSPACE_ID`.

Listing 4-17. *One-to-One Relationship from* `Employee` *to* `ParkingSpace`

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne
    @JoinColumn(name="PSPACE_ID")
    private ParkingSpace parkingSpace;
    // ...
}
```

Bidirectional One-to-One Mappings

It often happens that the target entity of the one-to-one has a relationship back to the source entity; for example, `ParkingSpace` has a reference back to the `Employee` that uses it. When this is the case, we call it a bidirectional one-to-one relationship. As we saw previously, we actually have two separate one-to-one mappings, one in each direction, but we call the combination of the two a bidirectional one-to-one relationship. To make our existing one-to-one employee and parking space example bidirectional, we need only change the `ParkingSpace` to point back to the `Employee`. Figure 4-13 shows the bidirectional relationship.

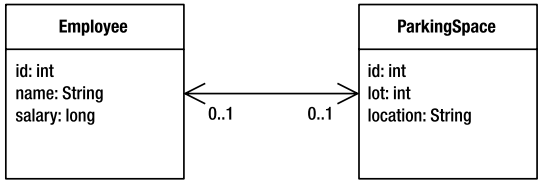

Figure 4-13. *One-to-one relationship between* `Employee` *and* `ParkingSpace`

We already learned that the entity table that contains the join column determines the entity that is the owner of the relationship. In a bidirectional one-to-one relationship both of the mappings are one-to-one mappings, and either side may be the owner, so the join column may end up being on one side or the other. This would normally be a data modeling decision and not a Java programming decision, and it would likely be decided based upon the most frequent direction of traversal.

Consider the `ParkingSpace` entity class shown in Listing 4-18. This example assumes the table mapping shown in Figure 4-12, and it assumes that `Employee` is the owning side of the relationship. We now have to add a reference from `ParkingSpace` back to `Employee`. This is achieved by adding the `@OneToOne` relationship annotation on the `employee` field. As part of the

annotation we must add a `mappedBy` element to indicate that the owning side is the `Employee` and not the `ParkingSpace`. Because `ParkingSpace` is the inverse side of the relationship, it does not have to supply the join column information.

Listing 4-18. *Inverse Side of a Bidirectional One-to-One Relationship*

```
@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    // ...
}
```

The `mappedBy` element in the one-to-one mapping of the `employee` attribute of `ParkingSpace` is needed to refer to the `parkingSpace` attribute in the `Employee` class. The value of `mappedBy` is the name of the attribute in the owning entity that points back to the inverse entity.

The two rules, then, for bidirectional one-to-one associations are:

1. The `@JoinColumn` annotation goes on the mapping of the entity that is mapped to the table containing the join column, or the owner of the relationship. This may be on either side of the association.

2. The `mappedBy` element should be specified in the `@OneToOne` annotation in the entity that does not define a join column, or the inverse side of the relationship.

It would not be legal to have a bidirectional association that had `mappedBy` on both sides just as it would be incorrect to not have it on either side. The difference is that if it were absent on both sides of the relationship, then the provider would treat each side as an independent unidirectional relationship. This would be fine except that it would assume that each side was the owner and that each had a join column.

Bidirectional many-to-one relationships are explained later as part of the discussion of multi-valued bidirectional associations.

One-to-One Primary Key Mappings

A specific case of a unique one-to-one relationship is when the primary keys of the related entities are always guaranteed to match. The two entities must always have been created with the same identifiers, and in the database each primary key could also be used as a foreign key to the other entity. Of course the direction in which the actual constraint occurs should dictate which side is the owner since the owner is the one with the foreign key constraint in its table.

Imagine if every time an employee got hired, his or her employee id was used as their parking space id (and employees were never allowed to change parking spaces!). This relationship would be modeled in the database as shown in Figure 4-14.

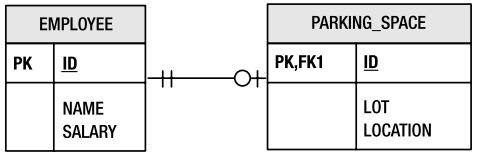

Figure 4-14. `EMPLOYEE` *and* `PARKING_SPACE` *tables with shared primary keys*

The difference between mapping this kind of one-to-one relationship and the previous kind is that there is no additional foreign key column in either table. The primary key in the `PARKING_SPACE` table is also a foreign key to the `EMPLOYEE` table. When this is the case, then an `@PrimaryKeyJoinColumn` is used instead of an `@JoinColumn` annotation. Because the foreign key is now defined from `ParkingSpace` to `Employee`, we need to reverse the ownership of the relationship and make `ParkingSpace` the owner. We specify the `@PrimaryKeyJoinColumn` annotation on the `employee` field of the `ParkingSpace` entity.

Listing 4-19 shows the revised `Employee` and `ParkingSpace` entities. Note that since the `Employee` is now the inverse side of the relationship, the join column mapping is removed, and the `@OneToOne` annotation must now specify the `mappedBy` element.

Listing 4-19. *One-to-One Primary Key Relationship*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToOne(mappedBy="employee")
    private ParkingSpace parkingSpace;
    // ...
}

@Entity
public class ParkingSpace {
    @Id private int id;
    private int lot;
    private String location;
    @OneToOne
    @PrimaryKeyJoinColumn
    private Employee employee;
    // ...
}
```

We did not have to specify anything more than the `@PrimaryKeyJoinColumn` annotation since the entities had simple primary keys. If compound primary keys had been used, then additional information would need to be specified in the `@PrimaryKeyJoinColumn` annotation. We will discuss this case in Chapter 8.

Collection-Valued Associations

When the source entity references one or more target entity instances, a many-valued association or associated collection is used. Both the one-to-many and many-to-many mappings fit the criteria of having many target entities, and although the one-to-many association is the most frequently used, many-to-many mappings are useful as well when there is sharing in both directions.

One-to-Many Mappings

When an entity is associated with a `Collection` of other entities, it is most often in the form of a one-to-many mapping. For example, a department would normally have a number of employees. Figure 4-15 shows the `Employee` and `Department` relationship that we showed earlier in the section Many-to-One Mappings, only this time the relationship is bidirectional in nature.

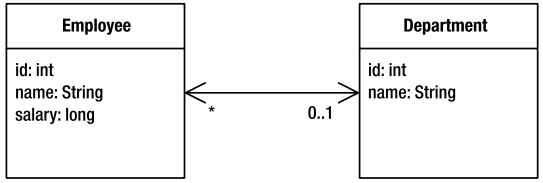

Figure 4-15. *Bidirectional* `Employee` *and* `Department` *relationship*

When a relationship is bidirectional, there are actually two mappings, one for each direction. A bidirectional one-to-many relationship always implies a many-to-one mapping back to the source, so in our `Employee` and `Department` example there is a one-to-many mapping from `Department` to `Employee` and a many-to-one mapping from `Employee` back to `Department`. We could just as easily say that the relationship is a bidirectional many-to-one if we were looking at it from the `Employee` perspective. They are equivalent, since bidirectional many-to-one relationships imply a one-to-many mapping back from the target to source, and vice versa.

When a source entity has an arbitrary number of target entities stored in its collection, there is no scalable way to store those references in the database table that it maps to. How would it store an arbitrary number of foreign keys in a single row? It must, rather, let the tables of the entities in the collection have foreign keys back to the source entity table. This is why the one-to-many association is almost always bidirectional and never the owning side.

Furthermore, if the target entity tables have foreign keys that point back to the source entity table then the target entities themselves should have many-to-one associations back to the source entity object. Having a foreign key in a table for which there is no association in the corresponding entity object model is not in keeping with the data model and not supported by the API.

Let's look at a concrete example of a one-to-many mapping based on the `Employee` and `Department` example shown in Figure 4-15. The tables for this relationship are exactly the same as those shown in Figure 4-10, where we talked about many-to-one relationships. The only difference between the many-to-one example and this one is that we are now implementing the inverse side of the relationship. Because `Employee` has the join column and is the owner of the relationship, the `Employee` class is unchanged from Listing 4-16.

On the Department side of the relationship, we need to map the employees collection of Employee entities as a one-to-many association using the @OneToMany annotation. Listing 4-20 shows the Department class that uses this annotation. Note that because this is the inverse side of the relationship, we need to include the mappedBy element just as we did in the bidirectional one-to-one relationship example.

Listing 4-20. *One-to-Many Relationship*

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    // ...
}
```

There are a couple of noteworthy points to mention about this class. The first is that a generic type-parameterized Collection is being used to store the Employee entities. This provides the strict typing that guarantees that only objects of type Employee will exist in the Collection. This, in and of itself, is quite useful since it not only provides compile-time checking of our code but also saves us having to perform cast operations when we retrieve the Employee instances from the collection.

The Java Persistence API assumes the availability of generics; however, it is still perfectly acceptable to use a Collection that is not type-parameterized. We might just as well have defined the Department class without using generics but defining only a simple Collection type, as we would have done in releases of standard Java previous to Java SE 5 (except for JDK 1.0 or 1.1 when java.util.Collection was not even standardized!). If we did, then we would need to specify the type of entity that will be stored in the Collection that is needed by the persistence provider. The code is shown in Listing 4-21 and looks almost identical, except for the targetEntity element that indicates the entity type.

Listing 4-21. *Using* targetEntity

```
@Entity
public class Department {
    @Id private int id;
    private String name;
    @OneToMany(targetEntity=Employee.class, mappedBy="department")
    private Collection employees;
    // ...
}
```

There are two important points to remember when defining bidirectional one-to-many (or many-to-one) relationships:

1. The many-to-one side is the owning side, so the join column is defined on that side.
2. The one-to-many mapping is the inverse side, so the mappedBy element must be used.

Failing to specify the mappedBy element in the @OneToMany annotation will cause the provider to treat it as a unidirectional one-to-many relationship that is defined to use a join table (described later). This is an easy mistake to make and should be the first thing you look for if you see a missing table error with a name that has two entity names concatenated together.

Many-to-Many Mappings

When one or more entities are associated with a Collection of other entities and the entities have overlapping associations with the same target entities, then we must model it as a many-to-many relationship. Each of the entities on each side of the relationship will have a collection-valued association that contains entities of the target type. Figure 4-16 shows a many-to-many relationship between Employee and Project. Each employee can work on multiple projects, and each project can be worked on by multiple employees.

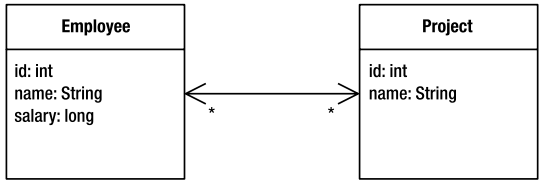

Figure 4-16. *Bidirectional many-to-many relationship*

A many-to-many mapping is expressed on both the source and target entities as an @ManyToMany annotation on the collection attributes. For example, in Listing 4-22 the Employee has a projects attribute that has been annotated with @ManyToMany. Likewise, the Project entity has an employees attribute that has also been annotated with @ManyToMany.

Listing 4-22. *Many-to-Many Relationship Between* Employee *and* Project

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    private Collection<Project> projects;
    // ...
}

@Entity
public class Project {
    @Id private int id;
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}
```

There are some important differences between this many-to-many relationship and the one-to-many relationship that we discussed earlier. The first is just a mathematical inevitability and is that when a many-to-many relationship is bidirectional, both sides of the relationship are many-to-many mappings.

The second difference is that there are no join columns on either side of the relationship. We will see in the next section that the only way to implement a many-to-many relationship is with a separate join table. The consequence of not having any join columns in either of the entity tables is that there is no way to determine which side is the owner of the relationship. Because every bidirectional relationship has to have both an owning side and an inverse side, we must pick one of the two entities to be the owner. In this example we have picked `Employee` to be owner of the relationship, but we could have just as easily picked `Project` instead. As in every other bidirectional relationship, the inverse side must use the `mappedBy` element to identify the owning attribute.

Note that no matter which side is designated as the owner, the other side should include the `mappedBy` element, otherwise the provider will think that both sides are the owner and that the mappings are separate unidirectional relationships.

Using Join Tables

Since the multiplicity of both sides of a many-to-many relationship is plural, neither of the two entity tables can store an unlimited set of foreign key values in a single entity row. We must use a third table to associate the two entity types. We call this association table a *join table*, and each many-to-many relationship must have one.

A join table consists simply of two foreign key or join columns to refer to each of the two entity types in the relationship. A collection of entities is then mapped as multiple rows in the table, each of which associates one entity with another. The set of rows that contains the same value in the foreign key column to an entity represents the associations that entity instance has with entity instances that it is related to.

In Figure 4-17 we see the `EMPLOYEE` and `PROJECT` tables for the `Employee` and `Project` entities and the `EMP_PROJ` join table that associates them. The `EMP_PROJ` table contains only foreign key columns that make up its compound primary key. The `EMP_ID` column refers to the `EMPLOYEE` primary key, while the `PROJ_ID` column refers to the `PROJECT` primary key.

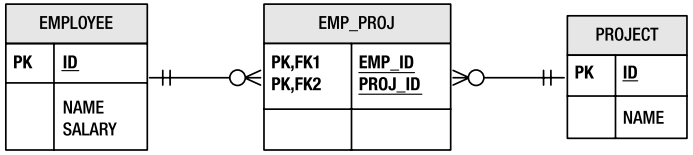

Figure 4-17. *Join table for a many-to-many relationship*

In order to map the tables described in Figure 4-17 we need to add some additional metadata to the `Employee` class that we have designated as the owner of the relationship. Listing 4-23 shows the many-to-many relationship with the accompanying join table annotations.

Listing 4-23. *Using a Join Table*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToMany
    @JoinTable(name="EMP_PROJ",
          joinColumns=@JoinColumn(name="EMP_ID"),
          inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Collection<Project> projects;
    // ...
}
```

The `@JoinTable` annotation is used to configure the join table for the relationship. The two join columns in the join table are distinguished by means of the owning and inverse sides. The join column to the owning side is described in the `joinColumns` element while the join column to the inverse side is specified by the `inverseJoinColumns` element. We can see from the previous example that the values of these elements are actually `@JoinColumn` annotations embedded within the `@JoinTable` annotation. This provides the ability to declare all of the information about the join columns within the table that defines them. The names are plural for the case when there may be multiple columns for each foreign key when either the owning entity or the inverse entity has a multipart primary key. This more complicated case will be discussed in Chapter 8.

In our example we fully specified the names of the join table and its columns because this is the most common case. But if we were generating the database schema from the entities, then we would not actually need to specify this information. We could have relied upon the default values that would be assumed and used when the persistence provider generates the table for us. When no `@JoinTable` annotation is present on the owning side, then a default join table named <Owner>_<Inverse> is assumed, where <Owner> is the name of the owning entity and <Inverse> is the name of the inverse or non-owning entity. Of course, the owner is basically picked at random by the developer so these defaults will apply according to the way the relationship is mapped and whichever entity is designated as the owning side.

The join columns will be defaulted according to the join column defaulting rules that were previously described in the section Using Join Columns. The default name of the join column that points to the owning entity is the name of the attribute on the inverse entity that points to the owning entity, appended by an underscore and the name of the primary key column of the owning entity table. So in our example the `Employee` is the owning entity, and the `Project` has an `employees` attribute that contains the collection of `Employee` instances. The `Employee` entity maps to the `EMPLOYEE` table and has a primary key column of `ID`, so the defaulted name of the join column to the owning entity would be `EMPLOYEES_ID`. The inverse join column would be likewise defaulted to be `PROJECTS_ID`.

It is fairly clear that the defaulted names of a join table and the join columns within it are not likely to match up with an existing table. This is why we mentioned that the defaults are really useful only if the database schema being mapped to was generated by the provider.

Unidirectional Collection Mappings

When an entity has a one-to-many mapping to a target entity but the `@OneToMany` annotation does not include the `mappedBy` element, it is assumed to be in a unidirectional relationship with the target entity. This means that the target entity does not have a many-to-one mapping back to the source entity. Figure 4-18 shows a unidirectional one-to-many association between `Employee` and `Phone`.

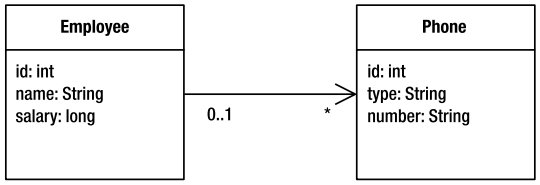

Figure 4-18. *Unidirectional one-to-many relationship*

Consider the data model in Figure 4-19. There is no join column to store the association back from `Phone` to `Employee`. Therefore, we have used a join table to associate the `Phone` entity with the `Employee` entity.

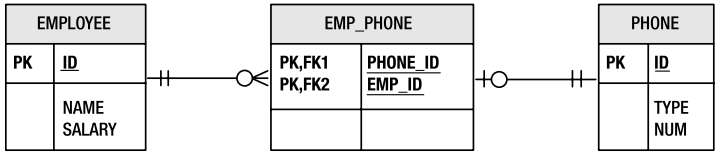

Figure 4-19. *Join table for a unidirectional one-to-many relationship*

Similarly, when one side of a many-to-many relationship does not have a mapping to the other, then it is a unidirectional relationship. The join table must still be used; the only difference is that only one of the two entity types actually uses the table to load its related entities or updates it to store additional entity associations.

In both of these two unidirectional collection-valued cases the source code is similar to the earlier examples, but there is no collection attribute in the target entity, and the `mappedBy` element will not be present in the `@OneToMany` annotation on the source entity. The join table must now be specified as part of the mapping. Listing 4-24 shows `Employee` with a one-to-many relationship to `Phone` using a join table.

Listing 4-24. *Unidirectional One-to-Many Relationship*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @OneToMany
    @JoinTable(name="EMP_PHONE",
          joinColumns=@JoinColumn(name="EMP_ID"),
          inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
    private Collection<Phone> phones;
    // ...
}
```

Note that when generating the schema, default naming for the join columns is slightly different in the unidirectional case because there is no inverse attribute. The name of the join table would default to `EMPLOYEE_PHONE` and would have a join column named `EMPLOYEE_ID` after the name of the `Employee` entity and its primary key column. The inverse join column would be named `PHONES_ID`, which is the concatenation of the `phones` attribute in the `Employee` entity and the `ID` primary key column of the `PHONE` table.

■ **Tip** Some vendors may provide support for a unidirectional one-to-many target foreign key mapping, where a join column exists in the target table but no relationship exists from the target entity back to the source entity. This is not supported in the current version of the Java Persistence API.

Using Different Collections

Different types of collections may be used to store multivalued entity associations. Depending upon the needs of the application, any of `Collection`, `Set`, `List`, and `Map` may be used for multivalued associations. There are only a couple of basic rules that guide their usage.

The first step is to define the collection to be any one of the interface types just mentioned. Implementation classes may not be used in the definition of the attribute but may be used as initial values set by the constructor or in an initialization method of the entity class.

The implementation class may be invoked through its implementation API at any time on a new object until the entity becomes managed or has been persisted by means of an `EntityManager.persist()` call. After that point, the interface must always be used when operating on the collection, whether it has been read in from the database or has been detached from the entity manager. This is because when the entity instance becomes managed, the persistence provider may have replaced the initial concrete instance with an alternate `Collection` implementation class of its own.

Using a Set or a Collection

The most common collection type used in associations is the standard `Collection` superinterface. This is used when it doesn't matter which implementation is underneath and when the common `Collection` methods are all that are required to access the entities stored in it.

A Set will prevent duplicate elements from being inserted and may be a simpler and more concise collection model, while a vanilla Collection interface is the most generic. Neither of these interfaces require accompanying annotations to further specify them. They are used the same way that they would be used if they held non-persistent objects.

Using a List

Another common collection type is the List. A List is often used when the entities are to be retrieved in some user-defined order, which may be optionally specified using an @OrderBy annotation. The value of the annotation is a string that indicates one or more fields or properties that are to be used to determine the order of the results, each of which may be optionally followed by an ASC or DESC keyword to define whether the attribute should be ordered in ascending or descending order. If the sequence direction is not specified, then the property will be listed in ascending order. Multiple attributes may be specified, each with their own ascending or descending sequence. If none are included then the list will be ordered using the primary keys of the entities. If the annotation is not present then the List will be in some undefined order, typically in the order returned by the database in the absence of any ORDER BY clause.

Let's take our example in Listing 4-20 and use a List instead of a Collection interface in the employees attribute of Department. By adding an @OrderBy annotation on the mapping, we will indicate that we want the employees to be ordered in ascending order according to the name attribute of the Employee. Listing 4-25 shows the updated example.

Listing 4-25. *One-to-Many Relationship Using a List*

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @OrderBy("name ASC")
    private List<Employee> employees;
    // ...
}
```

We needn't have included the ASC in the @OrderBy annotation since it would be ascending by default, but it is good documentation style to include it.

We might also want to have sub-orderings using multiple attributes, and we could do that by specifying comma-separated <attribute name ASC/DESC> pairs in the annotation. For example, if Employee had a status, then we might have ordered by status and then by name by using an @OrderBy annotation of @OrderBy("status DESC, name ASC"). Of course the prerequisite for using an attribute in an @OrderBy annotation is that the attribute type be comparable.

Order is not preserved, however, when writing the List out to the database. Since database rows do not have any inherent order, the entities in a Java List cannot retain their order unless a designated database column has been created to store it. Then when an entity gets added to the front of the List, each of the other entities in the List must be updated to reflect their new order assignment in the List.

The Java Persistence API does not include support for persistent order preservation that is not based upon some state of the entity, so changing the order of the items in a List in memory will not cause that order to be stored in the database at commit time. The order specified in

@OrderBy, which depends upon some aspect of the entity state, will be used only when reading the List back into memory. The rule of thumb, then, is that the entity List order should always be kept consistent by the application logic. Persisting the order of the List is being considered for inclusion in a subsequent version of the specification.

Using a Map

Sometimes a collection of entities is stored as a Map keyed by some attribute of the target entity type. When this is the case, the collection type may be declared to be of type Map, and the @MapKey annotation may be applied to the collection.

The map key designates the attribute of the target entity that is to be used to key on. It may be any persistent attribute on the target entity that is comparable and responds appropriately to the hashCode() and equals() methods. It should also be unique, at least in the collection domain, so that values are not lost or overwritten in memory.

If, for example, we want to use a Map instead of a List to store our employees in a department, then we could key the map on the employee name for quick employee lookup. The relevant Department code would look like Listing 4-26.

Listing 4-26. *One-to-Many Relationship Using a* Map

```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @MapKey(name="name")
    private Map<String, Employee> employees;
    // ...
}
```

If, for some reason we did not want or were not able to use the generic version of Map<KeyType, ValueType>, then we would instead define it using an old-style non-parameterized Map as shown in Listing 4-27.

Listing 4-27. *One-to-Many Relationship Using a Non-parameterized* Map

```
@Entity
public class Department {
    // ...
    @OneToMany(targetEntity=Employee.class, mappedBy="department")
    @MapKey(name="name")
    private Map employees;
    // ...
}
```

When the collection is of type Map but no @MapKey annotation is specified, the entities will be keyed by their primary key attributes. Entities that have multiple primary key fields will be keyed on instances of the primary key class.

Lazy Relationships

In previous sections we saw how we could configure an attribute to be loaded when it got accessed and not necessarily before. We learned that lazy loading at the attribute level is not normally going to be very beneficial.

At the relationship level, however, lazy loading can be a big boon to enhancing performance. It can reduce the amount of SQL that gets executed and speed up queries and object loading considerably.

The fetch mode can be specified on any of the four relationship mapping types. When not specified on a single-valued relationship the related object is guaranteed to be loaded eagerly. Collection-valued relationships default to be lazily loaded, but because lazy loading is only a hint to the provider, they may be loaded eagerly if the provider decides to do so.

In bidirectional relationship cases, the fetch mode may be lazy on one side but eager on the other. This kind of configuration is actually quite common since relationships are often accessed in different ways depending upon the direction from which navigation occurs.

An example of overriding the default fetch mode would be if we didn't want to load the `ParkingSpace` for an `Employee` every time we loaded the `Employee`. Listing 4-28 shows the `parkingSpace` attribute configured to use lazy loading.

Listing 4-28. *Changing the Fetch Mode on a Relationship*

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    // ...
}
```

■ **Tip** A relationship that is specified or defaulted to be lazily loaded may or may not cause the related object to be loaded when the getter method is used to access the object. The object may be a proxy, so it may take actually invoking a method on it to cause it to be faulted in.

Summary

Mapping objects to relational databases is of critical importance to persistence applications. Dealing with the impedance mismatch requires a sophisticated suite of metadata. The Java Persistence API not only provides this metadata, but also facilitates easy and convenient development.

In this chapter we went through the process of mapping entity state that included simple Java types, large objects, enumerated types, and temporal types. We also used the metadata to do meet-in-the-middle mapping to specific table names and columns.

We went over how identifiers are generated and described four different strategies of generation. We saw the different strategies in action and differentiated them from each other.

We then reviewed some of the relationship concepts and applied them to object-relational mapping metadata. We used join columns and join tables to map single-valued and collection-valued associations and went over some examples of using different kinds of `Collection` types.

In the next chapter we will discuss using entity managers and persistence contexts in more advanced ways than we did previously, delving into the practices and nuances of injecting and using them in Java EE and Java SE environments.

CHAPTER 5

Entity Manager

Entities do not persist themselves when they are created. Nor do they remove themselves from the database when they are garbage-collected. It is the business logic of the application that must manipulate entities to manage their persistent life cycle. The Java Persistence API provides the `EntityManager` interface for this purpose in order to let applications manage and search for entities in the relational database. At first this might seem like a limitation of the Java Persistence API. If the persistence runtime knows which objects are persistent, why should the application have to be involved in the process? Rest assured that this design is both deliberate and far more beneficial to the application than any transparent persistence solution. Persistence applications are a partnership between the application and persistence provider. The Java Persistence API brings a level of control and flexibility that cannot otherwise be achieved without the active participation of the application.

In Chapter 2 we introduced the `EntityManager` interface and described some of the basic operations that it provides for operating on entities. We extended that discussion in Chapter 3 to include an overview of the Java EE environment and the types of services that impact persistence applications. Finally, in the previous chapter we described object-relational mapping, the key to building entities out of objects. With that groundwork in place we are ready to revisit entity managers, persistence contexts, and persistence units, and to begin a more in-depth discussion of these concepts.

Persistence Contexts

Let's begin by reintroducing the core terms of the Java Persistence API. A persistence unit is a named configuration of entity classes. A persistence context is a managed set of entity instances. Every persistence context is associated with a persistence unit, restricting the classes of the managed instances to the set defined by the persistence unit. Saying that an entity instance is managed means that it is contained within a persistence context and that it may be acted upon by an entity manager. It is for this reason that we say an entity manager manages a persistence context.

Understanding the persistence context is the key to understanding the entity manager. An entity's inclusion or exclusion from a persistence context will determine the outcome of any persistent operations on it. If the persistence context participates in a transaction, then the in-memory state of the managed entities will get synchronized to the database. Yet despite the important role that it plays, the persistence context is never actually visible to the application. It is always accessed indirectly through the entity manager and assumed to be there when we need it.

So far so good, but how does the persistence context get created and when does this occur? How does the entity manager figure into the equation? This is where it starts to get interesting.

Entity Managers

Up to this point we have demonstrated only basic entity manager operations in both the Java SE and Java EE environments. We have reached a point, however, where we can finally reveal the full range of entity manager configurations. The Java Persistence API defines no less than *three* different types of entity managers, each of which has a different approach to persistence context management that is tailored to a different application need. As we will see, the persistence context is just one part of the puzzle.

Container-Managed Entity Managers

In the Java EE environment, the most common way to acquire an entity manager is by using the `@PersistenceContext` annotation to inject one. An entity manager obtained in this way is called *container-managed*, because the container manages the life cycle of the entity manager. The application does not have to create it or close it. This is the style of entity manager we demonstrated in Chapter 3.

Container-managed entity managers come in two varieties. The style of a container-managed entity manager determines how it works with persistence contexts. The first and most common style is called *transaction-scoped*. This means that the persistence contexts managed by the entity manager are determined by the active JTA transaction. The second style is called *extended*. Extended entity managers work with a single persistence context that is tied to the life cycle of a stateful session bean.

Transaction-Scoped

All of the entity manager examples that we have shown so far for the Java EE environment have been transaction-scoped entity managers. A transaction-scoped entity manager is returned whenever the reference created by the `@PersistenceContext` annotation is resolved. As we mentioned in Chapter 3, a transaction-scoped entity manager is stateless, meaning that it can be safely stored on any Java EE component. Because the container manages it for us, it is also basically maintenance-free.

Once again, let's introduce a stateless session bean that uses a transaction-scoped entity manager. Listing 5-1 shows the bean class for a session bean that manages project information. The entity manager is injected into the `em` field using the `@PersistenceContext` annotation and is then used in the business methods of the bean.

Listing 5-1. *The* `ProjectService` *Session Bean*

```
@Stateless
public class ProjectServiceBean implements ProjectService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;
```



```
    public void assignEmployeeToProject(int empId, int projectId) {
        Project project = em.find(Project.class, projectId);
        Employee employee = em.find(Employee.class, empId);
        project.getEmployees().add(employee);
        employee.getProjects().add(project);
    }

    // ...
}
```

We described the transaction-scoped entity manager as stateless. If that is the case, how can it work with a persistence context? The answer lies with the JTA transaction. All container-managed entity managers depend on JTA transactions. The reason for this is because they can use the transaction as a way to track persistence contexts. Every time an operation is invoked on the entity manager, it checks to see if a persistence context is associated with the transaction. If it finds one, the entity manager will use this persistence context. If it doesn't find one, then it creates a new persistence context and associates it with the transaction. When the transaction ends, the persistence context goes away.

Let's walk through an example. Consider the `assignEmployeeToProject()` method from Listing 5-1. The first thing the method does is search for the `Employee` and `Project` instances using the `find()` operation. When the first `find()` method is invoked, the container checks for a transaction. By default, the container will ensure that a transaction is active whenever a session bean method starts, so the entity manager in this example will find one ready. It then checks for a persistence context. This is the first time any entity manager call has occurred, so there isn't a persistence context yet. The entity manager creates a new one and uses it to find the employee.

When the entity manager is used to search for the employee, it checks the transaction again and this time finds the one it created when searching for the project. It then reuses this persistence context to search for the employee. At this point `employee` and `project` are both managed entity instances. The employee is then added to the project, updating both the `employee` and `project` entities. When the method call ends, the transaction is committed. Because the `employee` and `project` instances were managed, the persistence context is able to detect any state changes in them, and it updates the database during the commit. When the transaction is over, the persistence context goes away.

This process is repeated every time one or more entity manager operations are invoked within a transaction.

Extended

In order to describe the extended entity manager, we must first talk a little about stateful session beans. As we learned in Chapter 3, stateful session beans are designed to hold conversational state. Once acquired by a client, the same bean instance is used for the life of the conversation until the client invokes one of the methods marked `@Remove` on the bean. While the conversation is active, the business methods of the client may store and access information using the fields of the bean.

Let's try using a stateful session bean to help manage a department. Our goal is to create a business object for a `Department` entity that provides business operations relating to that entity. Listing 5-2 shows our first attempt. The business method `init()` is called by the client to initialize the department id. We then store this department id on the bean instance, and the `addEmployee()` method uses it to find the department and make the necessary changes. From the perspective of the client, they only have to set the department id once, and then subsequent operations always refer to the same department.

Listing 5-2. *First Attempt at Department Manager Bean*

```
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;
    int deptId;

    public void init(int deptId) {
        this.deptId = deptId;
    }

    public void setName(String name) {
        Department dept = em.find(Department.class, deptId);
        dept.setName(name);
    }

    public void addEmployee(int empId) {
        Department dept = em.find(Department.class, deptId);
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
    }

    // ...

    @Remove
    public void finished() {
    }
}
```

The first thing that should stand out when looking at this bean is that it seems unnecessary to have to search for the department every time. After all, we have the department id, why not just store the `Department` entity instance as well? Listing 5-3 revises our first attempt by searching for the department once during the `init()` method and then reusing the entity instance for each business method.

Listing 5-3. *Second Attempt at Department Manager Bean*

```
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;
    Department dept;

    public void init(int deptId) {
        dept = em.find(Department.class, deptId);
    }

    public void setName(String name) {
        dept.setName(name);
    }

    public void addEmployee(int empId) {
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
    }

    // ...

    @Remove
    public void finished() {
    }
}
```

This version looks better-suited to the capabilities of a stateful session bean. It is certainly more natural to reuse the `Department` entity instance instead of searching for it each time. But there is a problem. The entity manager in Listing 5-3 is transaction-scoped. Assuming there is no active transaction from the client, every method on the bean will start and commit a new transaction since the default transaction attribute for each method is `REQUIRED`. Because there is a new transaction for each method, the entity manager will use a different persistence context each time.

Even though the `Department` instance still exists, the persistence context that used to manage it went away when the transaction associated with the `init()` call ended. We refer to the `Department` entity in this case as being *detached* from a persistence context. The instance is still around and can be used, but any changes to its state will be ignored. For example, invoking `setName()` will change the name in the entity instance, but the changes will never be reflected in the database.

This is the situation that the extended entity manager is designed to solve. Designed specifically for stateful session beans, it prevents entities from becoming detached when transactions end. Before we go too much further, let's introduce our third and final attempt at a department manager bean. Listing 5-4 shows our previous example updated to use an extended persistence context.

Listing 5-4. *Using an Extended Entity Manager*

```
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Department dept;

    public void init(int deptId) {
        dept = em.find(Department.class, deptId);
    }

    public void setName(String name) {
        dept.setName(name);
    }

    public void addEmployee(int empId) {
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
    }

    // ...

    @Remove
    public void finished() {
    }
}
```

As you can see, we changed only one line. The `@PersistenceContext` annotation that we introduced in Chapter 3 has a special `type` element that may be set to either `TRANSACTION` or `EXTENDED`. These constants are defined by the `PersistenceContextType` enumerated type. `TRANSACTION` is the default and corresponds to the transaction-scoped entity managers we have been using up to now. `EXTENDED` means that an extended entity manager should be used.

With this change made, the department manager bean now works as expected. Extended entity managers create a persistence context when a stateful session bean instance is created that lasts until the bean is removed. Unlike the persistence context of a transaction-scoped entity manager that begins when the transaction begins and lasts until the end of a transaction, the persistence context of an extended entity manager will last for the entire length of the conversation. Because the `Department` entity is still managed by the same persistence context, any time it is used in a transaction, any changes will be automatically written to the database.

The extended persistence context allows stateful session beans to be written in a way that is more natural with respect to their capabilities. Later we will discuss special limitations on the transaction management of extended entity managers, but by and large they are well-suited to the type of example we have shown here.

The biggest limitation of the extended entity manager is that it requires a stateful session bean. Despite having been available in the EJB specification for many years, stateful session beans are still not widely used. Partly due to the poor quality of early vendor implementations, stateful session beans gained a reputation for poor performance and poor scalability. Even though modern servers are very efficient in their management of stateful session beans, developer skepticism remains. Given that the HTTP session offers similar capabilities and is readily available without developing new beans, developers have traditionally chosen that route over stateful session beans for conversational data.

More importantly, Java EE applications are largely stateless in nature. Many business operations do not require the kind of conversational state that stateful session beans provide. But that said, with the new ease-of-use features introduced in EJB 3.0 and the extended persistence context as a major new feature custom-tailored to stateful session beans, they may see more use in the future.

Application-Managed Entity Managers

In Chapter 2 we introduced the Java Persistence API with an example written using Java SE. The entity manager in that example, and any entity manager that is created from the `createEntityManager()` call of an `EntityManagerFactory` instance, is what we call an *application-managed* entity manager. This name comes from the fact that the application manages the life cycle of the entity manager instead of the container.

Although we expect the majority of applications to be written using container-managed entity managers, application-managed entity managers still have a role to play. They are the only entity manager type available in Java SE, and as we will see, they can be used in Java EE as well.

Creating an application-managed entity manager is simple enough. All you need is an `EntityManagerFactory` to create the instance. What separates Java SE and Java EE for application-managed entity managers is not how you create the entity manager but how you get the factory. Listing 5-5 demonstrates use of the `Persistence` class to bootstrap an `EntityManagerFactory` instance that is then used to create an entity manager.

Listing 5-5. *Application-Managed Entity Managers in Java SE*

```
public class EmployeeClient {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();

        Collection emps = em.createQuery("SELECT e FROM Employee e")
                            .getResultList();
        for (Iterator i = emps.iterator(); i.hasNext();) {
            Employee e = (Employee) i.next();
            System.out.println(e.getId() + ", " + e.getName());
        }

        em.close();
        emf.close();
    }
}
```

The Persistence class offers two variations of the same createEntityManager() method that may be used to create an EntityManagerFactory instance for a given persistence unit name. The first, specifying only the persistence unit name, returns the factory created with the default properties defined in the persistence.xml file. The second form of the method call allows a map of properties to be passed in, adding to or overriding the properties specified in persistence.xml. This form is useful when required JDBC properties may not be known until the application is started, perhaps with information provided as command-line parameters. We will discuss persistence unit properties in Chapter 11.

Creating an application-managed entity manager in Java EE requires using the @PersistenceUnit annotation to declare a reference to the EntityManagerFactory for a persistence unit. Once acquired, the factory can be used to create an entity manager, which may be used just as it would in Java SE. Listing 5-6 demonstrates injection of an EntityManagerFactory into a servlet and the use of it to create a short-lived entity manager in order to verify a user id.

Listing 5-6. *Application-Managed Entity Managers in Java EE*

```
public class LoginServlet extends HttpServlet {
    @PersistenceUnit(unitName="EmployeeService")
    EntityManagerFactory emf;

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
{
        String userId = request.getParameter("user");

        // check valid user
        EntityManager em = emf.createEntityManager();
        try {
            User user = em.find(User.class, userId);
            if (user == null) {
                // return error page
                // ...
            }
        } finally {
            em.close();
        }

        // ...
    }
}
```

One thing in common in both of these examples is that the entity manager is explicitly closed with the close() call when it is no longer needed. This is one of the lifecycle requirements of an entity manager that must be performed manually in the case of application-managed entity managers and that is normally taken care of automatically by container-managed entity managers. Likewise, the EntityManagerFactory instance must also be closed, but only in the Java SE application. In Java EE, the container closes the factory automatically, so no extra steps are required.

In terms of the persistence context, the application-managed entity manager is similar to an extended container-managed entity manager. When an application-managed entity manager

is created, it creates it own private persistence context that lasts until the entity manager is closed. This means that any entities managed by the entity manager will remain that way, independent of any transactions.

The role of the application-managed entity manager in Java EE is somewhat specialized. If resource-local transactions are required for an operation, an application-managed entity manager is the only type of entity manager that can be configured with that transaction type within the server. As we will describe in the next section, the transaction requirements of an extended entity manager can make them difficult to deal with in some situations. Application-managed entity managers can be safely used on stateful session beans to accomplish similar goals.

Transaction Management

Developing a persistence application is as much about transaction management as it is about object-relational mapping. Transactions define when new, changed, or removed entities are synchronized to the database. Understanding how persistence contexts interact with transactions is a fundamental part of working with the Java Persistence API.

Note that we said persistence contexts, not entity managers. There are several different entity manager types, but all use a persistence context internally. The entity manager type determines the lifetime of a persistence context, but all persistence contexts behave the same way when they are associated with a transaction.

There are two transaction-management types supported by the Java Persistence API. The first is resource-local transactions. These are the native transactions of the JDBC drivers that are referenced by a persistence unit. The second transaction-management type is the Java Transaction API, or JTA transactions. These are the transactions of the Java EE server, supporting multiple participating resources, transaction lifecycle management, and distributed XA transactions.

Container-managed entity managers always use JTA transactions, while application-managed entity managers may use either type. Because JTA is not typically available in Java SE applications, the provider need only support resource-local transactions in that environment. The default and preferred transaction type for Java EE applications is JTA. As we will describe in the next section, propagating persistence contexts with JTA transactions is a major benefit to enterprise persistence applications.

The transaction type is defined for a persistence unit and is configured using the persistence.xml file. We will discuss this setting and how to apply it in Chapter 11.

JTA Transaction Management

In order to talk about JTA transactions, we must first discuss the difference between transaction synchronization, transaction association, and transaction propagation. *Transaction synchronization* is the process by which a persistence context is registered with a transaction so that the persistence context may be notified when a transaction commits. The provider uses this notification to ensure that a given persistence context is correctly flushed to the database. *Transaction association* is the act of binding a persistence context to a transaction. You can also think of this as the *active* persistence context within the scope of that transaction. *Transaction propagation* is the process of sharing a persistence context between multiple container-managed entity managers in a single transaction.

There can be only one persistence context associated with and propagated across a JTA transaction. All container-managed entity managers in the same transaction must share the same propagated persistence context.

Transaction-Scoped Persistence Contexts

As the name suggests, a transaction-scoped persistence context is tied to the life cycle of the transaction. It is created by the container during a transaction and will be closed when the transaction completes. Transaction-scoped entity managers are responsible for creating transaction-scoped persistence contexts automatically when needed. We say only when needed because transaction-scoped persistence context creation is *lazy*. An entity manager will create a persistence context only when a method is invoked on the entity manager and when there is no persistence context available.

When a method is invoked on the transaction-scoped entity manager, it must first check if there is a propagated persistence context. If one exists, the entity manager uses this persistence context to carry out the operation. If one does not exist, the entity manager requests a new persistence context from the persistence provider and then marks this new persistence context as the propagated persistence context for the transaction before carrying out the method call. All subsequent transaction-scoped entity manager operations, in this component or any other, will thereafter use this newly created persistence context. This behavior works independently of whether or not container-managed or bean-managed transaction demarcation has been used.

Propagation of the persistence context simplifies the building of enterprise applications. When an entity is updated by a component inside of a transaction, any subsequent references to the same entity will always correspond to the correct instance, no matter what component obtains the entity reference. Propagating the persistence context gives developers the freedom to build loosely coupled applications knowing that they will always get the right data even though they are not sharing the same entity manager instance.

To demonstrate propagation of a transaction-scoped persistence context, let's introduce an audit service bean that stores information about a successfully completed transaction. Listing 5-7 shows the complete bean implementation. The `logTransaction()` method ensures that an employee id is valid by attempting to find the employee using the entity manager.

Listing 5-7. `AuditService` *Session Bean*

```
@Stateless
public class AuditServiceBean implements AuditService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public void logTransaction(int empId, String action) {
        // verify employee number is valid
        if (em.find(Employee.class, empId) == null) {
            throw new IllegalArgumentException("Unknown employee id");
        }
        LogRecord lr = new LogRecord(empId, action);
        em.persist(lr);
    }
}
```


Now consider the fragment from the `EmployeeService` session bean example shown in Listing 5-8. After an employee is created, the `logTransaction()` method of the `AuditService` session bean is invoked to record the "created employee" event.

Listing 5-8. *Logging* `EmployeeService` *Transactions*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    @EJB AuditService audit;

    public void createEmployee(Employee emp) {
        em.persist(emp);
        audit.logTransaction(emp.getId(), "created employee");
    }

    // ...
}
```

Even though the newly created `Employee` is not yet in the database, the audit bean is able to find the entity and verify that it exists. This works because the two beans are actually sharing the same persistence context. The transaction attribute of the `createEmployee()` method is `REQUIRED` by default since no attribute has been explicitly set. The container will guarantee that a transaction is started before the method is invoked. When `persist()` is called on the entity manager, the container checks to see if a persistence context is already associated with the transaction. Let's assume in this case that this was the first entity manager operation in the transaction, so the container creates a new persistence context and marks it as the propagated one.

When the `logTransaction()` method starts, it issues a `find()` call on the entity manager from the `AuditServiceBean`. We are guaranteed to be in a transaction, since the transaction attribute is also `REQUIRED` and the container-managed transaction from `createEmployee()` has been extended to this method by the container. When the `find()` method is invoked, the container again checks for an active persistence context. It finds the one created in the `createEmployee()` method and uses that persistence context to search for the entity. Since the newly created `Employee` instance is managed by this persistence context, it is returned successfully.

Now consider the case where `logTransaction()` has been declared with the `REQUIRES_NEW` transaction attribute instead of the default `REQUIRED`. Before the `logTransaction()` method call starts, the container will suspend the transaction inherited from `createEmployee()` and start a new transaction. When the `find()` method is invoked on the entity manager, it will check the current transaction for an active persistence context only to determine that one does not exist. A new persistence context will be created starting with the `find()` call, and this persistence context will be the active persistence context for the remainder of the `logTransaction()` call. Since the transaction started in `createEmployee()` has not yet committed, the newly created `Employee` instance is not in the database and therefore is not visible to this new persistence context. The `find()` method will return `null`, and the `logTransaction()` method will throw an exception as a result.

The rule of thumb for persistence context propagation is that the persistence context propagates as the JTA transaction propagates. Therefore it is important to understand not only when transactions begin and end, but also when a business method expects to inherit the transaction context from another method and when doing so would be incorrect. Having a clear plan for transaction management in your application is key to getting the most out of persistence context propagation.

Extended Persistence Contexts

The life cycle of an extended persistence context is tied to the stateful session bean to which it is bound. Unlike a transaction-scoped entity manager that creates a new persistence context for each transaction, the extended entity manager of a stateful session bean always uses the same persistence context. The stateful session bean is associated with a single extended persistence context that is created when the bean instance is created and closed when the bean instance is removed. This has implications for both the association and propagation characteristics of the extended persistence context.

Transaction association for extended persistence contexts is *eager*. In the case of container-managed transactions, as soon as a method call starts on the bean, the container automatically associates the persistence context with the transaction. Likewise in the case of bean-managed transactions; as soon as `UserTransaction.begin()` is invoked within a bean method, the container intercepts the call and performs the same association.

Because a transaction-scoped entity manager will use an existing persistence context associated with the transaction before it will create a new persistence context, it is possible to share an extended persistence context with other transaction-scoped entity managers. So long as the extended persistence context is propagated before any transaction-scoped entity managers are accessed, the same extended persistence context will be shared by all components.

Similar to the auditing `EmployeeServiceBean` we demonstrated in Listing 5-8, consider the same change made to a stateful session bean `DepartmentManagerBean` to audit when an employee is added to a department. Listing 5-9 shows this example.

Listing 5-9. *Logging Department Changes*

```
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Department dept;
    @EJB AuditService audit;

    public void init(int deptId) {
        dept = em.find(Department.class, deptId);
    }
```



```
    public void addEmployee(int empId) {
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
        audit.logTransaction(emp.getId(),
                             "added to department " + dept.getName());
    }

    // ...
}
```

The `addEmployee()` method has a default transaction attribute of `REQUIRED`. Since the container eagerly associates extended persistence contexts, the extended persistence context stored on the session bean will be immediately associated with the transaction when the method call starts. This will cause the relationship between the managed `Department` and `Employee` entities to be persisted to the database when the transaction commits. It also means that the extended persistence context will now be shared by other transaction-scoped persistence contexts used in methods called from `addEmployee()`.

The `logTransaction()` method in this example will inherit the transaction context from `addEmployee()` since its transaction attribute is the default `REQUIRED` and a transaction is active during the call to `addEmployee()`. When the `find()` method is invoked, the transaction-scoped entity manager checks for an active persistence context and will find the extended persistence context from the `DepartmentManagerBean`. It will then use this persistence context to execute the operation. All of the managed entities from the extended persistence context become visible to the transaction-scoped entity manager.

Persistence Context Collision

We said earlier that only one persistence context could be propagated with a JTA transaction. We also said that the extended persistence context would always try to make itself the active persistence context. This can quickly lead to situations where the two persistence contexts collide with each other. Consider, for example, that a stateless session bean with a transaction-scoped entity manager creates a new persistence context and then invokes a method on a stateful session bean with an extended persistence context. During the eager association of the extended persistence context, the container will check to see if there is already an active persistence context. If there is, it must be the same as the extended persistence context that it is trying to associate, or an exception will be thrown. In this example, the stateful session bean will find the transaction-scoped persistence context created by the stateless session bean, and the call into the stateful session bean method will fail. There can only be one active persistence context for a transaction.

While extended persistence context propagation is useful if a stateful session bean with an extended persistence context is the first EJB to be invoked in a call chain, it limits the situations in which other components can call into the stateful session bean if they too are using entity managers. This may or may not be common depending on your application architecture, but it is something to keep in mind when planning dependencies between components.

One way to work around this problem is to change the default transaction attribute for the stateful session bean that uses the extended persistence context. If the default transaction attribute is `REQUIRES_NEW`, then any active transaction will be suspended before the stateful

session bean method starts, allowing it to associate its extended persistence context with the new transaction. This is a good strategy if the stateful session bean calls in to other stateless session beans and needs to propagate the persistence context. Note that excessive use of the REQUIRES_NEW transaction attribute can lead to application performance problems as many more transactions than normal will be created and active transactions will be suspended and resumed.

If the stateful session bean is largely self-contained, that is, it does not call other session beans and does not need its persistence context propagated, then a default transaction attribute type of NOT_SUPPORTED may be worth considering. In this case, any active transaction will be suspended before the stateful session bean method starts, but no new transaction will be started. If there are some methods that need to write data to the database, then those methods can be overridden to use the REQUIRES_NEW transaction attribute.

Listing 5-10 repeats the DepartmentManager bean, this time with some additional getter methods and customized transaction attributes. We have set the default transaction attribute to REQUIRES_NEW to force a new transaction by default when a business method is invoked. For the getName() method, we don't need a new transaction since no changes are being made, so it has been set to NOT_SUPPORTED. This will suspend the current transaction but won't result in a new transaction being created. With these changes, the DepartmentManager bean may be accessed in any situation, even if there is already an active persistence context.

Listing 5-10. *Customizing Transaction Attributes to Avoid Collision*

```
@Stateful
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Department dept;
    @EJB AuditService audit;

    public void init(int deptId) {
        dept = em.find(Department.class, deptId);
    }

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public String getName() { return dept.getName(); }
    public void setName(String name) { dept.setName(name); }

    public void addEmployee(int empId) {
        Employee emp = em.find(empId, Employee.class);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
        audit.logTransaction(emp.getId(),
                             "added to department " + dept.getName());
    }

    // ...
}
```


Finally, one last option to consider is using an application-managed entity manager instead of an extended entity manager. If there is no need to propagate the persistence context, then the extended entity manager is not adding a lot of value over an application-managed entity manager. The stateful session bean can safely create an application-managed entity manager, store it on the bean instance, and use it for persistence operations without having to worry about whether or not an active transaction already has a propagated persistence context. An example of this technique is demonstrated later in the section Application-Managed Persistence Contexts.

Persistence Context Inheritance

The restriction of only one stateful session bean with an extended persistence context being able to participate in a JTA transaction can be a limitation in some situations. For example, the pattern we followed earlier in this chapter for the extended persistence context was to encapsulate the behavior of an entity behind a stateful session façade. In our example, clients worked with a DepartmentManager session bean instead of the actual Department entity instance. Since a department has a manager, it makes sense to extend this façade to the Employee entity as well.

Listing 5-11 shows changes to the DepartmentManager bean so that it returns an EmployeeManager stateful session bean from the getManager() method in order to represent the manager of the department. The EmployeeManager stateful session bean is injected and then initialized during the invocation of the init() method.

Listing 5-11. *Creating and Returning a Stateful Session Bean*

```
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Department dept;
    @EJB EmployeeManager manager;

    public void init(int deptId) {
        dept = em.find(Department.class, deptId);
        manager.init();
    }

    public EmployeeManager getManager() {
        return manager;
    }

    // ...
}
```

Should the init() method succeed or fail? So far based on what we have described, it looks like it should fail. When init() is invoked on the DepartmentManager bean, its extended persistence context will be propagated with the transaction. In the subsequent call to init() on the EmployeeManager bean, it will attempt to associate its own extended persistence context with the transaction, causing a collision between the two.

Perhaps surprisingly, this example actually works. When a stateful session bean with an extended persistence context creates another stateful session bean that also uses an extended persistence context, the child will inherit the parent's persistence context. The `EmployeeManager` bean inherits the persistence context from the `DepartmentManager` bean when it is injected into the `DepartmentManager` instance. The two beans can now be used together within the same transaction.

Application-Managed Persistence Contexts

Like container-managed persistence contexts, application-managed persistence contexts may be synchronized with JTA transactions. Synchronizing the persistence context with the transaction means that a flush will occur if the transaction commits, but the persistence context will not be considered associated by any container-managed entity managers. There is no limit to the number of application-managed persistence contexts that may be synchronized with a transaction, but only one container-managed persistence context will ever be associated. This is one of the most important differences between application-managed and container-managed entity managers.

An application-managed entity manager participates in a JTA transaction in one of two ways. If the persistence context is created inside the transaction, then the persistence provider will automatically synchronize the persistence context with the transaction. If the persistence context was created earlier (outside of a transaction or in a transaction that has since ended), the persistence context may be manually synchronized with the transaction by calling `joinTransaction()` on the `EntityManager` interface. Once synchronized, the persistence context will automatically be flushed when the transaction commits.

Listing 5-12 shows a variation of the `DepartmentManagerBean` from Listing 5-11 that uses an application-managed entity manager instead of an extended entity manager.

Listing 5-12. *Using Application-Managed Entity Managers with JTA*

```
@Stateful
public class DepartmentManagerBean implements DepartmentManager {
    @PersistenceUnit(unitName="EmployeeService")
    EntityManagerFactory emf;
    EntityManager em;
    Department dept;

    public void init(int deptId) {
        em = emf.createEntityManager();
        dept = em.find(Department.class, deptId);
    }

    public String getName() {
        return dept.getName();
    }
```



```
    public void addEmployee(int empId) {
        em.joinTransaction();
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
    }

    // ...

    @Remove
    public void finished() {
        em.close();
    }
}
```

Instead of injecting an entity manager, we are injecting an entity manager factory. Prior to searching for the entity, we manually create a new application-managed entity manager using the factory. Because the container does not manage its life cycle, we have to close it later when the bean is removed during the call to `finished()`. Like the container-managed extended persistence context, the `Department` entity remains managed after the call to `init()`. When `addEmployee()` is called, there is the extra step of calling `joinTransaction()` to notify the persistence context that it should synchronize itself with the current JTA transaction. Without this call, the changes to `Department` would not be flushed to the database when the transaction commits.

Because application-managed entity managers do not propagate, the only way to share managed entities with other components is to share the `EntityManager` instance. This can be achieved by passing the entity manager around as an argument to local methods or by storing the entity manager in a common place such as an HTTP session. Listing 5-13 demonstrates a servlet creating an application-managed entity manager and using it to instantiate the `EmployeeService` class we defined in Chapter 2. In these cases, care must be taken to ensure that access to the entity manager is done in a thread-safe manner. While `EntityManagerFactory` instances are thread-safe, `EntityManager` instances are not. Also, application code must not call `joinTransaction()` on the same entity manager in multiple concurrent transactions.

Listing 5-13. *Sharing an Application-Managed Entity Manager*

```
public class EmployeeServlet extends HttpServlet {
    @PersistenceUnit(unitName="EmployeeService")
    EntityManagerFactory emf;
    @Resource UserTransaction tx;

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // ...
        int id = Integer.parseInt(request.getParameter("id"));
        String name = request.getParameter("name");
        long salary = Long.parseLong(request.getParameter("salary"));
        tx.begin();
```

```
        EntityManager em = emf.createEntityManager();
        try {
            EmployeeService service = new EmployeeService(em);
            service.createEmployee(id, name, salary);
        } finally {
            em.close();
        }
        tx.commit();
        // ...
    }
}
```

Listing 5-13 demonstrates an additional characteristic of the application-managed entity manager in the presence of transactions. If the persistence context becomes synchronized with a transaction, changes will still be written to the database when the transaction commits, even if the entity manager is closed. This allows entity managers to be closed at the point where they are created without the need to worry about closing them after the transaction ends. Note that closing an application-managed entity manager still prevents any further use of the entity manager. It is only the persistence context that continues until the transaction has completed.

There is a danger in mixing multiple persistence contexts in the same JTA transaction. This occurs when multiple application-managed persistence contexts become synchronized with the transaction or when application-managed persistence contexts become mixed with container-managed persistence contexts. When the transaction commits, each persistence context will receive notification from the transaction manager that changes should be written to the database. This will cause each persistence context to be flushed.

What happens if an entity with the same primary key is used in more than one persistence context? Which version of the entity gets stored? The unfortunate answer is that there is no way to know for sure. The container does not guarantee any ordering when notifying persistence contexts of transaction completion. As a result, it is critical for data integrity that entities never be used by more than one persistence context in the same transaction. When designing your application, we recommend picking a single persistence context strategy (container-managed or application-managed) and sticking to that strategy consistently.

Resource-Local Transactions

Resource-local transactions are controlled explicitly by the application. The application server, if there is one, has no part in the management of the transaction. Applications interact with resource-local transactions by acquiring an implementation of the `EntityTransaction` interface from the entity manager. The `getTransaction()` method of the `EntityManager` interface is used for this purpose.

The `EntityTransaction` interface is designed to imitate the `UserTransaction` interface defined by the Java Transaction API, and the two behave very similarly. The main difference is that `EntityTransaction` operations are implemented in terms of the transaction methods on the JDBC `Connection` interface. Listing 5-14 shows the complete `EntityTransaction` interface.

Listing 5-14. *The* `EntityTransaction` *Interface*

```
public interface EntityTransaction {
    public void begin();
    public void commit();
    public void rollback();
    public void setRollbackOnly();
    public void getRollbackOnly();
    public void isActive();
}
```

There are only six methods on the `EntityTransaction` interface. The `begin()` method starts a new resource transaction. If a transaction is active, `isActive()` will return true. Attempting to start a new transaction while a transaction is active will result in an `IllegalStateException` being thrown. Once active, the transaction may be committed by invoking `commit()` or rolled back by invoking `rollback()`. Both operations will fail with an `IllegalStateException` if there is no active transaction. A `PersistenceException` will be thrown if an error occurs during rollback, while a `RollbackException` will be thrown if the commit fails.

If a persistence operation fails while an `EntityTransaction` is active, the provider will mark it for rollback. It is the application's responsibility to ensure that the rollback actually occurs by calling `rollback()`. If the transaction is marked for rollback and a commit is attempted, a `RollbackException` will be thrown. To avoid this exception, the `getRollbackOnly()` method may be called to determine whether the transaction is in a failed state. Until the transaction is rolled back, it is still active and will cause any subsequent commit or begin operation to fail.

Listing 5-15 shows a Java SE application that uses the `EntityTransaction` API to perform a password change for users who failed to update their passwords before they expired.

Listing 5-15. *Using the* `EntityTransaction` *Interface*

```
public class ExpirePasswords {
    public static void main(String[] args) {
        int maxAge = Integer.parseInt(args[0]);
        String defaultPassword = args[1];

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("admin");
        try {
            EntityManager em = emf.createEntityManager();

            Calendar cal = Calendar.getInstance();
            cal.add(Calendar.DAY_OF_YEAR, -maxAge);

            em.getTransaction().begin();
            Collection expired =
                em.createQuery("SELECT u FROM User u WHERE u.lastChange > ?1")
                  .setParameter(1, cal)
                  .getResultList();
```

```
            for (Iterator i = expired.iterator(); i.hasNext();) {
                User u = (User) i.next();
                System.out.println("Expiring password for " + u.getName());
                u.setPassword(defaultPassword);
            }
            em.getTransaction().commit();
            em.close();
        } finally {
            emf.close();
        }
    }
}
```

Within the application server, JTA transaction management is the default and should be used by most applications. One example use of resource-local transactions in the Java EE environment might be for logging. If your application requires an audit log stored in the database that must be written regardless of the outcome of any JTA transactions, then a resource-local entity manager may be used to persist data outside of the current transaction. Resource transactions may be freely started and committed any number of times within a JTA transaction without impacting the state of the JTA transactions.

Listing 5-16 shows an example of a stateless session bean that provides audit logging that will succeed even if the active JTA transaction fails.

Listing 5-16. *Using Resource-Local Transactions in the Java EE Environment*

```
@Stateless
public class LogServiceBean implements LogService {
    @PersistenceUnit(unitName="logging")
    EntityManagerFactory emf;

    public void logAccess(int userId, String action) {
        EntityManager em = emf.createEntityManager();
        try {
            LogRecord lr = new LogRecord(userId, action);
            em.getTransaction().begin();
            em.persist(lr);
            em.getTransaction().commit();
        } finally {
            em.close();
        }
    }
}
```

Of course, you could make the argument that this is overkill for a simple logging bean. Direct JDBC would probably work just as easily, but these same log records may have uses elsewhere in the application. It is a trade-off in configuration (defining a completely separate persistence unit in order to enable the resource-local transactions) versus the convenience of having an object-oriented representation of a log record.

Transaction Rollback and Entity State

When a database transaction is rolled back, all of the changes made during the transaction are abandoned. The database reverts to whatever state it was in before the transaction began. But as mentioned in Chapter 2, the Java memory model is not transactional. There is no way to take a snapshot of object state and revert to it later if something goes wrong. One of the harder parts of using an object-relational mapping solution is that while we can use transactional semantics in our application to control whether or not data is committed to the database, we can't truly apply the same techniques to the in-memory persistence context that manages our entity instances.

Any time we are working with changes that must be persisted to the database, we are working with a persistence context synchronized with a transaction. At some point during the life of the transaction, usually just before it commits, the changes we require will be translated into the appropriate SQL statements and sent to the database. Whether we are using JTA transactions or resource-local transactions is irrelevant. We have a persistence context participating in a transaction with changes that need to be made.

If that transaction rolls back, two things happen. The first is that the database transaction will be rolled back. The next thing that happens is that the persistence context is cleared, detaching all of our managed entity instances. If the persistence context was transaction-scoped, then it is removed.

Because the Java memory model is not transactional, we are basically left with a bunch of detached entity instances. More importantly, these detached instances reflect the entity state exactly as it was at the point when the rollback occurred. Faced with a rolled-back transaction and detached entities, you might be tempted to start a new transaction, merge the entities into the new persistence context, and start over. The following issues need to be considered in this case:

- If there is a new entity that uses automatic primary key generation, there may be a primary key value assigned to the detached entity. If this primary key was generated from a database sequence or table, the operation to generate the number may have been rolled back with the transaction. This means that the same sequence number could be given out again to a different object. Clear the primary key before attempting to persist the entity again, and do not rely on the primary key value in the detached entity.
- If your entity uses a version field for locking purposes that is automatically maintained by the persistence provider, it may be set to an incorrect value. The value in the entity will not match the correct value stored in the database. We will cover locking and versioning in Chapter 9.

If you need to reapply some of the changes that failed and are currently sitting in the detached entities, consider selectively copying the changed data into new managed entities. This guarantees that the merge operation will not be compromised by stale data left in the detached entity. To merge failed entities into a new persistence context, some providers may offer additional options that avoid some or all of these issues.

Choosing an Entity Manager

With three different entity manager types, each with a different life cycle and different rules about transaction association and propagation, it can all be a little overwhelming. What style is

right for your application? Application-managed or container-managed? Transaction-scoped or extended?

Generally speaking, we believe that container-managed, transaction-scoped entity managers are the best model for most applications. This is the design that originally inspired the Java Persistence API and is the model that commercial persistence providers have been using for years. The selection of this style to be the default for Java EE applications was no accident. It offers the best combination of flexible transaction propagation with easy-to-understand semantics.

Extended persistence contexts are effectively a new programming model introduced by this specification. Although commercial vendors have had similar features to allow entities to remain managed after commit, never before has such a feature been fully integrated into the life cycle of a Java EE component, in this case the stateful session bean. There are some interesting new techniques possible with the extended persistence context (some of which we will describe later in this chapter), but these may not apply to all applications.

In most enterprise applications, application-managed entity managers are unlikely to see much use. There is rarely a need for persistence contexts that are not associated with a container transaction and that remain isolated from the rest of the container-managed persistence contexts. The lack of propagation means that application-managed entity managers must be passed around as method arguments or stored in a shared object in order to share the persistence context. Evaluate application-managed entity managers based on your expected transactional needs and the size and complexity of your application.

More than anything, we recommend that you try to be consistent in how entity managers are selected and applied. Mixing all three entity manager types into an application is likely to be frustrating as the different entity manager types can intersect in unexpected ways.

Entity Manager Operations

Armed with information about the different entity manager types and how they work with persistence contexts, we can now revisit the basic entity manager operations we introduced in Chapter 2 and reveal more of the details. The following sections describe the entity manager operations with respect to the different entity manager and persistence context types.

Persisting an Entity

The `persist()` method of the `EntityManager` interface accepts a new entity instance and causes it to become managed. If the entity to be persisted is already managed by the persistence context, then it is ignored. The `contains()` operation can be used to check whether an entity is already managed, but it is very rare that this should be required. It should not come as a surprise to the application to find out which entities are managed and which are not. The design of the application dictates when entities become managed.

For an entity to be managed does not mean that it is persisted to the database right away. The actual SQL to create the necessary relational data will not be generated until the persistence context is synchronized with the database, typically only when the transaction commits. However, once a new entity is managed, any changes to that entity may be tracked by the persistence context. Whatever state exists on the entity when the transaction commits is what will be written to the database.

When `persist()` is invoked outside of a transaction, the behavior depends on the type of entity manager. A transaction-scoped entity manager will throw a `TransactionRequiredException` as there is no persistence context available in which to make the entity managed. Application-managed and extended entity managers will accept the persist request, causing the entity to become managed, but no immediate action will be taken until a new transaction begins and the persistence context becomes synchronized with the transaction. In effect, this queues up the change to happen at a later point in time. It is only when the transaction commits that changes will be written out to the database.

The `persist()` operation is intended for new entities that do not already exist in the database. If the provider immediately determines that this is not true, then an `EntityExistsException` will be thrown. If the provider does not make this determination and the primary key is in fact a duplicate, then an exception will be thrown when the persistence context is synchronized to the database.

Up to this point we have been discussing the persistence of entities only without relationships. But, as we learned in Chapter 4, the Java Persistence API supports a wide variety of relationship types. In practice, most entities are in a relationship with at least one other entity. Consider the following sequence of operations:

```
Department dept = em.find(Department.class, 30);
Employee emp = new Employee();
emp.setId(53);
emp.setName("Peter");
emp.setDepartment(dept);
dept.getEmployees().add(emp);
em.persist(emp);
```

Despite the brevity of this example, we have covered a lot of points relating to persisting a relationship. We begin by retrieving a pre-existing `Department` instance. A new `Employee` instance is then created, supplying the primary key and basic information about the `Employee`. We then assign the employee to the department, by setting the `department` attribute of the `Employee` to point to the `Department` instance we retrieved earlier. Because the relationship is bidirectional, we then add the new `Employee` instance to the `employees` collection in the `Department` instance. Finally the new `Employee` instance is persisted with the call to `persist()`. Assuming a transaction then commits, the new entity will be stored in the database.

An interesting thing about this example is that the `Department` is a passive participant despite the `Employee` instance being added to its collection. The `Employee` entity is the owner of the relationship because it is in a many-to-one relationship with the `Department`. As we mentioned in the previous chapter, the source side of the relationship is the owner, while the target is the inverse in this type of relationship. When the `Employee` is persisted, the foreign key to the `Department` is written out to the table mapped by the `Employee`, and no actual change is made to the `Department` entity's physical representation. Had we only added the employee to the collection and not updated the other side of the relationship, nothing would have been persisted to the database.

Finding an Entity

The ever-present `find()` method is the workhorse of the entity manager. Whenever an entity needs to be located by its primary key, `find()` is usually the best way to go. Not only does it have

simple semantics, but most persistence providers will also optimize this operation to use an in-memory cache that minimizes trips to the database.

The `find()` operation returns a managed entity instance in all cases except when invoked outside of a transaction on a transaction-scoped entity manager. In this case, the entity instance is returned in a detached state. It is not associated with any persistence context.

There exists a special version of `find()` that may be used in one particular situation. That situation is when a relationship is being created between two entities in a one-to-one or many-to-one relationship where the target entity already exists and its primary key is well-known. Since we are only creating a relationship, it may not be necessary to fully load the target entity in order to create the foreign key reference to it. Only its primary key is required. The `getReference()` operation may be used for this purpose. Consider the following example:

```
Department dept = em.getReference(Department.class, 30);
Employee emp = new Employee();
emp.setId(53);
emp.setName("Peter");
emp.setDepartment(dept);
dept.getEmployees().add(emp);
em.persist(emp);
```

The only difference between this sequence of operations and the ones we demonstrated earlier is that the `find()` call has been replaced with a call to `getReference()`. When the `getReference()` call is invoked, the provider may return a proxy to the `Department` entity without actually retrieving it from the database. So long as only its primary key is accessed, `Department` data does not need to be fetched. Instead, when the `Employee` is persisted, the primary key value will be used to create the foreign key to the corresponding `Department` entry. The `getReference()` call is effectively a performance optimization that removes the need to retrieve the target entity instance.

There are some drawbacks to using `getReference()` that must be understood. The first is that if a proxy is used, then it may throw an `EntityNotFoundException` exception if it is unable to locate the real entity instance when an attribute other the primary key is accessed. The assumption with `getReference()` is that you are sure the entity with the correct primary key exists. If, for some reason, an attribute other than the primary key is accessed and the entity does not exist, then an exception will be thrown. A corollary to this is that the object returned from `getReference()` may not be safe to use if it is no longer managed. If the provider returns a proxy, it will be dependent on there being an active persistence context to load entity state.

Given the very specific situation in which `getReference()` may be used, `find()` should be used in virtually all cases. The in-memory cache of a good persistence provider is effective enough that the performance cost of accessing an entity via its primary key will not usually be noticed. In the case of TopLink Essentials, it has a fully integrated shared object cache, so not only is local persistence context management efficient, but also all threads on the same server can benefit from the shared contents of the cache. The `getReference()` call is a performance optimization that should be used only when there is evidence to suggest that it will actually benefit the application.

Removing an Entity

Removing an entity is not a complex task, but it can require several steps depending on the number of relationships in the entity to be removed. At its most basic, removing an entity is simply a case of passing a managed entity instance to the `remove()` method of an entity manager. As soon as the associated persistence context becomes synchronized with a transaction and commits, the entity is removed. At least that is what we would like to happen. As we will soon show, removing an entity requires some attention to the relationships of an entity, or the integrity of the database can be compromised in the process.

Let's walk through a simple example. Consider the `Employee` and `ParkingSpace` relationship that we demonstrated in Chapter 4. The `Employee` has a unidirectional one-to-one relationship with the `ParkingSpace` entity. Now imagine that we execute the following code inside a transaction, where `empId` corresponds to an `Employee` primary key:

```
Employee emp = em.find(Employee.class, empId);
em.remove(emp.getParkingSpace());
```

When the transaction commits, we see the DELETE statement for the `PARKING_SPACE` table get generated, but then we get an exception containing a database error that shows that we have violated a foreign key constraint. It turns out that a referential integrity constraint exists between the `EMPLOYEE` table and the `PARKING_SPACE` table. The row was deleted from the `PARKING_SPACE` table, but the corresponding foreign key in the `EMPLOYEE` table was not set to `NULL`. To correct the problem we have to explicitly set the `parkingSpace` attribute of the `Employee` entity to `null` before the transaction commits:

```
Employee emp = em.find(Employee.class, empId);
ParkingSpace ps = em.getParkingSpace();
em.setParkingSpace(null);
em.remove(ps);
```

Relationship maintenance is the responsibility of the application. We are going to repeat this statement over the course of this book, but it cannot be emphasized enough. Almost every problem related to removing an entity always comes back to this issue. If the entity to be removed is the target of foreign keys in other tables, then those foreign keys must be cleared in order for the remove to succeed. The remove operation will either fail as it did here, or it will result in stale data being left in the foreign key columns referring to the removed entity in the event that there is no referential integrity.

An entity may be removed only if it is managed by a persistence context. This means that a transaction-scoped entity manager may be used to remove an entity only if there is an active transaction. Attempting to invoke `remove()` when there is no transaction will result in a `TransactionRequiredException` exception. Like the `persist()` operation we described earlier, application-managed and extended entity managers can remove an entity outside of a transaction, but the change will not take place in the database until a transaction involving the persistence context is committed.

After the transaction has committed, all entities that were removed in that transaction are left in the state that they were in before they were removed. A removed entity instance can be persisted again with the `persist()` operation, but the same issues with generated state that we discussed in the Transaction Rollback and Entity State section apply here as well.

Cascading Operations

By default, every entity manager operation applies only to the entity supplied as an argument to the operation. The operation will not cascade to other entities that have a relationship with the entity that is being operated on. For some operations, such as `remove()`, this is usually the desired behavior. We wouldn't want the entity manager to make incorrect assumptions about which entity instances should be removed as a side effect from some other operation. But the same does not hold true for operations such as `persist()`. Chances are that if we have a new entity and it has a relationship to another new entity, the two must be persisted together.

Consider the sequence of operations in Listing 5-17 that are required to create a new `Employee` entity with an associated `Address` entity and make the two persistent. The second call to `persist()` that makes the `Address` entity managed is bothersome. An `Address` entity is coupled to the `Employee` entity that holds on to it. Whenever a new `Employee` is created, it makes sense to cascade the `persist()` operation to the `Address` entity if it is present.

Listing 5-17. *Persisting Employee and Address Entities*

```
Employee emp = new Employee();
emp.setId(2);
emp.setName("Rob");
Address addr = new Address();
addr.setStreet("645 Stanton Way");
addr.setCity("Manhattan");
addr.setState("NY");
emp.setAddress(addr);
em.persist(addr);
em.persist(emp);
```

Fortunately the Java Persistence API provides a mechanism to define when operations such as `persist()` should be cascaded across relationships. The `cascade` element, in all of the logical relationship annotations (`@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`), defines the list of entity manager operations to be cascaded.

Entity manager operations are identified using the `CascadeType` enumerated type when listed as part of the `cascade` element. The `PERSIST`, `REFRESH`, `REMOVE`, and `MERGE` constants pertain to the entity manager operation of the same name. The constant `ALL` is shorthand for declaring that all four operations should be cascaded.

The following sections will define the cascading behavior of the `persist()` and `remove()` operations. We will introduce the `merge()` operation and its cascading behavior later in this chapter in the section Merging Detached Entities. Likewise, we will introduce the `refresh()` operation and its cascading behavior in Chapter 9.

Cascade Persist

To begin, let's consider the changes required to make the `persist()` operation cascade from `Employee` to `Address`. In the definition of the `Employee` class, there is an `@ManyToOne` annotation defined for the `address` relationship. To enable the cascade, we must add the `PERSIST` operation to the list of cascading operations for this relationship. Listing 5-18 shows a fragment of the `Employee` entity that demonstrates this change.

Listing 5-18. *Enabling Cascade Persist*

```
@Entity
public class Employee {
    // ...
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
    // ...
}
```

To leverage this change, we need only ensure that the `Address` entity has been set on the `Employee` instance before invoking `persist()` on it. As the entity manager encounters the `Employee` instance and adds it to the persistence context, it will navigate across the `address` relationship looking for a new `Address` entity to manage as well. In comparison to the approach in Listing 5-16, this change frees us from having to persist the `Address` separately.

Cascade settings are unidirectional. This means that it must be explicitly set on both sides of a relationship if the same behavior is intended for both situations. For example, in Listing 5-17, we only added the cascade setting to the `address` relationship in the `Employee` entity. If Listing 5-16 were changed to persist only the `Address` entity and not the `Employee` entity, then the `Employee` entity would not become managed, because the entity manager has not been instructed to navigate out from any relationships defined on the `Address` entity.

Even though it is legal to do so, it is still unlikely that we would add cascading operations from the `Address` entity to the `Employee` entity, because it is a child of the `Employee` entity. While causing the `Employee` instance to become managed as a side effect of persisting the `Address` instance is harmless, application code would not expect the same from the `remove()` operation, for example. Therefore we must be judicious in applying cascades, because there is an expectation of ownership in relationships that influences what developers expect when interacting with these entities.

In the Persisting an Entity section, we mentioned that the entity instance is ignored if it is already persisted. This is true, but the entity manager will still honor the `PERSIST` cascade in this situation. For example, consider our `Employee` entity again. If the `Employee` instance is already managed and a new `Address` instance is set in it, then invoking `persist()` again on the `Employee` instance will cause the `Address` instance to become managed. No changes will be made to the `Employee` instance since it is already managed.

As adding the `PERSIST` cascade is a very common and desirable behavior for relationships, it is possible to make this the default cascade setting for all relationships in the persistence unit. We will discuss this technique in Chapter 10.

Cascade Remove

At first glance, having the entity manager automatically cascade `remove()` operations may sound attractive. Depending on the cardinality of the relationship, it could eliminate the need to explicitly remove multiple entity instances. And yet, while we could cascade this operation in a number of situations, this should be applied only in certain cases. There are really only two cases where cascading the `remove()` operation makes sense: one-to-one and one-to-many relationships where there is a clear parent-child relationship. It can't be blindly applied to all one-to-one and one-to-many relationships because the target entities might also be participating in other relationships or might make sense as stand-alone entities. Because there is no way in

the logical annotations of the Java Persistence API to declare private ownership of the entities across a relationship, care must be taken when using the REMOVE cascade option.

With that warning out of the way, let's look at a situation where cascading the remove() operation makes sense. If an Employee entity is removed (hopefully an uncommon occurrence), it makes sense to cascade the remove() operation to both the ParkingSpace and Phone entities related to the Employee. These are both cases in which the Employee is the parent of the target entities. Listing 5-19 demonstrates the changes to the Employee entity class that enables this behavior. Note that we have added the REMOVE cascade in addition to the existing PERSIST option. Chances are, if an owning relationship is safe to use REMOVE, then it is also safe to use PERSIST.

Listing 5-19. *Enabling Cascade Remove*

```
@Entity
public class Employee {
    // ...
    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    ParkingSpace parkingSpace;
    @OneToMany(mappedBy="employee",
               cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    Collection<Phone> phones;
    // ...
}
```

Now let's take a step back and look at what it means to cascade the remove() operation. As it processes the Employee instance, the entity manager will navigate across the parkingSpace and phones relationships and invoke remove() on those entity instances as well. Like the remove() operation on a single entity, this is a database operation and has no effect at all on the in-memory links between the object instances. When the Employee instance becomes detached, its phones collection will still contain all of the Phone instances that were there before the remove() operation took place. The Phone instances are detached because they were removed as well, but the link between the two instances remains.

Because the remove() operation can be safely cascaded only from parent to child, it can't help the situation we encountered earlier in the Removing an Entity section. There is no setting that can be applied to a relationship from one entity to another that will cause it to be removed from a parent without also removing the parent in the process. For example, when trying to remove the ParkingSpace entity, we hit an integrity constraint violation from the database unless the parkingSpace field in the Employee entity is set to null. Setting the REMOVE cascade option on the @OneToOne annotation in the ParkingSpace entity would not cause it to be removed from the Employee; rather it would cause the Employee instance itself to become removed. Clearly this is not the behavior we desire. There are no shortcuts to relationship maintenance.

Clearing the Persistence Context

Occasionally it may be necessary to clear a persistence context of its managed entities. This is usually required only for application-managed and extended persistence contexts that are long-lived and have grown too large in size. For example, consider an application-managed entity manager that issues a query returning several hundred entity instances. Once changes are made to a handful of these instances and the transaction is committed, you have left in

memory hundreds of objects that you have no intention of changing any further. If you don't want to close the persistence context, then you need to be able to clear out the managed entities, or else the persistence context will continue to grow over time.

The clear() method of the EntityManager interface can be used to clear the persistence context. In many respects this is semantically equivalent to a transaction rollback. All entity instances managed by the persistence context become detached with their state left exactly as it was when the clear() operation was invoked. If a transaction was started at this point and then committed, nothing would be written out to the database because the persistence context is empty. The clear() operation is all or nothing. You cannot selectively cancel the management of any particular entity instance while the persistence context is still open.

While technically possible, clearing the persistence context when there are uncommitted changes is a dangerous operation. The persistence context is an in-memory structure, and clearing it simply detaches the managed entities. If you are in a transaction and changes have already been written to the database, they will not be rolled back when the persistence context is cleared. The detached entities that result from clearing the persistence context also suffer from all of the negative effects caused by a transaction rollback even though the transaction is still active. For example, identifier generation and versioning should be considered suspect for any entities detached as a result of using the clear() operation.

Synchronization with the Database

Any time the persistence provider generates SQL and writes it out to the database over a JDBC connection, we say that the persistence context has been flushed. All pending changes that require a SQL statement to become part of the transactional changes in the database have been written out and will be made permanent when the database transaction commits. It also means that any subsequent SQL operation that takes place after the flush will incorporate these changes. This is particularly important for SQL queries that are executed in a transaction that is also changing entity data.

If there are managed entities with changes pending, a flush is guaranteed to occur in two situations. The first is when the transaction commits. A flush of any required changes will occur before the database transaction has completed. The only other time a flush is guaranteed to occur is when the entity manager flush() operation is invoked. This method allows developers to manually trigger the same process that the entity manager internally uses to flush the persistence context.

That said, a flush of the persistence context could occur at any time if the persistence provider deems it necessary. An example of this is when a query is about to be executed and it depends on new or changed entities in the persistence context. Some providers will flush the persistence context to ensure that the query incorporates all pending changes. A provider might also flush the persistence context often if it uses an eager-write approach to entity updates. Most persistence providers defer SQL generation to the last possible moment for performance reasons, but this is not guaranteed.

Now that we have covered the circumstances where a flush can occur, let's look at exactly what it means to flush the persistence context. A flush basically consists of three components: new entities that need to be persisted, changed entities that need to be updated, and removed entities that need to be deleted from the database. All of this information is managed by the persistence context. It maintains links to all of the managed entities that will be created or changed as well as the list of entities that need to be removed.

When a flush occurs, the entity manager first iterates over the managed entities and looks for new entities that have been added to relationships with cascade persist enabled. This is logically equivalent to invoking `persist()` again on each managed entity just before the flush occurs. The entity manager also checks to ensure the integrity of all of the relationships. If an entity points to another entity that is not managed or has been removed, then an exception may be thrown.

The rules for determining whether or not the flush fails in the presence of an unmanaged entity can be complicated. Let's walk through an example that demonstrates the most common issues. Figure 5-1 shows an object diagram for an `Employee` instance and some of the objects that it is related to. The `emp` and `ps` entity objects are managed by the persistence context. The `addr` object is a detached entity from a previous transaction, and the `Phone` objects are new objects that have not been part of any persistence operation so far.

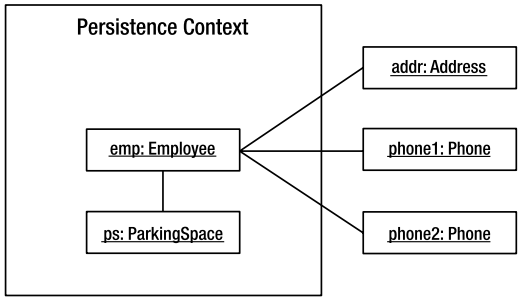

Figure 5-1. *Links to unmanaged entities from a persistence context*

To determine the outcome of flushing the persistence context given the arrangement shown in Figure 5-1, we must first look at the cascade settings of the `Employee` entity. Listing 5-20 shows the relationships as implemented in the `Employee` entity. Only the `phones` relationship has the `PERSIST` cascade option set. The other relationships are all defaulted so they will not cascade.

Listing 5-20. *Relationship Cascade Settings for Employee*

```
@Entity
public class Employee {
    // ...
    @OneToOne
    ParkingSpace parkingSpace;
    @OneToMany(mappedBy="employee", cascade=CascadeType.PERSIST)
    Collection<Phone> phones;
    @ManyToOne
    Address address;
    // ...
}
```


Starting with the `emp` object, let's walk through the flush process as if we are the persistence provider. The `emp` object is managed and has links to four other objects. The first step in the process is to navigate the relationships from this entity as if we are invoking `persist()` on it. The first object we encounter in this process is the `ps` object across the `parkingSpace` relationship. Since `ps` is also managed, we don't have to do anything further.

Next we navigate the `phones` relationship to the two `Phone` objects. These entities are new, and this would normally cause an exception, but since the `PERSIST` cascade option has been set, we perform the equivalent of invoking `persist()` on each `Phone` object. This makes the objects managed, making them part of the persistence context. The `Phone` objects do not have any further relationships to cascade the persist operation, so we are done here as well.

Next we reach the `addr` object across the `address` relationship. Since this object is detached, we would normally throw an exception, but this particular relationship is a special case in the flush algorithm. Any time a detached object that is the target of the one-to-one or many-to-one relationship is encountered where the source entity is the owner, the flush will still proceed because the act of persisting the owning entity does not depend on the target. The owning entity has the foreign key column and needs to store only the primary key value of the target entity.

This completes the flush of the `emp` object. The algorithm then moves to the `ps` object and starts the process again. Since there are no relationships from the `ps` object to any other, the flush process completes. So in this example even though three of the objects pointed to from the `emp` object are not managed, the overall flush completes successfully due to the cascade settings and rules of the flush algorithm.

Ideally during a flush all of the objects pointed to by a managed entity will also be managed entities themselves. If this is not the case, the next thing we need to be aware of is the `PERSIST` cascade setting. If the relationship has this setting, target objects in the relationship will also be persisted, making them managed before the flush completes. If the `PERSIST` cascade option is not set, an `IllegalStateException` exception will be thrown whenever the target of the relationship is not managed, except in the special case related to one-to-one and many-to-one relationships that we described previously.

In light of how the flush operation works, it is always safer to update relationships pointing to entities that will be removed before carrying out the `remove()` operation. A flush may occur at any time, so invoking `remove()` on an entity without clearing any relationships that point to the removed entity could result in an unexpected `IllegalStateException` exception if the provider decides to flush the persistence context before you get around to updating the relationships.

In Chapter 6, we will also discuss techniques to configure the data integrity requirements of queries so that the persistence provider is better able to determine when a flush of the persistence context is really necessary.

Detachment and Merging

Simply put, a detached entity is one that is no longer associated with a persistence context. It was managed at one point, but the persistence context may have ended or the entity may have been transformed in such a way that it has lost its association with the persistence context that used to manage it. The persistence context, if there still is one, is no longer tracking the entity. Any changes made to the entity won't be persisted to the database, but all of the state that was there on the entity when it was detached can still be used by the application. A detached entity cannot be used with any entity manager operation that requires a managed instance.

The opposite of detachment is merging. Merging is the process by which an entity manager integrates detached entity state into a persistence context. Any changes to entity state that were made on the detached entity overwrite the current values in the persistence context. When the transaction commits, those changes will be persisted. Merging allows entities to be changed "offline" and then have those changes incorporated later on.

The following sections will describe detachment and how detached entities can be merged back into a persistence context.

Detachment

There are two views on detachment. On one hand, it is a powerful tool that can be leveraged by applications in order to work with remote applications or to support access to entity data long after a transaction has ended. On the other hand, it can be a frustrating problem when the domain model contains lots of lazy-loading attributes and clients using the detached entities need to access this information.

There are many ways in which an entity can become detached. Each of the following situations will lead to detached entities:

- When the transaction that a transaction-scoped persistence context is associated with commits, all of the entities managed by the persistence context become detached.
- If an application-managed persistence context is closed, all of its managed entities become detached.
- If a stateful session bean with an extended persistence context is removed, all of its managed entities become detached.
- If the `clear()` method of an entity manager is used, it detaches all of the entities in the persistence context managed by that entity manager.
- When transaction rollback occurs, it causes all entities in all persistence contexts associated with the transaction to become detached.
- When an entity is serialized, the serialized form of the entity is detached from its persistence context.

Some of these situations may be intentional and planned for, such as detachment after the end of the transaction or serialization. Others may be unexpected, such as detachment due to rollback.

In Chapter 4, we introduced the `LAZY` fetch type that can be applied to any basic mapping or relationship. This has the effect of hinting to the provider that the loading of a basic or relationship attribute should be deferred until it is accessed for the first time. Although not commonly used on basic mappings, marking relationship mappings to be lazy loaded is an important part of performance tuning.

We need to consider, however, the impact of detachment on lazy loading. Consider the `Employee` entity shown in Listing 5-21. The `address` relationship will eagerly load because many-to-one relationships eagerly load by default. In the case of the `parkingSpace` attribute, which would also normally eagerly load, we have explicitly marked the relationship as being lazy loading. The `phones` relationship, as a one-to-many relationship, will also lazy load by default.

Listing 5-21. *Employee with Lazy-Loading Mappings*

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    private Address address;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    @OneToMany(mappedBy="employee")
    private Collection<Phone> phones;
    // ...
}
```

So long as the `Employee` entity is managed, everything works as we expect. When the entity is retrieved from the database, only the associated `Address` entity will be eagerly loaded. The provider will fetch the necessary entities the first time the `parkingSpace` and `phones` relationships are accessed.

If this entity becomes detached, the outcome of accessing the `parkingSpace` and `phones` relationships is suddenly a more complex issue. If the relationships were accessed while the entity was still managed, the target entities may also be safely accessed while the `Employee` entity is detached. If the relationships were not accessed while the entity was managed, then we have a problem.

The behavior of accessing an unloaded attribute when the entity is detached is not defined. Some vendors may attempt to resolve the relationship, while others may simply throw an exception or leave the attribute uninitialized. If the entity was detached due to serialization, there is virtually no hope of resolving the relationship. The only portable thing to do with attributes that are unloaded is leave them alone.

In the case where entities have no lazy-loading attributes, detachment is not a big deal. All of the entity state that was there in the managed version is still available and ready to use in the detached version of the entity. In the presence of lazy-loading attributes, care must be taken to ensure that all of the information you need to access offline is triggered while the entity is still managed. Later in the chapter we will demonstrate a number of strategies for planning for, and working with, detached entities.

Merging Detached Entities

The `merge()` operation is used to merge the state of a detached entity into a persistence context. The method is straightforward to use, requiring only the detached entity instance as an argument. There are some subtleties to using `merge()` that make it different to use than other entity manager methods. Consider the following example, which shows a session bean method that accepts a detached `Employee` parameter and merges it into the current persistence context:

```
public void updateEmployee(Employee emp) {
    em.merge(emp);
    emp.setLastAccessTime(new Date());
}
```

Assuming that a transaction begins and ends with this method call, any changes made to the `Employee` instance while it was detached will be written to the database. What will not be written, however, is the change to the last access time. The argument to `merge()` does not become managed as a result of the merge. A different managed entity (either a new instance or an existing managed version already in the persistence context) is updated to match the argument, and then this instance is returned from the `merge()` method. Therefore to capture this change, we need to use the return value from `merge()` since it is the managed entity. The following example shows the correct implementation:

```
public void updateEmployee(Employee emp) {
    Employee managedEmp = em.merge(emp);
    managedEmp.setLastAccessTime(new Date());
}
```

Returning a managed instance other than the original entity is a critical part of the merge process. If an entity instance with the same identifier already exists in the persistence context, the provider will overwrite its state with the state of the entity that is being merged, but the managed version that existed already must be returned to the client so that it can be used. If the provider did not update the `Employee` instance in the persistence context, then any references to that instance will become inconsistent with the new state being merged in.

When `merge()` is invoked on a new entity, it behaves similarly to the `persist()` operation. It adds the entity to the persistence context, but instead of adding the original entity instance, it creates a new copy and manages that instance instead. The copy that is created by the `merge()` operation is persisted as if the `persist()` method was invoked on it.

In the presence of relationships, the `merge()` operation will attempt to update the managed entity to point to managed versions of the entities referenced by the detached entity. If the entity has a relationship to an object that has no persistent identity, then the outcome of the merge operation is undefined. Some providers may allow the managed copy to point to the non-persistent object, while others may throw an exception immediately. The `merge()` operation may be optionally cascaded in these cases to prevent an exception from occurring. We will cover cascading of the `merge()` operation later in this section. If an entity being merged points to a removed entity, an `IllegalArgumentException` exception will be thrown.

Lazy-loading relationships are a special case in the merge operation. If a lazy-loading relationship was not triggered on an entity before it became detached, then that relationship will be ignored when the entity is merged. If the relationship was triggered while managed and then set to null while the entity was detached, then the managed version of the entity will likewise have the relationship cleared during the merge.

To illustrate the behavior of `merge()` with relationships, consider the object diagram shown in Figure 5-2. The detached `emp` object has relationships to three other objects. The `addr` and `dept` objects are detached entities from a previous transaction, while the `phone1` entity was recently created and persisted using the `persist()` operation and is now managed as a result. Inside the persistence context there is currently an `Employee` instance with a relationship to another managed `Address`. The existing managed `Employee` instance does not have a relationship to the newly managed `Phone` instance.

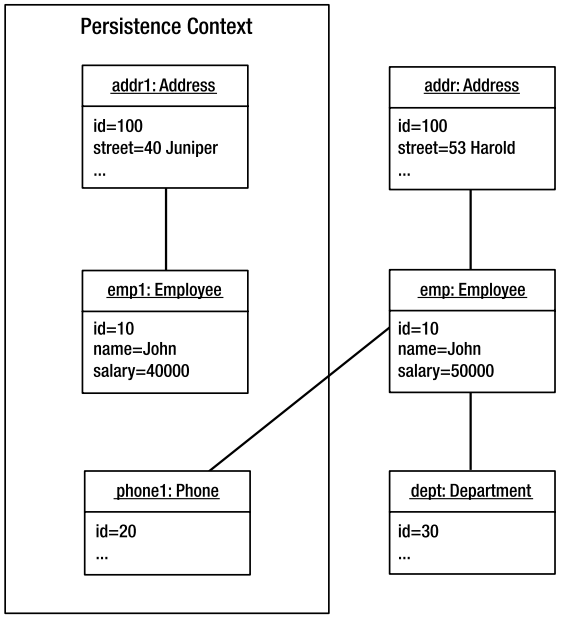

Figure 5-2. *Entity state prior to merge*

Let's consider the effect of invoking `merge()` on the `emp` object. The first thing that happens is that the provider checks the persistence context for a pre-existing entity instance with the same identifier. In this example, the `emp1` object from the persistence context matches the identifier from the `emp` object we are trying to merge. Therefore the basic state of the `emp` object overwrites the state of the `emp1` object in the persistence context, and the `emp1` object will be returned from the `merge()` operation.

The provider next considers the `Phone` and `Department` entities pointed to from `emp`. The `phone1` object is already managed, so the provider can safely update `emp1` to point to this instance. In the case of the `dept` object, the provider checks to see if there is already a persistent `Department` entity with the same identifier. In this case it finds one in the database and loads it into the persistence context. The `emp1` object is then updated to point to this version of the `Department` entity. The detached `dept` object does not become managed again.

Finally the provider checks the `addr` object referenced from `emp`. In this case it finds a pre-existing managed object `addr1` with the same identifier. Since the `emp1` object already points to the `addr1` object, no further changes are made. At this point let's look at the state of the object model after the merge. Figure 5-3 shows these changes.

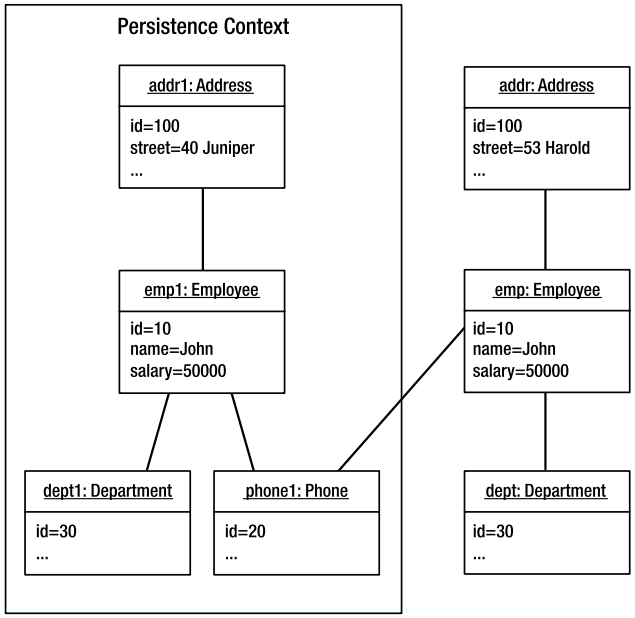

Figure 5-3. *Entity state after merge*

In Figure 5-3 we see that the `emp1` object has been updated to reflect the state changes from `emp`. The `dept1` object is new to the persistence context after being loaded from the database. The `emp1` object now points to both the `phone1` object and the `dept1` object in order to match the relationships of the `emp` object. The `addr1` object has not changed at all. The fact that the `addr1` object has not changed might come as a surprise. After all, the `addr` object had pending changes and it was pointed to by the `emp` object that was merged.

To understand why, we must return to the issue of cascading operations with the entity manager. By default, no operations are cascaded when an entity manager operation is applied to an entity instance. The `merge()` operation is no different in this regard. In order for the merge to be cascaded across relationships from an `Employee`, the `MERGE` cascade setting must be set on the relationship mappings. Otherwise we would have to invoke `merge()` on each related object.

Looking back at our example, the problem with the updated `Address` entity was that the `Employee` entity did not cascade the `merge()` operation to it. This had the unfortunate side effect of effectively discarding the changes we had made to the `Address` entity in favor of the version already in the persistence context. To obtain the behavior that we intended, we must either invoke `merge()` explicitly on the `addr` object or change the relationship mappings of the `Employee` entity to include the `MERGE` cascade option. Listing 5-22 shows the changed `Employee` class.

Listing 5-22. *Employee Entity with Merge Cascade Setting*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @ManyToOne(cascade=CascadeType.MERGE)
    private Address address;
    @ManyToOne
    private Department department;
    @OneToMany(mappedBy="employee", cascade=CascadeType.MERGE)
    private Collection<Phone> phones;
    // ...
}
```

With the `Employee` entity changed in this way, the merge operation will be cascaded to the `Address` and `Phone` entities pointed to by any `Employee` instances. This is equivalent to invoking `merge()` on each instance individually. Note that we did not cascade the merge operation to the `Department` entity. We generally only cascade operations down from parent to child, not upwards from child to parent. Doing so is not harmful but requires more effort from the persistence provider to search out changes. If the `Department` entity changes as well, it is better to cascade the merge from the `Department` to its associated `Employee` instances and then merge only a single `Department` instance instead of multiple `Employee` instances.

Merging detached entities with relationships can be a tricky operation. Ideally we want to merge the root of an object graph and have all related entities get merged in the process. This can work, but only if the `MERGE` cascade setting has been applied to all relationships in the graph. If it hasn't, you must merge each instance that is the target of a non-cascaded relationship one at a time.

Before we leave the topic of merging, we must mention that locking and versioning plays a vital role in ensuring data integrity in these situations. We will explore this topic in Chapter 9.

Working with Detached Entities

Let's begin with a scenario that is very common with modern web applications. A servlet calls out to a session bean in order to execute a query and receives a collection of entities in return. The servlet then places these entities into the request map and forwards the request to a JSP for presentation. This pattern is called Page Controller,[1] a variation of the Front Controller[2] pattern in which there is a single controller for each view instead of one central controller for all views. In the context of the familiar Model-View-Controller (MVC) architecture, the session bean provides the model, the JSP page is the view, and the servlet is our controller.

First consider the session bean that will produce the results that will be rendered by the JSP page. Listing 5-23 shows the bean implementation. In this example we are looking only at the `findAll()` method, which returns all of the `Employee` instances stored in the database.

1. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2002.
2. Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*. Upper Saddle River, N.J.: Prentice Hall PTR, 2003.

Listing 5-23. *The* `EmployeeService` *Session Bean*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    private EntityManager em;

    public List findAll() {
        return em.createQuery("SELECT e FROM Employee e")
                 .getResultList();
    }

    // ...
}
```

Listing 5-24 shows the source code for a simple servlet that invokes the `findAll()` method of the `EmployeeService` session bean to fetch all of the `Employee` entities in the database. It then places the results in the request map and delegates to the "listEmployees.jsp" JSP page to render the result.

Listing 5-24. *The View Employees Servlet*

```
public class EmployeeServlet extends HttpServlet {
    @EJB EmployeeService bean;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        List emps = bean.findAll();
        request.setAttribute("employees", emps);
        getServletContext().getRequestDispatcher("/listEmployees.jsp")
                           .forward(request, response);
    }
}
```

Finally, Listing 5-25 shows the last part of our MVC architecture, the JSP page to render the results. It uses the JavaServer Pages Standard Tag Library (JSTL) to iterate over the collection of `Employee` instances and display the name of each employee as well as the name of the department to which that employee is assigned. The `employees` variable accessed by the `<c:forEach/>` tag is the `List` of `Employee` instances that was placed in the request map by the servlet.

Listing 5-25. *JSP Page to Display Employee Information*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
  <head>
    <title>All Employees</title>
  </head>
```



```
  <body>
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Department</th>
        </tr>
      </thead>
      <tbody>
        <c:forEach items="${employees}" var="emp">
          <tr>
            <td><c:out value="${emp.name}"/></td>
            <td><c:out value="${emp.department.name}"/></td>
          </tr>
        </c:forEach>
      </tbody>
    </table>
  </body>
</html>
```

The `findAll()` method of the `EmployeeService` session bean uses `REQUIRED` container-managed transactions by default. Since the servlet invoking the method has not started a transaction, the container will start a new transaction before `findAll()` is invoked and commit the transaction after it finishes executing. As a result, the results of the query become detached before they are returned to the servlet.

This causes us a problem. In this example, the `department` relationship of the `Employee` class has been configured to use lazy fetching. As we learned previously in the section on Detachment, the only portable thing to do is leave them alone. In this example, however, we don't want to leave them alone. In order to display the department name for the employee, the JSP expression navigates to the `Department` entity from the `Employee` entity. Since this is a lazy-loading relationship, the results are unpredictable. It might work, but then it might not.

This scenario forms the basis of our challenge. In the following sections we will look at a number of strategies to either prepare the entities needed by the JSP page for detachment or avoid detachment altogether.

Planning for Detachment

Knowing that the results of the `findAll()` method will be used to display employee information and that the department name will be required as part of this process, we need to ensure that the `department` relationship of the `Employee` entity has been resolved before the entities become detached. There are several strategies that can be used to resolve lazy loaded associations in preparation for detachment. We will discuss two of them here, focusing on how to structure application code to plan for detachment. A third strategy, for Java Persistence QL queries called fetch joins, will be discussed in Chapter 7.

Triggering Lazy Loading

The first strategy to consider in resolving lazy-loading associations is to simply trigger the lazy loading behavior by accessing the field or relationship. It looks slightly odd in code since the return values of the getter methods are discarded, but nevertheless it has the desired effect. Listing 5-26 shows an alternate implementation of the `findAll()` method of the `EmployeeService` session bean. In this case we iterate over the `Employee` entities, triggering the `department` relationship before returning the original list from the method. Because `findAll()` is executed inside of a transaction, the `getDepartment()` call completes successfully, and the `Department` entity instance is guaranteed to be available when the `Employee` instance is detached.

Listing 5-26. *Triggering a Lazy-Loading Relationship*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    private EntityManager em;

    public List findAll() {
        List<Employee> emps = (List<Employee>)
            em.createQuery("SELECT e FROM Employee e")
              .getResultList();
        for (Employee emp : emps) {
            Department dept = emp.getDepartment();
            if (dept != null) {
                dept.getName();
            }
        }
        return emps;
    }

    // ...
}
```

One thing that might look odd from Listing 5-26 is that we not only invoked `getDepartment()` on the `Employee` instance, but we also invoked `getName()` on the `Department` instance. If you recall from Chapter 4, the entity returned from a lazy-loading relationship may actually be a proxy that further waits until a method is invoked on the proxy before the entity is faulted in. We have to invoke a method on the entity to guarantee that it is actually retrieved from the database. If this were a collection-valued relationship, the `size()` method of the `Collection` would be commonly used to force eager loading.

If lazy-loading basic mappings were used on either the `Employee` or `Department` entities, then those attributes would not be guaranteed to be present after detachment as well. This is another reason why configuring basic mappings to use lazy loading is not recommended. Developers often expect that a relationship is not eagerly loaded but can be caught off guard if a basic state field such as the `name` attribute of the `Employee` instance is missing.

Configuring Eager Loading

When an association is continuously being triggered for detachment scenarios, at some point it is worth revisiting whether or not the association should be lazy loaded in the first place. Carefully switching some relationships to eager loading can avoid a lot of special cases in code that attempt to trigger the lazy loading.

In this example, `Employee` has a many-to-one relationship with `Department`. The default fetch type for a many-to-one relationship is eager loading, but the class was modeled by explicitly using lazy loading. By removing the `LAZY` fetch type from the `department` relationship or by specifying the `EAGER` fetch type explicitly, we ensure that the `Department` instance is always available to the `Employee` instance.

Collection-valued relationships lazy load by default, so the `EAGER` fetch type must be explicitly applied to those mappings if eager loading is desired. Be judicious in configuring collection-valued relationships to be eagerly loaded, however, as it may cause excessive database access in cases where detachment is not a requirement.

Avoiding Detachment

The only complete solution to any detachment scenario is not to detach at all. If your code methodically triggers every lazy-loaded relationship or has marked every association on an entity to be eagerly loaded in anticipation of detachment, then this is probably a sign that an alternative approach is required.

Avoiding detachment really only boils down to two approaches. Either we don't work with entities in our JSP page, or we must keep a persistence context open for the duration of the JSP rendering process so that lazy-loading relationships can be resolved.

Not using entities means copying entity data into a different data structure that does not have the same lazy-loading behavior. One approach would be to use the Transfer Object[3] pattern, but that seems highly redundant given the POJO nature of entities. A better approach, which we will discuss in Chapters 6 and 7, is to use projection queries to retrieve only the entity state that will be displayed on the JSP page instead of retrieving full entity instances.

Keeping a persistence context open requires additional planning but allows the JSP page to work with entity data using the JavaBean properties of the entity class. In practical terms, keeping a persistence context open means that there is either an active transaction for entities fetched from transaction-scoped persistence contexts or that an application-managed or extended persistence context is in use. This obviously isn't an option when entities must be serialized to a separate tier or remote client, but it suits the web application scenario we described earlier. We'll cover each of these strategies here.

Transaction View

The persistence context created by a transaction-scoped entity manager remains open only as long as the transaction in which it was created has not ended. Therefore, in order to use a transaction-scoped entity manager to execute a query and be able to render the query results while resolving lazy-loading relationships, both operations must be part of the same transaction. When a transaction is started in the web tier and includes both session bean invocation and JSP page rendering before it is committed, we call this pattern a Transaction View.

---

3. Ibid.

The benefit of this approach is that any lazy-loading relationships encountered during the rendering of the view will be resolved because the entities are still managed by a persistence context. To implement this pattern in our example scenario, we start a bean-managed transaction before the `findAll()` method is invoked and commit the transaction after the JSP page has rendered the results. Listing 5-27 demonstrates this approach. Note that to conserve space we have omitted the handling of the checked exceptions thrown by the `UserTransaction` operations. The `commit()` method alone throws no less than six checked exceptions.

Listing 5-27. *Combining a Session Bean Method and JSP in a Single Transaction*

```
public class EmployeeServlet extends HttpServlet {
    @Resource UserTransaction tx;
    @EJB EmployeeService bean;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // ...
        try {
            tx.begin();
            List emps = bean.findAll();
            request.setAttribute("employees", emps);
            getServletContext().getRequestDispatcher("/listEmployees.jsp")
                               .forward(request, response);
        } finally {
            tx.commit();
        }
        // ...
    }
}
```

With this solution in place, the lazy-loading relationships of the `Employee` entity do not have to be eagerly resolved before the JSP page renders the results. The only downside to this approach is that the servlet must now manage transactions and recover from transaction failures. A lot of logic also has to be duplicated between all of the servlet controllers that need this behavior.

One way to work around this duplication is to introduce a common superclass for servlets that use the Transaction View pattern that encapsulates the transaction behavior. If, however, you are using the Front Controller pattern and controller actions are implemented using the Command[4] pattern, this may become more difficult to manage, particularly if the page flow is complex and multiple controllers collaborate to build a composite view. Then not only does each controller need to start transactions, but they also need to be aware of any transactions that were started earlier in the rendering sequence.

4. Gamma, Erich, Richard Helms, Ralph Johnson, and John Vlissades. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.

Another possible, though non-portable, solution is to move the transaction logic into a servlet filter. It allows us to intercept the HTTP request before the first controller servlet is accessed and wrap the entire request in a transaction. Such coarse-grained use of transactions is something that needs to be managed carefully, however. If applied to all HTTP requests equally, it may also cause trouble for requests that involve updates to the database. Assuming that these operations are implemented as session beans, the `REQUIRES_NEW` transaction attribute may be required in order to isolate entity updates and handle transaction failure without impacting the overriding global transaction.

Entity Manager per Request

For applications that do not encapsulate their query operations behind session bean façades, an alternative approach to the Transaction View pattern is to create a new application-managed entity manager to execute reporting queries, closing it only after the JSP page has been rendered. Because the entities returned from the query on the application-managed entity manager will remain managed until the entity manager is closed, it offers the same benefits as the Transaction View pattern without requiring an active transaction.

Listing 5-28 revisits our `EmployeeServlet` servlet again, this time creating an application-managed entity manager to execute the query. The results are placed in the map as before, and the entity manager is closed after the JSP page has finished rendering.

Listing 5-28. *Using an Application-Managed Entity Manager for Reporting*

```
public class EmployeeServlet extends HttpServlet {
    @PersistenceUnit(unitName="EmployeeService")
    EntityManagerFactory emf;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        EntityManager em = emf.createEntityManager();
        try {
            List emps = em.createQuery("SELECT e FROM Employee e")
                          .getResultList();
            request.setAttribute("employees", emps);
            getServletContext().getRequestDispatcher("/listEmployees.jsp")
                               .forward(request, response);
        } finally {
            em.close();
        }
    }
}
```

Unfortunately, we now have query logic embedded in our servlet implementation. The query is also no longer reusable the way it was when it was part of a stateless session bean. There are a couple of other options we can explore as a solution to this problem. Instead of executing the query directly, we could create a POJO service class that uses the application-managed entity manager created by the servlet to execute queries. This is similar to the first example we created in Chapter 2. We gain the benefit of encapsulating the query behavior inside business methods while being decoupled from a particular style of entity manager.

Alternatively we can place our query methods on a stateful session bean that uses an extended entity manager. When a stateful session bean uses an extended entity manager, its persistence context lasts for the lifetime of the session bean, which ends only when the user invokes a remove method on the bean. If a query is executed against the extended persistence context of a stateful session bean, the results of that query can continue to resolve lazy-loading relationships so long as the bean is still available.

Let's explore this option and see how it would look instead of the application-managed entity manager we showed in Listing 5-28. Listing 5-29 introduces a stateful session bean equivalent to the `EmployeeService` stateless session bean that we have been using so far. In addition to using the extended entity manager, we have also set the default transaction type to be `NOT_SUPPORTED`. There is no need for transactions because the results of the query will never be modified, only displayed.

Listing 5-29. *Stateful Session Bean with Query Methods*

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class EmployeeQueryBean implements EmployeeQuery {
    @PersistenceContext(type=PersistenceContextType.EXTENDED,
                        unitName="EmployeeService")
    EntityManager em;

    public List findAll() {
        return em.createQuery("SELECT e FROM Employee e")
                 .getResultList();
    }

    // ...

    @Remove
    public void finished() {
    }
}
```

Using this bean is very similar to using the application-managed entity manager. We create an instance of the bean, execute the query, and then remove the bean when the JSP page has finished rendering. Listing 5-30 shows this approach.

Listing 5-30. *Using an Extended Entity Manager for Reporting*

```
@EJB(name="queryBean", beanInterface=EmployeeQuery.class)
public class EmployeeServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        EmployeeQuery bean = createQueryBean();
        try {
            List emps = bean.findAll();
            request.setAttribute("employees", emps);
            getServletContext().getRequestDispatcher("/listEmployees.jsp")
                               .forward(request, response);
        } finally {
            bean.finished();
        }
    }

    private EmployeeQuery createQueryBean() throws ServletException {
        // look up queryBean
        // ...
    }
}
```

At first glance this might seem like an overengineered solution. We gain the benefit of decoupling queries from the servlet, but we have introduced a new session bean just to accomplish this goal. Furthermore, we are using stateful session beans with very short life-times. Doesn't that go against the accepted practice of how to use a stateful session bean?

To a certain extent this is true, but the extended persistence context invites us to experiment with new approaches. In practice, stateful session beans do not add a significant amount of overhead to an operation, even when used for short durations. As we will see later in the section Edit Session, moving the stateful session bean to the HTTP session instead of limiting it to a single request also opens up new possibilities for web application design.

Merge Strategies

Creating or updating information is a regular part of most enterprise applications. Users typically interact with an application via the web, using forms to create or change data as required. The most common strategy to handle these changes in a Java EE application that uses the Java Persistence API is to place the results of the changes into detached entity instances and merge the pending changes into a persistence context so that they can be written to the database.

Let's revisit our simple web application scenario again. This time, instead of simply viewing `Employee` information, the user is able to select an `Employee` and update basic information about that employee. The entities are queried for presentation in a form in one request and then updated in a second request when the user submits the form with changes entered.

Using a Session Façade, this operation is straightforward. The changed entity is updated and handed off to a stateless session bean to be merged. The only complexity involved is making sure that relationships properly merge by identifying cases where the `MERGE` cascade setting is required.

Similar to the question of whether we can avoid detaching entities to compensate for lazy loading concerns, the long-lived nature of application-managed and extended persistence contexts suggests that there may also be a way to apply a similar technique to this situation. Instead of querying entities in one HTTP request and throwing the entity instances away after the view has been rendered, we want to keep these entities around in a managed state so that they can be updated in a subsequent HTTP request and persisted merely by starting and committing a new transaction.

In the following sections we will revisit the traditional Session Façade approach to merging and then look at new techniques possible with the extended entity manager that will keep entities managed for the life of a user's editing session.

Session Façade

To use a Session Façade to capture changes to entities, we provide a business method that will merge changes made to a detached entity instance. In our example scenario, this means accepting an `Employee` instance and merging it into a transaction-scoped persistence context. Listing 5-31 shows an implementation of this technique in our `EmployeeService` session bean.

Listing 5-31. *Business Method to Update Employee Information*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    private EntityManager em;

    public void updateEmployee(Employee emp) {
        if (em.find(Employee.class, emp.getId()) == null) {
            throw new IllegalArgumentException("Unknown employee id: " + ➥
emp.getId());
        }
        em.merge(emp);
    }

    // ...
}
```

The `updateEmployee()` method in Listing 5-31 is straightforward. Given the detached `Employee` instance, it first attempts to check whether a matching identifier already exists. If no matching `Employee` is found, then an exception is thrown since we don't want to allow new `Employee` records to be created. Then we use the `merge()` operation to copy the changes into the persistence context, which are then saved when the transaction commits.

Using the façade from a servlet is a two-step approach. During the initial HTTP request to begin an editing session, the `Employee` instance is queried (typically using a separate method on the same façade) and used to create a web form on which the user can make their desired

changes. The detached instance is then stored in the HTTP session so it can be updated when the user submits the form from their browser. We need to keep the detached instance around in order to preserve any relationships or other state that will remain unchanged by the edit. Creating a new `Employee` instance and supplying only partial values could have many negative side effects when the instance is merged.

Listing 5-32 shows an `EmployeeUpdateServlet` servlet that collects the id, name, and salary information from the request parameters and invokes the session bean method to perform the update. The previously detached `Employee` instance is retrieved from the HTTP session, and then the changes indicated by the request parameters are set into it. We have omitted validation of the request parameters to conserve space, but ideally this should happen before the business method on the session bean is invoked.

Listing 5-32. *Using a Session Bean to Perform Entity Updates*

```
public class EmployeeUpdateServlet extends HttpServlet {
    @EJB EmployeeService bean;

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        int id = Integer.parseInt(request.getParameter("id"));
        String name = request.getParameter("name");
        long salary = Long.parseLong(request.getParameter("salary"));
        HttpSession session = request.getSession();
        Employee emp = (Employee) session.getAttribute("employee.edit");
        emp.setId(id);
        emp.setName(name);
        emp.setSalary(salary);
        bean.updateEmployee(emp);
        // ...
    }
}
```

If the amount of information being updated is very small, we can avoid the detached object and `merge()` operation entirely by locating the managed version and manually copying the changes into it. Consider the following example:

```
public void updateEmployee(int id, String name, long salary) {
    Employee emp = em.find(Employee.class, id);
    if (emp == null) {
        throw new IllegalArgumentException("Unknown employee id: " + id);
    }
    emp.setEmpName(name);
    emp.setSalary(salary);
}
```

The beauty of this approach is its simplicity, but that is also its primary limitation. Typical web applications today offer the ability to update large amounts of information in a single operation. To accommodate these situations with this pattern, there would either have to be business methods taking large numbers of parameters or many business methods that would have to

be invoked in sequence to completely update all of the necessary information. And, of course, once you have more than one method involved, then it becomes important to maintain a transaction across all of the update methods so that the changes are committed as a single unit.

As a result, despite the availability of this approach, the web tier still commonly collects changes into detached entities or transfer objects and passes the changed state back to session beans to be merged and written to the database.

Edit Session

With the introduction of the extended entity manager, we can take a different approach to building web applications that update entities. As we have discussed in this chapter, entities associated with an extended entity manager remain managed so long as the stateful session bean holding the extended entity manager is not removed. By placing a stateful session bean in a central location such as the HTTP session, we can operate on entities managed by the extended entity manager without having to merge in order to persist changes. We will refer to this as the Edit Session pattern to reflect the fact that the primary goal of this pattern is to encapsulate editing use cases using stateful session beans.

Listing 5-33 introduces a stateful session bean that represents an employee editing session. Unlike the `EmployeeService` session bean that contains a number of reusable business methods, this style of stateful session bean is targeted to a single application use case. In addition to using the extended entity manager, we have also set the default transaction type to be `NOT_SUPPORTED` with the exception of the `save()` method. There is no need for transactions for methods that simply access the `Employee` instance. It is only when we want to persist the changes that we need a transaction.

Listing 5-33. *Stateful Session Bean to Manage an Employee Editing Session*

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class EmployeeEditBean implements EmployeeEdit {
    @PersistenceContext(type=PersistenceContextType.EXTENDED,
                        unitName="EmployeeService")
    EntityManager em;
    Employee emp;

    public void begin(int id) {
        emp = em.find(Employee.class, id);
        if (emp == null) {
            throw new IllegalArgumentException("Unknown employee id: " + id);
        }
    }

    public Employee getEmployee() {
        return emp;
    }
```



```
    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void save() {}

    @Remove
    public void cancel() {}
}
```

Let's start putting the operations of the `EmployeeEdit` bean in context. When the HTTP request arrives and starts the editing session, we will create a new `EmployeeEdit` stateful session bean and invoke `begin()` using the id of the `Employee` instance that will be edited. The session bean then loads the `Employee` instance and caches it on the bean. The bean is then bound to the HTTP session so that it can be accessed again in a subsequent request once the user has changed the `Employee` information. Listing 5-34 shows the `EmployeeEditServlet` servlet that handles the HTTP request to begin a new editing session.

Listing 5-34. *Beginning an Employee Editing Session*

```
@EJB(name="EmployeeEdit", beanInterface=EmployeeEdit.class)
public class EmployeeEditServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        int id = Integer.parseInt(request.getParameter("id"));
        EmployeeEdit bean = getBean();
        bean.begin(id);
        HttpSession session = request.getSession();
        session.setAttribute("employee.edit", bean);
        request.setAttribute("employee", bean.getEmployee());
        getServletContext().getRequestDispatcher("/editEmployee.jsp")
                           .forward(request, response);
    }

    public EmployeeEdit getBean() throws ServletException {
        // lookup EmployeeEdit bean
        // ...
    }
}
```

Now let's look at the other half of the editing session, where we wish to commit the changes. When the user submits the form that contains the necessary `Employee` changes, the `EmployeeUpdateServlet` is invoked. It begins by retrieving the `EmployeeEdit` bean from the HTTP session. The request parameters with the changed values are then copied into the `Employee` instance obtained from called `getEmployee()` on the `EmployeeEdit` bean. If everything is in order, the `save()` method is invoked to write the changes to the database. Listing 5-35 shows the `EmployeeUpdateServlet` implementation. Note that we need to remove the bean from the HTTP session once the editing session has completed.

Listing 5-35. *Completing an Employee Editing Session*

```
public class EmployeeUpdateServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String name = request.getParameter("name");
        long salary = Long.parseLong(request.getParameter("salary"));
        HttpSession session = request.getSession();
        EmployeeEdit bean = (EmployeeEdit) session.getAttribute("employee.edit");
        session.removeAttribute("employee.edit");
        Employee emp = bean.getEmployee();
        emp.setName(name);
        emp.setSalary(salary);
        bean.save();
        // ...
    }
}
```

The pattern for using stateful session beans and extended entity managers in the web tier is as follows:

1. For each application use case that modifies entity data, we create a stateful session bean with an extended persistence context. This bean will hold onto all entity instances necessary to make the desired changes.
2. The HTTP request that initiates the editing use case creates an instance of the stateful session bean and binds it to the HTTP session. The entities are retrieved at this point and used to populate the web form for editing.
3. The HTTP request that completes the editing use case obtains the previously bound stateful session bean instance and writes the changed data from the web form into the entities stored on the bean. A method is then invoked on the bean to commit the changes to the database.

In our simple editing scenario this may seem somewhat excessive, but the beauty of this technique is that it can easily scale to accommodate editing sessions of any complexity. `Department`, `Project`, and other information may all be edited in one or even multiple sessions with the results accumulated on the stateful session bean until the application is ready to persist the results.

Another major benefit of this approach is that web application frameworks like JavaServer Faces can directly access the bean bound in the HTTP session from within JSP pages. The entity can be accessed both to display the form for editing and as the target of the form when the user submits the results. In this scenario the developer only has to ensure that the necessary save and cancel methods are invoked at the correct point in the application page flow.

There are a couple of other points that we need to mention about this approach. Once bound to the HTTP session, the session bean will remain there until it is explicitly removed or until the HTTP session expires. It is therefore important to ensure that the bean is removed once the editing session is complete, regardless of whether the changes will be saved or abandoned. The `HttpSessionBindingListener` callback interface can be used by applications to track when the HTTP session is destroyed and clean up corresponding session beans appropriately.

The HTTP session is not thread-safe, and neither are stateful session bean references. In some circumstances it may be possible for multiple HTTP requests from the same user to access the HTTP session concurrently. This is mostly an issue when requests take a long time to process and an impatient user refreshes the page or abandons their editing session for another part of the web application. In these circumstances the web application will either have to deal with possible exceptions occurring if the stateful session bean is accessed by more than one thread, or proxy the stateful session bean with a synchronized wrapper.

Summary

In this chapter we have presented a thorough treatment of the entity manager and its interactions with entities, persistence contexts, and transactions. As we have seen, the entity manager can be used in many different ways to accommodate a wide variety of application requirements.

We began by reintroducing the core terminology of the Java Persistence API and explored the persistence context. We then covered the three different types of entity manager: transaction-scoped, extended, and application-managed. We looked at how to acquire and use each type and the types of problems they are designed to solve.

In the Transaction Management section we looked at each of the entity manager types and how they relate to container-managed JTA transactions and the resource-local transactions of the JDBC driver. Transactions play an important role in all aspects of enterprise application development with the Java Persistence API.

Next we revisited the basic operations of the entity manager, this time armed with the full understanding of the different entity manager types and transaction-management strategies. We introduced the notion of cascading and looked at the impact of relationships on persistence.

In our discussion of detachment, we introduced the problem and looked at it both from the perspective of mobile entities to remote tiers and the challenge of merging offline entity changes back into a persistence context. We presented several strategies to minimize the impact of detachment and merging on application design by adopting design patterns specific to the Java Persistence API.

In the next chapter we will turn our attention to the query facilities of the Java Persistence API, showing how to create, execute, and work with the results of query operations.

CHAPTER 6

Using Queries

For most enterprise applications, getting data out of the database is at least as important as the ability to put new data in. From searching, to sorting, to analytics and business intelligence, efficiently moving data from the database to the application and presenting it to the user is a regular part of enterprise development. Doing so requires the ability to issue bulk queries against the database and interpret the results for the application. Although high-level languages and expression frameworks have attempted to insulate developers from the task of dealing with database queries at a low level, it's probably fair to say that most enterprise developers have worked with at least one SQL dialect at some point in their career.

Object-relational mapping adds another level of complexity to this task. Most of the time, the developer will want the results converted to entities so that the query results may be used directly by application logic. Similarly, if the domain model has been abstracted from the physical model via object-relational mapping, then it makes sense to also abstract queries away from SQL, which is not only tied to the physical model but also difficult to port between vendors. Fortunately, as we will see, the Java Persistence API can handle a diverse set of query requirements.

The Java Persistence API supports two query languages for retrieving entities and other persistent data from the database. The primary language is Java Persistence QL (JPQL), a database-independent query language that operates on the logical entity model as opposed to the physical data model. Queries may also be expressed in SQL in order to take advantage of the underlying database. We will discuss SQL queries with the Java Persistence API later in Chapter 9.

We will begin our discussion of queries with an introduction to Java Persistence QL, followed by an exploration of the query facilities provided by the `EntityManager` and `Query` interfaces.

Java Persistence QL

Before discussing JPQL, we must first look to its roots in the EJB specification. The EJB Query Language (EJB QL) was first introduced in the EJB 2.0 specification to allow developers to write portable finder and select methods for container-managed entity beans. Based on a small subset of SQL, it introduced a way to navigate across entity relationships both to select data and to filter the results. Unfortunately it placed strict limitations on the structure of the query, limiting results to either a single entity or a persistent field from an entity. Inner joins between entities were possible but used an odd notation. The initial release didn't even support sorting.

The EJB 2.1 specification tweaked EJB QL a little bit, adding support for sorting, and introduced basic aggregate functions; but again the limitation of a single result type hampered the use of aggregates. You could filter the data, but there was no equivalent to SQL GROUP BY and HAVING expressions.

Java Persistence Query Language significantly extends EJB QL, eliminating many weaknesses of the previous versions while preserving backwards compatibility. The following features are available above and beyond EJB QL:

- Single and multiple value result types
- Aggregate functions, with sorting and grouping clauses
- A more natural join syntax, including support for both inner and outer joins
- Conditional expressions involving subqueries
- Update and delete queries for bulk data changes
- Result projection into non-persistent classes

The next few sections provide a quick introduction to Java Persistence QL intended for readers familiar with SQL or previous versions of EJB QL. A complete tutorial and reference for Java Persistence QL can be found in Chapter 7.

Getting Started

The simplest JPQL query selects all of the instances of a single entity type. Consider the following query:

```
SELECT e
FROM Employee e
```

If this looks similar to SQL, it should. JPQL uses SQL syntax where possible in order to give developers experienced with SQL a head start in writing queries. The key difference between SQL and JPQL for this query is that instead of selecting from a table, an entity from the application domain model has been specified instead. The SELECT clause of the query is also slightly different, listing only the `Employee` alias `e`. This indicates that the result type of the query is the `Employee` entity, so executing this statement will result in a list of zero or more `Employee` instances.

Starting with an alias, we can navigate across entity relationships using the dot (.) operator. For example, if we want just the names of the employees, the following query will suffice:

```
SELECT e.name
FROM Employee e
```

Each part of the expression corresponds to a persistent field of the entity or an association leading to another entity or collection of entities. Since the `Employee` entity has a persistent field named `name` of type `String`, this query will result in a list of zero or more `String` objects.

We can also select an entity we didn't even list in the FROM clause. Consider the following example:

```
SELECT e.department
FROM Employee e
```

An employee has a many-to-one relationship with his or her department named `department`, so therefore the result type of the query is the `Department` entity.

Filtering Results

Just like SQL, JPQL supports the WHERE clause to set conditions on the data being returned. The majority of operators commonly available in SQL are available in JPQL, including basic comparison operators; IN, LIKE, and BETWEEN expressions; numerous function expressions (such as SUBSTRING and LENGTH); and subqueries. The key difference for JPQL is that entity expressions and not column references are used. Listing 6-1 demonstrates filtering using entity expressions in the WHERE clause.

Listing 6-1. *Filtering Criteria Using Entity Expressions*

```
SELECT e
FROM Employee e
WHERE e.department.name = 'NA42' AND
      e.address.state IN ('NY','CA')
```

Projecting Results

For applications that need to produce reports, a common scenario is selecting large numbers of entity instances, but only using a portion of that data. Depending on how an entity is mapped to the database, this can be an expensive operation if much of the entity data is discarded. It would be useful to return only a subset of the properties from an entity. The following query demonstrates selecting only the name and salary of each `Employee` instance:

```
SELECT e.name, e.salary
FROM Employee e
```

Joins Between Entities

The result type of a select query cannot be a collection; it must be a single valued object such as an entity instance or persistent field type. This means that expressions such as `e.phones` are illegal in the SELECT clause because they would result in `Collection` instances. Therefore, just as with SQL and tables, if we want to navigate along a collection association and return elements of that collection, then we must join the two entities together. Listing 6-2 demonstrates a join between `Employee` and `Phone` entities in order to retrieve all of the cell phone numbers for a specific department.

Listing 6-2. *Joining Two Entities Together*

```
SELECT p.number
FROM Employee e, Phone p
WHERE e = p.employee AND
      e.department.name = 'NA42' AND
      p.type = 'Cell'
```

In JPQL, joins may also be expressed in the FROM clause using the JOIN operator. The advantage of this operator is that the join can be expressed in terms of the association itself, and the query engine will automatically supply the necessary join criteria when it generates the SQL. Listing 6-3 shows the same query rewritten to use the JOIN operator. Just as in the previous query, the alias p is of type Phone, only this time it refers to each of the phones in the e.phones collection.

Listing 6-3. *Joining Two Entities Together Using the JOIN Operator*

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND
      p.type = 'Cell'
```

JPQL supports multiple join types, including inner and outer joins, as well as a technique called fetch joins for eagerly loading data associated to the result type of a query but not directly returned. See the Joins section in Chapter 7 for more information.

Aggregate Queries

The syntax for aggregate queries in JPQL is very similar to that of SQL. There are five supported aggregate functions (AVG, COUNT, MIN, MAX, and SUM), and results may be grouped in the GROUP BY clause and filtered using the HAVING clause. Once again, the difference is the use of entity expressions when specifying the data to be aggregated. Listing 6-4 demonstrates an aggregate query with JPQL.

Listing 6-4. *Query Returning Statistics for Departments with Five or More Employees*

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

Query Parameters

JPQL supports two types of parameter binding syntax. The first is positional binding, where parameters are indicated in the query string by a question mark followed by the parameter number. When the query is executed, the developer specifies the parameter number that should be replaced. Listing 6-5 demonstrates positional parameter syntax.

Listing 6-5. *Positional Parameter Notation*

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND
      e.salary > ?2
```

Named parameters may also be used and are indicated in the query string by a colon followed by the parameter name. When the query is executed, the developer specifies the parameter name that should be replaced. Listing 6-6 demonstrates named parameter syntax.

Listing 6-6. *Named Parameter Notation*

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND
      e.salary > :base
```

Defining Queries

The Java Persistence API provides the Query interface to configure and execute queries. An implementation of the Query interface for a given query is obtained through one of the factory methods in the EntityManager interface. The choice of factory method depends on the type of query (JPQL or SQL) and whether or not the query has been predefined. For now, we will restrict our discussion to JPQL queries. SQL query definition is discussed in Chapter 9.

There are two approaches to defining a query. A query may either be dynamically specified at runtime or configured in persistence unit metadata (annotation or XML) and referenced by name. Dynamic queries are nothing more than strings, and therefore may be defined on the fly as the need arises. Named queries, on the other hand, are static and unchangeable but are more efficient to execute as the persistence provider can translate the JPQL string to SQL once when the application starts as opposed to every time the query is executed.

The following sections compare the two approaches and discuss when one should be used instead of the other.

Dynamic Query Definition

A query may be defined dynamically by passing the JPQL query string to the createQuery() method of the EntityManager interface. There are no restrictions on the query definition. All JPQL query types are supported, as well as the use of parameters. The ability to build up a string at runtime and use it for a query definition is useful, particularly for applications where the user may specify complex criteria and the exact shape of the query cannot be known ahead of time.

An issue to consider with dynamic queries, however, is the cost of translating the JPQL string to SQL for execution. A typical query engine will have to parse the JPQL string into a syntax tree, get the object-relational mapping metadata for each entity in each expression, and then generate the equivalent SQL. For applications that issue many queries, the performance cost of dynamic query processing can become an issue.

Many query engines will cache the translated SQL for later use, but this can easily be defeated if the application does not use parameter binding and concatenates parameter values directly into query strings. This has the effect of generating a new and unique query every time a query that requires parameters is constructed.

Consider the session bean method shown in Listing 6-7 that searches for salary information given the name of a department and the name of an employee. There are two problems with this example, one performance-related and one security-related. Because the names are concatenated into the string instead of using parameter binding, it is effectively creating a new and unique query each time. One hundred calls to this method could potentially generate one hundred different query strings. This not only requires excessive parsing of JPQL but also almost certainly makes it difficult for the persistence provider if it attempts to build a cache of converted queries.

Listing 6-7. *Defining a Query Dynamically*

```
@Stateless
public class QueryServiceBean implements QueryService {
    @PersistenceContext(unitName="DynamicQueries")
    EntityManager em;

    public long queryEmpSalary(String deptName, String empName) {
        String query = "SELECT e.salary " +
                       "FROM Employee e " +
                       "WHERE e.department.name = '" + deptName + "' AND " +
                       "      e.name = '" + empName + "'";
        return (Long) em.createQuery(query).getSingleResult();
    }
}
```

The second problem with this example is that a malicious user could pass in a value that alters the query to his advantage. Consider a case where the department argument was fixed by the application but the user was able to specify the employee name (the manager of the department is querying the salaries of his or her employees, for example). If the name argument were actually the text "`'_UNKNOWN' OR e.name = 'Roberts'`". The actual query parsed by the query engine would be as follows:

```
SELECT e.salary
FROM Employee e
WHERE e.department.name = 'NA65' AND
      e.name = '_UNKNOWN' OR
      e.name = 'Roberts'
```

By introducing the OR condition, the user has effectively given himself access to the salary value for any employee in the company, since the original AND condition has a higher precedence than OR and the fake employee name is unlikely to belong to a real employee in that department.

Parameter binding defeats this type of security threat, because the original query string is never altered. The parameters are marshaled using the JDBC API and handled directly by the database. The text of a parameter string is effectively quoted by the database, so the malicious attack would actually end up producing the following query:

```
SELECT e.salary
FROM Employee e
WHERE e.department.name = 'NA65' AND
      e.name = '_UNKNOWN'' OR e.name = ''Roberts'
```

The single quotes used in the query parameter here have been escaped by prefixing them with an additional single quote. This removes any special meaning from them, and the entire sequence is treated as a single string value.

This type of problem may sound unlikely, but in practice many web applications take text submitted over a GET or POST request and blindly construct queries of this sort without considering side effects. One or two attempts that result in a parser stack trace displayed to the web page and the attacker will learn everything he needs to know about how to alter the query to his advantage.

Listing 6-8 shows the same method as in Listing 6-7 except that it uses named parameters instead. This not only reduces the number of unique queries parsed by the query engine, but it also eliminates the chance of the query being altered.

Listing 6-8. *Using Parameters with a Dynamic Query*

```
@Stateless
public class QueryServiceBean implements QueryService {
    private static final String QUERY =
        "SELECT e.salary " +
        "FROM Employee e " +
        "WHERE e.department.name = :deptName AND " +
        "      e.name = :empName ";

    @PersistenceContext(unitName="DynamicQueries")
    EntityManager em;

    public long queryEmpSalary(String deptName, String empName) {
        return (Long) em.createQuery(QUERY)
                        .setParameter("deptName", deptName)
                        .setParameter("empName", empName)
                        .getSingleResult();
    }
}
```

We recommend statically defined named queries in general, particularly for queries that are executed frequently. If dynamic queries are a necessity, take care to use parameter binding instead of concatenating parameter values into query strings in order to minimize the number of distinct query strings parsed by the query engine.

Named Query Definition

Named queries are a powerful tool for organizing query definitions and improving application performance. A named query is defined using the `@NamedQuery` annotation, which may be placed on the class definition for any entity. The annotation defines the name of the query, as well as the query text. Listing 6-9 shows how the query string used in Listing 6-8 would be declared as a named query.

Listing 6-9. *Defining a Named Query*

```
@NamedQuery(name="findSalaryForNameAndDepartment",
            query="SELECT e.salary " +
                  "FROM Employee e " +
                  "WHERE e.department.name = :deptName AND " +
                  "      e.name = :empName")
```

Note the use of string concatenation in the annotation definition. Formatting your queries visually aids in the readability of the query definition. Named queries are typically placed on the entity class that most directly corresponds to the query result, so the `Employee` entity would be a good location for this named query.

The name of the query is scoped to the persistence unit and must be unique within that scope. This is an important restriction to keep in mind, as commonly used query names such as "`findAll`" will have to be qualified for each entity. A common practice is to prefix the query name with the entity name. For example, the "`findAll`" query for the `Employee` entity would be named "`Employee.findAll`". It is undefined what should happen if two queries in the same persistence unit have the same name, but it is likely that either deployment of the application will fail or one will overwrite the other, leading to unpredictable results at runtime. Entity-scoped query names are planned for the next release of the Java Persistence API and will remove the need for this kind of prefixing.

If more than one named query is to be defined for a class, they must be placed inside of a `@NamedQueries` annotation, which accepts an array of one or more `@NamedQuery` annotations. Listing 6-10 shows the definition of several queries related to the `Employee` entity. Queries may also be defined (or redefined) using XML. This technique is discussed in Chapter 10.

Listing 6-10. *Multiple Named Queries for an Entity*

```
@NamedQueries({
    @NamedQuery(name="Employee.findAll",
                query="SELECT e FROM Employee e"),
    @NamedQuery(name="Employee.findByPrimaryKey",
                query="SELECT e FROM Employee e WHERE e.id = :id"),
    @NamedQuery(name="Employee.findByName",
                query="SELECT e FROM Employee e WHERE e.name = :name")
})
```

Because the query string is defined in the annotation, it cannot be altered by the application at runtime. This contributes to the performance of the application and helps to prevent the kind of security issues we discussed in the previous section. Due to the static nature of the query string, any additional criteria that are required for the query must be specified using

query parameters. Listing 6-11 demonstrates using the `createNamedQuery()` call on the `EntityManager` interface to create and execute a named query that requires a query parameter.

Listing 6-11. *Executing a Named Query*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public Employee findEmployeeByName(String name) {
         return (Employee) em.createNamedQuery("Employee.findByName")
                             .setParameter("name", name)
                             .getSingleResult();
    }

    // ...
}
```

Named parameters are the most practical choice for named queries as it effectively self-documents the application code that invokes the queries. Positional parameters are still supported, however, and may be used instead.

Parameter Types

As mentioned earlier, the Java Persistence API supports both named and positional parameters for JPQL queries. The query factory methods of the entity manager return an implementation of the `Query` interface. Parameter values are then set on this object using the `setParameter()` methods of the `Query` interface.

There are three variations of this method for both named parameters and positional parameters. The first argument is always the parameter name or number. The second argument is the object to be bound to the named parameter. `Date` and `Calendar` parameters also require a third argument that specifies whether the type passed to JDBC is a `java.sql.Date`, `java.sql.Time`, or `java.sql.TimeStamp` value.

Consider the following named query definition, which requires two named parameters:

```
@NamedQuery(name="findEmployeesAboveSal",
            query="SELECT e " +
                  "FROM Employee e " +
                  "WHERE e.department = :dept AND " +
                  "      e.salary > :sal")
```

This query highlights one of the nice features of JPQL in that entity types may be used as parameters. When the query is translated to SQL, the necessary primary key columns will be inserted into the conditional expression and paired with the primary key values from the parameter. It is not necessary to know how the primary key is mapped in order to write the query. Binding the parameters for this query is a simple case of passing in the required `Department` entity instance as well as a `long` representing the minimum salary value for

the query. Listing 6-12 demonstrates how to bind the entity and primitive parameters required by this query.

Listing 6-12. *Binding Named Parameters*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List findEmployeesAboveSal(Department dept, long minSal) {
         return em.createNamedQuery("findEmployeesAboveSal")
                  .setParameter("dept", dept)
                  .setParameter("sal", minSal)
                  .getResultList();
    }

    // ...
}
```

`Date` and `Calendar` parameters are a special case because they represent both dates and times. In Chapter 4, we discussed mapping temporal types by using the `@Temporal` annotation and the `TemporalType` enumeration. This enumeration indicates whether the persistent field is a date, time, or timestamp. When a query uses a `Date` or `Calendar` parameter, it must select the appropriate temporal type for the parameter. Listing 6-13 demonstrates binding parameters where the value should be treated as a date.

Listing 6-13. *Binding Date Parameters*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List findEmployeesHiredDuringPeriod(Date start, Date end) {
         return em.createQuery("SELECT e " +
                               "FROM Employee e " +
                               "WHERE e.startDate BETWEEN ?1 AND ?2")
                  .setParameter(1, start, TemporalType.DATE)
                  .setParameter(2, end, TemporalType.DATE)
                  .getResultList();
    }

    // ...
}
```

One thing to keep in mind with query parameters is that the same parameter can be used multiple times in the query string yet only needs to be bound once using the `setParameter()` method. For example, consider the following named query definition, where the "dept" parameter is used twice in the WHERE clause:

```
@NamedQuery(name="findHighestPaidByDepartment",
            query="SELECT e " +
                  "FROM Employee e " +
                  "WHERE e.department = :dept AND " +
                  "      e.salary = (SELECT MAX(e.salary) " +
                  "                  FROM Employee e " +
                  "                  WHERE e.department = :dept)")
```

To execute this query, the "dept" parameter only needs to be set once with `setParameter()` as in the following example:

```
public Employee findHighestPaidByDepartment(Department dept) {
    return (Employee) em.createNamedQuery("findHighestPaidByDepartment")
                        .setParameter("dept", dept)
                        .getSingleResult();
}
```

Executing Queries

The `Query` interface provides three different ways to execute a query, depending on whether or not the query returns results and how many results should be expected. For queries that return values, the developer may choose to call either `getSingleResult()` if the query is expected to return a single result or `getResultList()` if more than one result may be returned. The `executeUpdate()` method of the query interface is used to invoke bulk update and delete queries. We will discuss this method later in the section Bulk Update and Delete.

The simplest form of query execution is via the `getResultList()` method. It returns a collection containing the query results. If the query did not return any data, then the collection is empty. The return type is specified as a `List` instead of `Collection` in order to support queries that specify a sort order. If the query uses the ORDER BY clause to specify a sort order, then the results will be put into the result list in the same order. Listing 6-14 demonstrates how a query might be used to generate a menu for a command line application that displays the name of each employee working on a project as well as the name of the department that the employee is assigned to. The results are sorted by the name of the employee. Queries are unordered by default.

Listing 6-14. *Iterating over Sorted Results*

```
public void displayProjectEmployees(String projectName) {
    List result = em.createQuery("SELECT e " +
                                 "FROM Project p JOIN p.employees e " +
                                 "WHERE p.name = ?1 " +
                                 "ORDER BY e.name")
                    .setParameter(1, projectName)
                    .getResultList();
    int count = 0;
```

```
    for (Iterator i = result.iterator(); i.hasNext();) {
        Employee e = (Employee) i.next();
        System.out.println(++count + ": " + e.getName() + ", " +
                           e.getDepartment().getName());
    }
}
```

The `getSingleResult()` method is provided as a convenience for queries that return only a single value. Instead of iterating to the first result in a collection, the object is directly returned. It is important to note, however, that `getSingleResult()` behaves differently than `getResultList()` in how it handles unexpected results. Whereas `getResultList()` returns an empty collection when no results are available, `getSingleResult()` throws a `NoResultException` exception. Therefore if there is a chance that the desired result may not be found, then this exception needs to be handled.

If multiple results are available after executing the query instead of the single expected result, `getSingleResult()` will throw a `NonUniqueResultException` exception. Again, this can be problematic for application code if the query criteria may result in more than one row being returned in certain circumstances. Although `getSingleResult()` is convenient to use, be sure that the query and its possible results are well understood, otherwise application code may have to deal with an unexpected runtime exception. Unlike other exceptions thrown by entity manager operations, these exceptions will not cause the provider to roll back the current transaction, if there is one.

`Query` objects may be reused as often as needed so long as the same persistence context that was used to create the query is still active. For transaction-scoped entity managers, this limits the lifetime of the `Query` object to the life of the transaction. Other entity manager types may reuse `Query` objects until the entity manager is closed or removed.

Listing 6-15 demonstrates caching a `Query` object instance on the bean class of a stateful session bean that uses an extended persistence context. Whenever the bean needs to find the list of employees who are currently not assigned to any project, it reuses the same `unassignedQuery` object that was initialized during PostConstruct.

Listing 6-15. *Reusing a Query Object*

```
@Stateful
public class ProjectManagerBean implements ProjectManager {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;

    Query unassignedQuery;

    @PostConstruct
    public void init() {
        unassignedQuery =
            em.createQuery("SELECT e " +
                           "FROM Employee e " +
                           "WHERE e.projects IS EMPTY");
    }
```



```
    public List findEmployeesWithoutProjects() {
        return unassignedQuery.getResultList();
    }

    // ...
}
```

Working with Query Results

The *result type* of a query is determined by the expressions listed in the SELECT clause of the query. If the result type of a query is the `Employee` entity, then executing `getResultList()` will result in a collection of zero or more `Employee` entity instances. There is a wide variety of results possible depending on the makeup of the query. The following are just some of the types that may result from JPQL queries:

- Basic types, such as `String`, the primitive types, and JDBC types
- Entity types
- An array of `Object`
- User-defined types created from a constructor expression

For developers used to JDBC, the most important thing to remember when using the `Query` interface is that the results are not encapsulated in a `ResultSet`. The collection or single result corresponds directly to the result type of the query.

Whenever an entity instance is returned, it becomes managed by the active persistence context. If that entity instance is modified and the persistence context is part of a transaction, then the changes will be persisted to the database. The only exception to this rule is the use of transaction-scoped entity managers outside of a transaction. Any query executed in this situation returns detached entity instances instead of managed entity instances. To make changes on these detached entities, they must first be merged into a persistence context before they can be synchronized with the database.

A consequence of the long-term management of entities with application-managed and extended persistence contexts is that executing large queries will cause the persistence context to grow as it stores all of the managed entity instances that are returned. If many of these persistence contexts are holding onto large numbers of managed entities for long periods of time, then memory use may become a concern. The `clear()` method of the `EntityManager` interface may be used to clear application-managed and extended persistence contexts, removing unnecessary managed entities.

Optimizing Read-Only Queries

When the query results will not be modified, queries using transaction-scoped entity managers outside of a transaction are typically more efficient than queries executed within a transaction when the result type is an entity. When query results are prepared within a transaction, the persistence provider has to take steps to convert the results into managed entities. This usually entails taking a snapshot of the data for each entity in order to have a baseline to compare against when the transaction is committed. If the managed entities are never modified, then the effort of converting the results into managed entities is wasted.

Outside of a transaction, in some circumstances the persistence provider may be able to optimize the case where the results will be detached immediately. Therefore it can avoid the overhead of creating the managed versions. Note that this technique does not work on application-managed or extended entity managers, since their persistence context outlives the transaction. Any query result from this type of persistence context may be modified for later synchronization to the database even if there is no transaction.

When encapsulating query operations behind a stateless session façade, the easiest way to execute non-transactional queries is to use the `NOT_SUPPORTED` transaction attribute for the session bean method. This will cause any active transaction to be suspended, forcing the query results to be detached and enabling this optimization. Listing 6-16 shows an example of this technique.

Listing 6-16. *Executing a Query Outside of a Transaction*

```
@Stateless
public class QueryServiceBean implements QueryService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public List findAllDepartmentsDetached() {
        return em.createQuery("SELECT d FROM Department d")
                 .getResultList();
    }

    // ...
}
```

Special Result Types

The array of `Object` result occurs whenever a query involves more than one expression in the SELECT clause. Common examples include projection of entity fields and aggregate queries where grouping expressions or multiple functions are used. Listing 6-17 revisits the menu generator from Listing 6-14 using a projection query instead of returning full `Employee` entity instances. Each element of the `List` is cast to an array of `Object` that is then used to extract the employee and department name information.

Listing 6-17. *Handling Multiple Result Types*

```
public void displayProjectEmployees(String projectName) {
    List result = em.createQuery("SELECT e.name, e.department.name " +
                                 "FROM Project p JOIN p.employees e " +
                                 "WHERE p.name = ?1 " +
                                 "ORDER BY e.name")
                    .setParameter(1, projectName)
                    .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext();) {
        Object[] values = (Object[]) i.next();
        System.out.println(++count + ": " + values[0] + ", " + values[1]);
    }
}
```

Constructor expressions provide developers with a way to map array of `Object` result types to custom objects. Typically this is used to convert the results into JavaBean-style classes that provide getters for the different returned values. This makes the results easier to work with and makes it possible to use the results directly in an environment such as JavaServer Faces without additional translation.

A constructor expression is defined in JPQL using the NEW operator in the SELECT clause. The argument to the NEW operator is the fully qualified name of the class that will be instantiated to hold the results for each row of data returned. The only requirement on this class is that it has a constructor with arguments matching the exact type and order that will be specified in the query. Listing 6-18 shows an `EmpMenu` class defined in the package `example` that could be used to hold the results of the query that was executed in Listing 6-17.

Listing 6-18. *Defining a Class for Use in a Constructor Expression*

```
package example;

public class EmpMenu {
    private String employeeName;
    private String departmentName;

    public EmpMenu(String employeeName, String departmentName) {
        this.employeeName = employeeName;
        this.departmentName = departmentName;
    }

    public String getEmployeeName() { return employeeName; }
    public String getDepartmentName() { return departmentName; }
}
```

Listing 6-19 shows the same example as Listing 6-17 using the fully qualified EmpMenu class name in a constructor expression. Instead of working with array indexes, each result is cast to the EmpMenu class and used like a regular Java object.

Listing 6-19. *Using Constructor Expressions*

```
public void displayProjectEmployees(String projectName) {
    List result =
        em.createQuery("SELECT NEW example.EmpMenu(e.name, e.department.name) " +
                       "FROM Project p JOIN p.employees e " +
                       "WHERE p.name = ?1 " +
                       "ORDER BY e.name")
          .setParameter(1, projectName)
          .getResultList();
    int count = 0;
    for (Iterator i = result.iterator(); i.hasNext();) {
        EmpMenu menu = (EmpMenu) i.next();
        System.out.println(++count + ": " + menu.getEmployeeName() + ", " +
                           menu.getDepartmentName());
    }
}
```

Query Paging

Large result sets from queries are often a problem for many applications. In cases where it would be overwhelming to display the entire result set, or if the application medium makes displaying many rows inefficient (web applications, in particular), applications must be able to display ranges of a result set and provide users with the ability to control the range of data that they are viewing. The most common form of this technique is to present the user with a fixed-size table that acts as a sliding window over the result set. Each increment of results displayed is called a *page*, and the process of navigating through the results is called *pagination*.

Efficiently paging through result sets has long been a challenge for both application developers and database vendors. Before support existed at the database level, a common technique was to first retrieve all of the primary keys for the result set and then issue separate queries for the full results using ranges of primary key values. Later, database vendors added the concept of logical row number to query results, guaranteeing that so long as the result was ordered, the row number could be relied on to retrieve portions of the result set. More recently, the JDBC specification has taken this even further with the concept of scrollable result sets, which can be navigated forwards and backwards as required.

The Query interface provides support for pagination via the setFirstResult() and setMaxResults() methods. These methods specify the first result to be received (numbered from zero) and the maximum number of results to return relative to that point. A persistence provider may choose to implement support for this feature in a number of different ways, as not all databases benefit from the same approach. It's a good idea to become familiar with how your vendor approaches pagination and what level of support exists in the target database platform for your application.

■ Caution The setFirstResult() and setMaxResults() methods should not be used with queries that join across collection relationships (one-to-many and many-to-many) because these queries may return duplicate values. The duplicate values in the result set make it impossible to use a logical result position.

To better illustrate pagination support, consider the stateful session bean shown in Listing 6-20. Once created, it is initialized with the name of a query to count the total results and the name of a query to generate the report. When results are requested, it uses the page size and current page number to calculate the correct parameters for the setFirstResult() and setMaxResults() methods. The total number of results possible is calculated by executing the count query. By using the next(), previous(), and getCurrentResults() methods, presentation code can page through the results as required. If this session bean were bound into an HTTP session, it could be directly used by a JSP or JavaServer Faces page presenting the results in a data table.

Listing 6-20. *Stateful Session Report Pager*

```
@Stateful
public class ResultPagerBean implements ResultPager {
    @PersistenceContext(unitName="QueryPaging")
    private EntityManager em;

    private String reportQueryName;
    private int currentPage;
    private int maxResults;
    private int pageSize;

    public int getPageSize() {
        return pageSize;
    }

    public int getMaxPages() {
        return maxResults / pageSize;
    }

    public void init(int pageSize, String countQueryName,
                     String reportQueryName) {
        this.pageSize = pageSize;
        this.reportQueryName = reportQueryName;
        maxResults = (Long) em.createNamedQuery(countQueryName)
                              .getSingleResult();
        maxResults = resultCount.longValue();
        currentPage = 0;
    }
```

```
    public List getCurrentResults() {
        return em.createNamedQuery(reportQueryName)
                 .setFirstResult(currentPage * pageSize)
                 .setMaxResults(pageSize)
                 .getResultList();
    }

    public void next() {
        currentPage++;
    }

    public void previous() {
        currentPage--;
        if (currentPage < 0) {
            currentPage = 0;
        }
    }

    public int getCurrentPage() {
        return currentPage;
    }

    public void setCurrentPage(int currentPage) {
        this.currentPage = currentPage;
    }

    @Remove
    public void finished() {
    }
}
```

Queries and Uncommitted Changes

Executing queries against entities that have been created or changed in a transaction is a topic that requires special consideration. As we discussed in Chapter 5, the persistence provider will attempt to minimize the number of times the persistence context must be flushed within a transaction. Optimally this will occur only once, when the transaction commits. While the transaction is open and changes are being made, the provider relies on its own internal cache synchronization to ensure that the right version of each entity is used in entity manager operations. At most the provider may have to read new data from the database in order to fulfill a request. All entity operations other than queries can be satisfied without flushing the persistence context to the database.

Queries are a special case because they are executed directly as SQL against the database. Because the database executes the query and not the persistence provider, the active persistence context cannot usually be consulted by the query. As a result, if the persistence context has not been flushed and the database query would be impacted by the changes pending in the persistence context, incorrect data is likely to be retrieved from the query. The entity manager

`find()` operation, on the other hand, always checks the persistence context before going to the database, so this is not a concern.

The good news is that by default, the persistence provider will ensure that queries are able to incorporate pending transactional changes in the query result. It might accomplish this by flushing the persistence context to the database, or it might leverage its own runtime information to ensure the results are correct.

And yet, there are times when having the persistence provider ensure query integrity is not necessarily the behavior we need. The problem is that it is not always easy for the provider to determine the best strategy to accommodate the integrity needs of a query. There is no way the provider can logically determine at a fine-grained level which objects have changed and therefore need to be incorporated into the query results. If the provider solution to ensuring query integrity is to flush the persistence context to the database, then you might have a performance problem if this is a frequent occurrence.

To put this issue in context, consider a message board application, which has modeled conversation topics as `Conversation` entities. Each `Conversation` entity refers to one or more messages represented by a `Message` entity. Periodically, conversations are archived when the last message added to the conversation is more than 30 days old. This is accomplished by changing the `status` of the `Conversation` entity from "ACTIVE" to "INACTIVE". The two queries to obtain the list of active conversations and the last message date for a given conversation are shown in Listing 6-21.

Listing 6-21. *Conversation Queries*

```
@NamedQueries({
    @NamedQuery(name="findActiveConversations",
                query="SELECT c " +
                      "FROM Conversation c " +
                      "WHERE c.status = 'ACTIVE'"),
    @NamedQuery(name="findLastMessageDate",
                query="SELECT MAX(m.postingDate) " +
                      "FROM Conversation c JOIN c.messages m " +
                      "WHERE c = :conversation")
})
```

Listing 6-22 shows the session bean method used to perform this maintenance, accepting a `Date` argument that specifies the minimum age for messages in order to still be considered an active conversation. In this example, we see that two queries are being executed. The "findAllActiveConversations" query collects all of the active conversations, while the "findLastMessageDate" returns the last date that a message was added to a `Conversation` entity. As the code iterates over the `Conversation` entities, it invokes the "findLastMessageDate" query for each one. As these two queries are related, it is reasonable for a persistence provider to assume that the results of the "findLastMessageDate" query will depend on the changes being made to the `Conversation` entities. If the provider ensures the integrity of the "findLastMessageDate" query by flushing the persistence context, this could become a very expensive operation if hundreds of active conversations are being checked.

Listing 6-22. *Archiving Conversation Entities*

```
@Stateless
public class ConversationMaintenanceBean implements ConversationMaintenance {
    @PersistenceContext(unitName="MessageBoard")
    EntityManager em;

    public void archiveConversations(Date minAge) {
        List<Conversation> active = (List<Conversation>)
            em.createNamedQuery("findActiveConversations")
              .getResultList();
        Query maxAge = em.createNamedQuery("findLastMessageDate");
        for (Conversation c : active) {
            maxAge.setParameter("conversation", c);
            Date lastMessageDate  = (Date) maxAge.getSingleResult();
            if (lastMessageDate.before(minAge)) {
                c.setStatus("INACTIVE");
            }
        }
    }

    // ...
}
```

To give developers more control over the integrity requirements of queries, the `EntityManager` and `Query` interfaces support a `setFlushMode()` method to set the *flush mode*, an indicator to the provider how it should handle pending changes and queries. There are two possible flush mode settings, `AUTO` and `COMMIT`, which are defined by the `FlushModeType` enumerated type. The default setting is `AUTO`, which means that the provider should ensure that pending transactional changes are included in query results. If a query might overlap with changed data in the persistence context, then this setting will ensure that the results are correct.

The `COMMIT` flush mode tells the provider that queries don't overlap with changed data in the persistence context, so it does not need to do anything in order to get correct results. Depending on how the provider implements its query integrity support, this might mean that it does not have to flush the persistence context before executing a query since you have indicated that there is nothing in memory that will be queried from the database.

Although the flush mode is set on the entity manager, the flush mode is really a property of the persistence context. For transaction-scoped entity managers, that means the flush mode has to be changed in every transaction. Extended and application-managed entity managers will preserve their flush-mode setting across transactions.

Setting the flush mode on the entity manager applies to all queries, while setting the flush mode for a query limits the setting to that scope. Setting the flush mode on the query overrides the entity manager setting as you would expect. If the entity manager setting is `AUTO` and one query has the `COMMIT` setting, then the provider will guarantee query integrity for all of the

queries other than the one with the `COMMIT` setting. Likewise if the entity manager setting is `COMMIT` and one query has an `AUTO` setting, then only the query with the `AUTO` setting is guaranteed to incorporate pending changes from the persistence context.

Generally speaking, if you are going to execute queries in transactions where data is being changed, `AUTO` is the right answer. If you are concerned about the performance implications of ensuring query integrity, consider changing the flush mode to `COMMIT` on a per-query basis. Changing the value on the entity manager, while convenient, can lead to problems if more queries are added to the application later and they require `AUTO` semantics.

Coming back to the example at the start of this section, we can set the flush mode on the `Query` object for the "findLastMessageDate" query to `COMMIT` because it does not need to see the changes being made to the `Conversation` entities. The following fragment shows how this would be accomplished for the `archiveConversations()` method shown in Listing 6-22:

```
public void archiveConversations(Date minAge) {
    // ...
    Query maxAge = em.createNamedQuery("findLastMessageDate");
    maxAge.setFlushMode(FlushModeType.COMMIT);
    // ...
}
```

Bulk Update and Delete

Like their SQL counterparts, JPQL bulk update and delete statements are designed to make changes to large numbers of entities in a single operation without requiring the individual entities to be retrieved and modified using the entity manager. Unlike SQL, which operates on tables, JPQL update and delete statements must take the full range of mappings for the entity into account. These operations are challenging for vendors to implement correctly, and as a result, there are restrictions on the use of these operations that must be well understood by developers.

The full syntax for UPDATE and DELETE statements is described in Chapter 7. The following sections will describe how to use these operations effectively and the issues that may result when used incorrectly.

Using Bulk Update and Delete

Bulk update of entities is accomplished with the UPDATE statement. This statement operates on a single entity type and sets one or more single-valued properties of the entity (either a state field or a single-valued association) subject to the conditions in the WHERE clause. In terms of syntax, it is nearly identical to the SQL version with the exception of using entity expressions instead of tables and columns. Listing 6-23 demonstrates using a bulk update statement. Note that the use of the `REQUIRES_NEW` transaction attribute type is significant and will be discussed following the examples.

Listing 6-23. *Bulk Update of Entities*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="BulkQueries")
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void assignManager(Department dept, Employee manager) {
         em.createQuery("UPDATE Employee e " +
                        "SET e.manager = ?1 " +
                        "WHERE e.department = ?2 ")
           .setParameter(1, manager)
           .setParameter(2, dept)
           .executeUpdate();
    }
}
```

Bulk removal of entities is accomplished with the DELETE statement. Again, the syntax is the same as the SQL version except that the target in the FROM clause is an entity instead of a table and the WHERE clause is composed of entity expressions instead of column expressions. Listing 6-24 demonstrates bulk removal of entities.

Listing 6-24. *Bulk Removal of Entities*

```
@Stateless
public class ProjectServiceBean implements ProjectService {
    @PersistenceContext(unitName="BulkQueries")
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void removeEmptyProjects() {
         em.createQuery("DELETE FROM Project p " +
                        "WHERE p.employees IS EMPTY ")
           .executeUpdate();
    }
}
```

The first issue for developers to consider when using these statements is that the persistence context is not updated to reflect the results of the operation. Bulk operations are issued as SQL against the database, bypassing the in-memory structures of the persistence context. Therefore updating the salary of all of the employees will not change the current values for any entities managed in memory as part of a persistence context. The developer can rely only on entities retrieved after the bulk operation completes.

For developers using transaction-scoped persistence contexts, this means that the bulk operation should either execute in a transaction all by itself or be the first operation in the transaction. Running the bulk operation in its own transaction is the preferred approach as it minimizes the chance of the developer accidentally fetching data before the bulk change occurs. Executing the bulk operation and then working with entities after it completes is also

safe, because then any `find()` operation or query will go to the database to get current results. The examples in Listing 6-23 and Listing 6-24 used the `REQUIRES_NEW` transaction attribute to ensure that the bulk operations occurred within their own transactions.

A typical strategy for persistence providers dealing with bulk operations is to invalidate any in-memory cache of data related to the target entity. This forces data to be fetched from the database the next time it is required. How much cached data gets invalidated depends on the sophistication of the persistence provider. If the provider can detect that the update impacts only a small range of entities, then those specific entities may be invalidated, leaving other cached data in place. Such optimizations are limited, however, and if the provider cannot be sure of the scope of the change, then the entire cache must be invalidated. This can have performance impacts on the application if bulk changes are a frequent occurrence.

■ **Caution** SQL update and delete operations should never be executed on tables mapped by an entity. The JPQL operations tell the provider what cached entity state must be invalidated in order to remain consistent with the database. Native SQL operations bypass such checks and can quickly lead to situations where the in-memory cache is out of date with respect to the database.

The danger present in bulk operations and the reason they must occur first in a transaction is that any entity actively managed by a persistence context will remain that way, oblivious to the actual changes occurring at the database level. The active persistence context is separate and distinct from any data cache that the provider may use for optimizations. Consider the following sequence of operations:

1. A new transaction starts.
2. Entity A is created by calling `persist()` to make the entity managed.
3. Entity B is retrieved from a `find()` operation and modified.
4. A bulk remove deletes entity A.
5. A bulk update changes the same properties on entity B that were modified in step 3.
6. The transaction commits.

What should happen to entities A and B in this sequence? In the case of entity A, the provider has to assume that the persistence context is correct and so will still attempt to insert the new entity even though it should have been removed. In the case of entity B, again the provider has to assume that managed version is the correct version and will attempt to update the version in the database, undoing the bulk update change.

This brings us to the issue of extended persistence contexts. Bulk operations and extended persistence contexts are a particularly dangerous combination because the persistence context survives across transaction boundaries, but the provider will never refresh the persistence context to reflect the changed state of the database after a bulk operation has completed. When the extended persistence context is next associated with a transaction, it will attempt to synchronize its current state with the database. Since the managed entities in the persistence

context are now out of date with respect to the database, any changes made since the bulk operation could result in incorrect results being stored. In this situation, the only option is to refresh the entity state or ensure that the data is versioned in such a way that the incorrect change can be detected. Locking strategies and refreshing of entity state are discussed in Chapter 9.

Bulk Delete and Relationships

In our discussion of the `remove()` operation in the previous chapter, we emphasized that relationship maintenance is always the responsibility of the developer. The only time a cascading remove occurs is when the `REMOVE` cascade option is set for a relationship. Even then, the persistence provider won't automatically update the state of any managed entities that refer to the removed entity. As we are about to see, the same requirement holds true when using DELETE statements as well.

A DELETE statement in JPQL corresponds more or less to a DELETE statement in SQL. Writing the statement in JPQL gives you the benefit of working with entities instead of tables, but the semantics are exactly the same. This has implications in how applications must write DELETE statements in order to ensure that they execute correctly and leave the database in a consistent state.

DELETE statements do not cascade to related entities. Even if the `REMOVE` cascade option is set on a relationship, it will not be followed. It is your responsibility to ensure that relationships are correctly updated with respect to the entities that have been removed. The persistence provider also has no control over constraints in the database. If you attempt to remove data that is the target of a foreign key relationship in another table, you will get a referential integrity constraint violation in return.

Let's look at an example that puts these issues in context. Consider, for example, that a company wishes to reorganize its department structure. We want to delete a number of departments and then assign the employees to new departments. The first step is to delete the old departments, so the following statement is to be executed:

```
DELETE FROM Department d
WHERE d.name IN ('CA13', 'CA19', 'NY30')
```

This is a straightforward operation. We want to remove the department entities that match the given list of names using a DELETE statement instead of querying for the entities and using the `remove()` operation to dispose of them. But when this query is executed, a `PersistenceException` exception is thrown, reporting that a foreign key integrity constraint has been violated. Therefore, another table has a foreign key reference to one of the rows we are trying to delete. Checking the database, we see that the table mapped by the `Employee` entity has a foreign key constraint against the table mapped by the `Department` entity. Since the foreign key value in the `Employee` table is not `NULL`, the parent key from the `Department` table can't be removed.

Therefore we need to first update the `Employee` entities in question to make sure that they do not point to the department we are trying to delete:

```
UPDATE Employee e
SET e.department = null
WHERE e.department.name IN ('CA13', 'CA19', 'NY30')
```


With this change the original DELETE statement will work as expected. Now consider what would have happened if the integrity constraint had not been in the database. The DELETE operation would have completed successfully, but the foreign key values would still be sitting in the `Employee` table. The next time the persistence provider tried to load the `Employee` entities with dangling foreign keys, it would be unable to resolve the target entity. The outcome of this operation is vendor-specific but will most likely lead to a `PersistenceException` exception being thrown, complaining of the invalid relationship.

Query Hints

Query hints are the Java Persistence API extension point for vendor-specific query features. A hint is simply a string name and object value. The meaning of both the name and value is entirely up to the persistence provider. Every query may be associated with any number of hints, set either in persistence unit metadata as part of the `@NamedQuery` annotation, or on the `Query` interface itself using the `setHint()` method.

We left query hints until the end of this chapter because they are the only feature in the query API that has no standard usage. Everything about hints is vendor-specific. The only guarantee provided by the specification is that providers must ignore hints that they do not understand. Listing 6-25 demonstrates the "toplink.cache-usage" hint supported by the Reference Implementation of the Java Persistence API to indicate that the cache should not be checked when reading an `Employee` from the database. Unlike the `refresh()` method of the `EntityManager` interface, this hint will not cause the query result to override the current cached value.

Listing 6-25. *Using Query Hints*

```
public Employee findEmployeeNoCache(int empId) {
    Query q = em.createQuery("SELECT e FROM Employee e WHERE e.id = ?1");
    // force read from database
    q.setHint("toplink.cache-usage", "DoNotCheckCache");
    q.setParameter(1, empId);
    try {
        return (Employee) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}
```

If this query were to be executed frequently, a named query would be more efficient. The following named query definition incorporates the cache hint used earlier:

```
@NamedQuery(name="findEmployeeNoCache",
            query="SELECT e FROM Employee e WHERE e.id = :empId",
            hints={@QueryHint(name="toplink.cache-usage", value="DoNotCheckCache")})
```

The `hints` element accepts an array of `@QueryHint` annotations, allowing any number of hints to be set for a query.

Query Best Practices

The typical application using the Java Persistence API is going to have many queries defined. It is the nature of enterprise applications that information is constantly being queried from the database, for everything from complex reports to drop-down lists in the user interface. Therefore efficiently using queries can have a major impact on your application as a whole.

Named Queries

First and foremost, we recommend named queries whenever possible. Persistence providers will often take steps to precompile JPQL named queries to SQL as part of the deployment or initialization phase of an application. This avoids the overhead of continuously parsing JPQL and generating SQL. Even with a cache for converted queries, dynamic query definition will always be less efficient than using named queries.

Named queries also enforce the best practice of using query parameters. Query parameters help to keep the number of distinct SQL strings parsed by the database to a minimum. Since databases typically keep a cache of SQL statements on hand for frequently accessed queries, this is an essential part of ensuring peak database performance.

As we discussed in the Dynamic Query Definition section, query parameters also help to avoid security issues caused by concatenating values into query strings. For applications exposed to the web, security has to be a concern at every level of an application. You can either spend a lot of effort trying to validate input parameters, or you can use query parameters and let the database do the work for you.

When naming queries, decide on a naming strategy early in the application development cycle with the understanding that the query namespace is global for each persistence unit. Collisions between query names are likely to be a common frustration if there is no established naming pattern.

Finally, using named queries allows for JPQL queries to be overridden with SQL queries or even with vendor-specific languages and expression frameworks. For applications migrating from an existing object-relational mapping solution, it is quite likely that the vendor will provide some support for invoking their existing query solution using the named query facility in the Java Persistence API. We will discuss SQL named queries in Chapter 9.

Report Queries

If you are executing queries that return entities for reporting purposes and have no intention of modifying the results, consider executing queries using a transaction-scoped entity manager but outside of a transaction. The persistence provider may be able to detect the lack of a transaction and optimize the results for detachment, often by skipping some of the steps required to create an interim managed version of the entity results.

Likewise, if an entity is expensive to construct due to eager relationships or a complex table mapping, consider selecting individual entity properties using a projection query instead of retrieving the full entity result. If all you need is the name and office phone number for 500 employees, selecting only those two fields is likely to be far more efficient than fully constructing 1,000 entity instances.

Query Hints

It is quite likely that vendors will entice you with a variety of hints to enable different performance optimizations for queries. Query hints may well be an essential tool in meeting your performance expectations. We strongly advise, however, that you resist the urge to embed query hints in your application code. The ideal location for query hints is in an XML mapping file (which we will be describing in Chapter 10), or at the very least as part of a named query definition. Hints are often highly dependent on the target platform and may well change over time as different aspects of the application impact the overall balance of performance. Keep hints decoupled from your code if at all possible.

Stateless Session Beans

We tried to demonstrate as many examples as possible in the context of a stateless session bean method, as we believe that this is the best way to organize queries in a Java EE application. Using the stateless session bean has a number of benefits over simply embedding queries all over the place in application code:

- Clients can execute queries by invoking an appropriately named business method instead of relying on a cryptic query name or multiple copies of the same query string.
- Stateless session bean methods can optimize their transaction usage depending on whether or not the results need to be managed or detached.
- Using a transaction-scoped persistence context ensures that large numbers of entity instances don't remain managed long after they are needed.
- For existing entity bean applications, the stateless session bean is the ideal vehicle for migrating finder queries away from the entity bean home interface. We will discuss this technique in Chapter 13.

This is not to say that other components are unsuitable locations for queries, but stateless session beans are a well-established best practice for hosting queries in the Java EE environment.

Bulk Update and Delete

If bulk update and delete operations must be used, ensure that they are executed only in an isolated transaction where no other changes are being made. There are many ways in which these queries can negatively impact an active persistence context. Interweaving these queries with other non-bulk operations requires careful management by the application.

Entity versioning and locking requires special consideration when bulk update operations are used. Bulk delete operations can have wide ranging ramifications depending on how well the persistence provider can react and adjust entity caching in response. Therefore we view bulk update and delete operations as being highly specialized, to be used with care.

Provider Differences

Take time to become familiar with the SQL that your persistence provider generates for different JPQL queries. Although understanding SQL is not necessary for writing JPQL queries,

knowing what happens in response to the various JPQL operations is an essential part of performance tuning. Joins in JPQL are not always explicit, and you may find yourself surprised at the complex SQL generated for a seemingly simple JPQL query.

The benefits of features such as query paging are also dependent on the approach used by your persistence provider. There are a number of different techniques that can be used to accomplish pagination, many of which suffer from performance and scalability issues. Because the Java Persistence API can't dictate a particular approach that will work well in all cases, become familiar with the approach used by your provider and whether or not it is configurable.

Finally, understanding the provider strategy for when and how often it flushes the persistence context is necessary before looking at optimizations such as changing the flush mode. Depending on the caching architecture and query optimizations used by a provider, changing the flush mode may or may not make a difference to your application.

Summary

We began this chapter with an introduction to JPQL, the query language defined by the Java Persistence API. We briefly discussed the origins of JPQL and its role in writing queries that interact with entities. We also provided an overview of major JPQL features for developers already experienced with SQL or previous versions of EJB QL.

In the discussion on executing queries, we introduced the methods for defining queries both dynamically at runtime and statically as part of persistence unit metadata. We looked at the `Query` interface and the types of query results possible using JPQL. We also looked at parameter binding, strategies for handling large result sets and how to ensure that queries in transactions with modified data complete successfully.

In the section on bulk update and delete we looked at how to execute these types of queries and how to ensure that they are used safely by the application. We provided details on how persistence providers deal with bulk operations and the impact that they have on the active persistence context.

We ended our discussion of query features with a look at query hints. We showed how to specify hints and provided an example using hints supported by the Reference Implementation of the Java Persistence API.

Finally, we summarized our view of best practices relating to queries, looking at named queries, different strategies for the various query types, as well as the implementation details that need to be understood for different persistence providers.

In the next chapter, we will continue to focus on queries by examining JPQL in detail.

CHAPTER 7

■ ■ ■

Query Language

Based on the EJB Query Language (EJB QL) first introduced in EJB 2.0, the Java Persistence Query Language (JPQL), is a portable query language designed to combine the syntax and simple query semantics of SQL with the expressiveness of an object-oriented expression language. Queries written using this language can be portably compiled to SQL on all major database servers.

In the last chapter, we looked at programming using the query interfaces and presented a brief introduction to JPQL for users already experienced with SQL. This chapter will explore the query language in detail, breaking the language down piece by piece with examples to demonstrate all of its features.

Introduction

In order to describe what JPQL is, it is important to make clear what it is not. JPQL is not SQL. Despite the similarities between the two languages in terms of keywords and overall structure, there are very important differences. Attempting to write JPQL as if it were SQL is the easiest way to get frustrated with the language. The similarities between the two languages are intentional in order to give developers a feel for what the language can accomplish, but the object-oriented nature of the language requires a different mode of thinking.

If JPQL is not SQL, then what is it? Put simply, JPQL is a language for querying entities. Instead of tables and rows, the currency of the language is entities and objects. It provides us with a way to express queries in terms of entities and their relationships, operating on the persistent state of the entity as defined in the object model, not in the physical database model.

If the Java Persistence API supports SQL queries, why introduce a new query language? There are a couple of important reasons to consider JPQL over SQL. The first is portability. It may be translated into the SQL dialect of all major database vendors. The second reason is that queries are literally written against the domain model of persistent entities. Queries may be written without any need to know exactly how the entities are mapped to the database. We hope that the examples in this chapter will demonstrate the power present in even the simplest JPQL expressions.

Adopting JPQL does not mean losing all of the SQL features you have grown accustomed to using. A broad selection of SQL features are directly supported, including subqueries, aggregate queries, update and delete statements, numerous SQL functions, and more.

Terminology

Queries fall into one of four categories: select, aggregate, update, and delete queries. Select queries retrieve persistent state from one or more entities, filtering results as required. Aggregate queries are variations of select queries that group the results and produce summary data. Together, select and aggregate queries are sometimes called *report queries*, since they are primarily focused on generating data for reporting. Update and delete queries are used to conditionally modify or remove entire sets of entities. Each query type will be described in detail in its own section as the chapter progresses.

Queries operate on the set of entities defined by a persistence unit. This set of entities is known as the *abstract persistence schema*, the collection of which defines the overall domain from which results may be retrieved.

■ **Note** To allow this chapter to be used as a companion to the Query Language chapter of the Java Persistence API specification, the same terminology is used where possible.

In query expressions, entities are referred to by name. If an entity has not been explicitly named (using the `name` attribute of the `@Entity` annotation, for example) the unqualified class name is used by default. This name is the *abstract schema name* of the entity in the context of a query.

Entities are composed of one or more persistence properties implemented as fields or JavaBean properties. The *abstract schema type* of a persistent property on an entity refers to the class or primitive type used to implement that property. For example, if the `Employee` entity has a property `name` of type `String`, then the abstract schema type of that property in query expressions is `String` as well. Simple persistent properties with no relationship mapping comprise the persistent state of the entity and are referred to as *state fields*. Persistent properties that are also relationships are called *association fields*.

As we saw in the last chapter, queries may be defined dynamically or statically. The examples in this chapter will consist of queries that may be used either dynamically or statically depending on the needs of the application.

Finally, it is important to note that queries are not case sensitive except in two cases. Entity names and property names must be specified exactly as they are named.

Example Data Model

Figure 7-1 shows the domain model for the queries in this chapter. Continuing the examples we have been using throughout the book, it demonstrates many different relationship types, including unidirectional, bidirectional, and self-referencing relationships. We have added the role names to this diagram to make the relationship property names explicit.

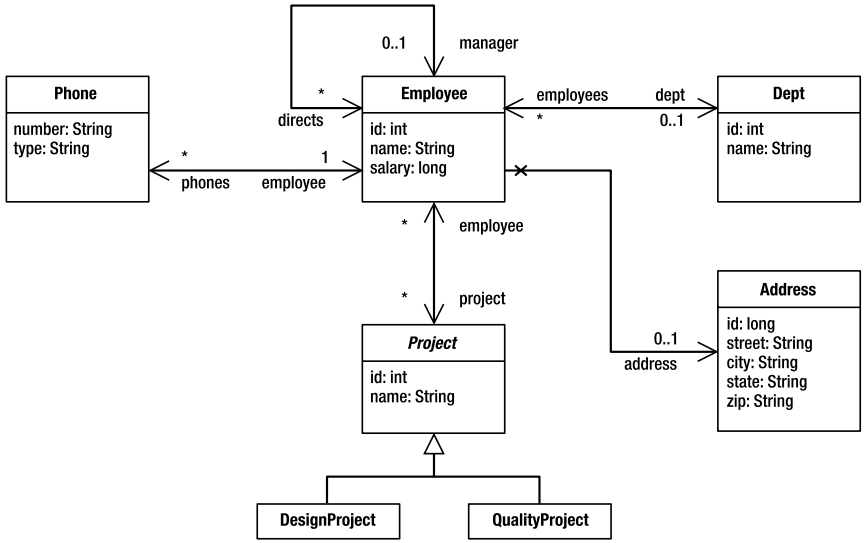

Figure 7-1. *Example application domain model*

The object relational mappings for this model are not included in this chapter except where we describe the SQL equivalent of a particular query. Knowing how a mapping is implemented is only a performance concern and is not necessary to write queries since the query language is based entirely on the object model and the logical relationships between entities. It is the job of the query translator to take the object-oriented query expressions and interpret the mapping metadata in order to produce the SQL required to execute the query on the database.

Example Application

Learning a new language can be a challenging experience. It's one thing to read through page after page of text describing the features of the language, but it's another thing completely to put these features into practice. To get used to writing queries, consider using an application like the one shown in Listing 7-1. This simple application reads queries from the console and executes them against the entities from a particular persistence unit.

Listing 7-1. *Application for Testing Queries*

```
package persistence;

import java.io.*;
import java.util.*;
import javax.persistence.*;
import org.apache.commons.lang.builder.*;
```

```
public class QueryTester {

    public static void main(String[] args) throws Exception {
        String unitName = args[0];

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory(unitName);
        EntityManager em = emf.createEntityManager();
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(System.in));

        for (;;) {
            System.out.print("JPQL> ");
            String query = reader.readLine();
            if (query.equals("quit")) {
                break;
            }
            if (query.length() == 0) {
                continue;
            }

            try {
                List result = em.createQuery(query).getResultList();
                if (result.size() > 0) {
                    int count = 0;
                    for (Object o : result) {
                        System.out.print(++count + " ");
                        printResult(o);
                    }
                } else {
                    System.out.println("0 results returned");
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    private static void printResult(Object result) throws Exception {
        if (result == null) {
            System.out.print("NULL");
        } else if (result instanceof Object[]) {
            Object[] row = (Object[]) result;
            System.out.print("[");
            for (int i = 0; i < row.length; i++) {
                printResult(row[i]);
            }
```



```
            System.out.print("]");
        } else if (result instanceof Long ||
                   result instanceof Double ||
                   result instanceof String) {
            System.out.print(result.getClass().getName() + ": " + result);
        } else {
            System.out.print(ReflectionToStringBuilder.toString(result, ➥
ToStringStyle.SHORT_PREFIX_STYLE));
        }
        System.out.println();
    }
}
```

The only requirement for using this application is the name of a persistence unit containing the entities you wish to query against. The application will read the persistence unit name from the command line and attempt to create an `EntityManagerFactory` for that name. If initialization is successful, queries may be typed at the `JPQL>` prompt. The query will be executed and the results printed out. The format of each result is the class name followed by each of the properties for that class. This example uses the Apache Jakarta Commons-Lang library to generate the object summary. Listing 7-2 demonstrates a sample session with the application.

Listing 7-2. *Example Session with* `QueryTester`

```
JPQL> SELECT p FROM Phone p WHERE p.type NOT IN ('office', 'home')
1 Phone[id=5,number=516-555-1234,type=cell,employee=Employee@13c0b53]
2 Phone[id=9,number=650-555-1234,type=cell,employee=Employee@193f6e2]
3 Phone[id=12,number=650-555-1234,type=cell,employee=Employee@36527f]
4 Phone[id=18,number=585-555-1234,type=cell,employee=Employee@bd6a5f]
5 Phone[id=21,number=650-555-1234,type=cell,employee=Employee@979e8b]
JPQL> SELECT d.name, AVG(e.salary) FROM Department d JOIN d.employees e ➥
GROUP BY d.name
1 [java.lang.String: QA
java.lang.Double: 52500.0
]
2 [java.lang.String: Engineering
java.lang.Double: 56833.333333333336
]
JPQL> quit
```

Select Queries

Select queries are the primary query type and facilitate the bulk retrieval of data from the database. Not surprisingly, select queries are also the most common form of query used in applications. The overall form of a select query is as follows:

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[ORDER BY <order_by_clause>]
```

The simplest form of a select query consists of two mandatory parts, the SELECT clause and the FROM clause. The SELECT clause defines the format of the query results, while the FROM clause defines the entity or entities from which the results will be obtained. Consider the following complete query that retrieves all of the employees in the company:

```
SELECT e
FROM Employee e
```

The structure of this query is very similar to a SQL query but with a couple of important differences. The first difference is that the domain of the query defined in the FROM clause is not a table but an entity, in this case the `Employee` entity. As in SQL, it has been aliased to the identifier `e`. This aliased value is known as an *identification variable* and is the key by which the entity will be referred to in the rest of the select statement. Unlike queries in SQL, where a table alias is optional, the use of identification variables is mandatory in JPQL.

The second difference is that the SELECT clause does not enumerate the fields of the table or use a wildcard to select all of the fields. Instead, only the identification variable is listed in order to indicate that the result type of the query is the `Employee` entity, not a tabular set of rows.

As the query processor iterates over the result set returned from the database, it converts the tabular row and column data into a set of entity instances. The `getResultList()` method of the `Query` interface will return a collection of zero or more `Employee` objects after evaluating the query.

Despite the differences in structure and syntax, every query is translatable to SQL. In order to execute a query, the query engine first builds an optimal SQL representation of the JPQL query. The resulting SQL query is what actually gets executed on the database. In this simple example the resulting SQL would look something like this:

```
SELECT id, name, salary, manager_id, dept_id, address_id
FROM emp
```

The SQL statement must read in all of the mapped columns required to create the entity instance, including foreign key columns. Even if the entity is cached in memory, the query engine will still typically read all required data in order to ensure that the cached version is up to date. Note that, had the relationships between the `Employee` and the `Department` or `Address` entities required eager loading, the SQL statement would either be extended to retrieve the extra data or multiple statements would have been batched together in order to completely construct the `Employee` entity. Every vendor will provide some method for displaying the SQL it generates from translating JPQL. For performance tuning in particular, being familiar with how your vendor approaches SQL generation can help you write more efficient queries.

Now that we have looked at a simple query and covered the basic terminology, the following sections will move through each of the clauses of the select query, explaining the syntax and features available.

The SELECT Clause

The SELECT clause of a query can take several forms, including simple and complex path expressions, transformation functions, multiple expressions (including constructor expressions), and aggregate functions. The following sections introduce path expressions and discuss the different styles of SELECT clauses and how they determine the result type of the query. Aggregate functions are detailed later in the chapter in the section on Aggregate Queries.

Path Expressions

Path expressions are the building blocks of queries. They are used to navigate out from an entity, either across a relationship to another entity (or collection of entities) or to one of the persistent properties of an entity. Navigation that results in one of the persistent state fields (either field or property) of an entity is referred to as a *state field path*. Navigation that leads to a single entity is referred to as a *single-valued association path*, while navigation to a collection of entities is referred to as a *collection-valued association path*.

The dot operator (`.`) signifies path navigation in an expression. For example, if the `Employee` entity has been mapped to the identification variable `e`, then `e.name` is a state field path expression resolving to the employee name. Likewise, the path expression `e.department` is a single-valued association from the employee to the department to which he or she is assigned. Finally, `e.directs` is a collection-valued association that resolves to the collection of employees reporting to an employee who is also a manager.

What makes path expressions so powerful is that they are not limited to a single navigation. Rather, navigation expressions can be chained together to traverse complex entity graphs, so long as the path moves from left to right across single-valued associations. A path cannot continue from a state field or collection-valued association. Using this technique, we can construct path expressions such as `e.department.name`, which is the name of the department to which the employee belongs.

Path expressions are used in every clause of a select query, determining everything from the result type of the query to the conditions under which the results should be filtered. Experience with path expressions is the key to writing effective queries.

Entities and Objects

The first and simplest form of the SELECT clause is a single identification variable. The result type for a query of this style is the entity to which the identification variable is associated. For example, the following query returns all of the departments in the company:

```
SELECT d
FROM Department d
```

The keyword OBJECT may be used to indicate that the result type of the query is the entity bound to the identification variable. It has no impact on the query, but it may be used as a visual clue:

```
SELECT OBJECT(d)
FROM Department d
```

The only problem with using OBJECT is that even though path expressions can resolve to an entity type, the syntax of the OBJECT keyword is limited to identification variables. The expression `OBJECT(e.department)` is illegal even though `Department` is an entity type. For that reason, we do not recommend the OBJECT syntax. It exists primarily for compatibility with previous versions of the language that required the OBJECT keyword on the assumption that a future revision to SQL would include the same terminology.

A path expression resolving to a state field or single-valued association may also be used in the SELECT clause. The result type of the query in this case becomes the type of the path expression, either the state field type or the entity type of a single-valued association. The following query returns the names for all employees:

```
SELECT e.name
FROM Employee e
```

The result type of the path expression in the SELECT clause is `String`, so executing this query using `getResultList()` will produce a collection of zero or more `String` objects.

Entities reached from a path expression may also be returned. The following query demonstrates returning a different entity as a result of path navigation:

```
SELECT e.department
FROM Employee e
```

The result type of this query is the `Department` entity since that is the result of traversing the `department` relationship from `Employee` to `Department`. Executing the query will therefore result in a collection of zero or more `Department` objects, including duplicates.

To remove the duplicates, the DISTINCT operator must be used:

```
SELECT DISTINCT e.department
FROM Employee e
```

The DISTINCT operator is functionally equivalent to the SQL operator of the same name. Once the result set is collected, duplicate values (using entity identity if the query result type is an entity) are removed so that only unique results are returned.

The result type of a select query is the type corresponding to each row in the result set produced by executing the query. This may include entities, primitive types and other persistent attribute types, but never a collection type. The following query is illegal:

```
SELECT d.employees
FROM Department d
```

The path expression `d.employees` is a collection-valued path that produces a collection type. Restricting queries in this way prevents the provider from having to combine successive rows from the database into a single result object.

Combining Expressions

Multiple expressions may be specified in the same SELECT clause by separating them with commas. The result type of the query in this case is an array of type `Object`, where the elements of the array are the results of resolving the expressions in the order in which they appeared in the query.

Consider the following query that returns only the name and salary of an employee:


```
SELECT e.name, e.salary
FROM Employee e
```

When executed, a collection of zero or more instances of arrays of type `Object` will be returned. Each array in this example has two elements, the first being a `String` containing the employee name and the second being a `Double` containing the employee salary. The practice of reporting only a subset of the state fields from an entity is called *projection* because the entity data is projected out from the entity into tabular form.

Projection is a useful technique for web applications where only a few pieces of information are displayed from a large set of entity instances. Depending on how the entity has been mapped, it may require a complex SQL query to fully retrieve the entity state. If only two fields are required, then the extra effort spent constructing the entity instance may have been wasted. A projection query that returns only the minimum amount of data is more useful in these cases.

Constructor Expressions

A more powerful form of SELECT clause involving multiple expressions is the constructor expression, which specifies that the results of the query are to be stored using a user-specified object type. Consider the following query:

```
SELECT NEW example.EmployeeDetails(e.name, e.salary, e.department.name)
FROM Employee e
```

The result type of this query is the type `example.EmployeeDetails`. As the query processor iterates over the results of the query, it instantiates new instances of `EmployeeDetails` using the constructor that matches the expression types listed in the query. In this case the expression types are `String`, `Double`, and `String`, so the query engine will search for a constructor with those class types for arguments. Each row in the resulting query collection is therefore an instance of `EmployeeDetails` containing the employee name, salary, and department name.

The result object type must be referred to using the fully qualified name of the object. The class does not have to be mapped to the database in any way, however. Any class with a constructor compatible with the expressions listed in the SELECT clause can be used in a constructor expression.

Constructor expressions are powerful tools for constructing coarse-grained data transfer objects or view objects for use in other application tiers. Instead of manually constructing these objects, a single query can be used to gather together view objects ready for presentation on a web page.

Inheritance and Polymorphism

The Java Persistence API supports inheritance between entities. As a result, the query language supports polymorphic results where multiple subclasses of an entity can be returned by the same query.

In the example model, `Project` is an abstract base class for `QualityProject` and `DesignProject`. If an identification variable is formed from the `Project` entity, then the query results will include a mixture of `QualityProject` and `DesignProject` objects, and the results may be cast to these classes as necessary. There is no special syntax to enable this behavior. The following query retrieves all projects with at least one employee:

```
SELECT p
FROM Project p
WHERE p.employees IS NOT EMPTY
```

The impact that inheritance between entities has on the generated SQL is important to understand for performance reasons and will be described in Chapter 8.

The FROM Clause

The FROM clause is used to declare one or more identification variables, optionally derived from joined relationships, that form the domain over which the query should draw its results. The syntax of the FROM clause consists of one or more identification variables and join clause declarations.

Identification Variables

The identification variable is the starting point for all query expressions. Every query must have at least one identification variable defined in the FROM clause, and that variable must correspond to an entity type. When an identification variable declaration does not use a path expression (that is, when it is a single entity name), it is referred to as a *range variable declaration*. This terminology comes from set theory as the variable is said to range over the entity.

Range variable declarations use the following syntax: `<entity_name> [AS] <identifier>`. The identifier must follow the standard Java naming rules and may be referenced throughout the query in a case-insensitive manner. Multiple declarations may be specified by separating them with commas.

Path expressions may also be aliased to identification variables in the case of joins and subqueries. The syntax for identification variable declarations in these cases will be covered in their respective sections next.

Joins

A join is a query that combines results from multiple entities. Joins in JPQL queries are logically equivalent to SQL joins. Ultimately, once the query is translated to SQL, it is quite likely that the joins between entities will produce similar joins amongst the tables to which the entities are mapped. Understanding when joins occur is therefore important to writing efficient queries.

Joins occur whenever any of the following conditions are met in a select query:

1. Two or more range variable declarations are listed in the FROM clause
2. The JOIN operator is used to extend an identification variable using a path expression
3. A path expression anywhere in the query navigates across an association field, to the same or a different entity

The semantics of a join between entities are the same as SQL joins between tables. Most queries contain a series of join conditions, expressions that define the rules for matching one entity to another. Join conditions may be specified explicitly, such as using the JOIN operator in the FROM clause of a query, or implicitly as a result of path navigation.

An *inner join* between two entities returns the objects from both entity types that satisfy all of the join conditions. Path navigation from one entity to another is a form of inner join. The *outer join* of two entities is the set of objects from both entity types that satisfy the join conditions plus the set of objects from one entity type (designated as the *left* entity) that have no matching join condition in the other.

In the absence of join conditions between two entities, queries will produce a Cartesian product. Each object of the first entity type is paired with each object of the second entity type, squaring the number of results. Cartesian products are rare with JPQL queries given the navigation capabilities of the language, but they are possible if two range variable declarations in the FROM clause are specified without additional conditions specified in the WHERE clause.

Further discussion and examples of each join style are provided in the following sections.

Inner Joins

All of the example queries so far have been using the simplest form of FROM clause, a single entity type aliased to an identification variable. However, as a relational language, JPQL supports queries that draw on multiple entities and the relationships between them.

Inner joins between two entities may be specified in one of two ways. The first and preferred form is the JOIN operator in the FROM clause. The second form requires multiple range variable declarations in the FROM clause and WHERE clause conditions to provide the join conditions.

The JOIN Operator and Collection Association Fields The syntax of an inner join using the JOIN operator is `[INNER] JOIN <path_expression> [AS] <identifier>`. Consider the following query:

```
SELECT p
FROM Employee e JOIN e.phones p
```

This query uses the JOIN operator to join the `Employee` entity to the `Phone` entity across the `phones` relationship. The join condition in this query is defined by the object-relational mapping of the `phones` relationship. No additional criteria need to be specified in order to link the two entities. By joining the two entities together, this query returns all of the `Phone` entity instances associated with employees in the company.

The syntax for joins is similar to the JOIN expressions supported by ANSI SQL. For readers who may not be familiar with this syntax, consider the equivalent SQL form of the previous query written using the traditional join form:

```
SELECT p.id, p.phone_num, p.type, p.emp_id
FROM emp e, phone p
WHERE e.id = p.emp_id
```

The table mapping for the `Phone` entity replaces the expression `e.phones`. The WHERE clause also includes the criteria necessary to join the two tables together across the join columns defined by the `phones` mapping.

Note that the `phones` relationship has been mapped to the identification variable `p`. Even though the `Phone` entity does not directly appear in the query, the target of the `phones` relationship is the `Phone` entity, and this determines the identification variable type. This implicit determination of the identification variable type can take some getting used to. Familiarity

with how relationships are defined in the object model is necessary to navigate through a written query.

Each occurrence of p outside of the FROM clause now refers to a single phone owned by an employee. Even though a collection association field was specified in the JOIN clause, the identification variable is really referring to entities reached by that association, not the collection itself. The variable can now be used as if the Phone entity was listed directly in the FROM clause. For example, instead of returning Phone entity instances, phone numbers can be returned instead:

```
SELECT p.number
FROM Employee e JOIN e.phones p
```

In the definition of path expressions earlier, it was noted that a path couldn't continue from a state field or collection association field. To work around this situation, the collection association field must be joined in the FROM clause so that a new identification variable is created for the path, allowing it to be the root for new path expressions.

IN VS. JOIN

The JOIN operator is new with the Java Persistence API. Previous versions of EJB QL defined by the EJB 2.0 and EJB 2.1 specifications used a special operator IN in the FROM clause to map collection associations to identification variables. The equivalent form of the query used earlier in this section may be specified as:

```
SELECT DISTINCT p
FROM Employee e, IN(e.phones) p
```

The IN operator is intended to indicate that the variable p is an enumeration of the phones collection. We believe that the JOIN operator is a more powerful and expressive way to declare relationships in a query. The IN operator is still supported, but use of the JOIN operator is recommended.

The JOIN Operator and Single-Valued Association Fields The JOIN operator works with both collection-valued association path expressions and single-valued association path expressions. Consider the following example:

```
SELECT d
FROM Employee e JOIN e.department d
```

This query defines a join from Employee to Department across the department relationship. This is semantically equivalent to using a path expression in the SELECT clause to obtain the department for the employee. For example, the following query should result in similar if not identical SQL representations involving a join between the Employee and Department entities:

```
SELECT e.department
FROM Employee e
```


The primary use case for using a single-valued association path expression in the FROM clause as opposed to just using a path expression in the SELECT clause is for outer joins. Path navigation is equivalent to the inner join of all associated entities traversed in the path expression.

Implicit inner joins resulting from path expressions is something that developers should be aware of. Consider the following example that returns the distinct departments based in California that are participating in the "Release1" project:

```
SELECT DISTINCT e.department
FROM Project p JOIN p.employees e
WHERE p.name = 'Release1' AND
      e.address.state = 'CA'
```

There are actually four logical joins here, not two. The translator will treat the query as if it had been written with explicit joins between the various entities. We will cover the syntax for multiple joins later, but for now consider the following query that is equivalent to the previous query, reading the join conditions from left to right:

```
SELECT DISTINCT d
FROM Project p JOIN p.employees e JOIN e.department d JOIN e.address a
WHERE p.name = 'Release1' AND
      a.state = 'CA'
```

We say four logical joins because the actual physical mapping may involve more tables. In this case the Employee and Project entities are related via a many-to-many association using a join table. Therefore the actual SQL for such a query uses five tables, not four:

```
SELECT DISTINCT d.id, d.name
FROM project p, emp_projects ep, emp e, dept d, address a
WHERE p.id = ep.project_id AND
      ep.emp_id = e.id AND
      e.dept_id = d.id AND
      e.address_id = a.id AND
      p.name = 'Release1' AND
      a.state = 'CA'
```

The first form of the query is certainly easier to read and understand. However, during performance tuning it may be helpful to understand how many joins can occur as the result of seemingly trivial path expressions.

Join Conditions in the WHERE Clause SQL queries have traditionally joined tables together by listing the tables to be joined in the FROM clause and supplying criteria in the WHERE clause of the query to determine the join conditions. To join two entities without using a relationship, use a range variable declaration for each entity in the FROM clause.

The previous join example between the Employee and Department entities could also have been written like this:

```
SELECT DISTINCT d
FROM Department d, Employee e
WHERE d = e.department
```

This style of query is usually used to compensate for the lack of an explicit relationship between two entities in the domain model. For example, there is no association between the `Department` entity and the `Employee` who is the manager of the department. We can use a join condition in the WHERE clause to make this possible:

```
SELECT d, m
FROM Department d, Employee m
WHERE d = m.department AND
      m.directs IS NOT EMPTY
```

In this example we are using one of the special collection expressions, IS NOT EMPTY, to check that the collection of direct reports to the employee is not empty. Any employee with a non-empty collection of directs is by definition a manager.

Multiple Joins More than one join may be cascaded if necessary. For example, the following query returns the distinct set of projects belonging to employees who belong to a department:

```
SELECT DISTINCT p
FROM Department d JOIN d.employees e JOIN e.projects p
```

The query processor interprets the FROM clause from left to right. Once a variable has been declared, it may be subsequently referenced by other JOIN expressions. In this case the `projects` relationship of the `Employee` entity is navigated once the employee variable has been declared.

Outer Joins

An outer join between two entities produces a domain where only one side of the relationship is required to be complete. In other words, the outer join of `Employee` to `Department` across the employee `department` relationship returns all employees and the department to which the employee has been assigned, but the department is returned only if it is available.

An outer join is specified using the following syntax: `LEFT [OUTER] JOIN <path_expression> [AS] <identifier>`. The following query demonstrates an outer join between two entities:

```
SELECT e, d
FROM Employee e LEFT JOIN e.department d
```

If the employee has not been assigned to a department, then the department object (the second element of the `Object` array) will be null. For readers familiar with Oracle SQL, the previous query would be equivalent to the following:

```
SELECT e.id, e.name, e.salary, e.manager_id, e.dept_id, e.address_id,
       d.id, d.name
FROM emp e, dept d
WHERE e.dept_id = d.id (+)
```

Fetch Joins

Fetch joins are intended to help application designers optimize their database access and prepare query results for detachment. They allow queries to specify one or more relationships that should be navigated and prefetched by the query engine so that they are not lazy loaded later at runtime.

For example, if we have an `Employee` entity with a lazy loading relationship to its address, the following query can be used to indicate that the relationship should be resolved eagerly during query execution:

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

Note that no identification variable is set for the `e.address` path expression. This is because even though the `Address` entity is being joined in order to resolve the relationship, it is not part of the result type of the query. The result of executing the query is still a collection of `Employee` entity instances, except that the `address` relationship on each entity will not cause a secondary trip to the database when it is accessed. This also allows the `address` relationship to be accessed safely if the `Employee` entity becomes detached. A fetch join is distinguished from a regular join by adding the FETCH keyword to the JOIN operator.

In order to implement fetch joins, the query is rewritten to turn the fetched association into a regular join of the appropriate type: inner by default or outer if the LEFT keyword was specified. The SELECT expression of the query is then expanded to include the joined relationship.

Consider the changes required to the previous example in order to implement the fetch join:

```
SELECT e, a
FROM Employee e JOIN e.address a
```

As the results are processed from this query, the query engine creates the `Address` entity in memory and assigns it to the `Employee` entity but then drops it from the result collection that it builds for the client. This eagerly loads the `address` relationship, which may then get accessed normally via the `Employee` entity.

A consequence of implementing fetch joins in this way is that fetching a collection association results in duplicate results. For example, consider a department query where the `employees` relationship of the `Department` entity is eagerly fetched. The fetch join query, this time using an outer join to ensure that departments without employees are retrieved, would be written as follows:

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
```

The actual query executed replaces the fetch with an outer join across the `employees` relationship:

```
SELECT d, e
FROM Department d LEFT JOIN d.employees e
```

Once again, as the results are processed the `Employee` entity is constructed in memory but dropped from the result collection. Each `Department` entity now has a fully resolved `employees` collection, but the client receives one reference to each department per employee. For example, if four departments with five employees each were retrieved, the result would be a collection of 20 `Department` instances, with each department duplicated five times. The actual entity instances all point back to the same managed versions, but the results are somewhat odd at the very least.

To eliminate the duplicate values, either the DISTINCT operator must be used or the results must be placed into a data structure such as a `Set`. Since it is not possible to write a SQL query that uses the DISTINCT operator while preserving the semantics of the fetch join, the provider will have to eliminate duplicates in memory after the results have been fetched. This could have performance implications for large result sets.

Given the somewhat peculiar results generated from a fetch join to a collection, it may not be the most appropriate way to eagerly load related entities in all cases. If a collection requires eager fetching on a regular basis, then it is worth considering making the relationship eager by default. Some persistence providers also offer batch reads as an alternative to fetch joins that issue multiple queries in a single batch and then correlate the results to eagerly load relationships.

The WHERE Clause

The WHERE clause of a query is used to specify filtering conditions to reduce the result set. In this section we will explore the features of the WHERE clause and the types of expressions that can be formed to filter query results.

The definition of the WHERE clause is deceptively simple. It is simply the keyword WHERE, followed by a conditional expression. However, as the following sections demonstrate, JPQL supports a powerful set of conditional expressions to filter the most sophisticated of queries.

Input Parameters

Input parameters for queries may be specified using either positional or named notation. Positional notation is defined by prefixing the variable number with a question mark. Consider the following query:

```
SELECT e
FROM Employee e
WHERE e.salary > ?1
```

Using the `Query` interface, any double value can be bound into the first parameter in order to indicate the lower bound for employee salaries in this query. The same positional parameter may occur more than once in the query. The value bound into the parameter will be substituted for each of its occurrences.

Named parameters are specified using a colon followed by an identifier. Here is the same query, this time using a named parameter:

```
SELECT e
FROM Employee e
WHERE e.salary > :sal
```

Input parameters were covered in detail in Chapter 6.

Basic Expression Form

Much of the conditional expression support in JPQL is borrowed directly from SQL. This is intentional and serves to ease the transition for developers already familiar with SQL. The key difference between conditional expressions in JPQL and SQL is that JPQL expressions can leverage identification variables and path expressions to navigate relationships during expression evaluation.

Conditional expressions are constructed in the same style as SQL conditional expressions, using a combination of logical operators, comparison expressions, primitive and function operations on fields, and so on. Although a summary of the operators is provided later, the grammar for conditional expressions is not repeated here. The Java Persistence API specification contains the grammar in Backus-Naur form (BNF) and is the place to look for the exact rules about using basic expressions. The following sections do, however, explain the higher-level operators and expressions, particularly those unique to JPQL, and they provide examples for each.

Literal syntax is also similar to SQL. Single quotes are used for string literals and escaped within a string by prefixing the quote with another single quote. Numeric expressions are defined according to the conventions of the Java programming language. Boolean values are represented by the literals TRUE and FALSE. There is no support in the query language for date literals.

Operator precedence is as follows:

1. Navigation operator (.)
2. Unary +/–
3. Multiplication (*) and division (/)
4. Addition (+) and subtraction (–)
5. Comparison operators: =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
6. Logical operators (AND, OR, NOT)

BETWEEN Expressions

The BETWEEN operator may be used in conditional expressions to determine whether or not the result of an expression falls within an inclusive range of values. Numeric, string, and date expressions may be evaluated in this way. Consider the following example:

```
SELECT e
FROM Employee e
WHERE e.salary BETWEEN 40000 AND 45000
```

Any employee making $40,000 to $45,000 inclusively is included in the results. This is identical to the following query using basic comparison operators:

```
SELECT e
FROM Employee e
WHERE e.salary >= 40000 AND e.salary <= 45000
```

The BETWEEN operator may also be negated with the NOT operator.

LIKE Expressions

JPQL supports the SQL LIKE condition to provide for a limited form of string pattern matching. Each LIKE expression consists of a string expression to be searched and a pattern string and optional escape sequence that defines the match conditions. The wildcard characters used by the pattern string are the underscore (_) for single character wildcards and the percent sign (%) for multicharacter wildcards.

```
SELECT d
FROM Department d
WHERE d.name LIKE '__Eng%'
```

Example department names to match this query would be "CAEngOtt" or "USEngCal", but not "CADocOtt". Note that pattern matches are case-sensitive.

If the pattern string contains an underscore or percent sign that should be literally matched, the ESCAPE clause may be used to specify a character that, when prefixing a wildcard character, indicates that it should be treated literally:

```
SELECT d
FROM Department d
WHERE d.name LIKE 'QA\_%' ESCAPE '\'
```

By escaping the underscore, it becomes a mandatory part of the expression. For example, "QA_East" would match, but "QANorth" would not.

Subqueries

Subqueries may be used in the WHERE and HAVING clauses of a query. A subquery is a complete select query inside a pair of parentheses that is embedded within a conditional expression. The results of executing the subquery (which will either be a scalar result or a collection of values) are then evaluated in the context of the conditional expression. Subqueries are a powerful technique for solving the most complex query scenarios.

Consider the following query:

```
SELECT e
FROM Employee e
WHERE e.salary = (SELECT MAX(e.salary)
                  FROM Employee e)
```

This query returns the employee with the highest salary from among all employees. A subquery consisting of an aggregate query (described later in this chapter) is used to return the maximum salary value, and then this result is used as the key to filter the employee list by salary. A subquery may be used in most conditional expressions and may appear on either the left or right side of an expression.

The scope of an identifier variable name begins in the query where it is defined and extends down into any subqueries. Identifiers in the main query may be referenced by a subquery, and identifiers introduced by a subquery may be referenced by any subquery that it creates. If a subquery declares an identifier variable of the same name, then it overrides the parent declaration and prevents the subquery from referring to the parent variable. In the previous example,

the declaration of the identification variable e in the subquery overrides the same declaration from the parent query.

■ **Note** Overriding an identification variable name in a subquery is not guaranteed to be supported by all providers. Unique names should be used to ensure portability.

The ability to refer to a variable from the main query in the subquery allows the two queries to be correlated. Consider the following example:

```
SELECT e
FROM Employee e
WHERE EXISTS (SELECT p
              FROM Phone p
              WHERE p.employee = e AND p.type = 'Cell')
```

This query returns all of the employees who have a cell phone number. This is also an example of a subquery that returns a collection of values. The EXISTS expression in this example returns true if any results are returned by the subquery. Note that the WHERE clause of the subquery references the identifier variable e from the main query and uses it to filter the subquery results. Conceptually, the subquery can be thought of as executing once for each employee. In practice, many database servers will optimize these types of queries into joins or inline views in order to maximize performance.

This query could also have been written using a join between the `Employee` and `Phone` entities with the DISTINCT operator used to filter the results. The advantage in using the correlated subquery is that the main query remains unburdened by joins to other entities. Quite often if a join is used only to filter the results, there is an equivalent subquery condition that may alternately be used in order to remove constraints on the join clause of the main query or even to improve query performance.

The FROM clause of a subquery may also create new identification variables out of path expressions using an identification variable from the main query. For example, the previous query could also have been written as follows:

```
SELECT e
FROM Employee e
WHERE EXISTS (SELECT p
              FROM e.phones p
              WHERE p.type = 'Cell')
```

In this version of the query, the subquery uses the collection association path `phones` from the `Employee` identification variable e in the subquery. This is then mapped to a local identification variable p that is used to filter the results by phone type. Each occurrence of p refers to a single phone associated with the employee.

To better illustrate how the translator handles this query, consider the equivalent query written in SQL:

```
SELECT e.id, e.name, e.salary, e.manager_id, e.dept_id, e.address_id
FROM emp e
WHERE EXISTS (SELECT 1
              FROM phone p
              WHERE p.emp_id = e.id AND
                    p.type = 'Cell')
```

The expression `e.phones` is converted to the table mapped by the `Phone` entity. The WHERE clause for the subquery then adds the necessary join condition to correlate the subquery to the primary query, in this case the expression `p.emp_id = e.id`. The join criteria applied to the `PHONE` table results in all of the phones owned by the related employee. Returning the literal `1` from the subquery is a standard practice with SQL EXISTS expressions because the actual columns selected by the subquery do not matter; only the number of rows is relevant. Because literals are not allowed in the SELECT clause, the entity must still be selected even though it will be ignored when the SQL is generated.

IN Expressions

The IN expression may be used to check whether a single-valued path expression is a member of a collection. The collection may be defined inline as a set of literal values or may be derived from a subquery. The following query demonstrates the literal notation by selecting all of the employees who live in New York or California:

```
SELECT e
FROM Employee e
WHERE e.address.state IN ('NY', 'CA')
```

The subquery form of the expression is similar, replacing the literal list with a nested query. The following query returns employees who work in departments that are contributing to projects beginning with the prefix "QA":

```
SELECT e
FROM Employee e
WHERE e.department IN (SELECT DISTINCT d
                       FROM Department d JOIN d.employees de JOIN de.projects p
                       WHERE p.name LIKE 'QA%')
```

The IN expression may also be negated using the NOT operator. For example, the following query returns all of the `Phone` entities with a phone number other than office or home:

```
SELECT p
FROM Phone p
WHERE p.type NOT IN ('Office', 'Home')
```

Collection Expressions

The IS EMPTY operator is the logical equivalent of IS NULL for collections. Queries may use the IS EMPTY operator or its negated form IS NOT EMPTY to check whether a collection association path resolves to an empty collection or has at least one value. For example, the following query returns all employees who are managers by virtue of having at least one direct report:


```
SELECT e
FROM Employee e
WHERE e.directs IS NOT EMPTY
```

Note that IS EMPTY expressions are translated to SQL as subquery expressions. The query translator may make use of an aggregate subquery or use the SQL EXISTS expression. Therefore the following query is equivalent to the previous one:

```
SELECT m
FROM Employee m
WHERE (SELECT COUNT(e)
       FROM Employee e
       WHERE e.manager = m) > 0
```

The MEMBER OF operator and its negated form NOT MEMBER OF are a shorthand way of checking whether an entity is a member of or not a member of a collection association path. The following query returns all managers who are incorrectly entered as reporting to themselves:

```
SELECT e
FROM Employee e
WHERE e MEMBER OF e.directs
```

A more typical use of the MEMBER OF operator is in conjunction with an input parameter. For example, the following query selects all employees who are assigned to a designated project:

```
SELECT e
FROM Employee e
WHERE :project MEMBER OF e.projects
```

Like the IS EMPTY expression, the MEMBER OF expression will be translated to SQL using either an EXISTS expression or the subquery form of the IN expression. The previous example is equivalent to the following query:

```
SELECT e
FROM Employee e
WHERE :project IN (SELECT p
                   FROM e.projects p)
```

EXISTS Expressions

The EXISTS condition returns true if a subquery returns any rows. Examples of EXISTS were demonstrated earlier in the introduction to subqueries. The EXISTS operator may also be negated with the NOT operator. The following query selects all employees who do not have a cell phone:

```
SELECT e
FROM Employee e
WHERE NOT EXISTS (SELECT p
                  FROM e.phones p
                  WHERE p.type = 'Cell')
```

ANY, ALL, and SOME Expressions

The ANY, ALL, and SOME operators may be used to compare an expression to the results of a subquery. Consider the following example:

```
SELECT e
FROM Employee e
WHERE e.directs IS NOT EMPTY AND
      e.salary < ALL (SELECT d.salary
                      FROM e.directs d)
```

This query returns all of the managers who are paid less than all of the employees who work for them. The subquery is evaluated, and then each value of the subquery is compared to the left-hand expression, in this case the manager salary. When the ALL operator is used, the comparison between the left side of the equation and all subquery results must be true for the overall condition to be true.

The ANY operator behaves similarly, but the overall condition is true so long as at least one of the comparisons between the expression and the subquery result are true. For example, if ANY were specified instead of ALL in the previous example, then the result of the query would be all of the managers who were paid less than at least one of their employees. The SOME operator is an alias for the ANY operator.

There is symmetry between IN expressions and the ANY operator. Consider the following variation of the project department example used previously:

```
SELECT e
FROM Employee e
WHERE e.department = ANY (SELECT DISTINCT d
                          FROM Department d JOIN d.employees de JOIN de.projects p
                          WHERE p.name LIKE 'QA%')
```

Function Expressions

Conditional expressions may leverage a number of functions that can be used to modify query results in the WHERE and HAVING clauses of a select query. Table 7-1 summarizes the syntax for each of the supported function expressions.

Table 7-1. *Supported Function Expressions*

Function	Description
ABS(*number*)	The ABS function returns the unsigned version of the *number* argument. The result type is the same as the argument type (integer, float, or double).
CONCAT(string1, string2)	The CONCAT function returns a new string that is the concatenation of its arguments, *string1* and *string2*.
CURRENT_DATE	The CURRENT_DATE function returns the current date as defined by the database server.
CURRENT_TIME	The CURRENT_TIME function returns the current time as defined by the database server.

Function	Description
CURRENT_TIMESTAMP	The CURRENT_TIMESTAMP function returns the current timestamp as defined by the database server.
LENGTH(*string*)	The LENGTH function returns the number of characters in the *string* argument.
LOCATE(*string1*, *string2* [, *start*])	The LOCATE function returns the position of *string2* in *string1*, optionally starting at the position indicated by *start*. The result is zero if the string cannot be found.
LOWER(*string*)	The LOWER function returns the lowercase form of the *string* argument.
MOD(*number1*, *number2*)	The MOD function returns the modulus of numeric arguments *number1* and *number2* as an integer.
SIZE(*collection*)	The SIZE function returns the number of elements in the collection, or zero if the collection is empty.
SQRT(*number*)	The SQRT function returns the square root of the *number* argument as a double.
SUBSTRING(*string*, *start*, *end*)	The SUBSTRING function returns a portion of the input *string*, starting at the index indicated by *start* up to *length* characters. String indexes are measured starting from one.
UPPER(*string*)	The UPPER function returns the uppercase form of the *string* argument.
TRIM([[LEADING|TRAILING|BOTH] [*char*] FROM] *string*)	The TRIM function removes leading and/or trailing characters from a string. If the optional LEADING, TRAILING, or BOTH keyword is not used, then both leading and trailing characters are removed. The default trim character is the space character.

The SIZE function requires special attention, as it is shorthand notation for an aggregate subquery. For example, consider the following query that returns all departments with only two employees:

```
SELECT d
FROM Department d
WHERE SIZE(d.employees) = 2
```

Similar to the collection expressions IS EMPTY and MEMBER OF, the SIZE function will be translated to SQL using a subquery. The equivalent form of the previous example using a subquery is as follows:

```
SELECT d
FROM Department d
WHERE (SELECT COUNT(e)
       FROM d.employees e) = 2
```

The ORDER BY Clause

Queries may optionally be sorted using one or more expressions comprised of identification variables, a path expression resolving to a single entity, or a path expression resolving to a persistent state field. The optional keywords ASC or DESC after the expression may be used to indicate ascending or descending sorts respectively. The default sort order is ascending.

The following example demonstrates sorting by a single field:

```
SELECT e
FROM Employee e
ORDER BY e.name DESC
```

Multiple expressions may also be used to refine the sort order:

```
SELECT e
FROM Employee e JOIN e.department d
ORDER BY d.name, e.name DESC
```

If the SELECT clause of the query uses state field path expressions, then the ORDER BY clause is limited to the same path expressions used in the SELECT clause. For example, the following query is not legal:

```
SELECT e.name
FROM Employee e
ORDER BY e.salary DESC
```

Because the result type of the query is the employee name, which is of type `String`, the remainder of the `Employee` state fields are no longer available for ordering.

Aggregate Queries

An aggregate query is a variation of a normal select query. An aggregate query groups results and applies aggregate functions to obtain summary information about query results. A query is considered an aggregate query if it uses an aggregate function or possesses a GROUP BY clause and/or a HAVING clause. The most typical form of aggregate query involves the use of one or more grouping expressions and aggregate functions in the SELECT clause paired with grouping expressions in the GROUP BY clause. The syntax of an aggregate query is as follows:

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[GROUP BY <group_by_clause>]
[HAVING <conditional_expression>]
[ORDER BY <order_by_clause>]
```

The SELECT, FROM, and WHERE clauses behave largely the same as previously described under select queries, with the exception of some restrictions on how the SELECT clause is formulated.

The power of an aggregate query comes from the use of aggregate functions over grouped data. Consider the following simple aggregate example:


```
SELECT AVG(e.salary)
FROM Employee e
```

This query returns the average salary of all employees in the company. AVG is an aggregate function that takes a numeric state field path expression as an argument and calculates the average over the group. Because there was no GROUP BY clause specified, the group here is the entire set of employees. This was the only form of aggregate query supported by EJB QL as defined in the EJB 2.1 specification.

Now consider this variation, where the result has been grouped by the department name:

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

This query returns the names of all departments and the average salary of the employees in that department. The `Department` entity is joined to the `Employee` entity across the `employees` relationship and then formed into a group defined by the department name. The AVG function then calculates its result based on the employee data in this group.

This can be extended further to filter the data so that manager salaries are not included:

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
WHERE e.directs IS EMPTY
GROUP BY d.name
```

Finally, we can extend this one last time to return only the departments where the average salary is greater than $50,000. Consider the following version of the previous query:

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
WHERE e.directs IS EMPTY
GROUP BY d.name
HAVING AVG(e.salary) > 50000
```

To better understand this query, let's go through the logical steps that took place to execute it. Databases use many techniques to optimize these types of queries, but conceptually the same process is being followed. First, the following non-grouping query is executed:

```
SELECT d.name, e.salary
FROM Department d JOIN d.employees e
WHERE e.directs IS EMPTY
```

This will produce a result set consisting of all department name and salary value pairs. The query engine then starts a new result set and makes a second pass over the data, collecting all of the salary values for each department name and handing them off to the AVG function. This function then returns the group average, which is then checked against the criteria from the HAVING clause. If the average value is greater than $50,000 then the query engine generates a result row consisting of the department name and average salary value.

The following sections describe the aggregate functions available for use in aggregate queries and the use of the GROUP BY and HAVING clauses.

Aggregate Functions

There are five aggregate functions that may be placed in the select clause of a query: AVG, COUNT, MAX, MIN, and SUM.

AVG

The AVG function takes a state field path expression as an argument and calculates the average value of that state field over the group. The state field type must be numeric, and the result is returned as a `Double`.

COUNT

The COUNT function takes either an identification variable or a path expression as its argument. This path expression may resolve to a state field or a single-valued association field. The result of the function is a `Long` value representing the number of values in the group. The argument to the COUNT function may optionally be preceded with the keyword DISTINCT, in which case duplicate values are eliminated before counting.

The following query counts the number of phones associated with each employee as well as the number of distinct number types (cell, office, home, and so on):

```
SELECT e, COUNT(p), COUNT(DISTINCT p.type)
FROM Employee e JOIN e.phones p
GROUP BY e
```

MAX

The MAX function takes a state field expression as an argument and returns the maximum value in the group for that state field.

MIN

The MIN function takes a state field expression as an argument and returns the minimum value in the group for that state field.

SUM

The SUM function takes a state field expression as an argument and calculates the sum of the values in that state field over the group. The state field type must be numeric, and the result type must correspond to the field type. For example, if a `Double` field is summed, then the result will be returned as a `Double`. If a `Long` field is summed, then the response will be returned as a `Long`.

The GROUP BY Clause

The GROUP BY clause defines the grouping expressions over which the results will be aggregated. A grouping expression must either be a single-valued path expression (state field or single-valued association field) or an identification variable. If an identification variable is used, the entity must not have any serialized state or large object fields.

The following query counts the number of employees in each department:

```
SELECT d.name, COUNT(e)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

Note that the same field expression used in the SELECT clause is repeated in the GROUP BY clause. All non-aggregate expressions must be listed this way. More than one aggregate function may be applied to the same GROUP BY clause:

```
SELECT d.name, COUNT(e), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

This variation of the query calculates the average salary of all employees in each department in addition to counting the number of employees in the department.

Multiple grouping expressions may also be used to further break down the results:

```
SELECT d.name, e.salary, COUNT(p)
FROM Department d JOIN d.employees e JOIN e.projects p
GROUP BY d.name, e.salary
```

Because there are two grouping expressions, the department name and employee salary must be listed in both the SELECT clause and GROUP BY clause. For each department, this query counts the number of projects assigned to employees based on their salary.

In the absence of a GROUP BY clause, the entire query is treated as one group, and the SELECT list may contain only aggregate functions. For example, the following query returns the number of employees and their average salary across the entire company:

```
SELECT COUNT(e), AVG(e.salary)
FROM Employee e
```

The HAVING Clause

The HAVING clause defines a filter to be applied after the query results have been grouped. It is effectively a secondary WHERE clause, and its definition is the same, the keyword HAVING followed by a conditional expression. The key difference with the HAVING clause is that its conditional expressions are limited to state fields or single-valued association fields previously identified in the GROUP BY clause.

Conditional expressions in the HAVING clause may also make use of aggregate functions. In many respects, the primary use of the HAVING clause is to restrict the results based on the aggregate result values. The following query uses this technique to retrieve all employees assigned to two or more projects:

```
SELECT e, COUNT(p)
FROM Employee e JOIN e.projects p
GROUP BY e
HAVING COUNT(p) >= 2
```

Update Queries

Update queries are a new feature in the Java Persistence API. They provide an equivalent to the SQL UPDATE statement but with JPQL conditional expressions. The form of an update query is:

```
UPDATE <entity name> [[AS] <identification variable>]
SET <update_statement> {, <update_statement>}*
[WHERE <conditional_expression>]
```

Each UPDATE statement consists of a single-valued path expression, assignment operator (=), and an expression. Expression choices for the assignment statement are slightly restricted compared to regular conditional expressions. The right side of the assignment must resolve to a literal, simple expression resolving to a basic type, function expression, identification variable, or input parameter. The result type of that expression must be compatible with the simple association path or persistent state field on the left side of the assignment.

The following simple example demonstrates the update query by giving employees who make $55,000 a year a raise to $60,000:

```
UPDATE Employee e
SET e.salary = 60000
WHERE e.salary = 55000
```

The WHERE clause of an UPDATE statement functions the same as a SELECT statement and may use the identification variable defined in the UPDATE clause in expressions. A slightly more complex but more realistic update query would be to award a $5,000 raise to employees who worked on a particular project:

```
UPDATE Employee e
SET e.salary = e.salary + 5000
WHERE EXISTS (SELECT p
              FROM e.projects p
              WHERE p.name = 'Release2')
```

More than one property of the target entity may be modified with a single UPDATE statement. For example, the following query updates the phone exchange for employees in the city of Ottawa and changes the terminology of the phone type from “Office” to “Business”:

```
UPDATE Phone p
SET p.number = CONCAT('288', SUBSTRING(p.number, LOCATE(p.number, '-'), 4)),
    p.type = 'Business'
WHERE p.employee.address.city = 'Ottawa' AND
      p.type = 'Office'
```

Delete Queries

Like the update query, the delete query is a new feature in the Java Persistence API. It provides equivalent capability as the SQL DELETE statement but with JPQL conditional expressions. The form of a delete query is:


```
DELETE FROM <entity name> [[AS] <identification variable>]
[WHERE <condition>]
```

The following example demonstrates removes all employees who are not assigned to a department:

```
DELETE FROM Employee e
WHERE e.department IS NULL
```

The WHERE clause for a DELETE statement functions the same as it would for a SELECT statement. All conditional expressions are available to filter the set of entities to be removed. If the WHERE clause is not provided, all entities of the given type are removed.

Delete queries are polymorphic. Any entity subclass instances that meet the criteria of the delete query will also be deleted. Delete queries do not honor cascade rules, however. No entities other than the type referenced in the query and its subclasses will be removed, even if the entity has relationships to other entities with cascade removes enabled.

Summary

In this chapter we have taken a complete tour of the Java Persistence Query Language, looking at the numerous query types and the syntax for each. We covered the history of the language, from its roots in the EJB 2.0 specification to the major enhancements introduced by the Java Persistence API.

In the section on select queries, we explored each query clause and incrementally built up more complex queries as the full syntax was described. We discussed identification variables and path expressions, which are used to navigate through the domain model in query expressions. We also looked at the various conditional expressions supported by the language.

In our discussion of aggregate queries we introduced the additional grouping and filtering clauses that extend select queries. We also demonstrated the various aggregate functions.

In the sections on update and delete queries, we described the full syntax for bulk update and delete statements, the runtime behavior of which was described in the previous chapter.

In the next chapter we switch back to object-relational mapping and cover advanced concepts such as inheritance, composite primary keys and associations, and multiple table mappings.

CHAPTER 8

Advanced Object-Relational Mapping

Every application is different, and while most have some elements of complexity in them, the difficult parts in one application will tend to be different than those in other types of applications. Chances are that whichever application you are working on at any given time will need to make use of at least one advanced feature of the API. This chapter will introduce and explain some of these more advanced ORM features.

As it turns out, an entity is just one of three different types of persistable classes that can be used with the Java Persistence API. The provider must also manage classes called mapped superclasses and embeddable classes. Collectively we refer to these three types as *managed classes* because they are the classes that must be managed by the provider in order for them to be made persistent.

Some of the features in this chapter are targeted at applications that need to reconcile the differences between an existing data model and an object model. For example, when the data in an entity table would be better decomposed in the object model as an entity and a dependent sub-object that is referenced by the entity, then the mapping infrastructure should be able to support that. Likewise, when the entity data is spread across multiple tables, the mapping layer should allow for this kind of configuration to be specified.

There has been no shortage of discussion in this book about how entities in the Java Persistence API are just regular Java classes and not the heavy persistent objects that were generated by EJB 2.1 entity bean compilers. One of the benefits of entities being regular Java classes is that they can adhere to already established concepts and practices that exist in object-oriented systems. One of the traditional object-oriented innovations is the use of inheritance and creating objects in a hierarchy in order to inherit state and behavior.

This chapter will discuss some of the more advanced mapping features and delve into some of the diverse possibilities offered by the API and the mapping layer. We will see how inheritance works within the framework of the Java Persistence API and how inheritance affects the model.

Embedded Objects

An *embedded object* is one that is dependent upon an entity for its identity. It has no identity of its own but is merely part of the entity state that has been carved off and stored in a separate Java object hanging off of the entity. In Java, embedded objects appear similar to relationships

in that they are referenced by an entity and appear in the Java sense to be the target of an association. In the database, however, the state of the embedded object is stored with the rest of the entity state in the database row, with no distinction between the state in the Java entity and that in its embedded object.

If the database row contains all of the data for both the entity and its embedded object, why have such an object anyway? Why not just define the fields of the entity to reference all of its persistence state instead of splitting it up into one or more sub-objects that are second-class persistent objects dependent upon the entity for their existence?

This brings us back to the object-relational impedance mismatch. Since the database record contains more than one logical type, it makes sense to make that relationship explicit in the object model of the application even though the physical representation is different. You could almost say that the embedded object is a more natural representation of the domain concept than a simple collection of attributes on the entity. Furthermore, once we have identified a grouping of entity state that makes up an embedded object, we can share the same embedded object type with any other entity that also has the same internal representation.[1]

An example of such reuse might be address information. Figure 8-1 shows an `EMPLOYEE` table that contains a mixture of basic employee information as well as columns that correspond to the home address of the employee.

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	STATE
	ZIP_CODE

Figure 8-1. `EMPLOYEE` *table with embedded address information*

The `STREET`, `CITY`, `STATE`, and `ZIP_CODE` columns combine logically to form the address. In the object model this is an excellent candidate to be abstracted into a separate `Address` embedded type instead of listing each attribute on the entity class itself. The entity class would then simply have an `address` attribute pointing to an embedded object of type `Address`. Figure 8-2 shows the relationship between `Employee` and `Address`. The UML composition association is used to denote that the `Employee` wholly owns the `Address` and that an instance of `Address` may not be shared by any other object other than the `Employee` instance that owns it.

With this representation, not only is the address information neatly encapsulated within an object, but if another entity such as `Company` also has address information, then it can also have an attribute that points to its own embedded `Address` object. We will describe this scenario in the next section.

1. Even though embedded types can be shared or reused, the instances cannot. An embedded object instance belongs to the entity that references it, and no other entity instance, of that entity type or any other, may reference the same embedded instance.

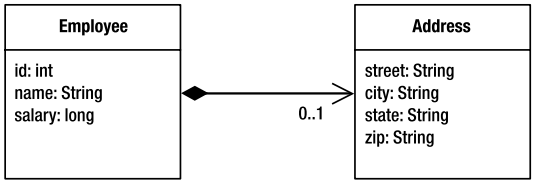

Figure 8-2. `Employee` *and* `Address` *relationship*

An embedded type is marked as such by adding the `@Embeddable` annotation to the class definition. This annotation serves to distinguish the class from other regular Java types. Once a class has been designated as embeddable, then its fields and properties will be persistable as part of an entity. Basic column mappings such as `@Basic`, `@Temporal`, `@Enumerated`, `@Lob`, and `@Column` may be added to the attributes of the embedded class, but it may not contain any relationships or other advanced mappings. Listing 8-1 shows the definition of the `Address` embedded type.

Listing 8-1. *Embeddable Address Type*

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
    // ...
}
```

To use this class in an entity, the entity must have an attribute of the same type annotated with the `@Embedded` annotation. Listing 8-2 shows the `Employee` class using an embedded `Address` object.

Listing 8-2. *Using an Embedded Object*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded private Address address;
    // ...
}
```

When the provider persists an instance of `Employee`, it will access the attributes of the `Address` object just as if they were present on the entity instance itself. Column mappings on the `Address` type really pertain to columns on the `EMPLOYEE` table, even though they are listed in a different type.

Support for an entity having a collection of embedded objects, an embedded object referencing other embedded objects, or an embedded object having relationships to entities is not in the current version of the Java Persistence API. It is also not portable to use embedded objects as part of inheritance hierarchies. These are features that some persistence providers support and that may be in future versions of the API.

The decision to use embedded objects or entities depends upon whether you think you will ever need to create relationships to them or from them. Embedded objects are not meant to be entities, and as soon as you start to treat them as entities then you should probably make them first-class entities instead of embedded objects if the data model permits it.

Sharing Embedded Object Classes

Before we got to our example we mentioned that an `Address` class could be reused in both `Employee` and `Company` entities. Ideally we would like the representation shown in Figure 8-3. Even though both the `Employee` and `Company` classes compose the `Address` class, this is not a problem, because each instance of `Address` will be used by only a single `Employee` or `Company` instance.

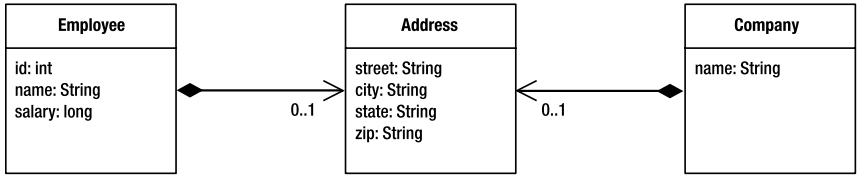

Figure 8-3. *Address shared by two entities*

Given that the column mappings of the `Address` embedded type apply to the columns of the containing entity, you might be wondering how sharing could be possible in the case where the two entity tables may have different column names for the same fields. Figure 8-4 demonstrates this problem. The `COMPANY` table matches the default and mapped attributes of the `Address` type we defined earlier, but the `EMPLOYEE` table in this example has been changed to match the address requirements of a person living in Canada. We need a way for an entity to map the embedded object according to its own entity table needs, and we have one in the `@AttributeOverride` annotation.

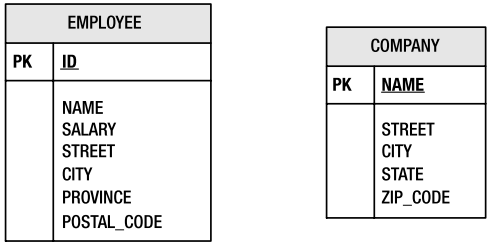

Figure 8-4. `EMPLOYEE` *and* `COMPANY` *tables*

We use an `@AttributeOverride` annotation for each attribute of the embedded object that we want to override in the entity. We annotate the embedded field or property in the entity and specify in the `name` element the field or property in the embedded object that we are overriding. The `column` element allows us to specify the column that the attribute is being mapped to in the entity table. We indicate this in the form of a nested `@Column` annotation. If we are overriding multiple fields or properties, then we can use the plural `@AttributeOverrides` annotation and nest multiple `@AttributeOverride` annotations inside of it.

In Listing 8-3 is an example of using `Address` in both `Employee` and `Company`. The `Company` entity uses the `Address` type without change, but the `Employee` entity specifies two attribute overrides to map the `state` and `zip` attributes of the `Address` to the `PROVINCE` and `POSTAL_CODE` columns of the `EMPLOYEE` table.

Listing 8-3. *Reusing an Embedded Object in Multiple Entities*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
    })
    private Address address;
    // ...
}
@Entity
public class Company {
    @Id private String name;
    @Embedded
    private Address address;
    // ...
    }
```

Compound Primary Keys

In some cases an entity needs to have a primary key or identifier that is composed of multiple fields, or from the database perspective the primary key in its table is made up of multiple columns. This is more common for legacy databases and also occurs when a primary key is composed of a relationship, a topic that we will discuss later in this chapter.

We have two options available to us for having compound primary keys in our entity, depending on how we want to structure our entity class. Both of them require that we use a separate class containing the primary key fields called a *primary key class*; the difference between the two options is determined by what the entity class contains.

Primary key classes must include method definitions for `equals()` and `hashCode()` in order to be able to be stored and keyed upon by the persistence provider, and their fields or

properties must be in the set of valid identifier types listed in the previous chapter. They must also be public, implement `Serializable`, and have a no-arg constructor.

As an example of a compound primary key, we will look at the `Employee` entity again, only this time the employee number is specific to the country where he or she works. Two employees in different countries can have the same employee number, but only one can be used within any given country. Figure 8-5 shows the `EMPLOYEE` table structured with a compound primary key to capture this requirement. Given this table definition, we will now look at how to map the `Employee` entity using the two different styles of primary key class.

Figure 8-5. `EMPLOYEE` *table with a compound primary key*

Id Class

The first and most basic type of primary key class is an *id class*. Each field of the entity that makes up the primary key is marked with the `@Id` annotation. The primary key class is defined separately and associated with the entity by using the `@IdClass` annotation on the entity class definition. Listing 8-4 demonstrates an entity with a compound primary key that uses an id class.

Listing 8-4. *Using an Id Class*

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    private long salary;
    // ...
}
```

The primary key class must contain fields or properties that match the primary key attributes in the entity in both name and type. Listing 8-5 shows the `EmployeeId` primary key class. It has two fields, one to represent the country and one to represent the employee number. We have also supplied `equals()` and `hashCode()` methods to allow the class to be used in sorting and hashing operations.

Listing 8-5. *The* `EmployeeId` *Id Class*

```
public class EmployeeId implements Serializable {
    private String country;
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country, int id) {
      this.country = country;
      this.id = id;
    }

    public String getCountry() { return country; }
    public int getId() { return id; }

    public boolean equals(Object o) {
        return ((o instanceof EmployeeId) &&
                country.equals(((EmployeeId)o).getCountry()) &&
                id == ((EmployeeId)o).getId());
    }

    public int hashCode() {
        return country.hashCode() + id;
    }
}
```

Note that there are no setter methods on the `EmployeeId` class. Once it has been constructed using the primary key values, it can't be changed. We do this to enforce the notion that a primary key value cannot be changed, even when it is made up of multiple fields. Because the `@Id` annotation was placed on the fields of the entity, the provider will also use field access when it needs to work with the primary key class.

The id class is useful as a structured object that encapsulates all of the primary key information. For example, when doing a query based upon the primary key, such as the `find()` method of the `EntityManager` interface, an instance of the id class can be used as an argument instead of some unstructured and unordered collection of primary key data. Listing 8-6 shows the definition of a method to search for an `Employee` instance given the name of the country and the employee number. A new instance of the `EmployeeId` class is constructed using the method arguments and then used as the argument to the `find()` method.

Listing 8-6. *Invoking a Primary Key Query on an Entity with an Id Class*

```
public Employee findEmployee(String country, int id) {
    return em.find(Employee.class, new EmployeeId(country, id));
}
```

■ **Tip** Because the argument to `find()` is of type `Object`, vendors may support passing in simple arrays or collections of primary key information. Passing arguments that are not primary key classes is nonportable.

Embedded Id Class

An entity that contains a single field of the same type as the primary key class is said to use an *embedded id class*. The embedded id class is just an embedded object that happens to be composed of the primary key components. Instead of annotating the embedded id class with an `@Embedded` annotation, though, we use an `@EmbeddedId` annotation to indicate that it is not just a regular embedded object but also a primary key class. When we use this approach there are no `@Id` annotations on the class, nor is the `@IdClass` annotation used. You can think of `@EmbeddedId` as the logical equivalent to putting both `@Id` and `@Embedded` on the field.

Like other embedded objects, the embedded id class must be annotated with `@Embeddable`, and access type must also match the access type of the entity that uses it. If the entity annotates its fields, then the embedded id class should also annotate its fields if such annotations are required. Listing 8-7 shows the `EmployeeId` class again, this time as an embeddable primary key class. The getter methods, `equals()` and `hashCode()` implementations, are the same as the previous version from Listing 8-5.

Listing 8-7. *Embeddable Primary Key Class*

```
@Embeddable
public class EmployeeId {
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country, int id) {
        this.country = country;
        this.id = id;
    }

    // ...
}
```

Using the embedded primary key class is no different than using a regular embedded type. Listing 8-8 shows the `Employee` entity adjusted to use the embedded version of the `EmployeeId` class. Note that since the column mappings are present on the embedded type, we do not specify the mapping for `EMP_ID` as was done in the case of the id class. If the embedded primary key class is used by more than one entity, then the `@AttributeOverride` annotation can be used to customize mappings just as you would for a regular embedded type. To return the `country` and `id` attributes of the primary key from getter methods, we must delegate to the embedded id object to obtain the values.

Listing 8-8. *Using an Embedded Id Class*

```
@Entity
public class Employee {
    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(String country, int id) {
        this.id = new EmployeeId(country, id);
    }

    public String getCountry() { return id.getCountry(); }
    public int getId() { return id.getId(); }
    // ...
}
```

We can create an instance of `EmployeeId` and pass it to the `find()` method just as we did for the id class example, but if we want to create the same query using JPQL and reference the primary key, then we have to traverse the embedded id class explicitly. Listing 8-9 shows this technique. Even though `id` is not a relationship, we still traverse it using the dot notation in order to access the members of the embedded class.

Listing 8-9. *Referencing an Embedded Id Class in a Query*

```
public Employee findEmployee(String country, int id) {
    return (Employee)
        em.createQuery("SELECT e " +
                       "FROM Employee e " +
                       "WHERE e.id.country = ?1 AND e.id.id = ?2")
          .setParameter(1, country)
          .setParameter(2, id)
          .getSingleResult();
}
```

Advanced Mapping Elements

Various other metadata may be specified on the `@Column` and `@JoinColumn` annotations, some of which applies to schema generation that will be discussed in Chapter 9. Other parts we can describe separately as applying to columns and join columns in the following sections.

Read-Only Mappings

The Java Persistence API does not really define any kind of read-only entity, although it will likely show up in a future release. The API does, however, define options to set individual mappings to be read-only using the `insertable` and `updatable` elements of the `@Column` and `@JoinColumn` annotations. These two settings default to true but may be set to false if we want to ensure that

the provider will not insert or update information in the table in response to changes in the entity instance. If the data in the mapped table already exists and we want to ensure that it will not be modified at runtime, then the `insertable` and `updatable` elements can be set to false, effectively preventing the provider from doing anything other than reading the entity from the database. Listing 8-10 demonstrates the `Employee` entity with read-only mappings.

Listing 8-10. *Making Entity Mappings Read-Only*

```
@Entity
public class Employee {
    @Id private int id;
    @Column(insertable=false, updatable=false)
    private String name;
    @Column(insertable=false, updatable=false)
    private long salary;

    @ManyToOne
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

We don't need to worry about the identifier mapping being modified, because it is illegal to modify identifiers. The other mappings, though, are marked as not being able to be inserted or updated, so we are assuming that there are already entities in the database to be read in and used. No new entities will be persisted, and existing entities will never be updated.

Note that this does not guarantee that the entity state will not change in memory. `Employee` instances could still get changed either inside or outside a transaction, but at transaction commit time or whenever the entities get flushed to the database, this state will not be saved. Be careful modifying read-only mappings in memory, however, as changing the entities may cause them to become inconsistent with the state in the database and could wreak havoc on any vendor-specific cache.

Even though all of these mappings are not updatable, the entity as a whole could still be deleted. A proper read-only feature will solve this problem once and for all in a future release.

Optionality

As we will see in Chapter 9 when we talk about schema generation, there exists metadata that either permits the database columns to be null or requires them to have values. While this setting will affect the physical database schema, there are also settings on some of the logical mappings that allow a basic mapping or a single-valued association mapping to be left empty or required to be specified in the object model. The element that requires or permits such behavior is the `optional` element in the `@Basic`, `@ManyToOne`, and `@OneToOne` annotations.

When the `optional` element is specified as false, it indicates to the provider that the field or property mapping may not be null. The API does not actually define what the behavior is in the case when it is, but the provider may choose to throw an exception or simply do something else. For basic mappings, it is only a hint and may be completely ignored. The `optional` element

may also be used by the provider when doing schema generation, since if optional is set to true, then the column in the database must also be nullable.

Because the API does not go into any detail about ordinality of the object model, there is a certain amount of nonportability associated with using it. An example of setting the manager to be a required attribute is shown in Listing 8-11. The default value for `optional` is true, making it necessary to be specified only if a false value is needed.

Listing 8-11. *Using Optional Mappings*

```
@Entity
public class Employee {
    // ...
    @ManyToOne(optional=false)
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

Advanced Relationships

If your object model is able to dictate your physical schema, then it is likely that you will not need to use many of the advanced relationship features that are offered by the API. The flexibility of being able to define a data model usually makes for a less demanding mapping configuration. It is when the tables are already in place that an object model must work around the data schema and go beyond the rudimentary relationship mappings that we have been using thus far. The following sections describe a few of the more common relationship issues that you may encounter.

Compound Join Columns

Now that we know how to create entities with compound primary keys it is not a far stretch to figure out that as soon as we have a relationship to an entity with a compound identifier, we will need some way to extend the way we currently reference it.

Up to this point we have dealt with the physical relationship mapping only as a join column, but if the primary key that we are referencing is composed of multiple fields, then we will need multiple join columns. This is why we have the plural `@JoinColumns` annotation that can hold as many join columns as we need to put into it.

There are no default values for join column names when we have multiple join columns. The simplest answer is to simply require that the user assign them, so when multiple join columns are used, both the `name` element and the `referencedColumnName` element, which indicates the name of the primary key column in the target table, must be specified.

Now that we are getting into more complex scenarios, let's add a more interesting relationship to the mix. Let's say that employees have managers and that each manager has a number of employees that work for him or her. You may not find that very interesting until you realize that managers are themselves employees, so the join columns are actually self-referential, that is, referring to the same table they are stored in. Figure 8-6 shows the `EMPLOYEE` table with this relationship.

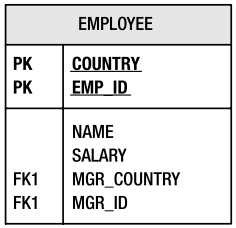

Figure 8-6. EMPLOYEE *table with self-referencing compound foreign key*

Listing 8-12 shows a version of the Employee entity that has a manager relationship, which is many-to-one from each of the managed employees to the manager, and a one-to-many directs relationship from the manager to its managed employees.

Listing 8-12. *Self-Referencing Compound Relationships*

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="MGR_COUNTRY", referencedColumnName="COUNTRY"),
        @JoinColumn(name="MGR_ID", referencedColumnName="EMP_ID")
    })
    private Employee manager;

    @OneToMany(mappedBy="manager")
    private Collection<Employee> directs;
    // ...
}
```

Any number of join columns can be specified, although in practice very seldom are there more than two. The plural form of @JoinColumns may be used on many-to-one or one-to-one relationships or more generally whenever the single @JoinColumn annotation is valid.

Another example to consider is in the join table of a many-to-many relationship. We can revisit the Employee and Project relationship described in Chapter 4 to take into account our compound primary key in Employee. The new table structure for this relationship is shown in Figure 8-7.

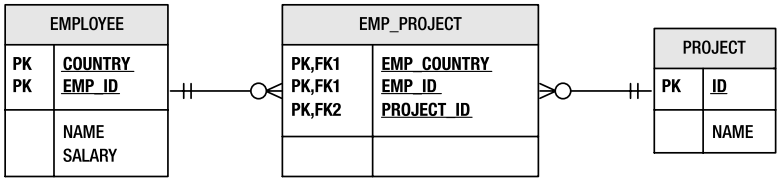

Figure 8-7. *Join table with a compound primary key*

If we keep the Employee entity as the owner, where the join table is defined, then the mapping for this relationship will be as shown in Listing 8-13.

Listing 8-13. *Join Table with Compound Join Columns*

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;

    @ManyToMany
    @JoinTable(
        name="EMP_PROJECT",
        joinColumns={
            @JoinColumn(name="EMP_COUNTRY", referencedColumnName="COUNTRY"),
            @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")},
        inverseJoinColumns=@JoinColumn(name="PROJECT_ID"))
    private Collection<Project> projects;
    // ...
}
```

Identifiers That Include a Relationship

It is possible for a compound primary key to actually include a relationship, which implies that the object cannot exist without participating in the relationship that is part of its identifier. Being that primary key fields may not be changed once they have been set, a relationship that is part of a primary key is likewise immutable. Such a relationship must be set in order for the entity to be created and must not be changed once the entity exists.

To demonstrate, let's reconsider our Project entity. Instead of having a unique numeric identifier, it will now consist of a name and a reference to the Department entity. Multiple projects of the same name may exist, but only one name can be used with a given department. Figure 8-8 shows the data model for this relationship.

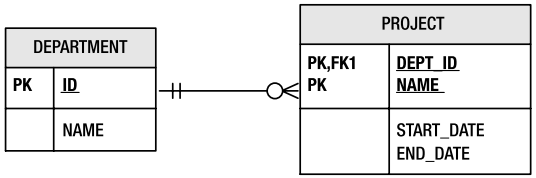

Figure 8-8. *Compound primary key with a foreign key dependency*

We first need to create a primary key class that will store the compound primary key. This primary key class will not contain the relationship mapping. Instead, it will just contain the basic mappings for the actual primary key columns. Listing 8-14 shows the `ProjectId` id class.

Listing 8-14. *The* `ProjectId` *Id Class*

```
public class ProjectId implements Serializable {
    private int deptId;
    private String name;

    public ProjectId() {}
    public ProjectId(int deptId, String name) {
        this.deptId = deptId;
        this.name = name;
    }
    // ...
}
```

Now we can update the `Project` entity to reference the `ProjectId` id class and declare the compound primary key. Listing 8-15 shows the `Project` entity. Note that we are placing the `@Id` annotations on the basic attributes, not on the relationship. We have introduced an `@ManyToOne` relationship to the `Department` entity that maps to the same `DEPT_ID` column as the `deptId` basic mapping. Because the basic mapping has been changed so that it is not insertable or updatable, it will be ignored when the entity is persisted. Instead, the foreign key mapped by the department relationship will be inserted into the `DEPT_ID` column. Structuring it in this way prevents the two mappings from colliding with each other when the provider writes entity changes to the database.

Listing 8-15. *Primary Key That Includes a Relationship*

```
@Entity
@IdClass(ProjectId.class)
public class Project {
    @Id
    @Column(name="DEPT_ID", insertable=false, updatable=false)
    private int deptId;
    @Id private String name;
```



```
    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;

    @Temporal(TemporalType.DATE)
    @Column(name="START_DATE")
    private Date startDate;
    @Temporal(TemporalType.DATE)
    @Column(name="END_DATE")
    private Date endDate;
    // ...
}
```

■ **Tip** Support for primary keys that include relationships or foreign keys is not explicitly mentioned in the current version of the Java Persistence API even though the specifics of doing so are not disallowed. It may be that some vendors will not support the duplicate mapping case that is described in this section or that a slightly different mapping practice from the one described here is required.

Mapping Relationship State

There are times when a relationship actually has state associated with it. For example, let's say that we want to maintain the date an employee was assigned to work on a project. Storing the state on the employee is possible but less helpful, since the date is really coupled to the employee's relationship to a particular project (a single entry in the many-to-many association). Taking an employee off of a project should really just cause the assignment date to go away, so storing it as part of the employee means that we have to ensure that the two are consistent with each other, which is kind of bothersome. In UML, we would show this kind of relationship using an association class. Figure 8-9 shows an example of this technique.

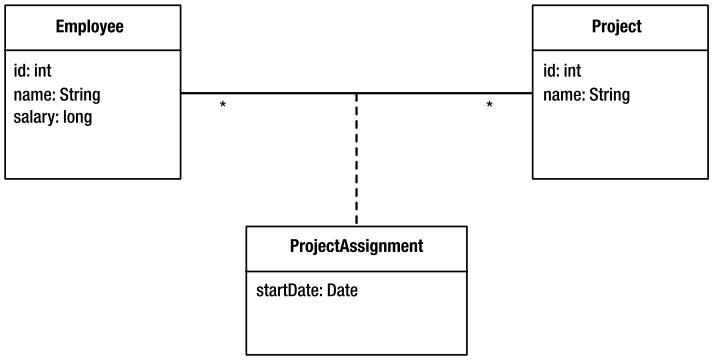

Figure 8-9. *Modeling state on a relationship using an association class*

In the database everything is rosy, since we can simply add a column to the join table. The data model provides natural support for relationship state. Figure 8-10 shows the many-to-many relationship between EMPLOYEE and PROJECT with an expanded join table.

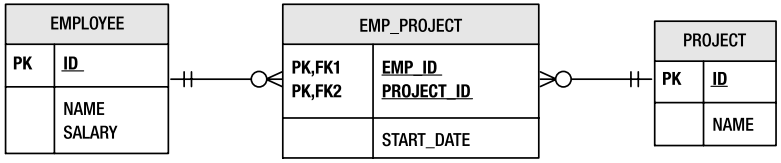

Figure 8-10. *Join table with additional state*

When we get to the object model, however, it becomes much more problematic. The issue is that Java has no inherent support for relationship state. Relationships are just object references or pointers, hence no state can ever exist on them. State exists on objects only, and relationships are not first-class objects.

The Java solution is to turn the relationship into an entity that contains the desired state and map the new entity to what was previously the join table. The new entity will have a many-to-one relationship to each of the existing entity types, and each of the entity types will have a one-to-many relationship back to the new entity representing the relationship. The primary key of the new entity will be the combination of the two relationships to the two entity types. Listing 8-16 shows all of the participants in the Employee and Project relationship.

Listing 8-16. *Mapping Relationship State with an Intermediate Entity*

```
@Entity
public class Employee {
    @Id private int id;
    // ...
    @OneToMany(mappedBy="employee")
    private Collection<ProjectAssignment> assignments;
    // ...
}

@Entity
public class Project {
    @Id private int id;
    // ...
    @OneToMany(mappedBy="project")
    private Collection<ProjectAssignment> assignments;
    // ...
}

@Entity
@Table(name="EMP_PROJECT")
@IdClass(ProjectAssignmentId.class)
```



```
public class ProjectAssignment {
    @Id
    @Column(name="EMP_ID", insertable=false, updatable=false)
    private int empId;
    @Id
    @Column(name="PROJECT_ID", insertable=false, updatable=false)
    private int projectId;

    @ManyToOne
    @JoinColumn(name="EMP_ID")
    private Employee employee;
    @ManyToOne
    @JoinColumn(name="PROJECT_ID")
    private Project project;

    @Temporal(TemporalType.DATE)
    @Column(name="START_DATE", updatable=false)
    private Date startDate;
    // ...
}

public class ProjectAssignmentId implements Serializable {
    private int empId;
    private int projectId;
    // ...
}
```

Here we have the two foreign key columns making up the primary key in the EMP_PROJECT join table, so in reality the primary key is actually entirely composed of relationships. The date at which the assignment was made could be manually set when the assignment is created, or it could be associated with a trigger that causes it to be set when the assignment is created in the database. Note that if a trigger were used, then the entity would need to be refreshed from the database in order to populate the assignment date field in the Java object.

Multiple Tables

The most common mapping scenarios are of the so-called *meet-in-the-middle* variety. This means that the data model and the object model already exist, or if one does not exist, then it is not created based on requirements from the other model. This is relevant because there are a number of features in the Java Persistence API that attempt to address concerns that arise in this case.

Up to this point we have assumed that an entity gets mapped to a single table and that a single row in that table represents an entity. In an existing or legacy data model, it was actually quite common to spread data, even data that was tightly coupled, across multiple tables. This was done for different administrative as well as performance reasons, one of which was to decrease table contention when specific subsets of the data were accessed or modified.

To account for this, entities may be mapped across multiple tables by making use of the `@SecondaryTable` annotation and its plural `@SecondaryTables` form. We call the default table or the table defined by the `@Table` annotation the *primary table* and any additional ones *secondary tables*. We can then distribute the data in an entity across rows in both the primary table and the secondary tables simply by defining the secondary tables as annotations on the entity and then specifying when we map each field or property which table the column is in. We do this by specifying the name of the table in the `table` element in `@Column` or `@JoinColumn`. We did not need to use this element earlier, because the default value of `table` is the name of the primary table.

The only bit that is left is to specify how to join the secondary table or tables to the primary table. We saw in Chapter 4 how the primary key join column is a special case of a join column where the join column is just the primary key column (or columns in the case of composite primary keys). Support for joining secondary tables to the primary table is limited to primary key join columns and is specified as an `@PrimaryKeyJoinColumn` annotation as part of the `@SecondaryTable` annotation.

To demonstrate the use of a secondary table, consider the data model shown in Figure 8-11. There is a primary key relationship between the `EMP` and `EMP_ADDRESS` tables. The `EMP` table stores the primary employee information, while address information has been moved to the `EMP_ADDRESS` table.

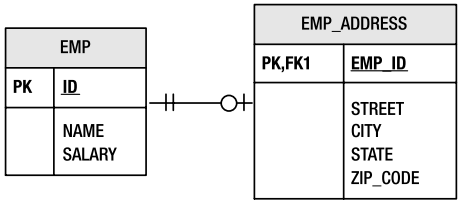

Figure 8-11. `EMP` *and* `EMP_ADDRESS` *tables*

To map this table structure to the `Employee` entity, we must declare `EMP_ADDRESS` as a secondary table and use the table element of the `@Column` annotation for every attribute stored in that table. Listing 8-17 shows the mapped entity. The primary key of the `EMP_ADDRESS` table is in the `EMP_ID` column. If it had been named `ID` then we would not have needed to use the name element in the `@PrimaryKeyJoinColumn` annotation. It defaults to the name of the primary key column in the primary table.

Listing 8-17. *Mapping an Entity Across Two Tables*

```
@Entity
@Table(name="EMP")
@SecondaryTable(name="EMP_ADDRESS",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="EMP_ID"))
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Column(table="EMP_ADDRESS")
    private String street;
    @Column(table="EMP_ADDRESS")
    private String city;
    @Column(table="EMP_ADDRESS")
    private String state;
    @Column(name="ZIP_CODE", table="EMP_ADDRESS")
    private String zip;
    // ...
}
```


In Chapter 4 we learned how to use the `schema` or `catalog` elements in `@Table` to qualify the primary table to be in a particular database schema or catalog. This is also valid in the `@SecondaryTable` annotation.

Previously when discussing embedded objects, we mapped the address fields of the `Employee` entity into an `Address` embedded type. With the address data in a secondary table, it is still possible to do this by specifying the mapped table name as part of the column information in the `@AttributeOverride` annotation. Listing 8-18 demonstrates this approach. Note that we have to enumerate all of the fields in the embedded type even though the column names may match the correct default values.

Listing 8-18. *Mapping an Embedded Type to a Secondary Table*

```
@Entity
@Table(name="EMP")
@SecondaryTable(name="EMP_ADDRESS",
                pkJoinColumns=@PrimaryKeyJoinColumn(name="EMP_ID"))
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="street", column=@Column(table="EMP_ADDRESS")),
        @AttributeOverride(name="city", column=@Column(table="EMP_ADDRESS")),
        @AttributeOverride(name="state", column=@Column(table="EMP_ADDRESS")),
        @AttributeOverride(name="zip",
                           column=@Column(name="ZIP_CODE", table="EMP_ADDRESS"))
    })
    private Address address;
    // ...
}
```

Let's consider a more complex example involving multiple tables and compound primary keys. Figure 8-12 shows the table structure we wish to map. In addition to the `EMPLOYEE` table, we have two secondary tables, `ORG_STRUCTURE` and `EMP_LOB`. The `ORG_STRUCTURE` table stores employee and manger reporting information. The `EMP_LOB` table stores large objects that are

infrequently fetched during normal query options. Moving large objects to a secondary table is a common design technique in many database schemas.

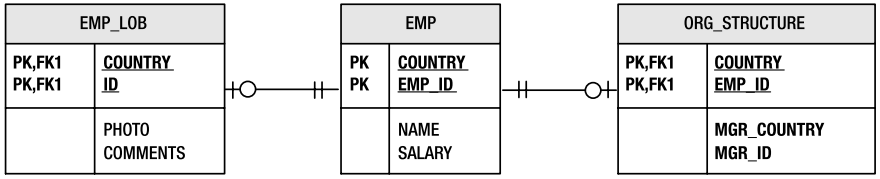

Figure 8-12. *Secondary tables with compound primary key relationships*

Listing 8-19 shows the `Employee` entity mapped to this table structure. We have reused the `EmployeeId` id class from Listing 8-5 in this example.

Listing 8-19. *Mapping an Entity with Multiple Secondary Tables*

```
@Entity
@IdClass(EmployeeId.class)
@SecondaryTables({
    @SecondaryTable(name="ORG_STRUCTURE", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="COUNTRY", referencedColumnName="COUNTRY"),
        @PrimaryKeyJoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")})
    @SecondaryTable(name="EMP_LOB", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="COUNTRY", referencedColumnName="COUNTRY"),
        @PrimaryKeyJoinColumn(name="ID", referencedColumnName="EMP_ID")})
})
public class Employee {
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;

    @Basic(fetch=FetchType.LAZY)
    @Lob
    @Column(table="EMP_LOB")
    private byte[] photo;

    @Basic(fetch=FetchType.LAZY)
    @Lob
    @Column(table="EMP_LOB")
    private char[] comments;
```



```
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="MGR_COUNTRY", referencedColumnName="COUNTRY",
                    table="ORG_STRUCTURE"),
        @JoinColumn(name="MGR_ID", referencedColumnName="EMP_ID",
                    table="ORG_STRUCTURE")
    })
    private Employee manager;
    // ...
}
```

We have thrown a few curves into this example to make it more interesting. The first is that we have defined `Employee` to have a composite primary key. This requires additional information to be provided for the `EMP_LOB` table, because its primary key is not named the same as the primary table. The next difference is that we are storing a relationship in the `ORG_STRUCTURE` secondary table. The `MGR_COUNTRY` and `MGR_ID` columns combine to reference the id of the manager for this employee. Since the employee has a composite primary key, the `manager` relationship must also specify a set of join columns instead of only one, and the `referencedColumnName` elements in those join columns refer to the primary key columns `COUNTRY` and `EMP_ID` in the entity's own primary table `EMPLOYEE`.

Inheritance

One of the common mistakes made by novice object-oriented developers is that they catch the vision of reuse and create complex inheritance hierarchies all for the sake of sharing a few methods. This will often lead to pain and hardship down the road as the application becomes difficult to debug and a challenge to maintain.

Most applications do enjoy the benefits of at least some inheritance in the object model. As with most things, moderation should be used, however, especially when it comes to mapping the classes to relational databases. Large hierarchies can often lead to significant performance reduction, and it may be that the cost of code reuse is higher than what you might want to pay.

In the following sections we will explain the support that exists in the API to map inheritance hierarchies and outline some of the repercussions.

Class Hierarchies

Being that this is a book about the Java Persistence API, the first and most obvious place to start talking about inheritance is in the Java object model. Entities are objects, after all, and should be able to inherit state and behavior from other entities. This is not only expected but also essential for the development of object-oriented applications.

What does it mean when one entity inherits state from its entity superclass? It may imply different things in the data model, but in the Java model it simply means that when a subclass entity is instantiated, then it has its own version or copy of both its locally defined state and its inherited state, all of which is persistent. While this basic premise is not at all surprising, it introduces the less obvious notion of what happens when an entity inherits from something other than another entity. Which classes is an entity allowed to extend, and what happens when it does?

Consider the class hierarchy shown in Figure 8-13. As we saw in Chapter 1 there are a number of ways that class inheritance can be represented in the database. In the object model there may even be a number of different ways to implement a hierarchy, some of which may include non-entity classes. We will use this example as we explore ways to persist inheritance hierarchies in the sections that follow.

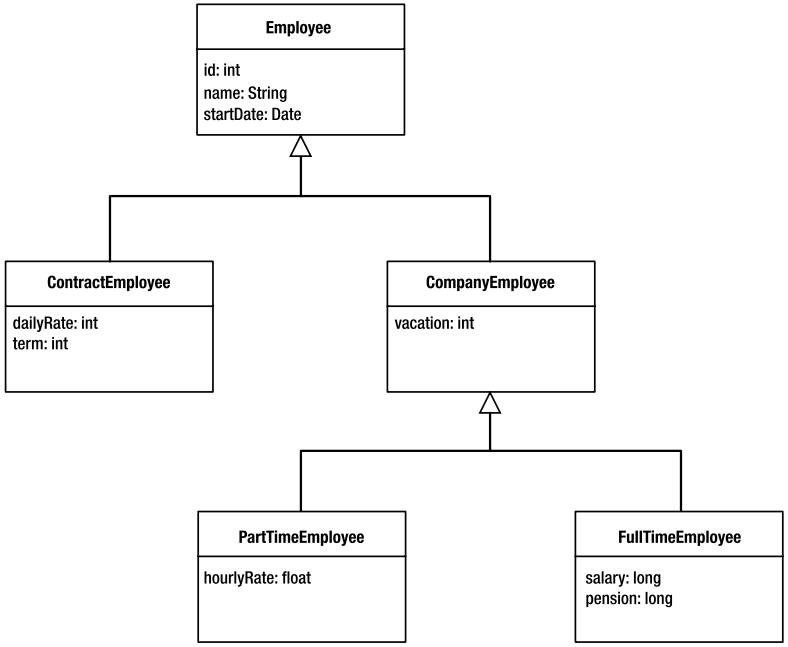

Figure 8-13. *Inheritance class hierarchy*

We differentiate between a class hierarchy, which is a set of various types of Java classes that extend each other in a tree, and an entity hierarchy, which is a tree consisting of persistent entity classes interspersed with non-entity classes. An entity hierarchy is rooted at the first entity class in the hierarchy.

Mapped Superclasses

The Java Persistence API defines a special kind of class called a *mapped superclass* that is quite useful as a superclass for entities. A mapped superclass provides a convenient class on which to store shared state and behavior that entities can inherit from, but it is itself not a persistent class and cannot act in the capacity of an entity. It cannot be queried over and cannot be the target of a relationship. Annotations such as `@Table` are not permitted on mapped superclasses since the state defined in them applies only to its entity subclasses.

Mapped superclasses can be compared to entities in somewhat the same way that an abstract class is compared to a concrete class; they can contain state and behavior but just can't be instantiated as persistent entities. An abstract class is of use only in relation to its concrete subclasses, and a mapped superclass is useful only as state and behavior that is inherited by the entity subclasses that extend it. They do not play a role in an entity inheritance hierarchy other than contributing that state and behavior to the entities that inherit from them.

Mapped superclasses may or may not be defined as abstract in their class definitions, but it is good practice to make them actual abstract Java classes. We don't know of any good use cases for creating concrete Java instances of them without ever being able to persist them, and chances are that if you happen to find one, then you probably want the mapped superclass to be an entity.

All of the default mapping rules that apply to entities also apply to the basic and relationship state in mapped superclasses. The biggest advantage of using mapped superclasses is being able to define partial shared state that should not be accessed on its own without the additional state that its entity subclasses add to it. If you are not sure whether to make a class an entity or a mapped superclass, then you need only ask yourself if you will ever need to query across or access an instance that is only exposed as an instance of that mapped class. This also includes relationships, since a mapped superclass can't be used as the target of a relationship. If you answer yes to any variant of that question, then you should probably make it a first class entity.

If we look back at Figure 8-13 we could conceivably treat the `CompanyEmployee` class as a mapped superclass instead of an entity. It defines shared state, but perhaps we have no reason to query over it.

A class is indicated as being a mapped superclass by annotating it with the `@MappedSuperclass` annotation. The class fragments from Listing 8-20 show how the hierarchy would be mapped with `CompanyEmployee` as a mapped superclass.

Listing 8-20. *Entities Inheriting from a Mapped Superclass*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}

@Entity
public class ContractEmployee extends Employee {
    @Column(name="D_RATE")
    private int dailyRate;
    private int term;
    // ...
}
```

```
@MappedSuperclass
public abstract class CompanyEmployee extends Employee {
    private int vacation;
    // ...
}

@Entity
public class FullTimeEmployee extends CompanyEmployee {
    private long salary;
    private long pension;
    // ...
}

@Entity
public class PartTimeEmployee extends CompanyEmployee {
    @Column(name="H_RATE")
    private float hourlyRate;
    // ...
}
```

Transient Classes in the Hierarchy

We call classes in an entity hierarchy that are not entities or mapped superclasses *transient classes*. Entities may extend transient classes either directly or indirectly through a mapped superclass. When an entity inherits from a transient class, then the state defined in the transient class is still inherited in the entity, but it is not persistent. In other words, the entity will have space allocated for the inherited state, according to the usual Java rules, but that state will not be managed by the persistence provider. It will be effectively ignored during the life cycle of the entity. The entity may manage that state manually through the use of lifecycle callback methods that we describe in Chapter 9, or other approaches, but the state will not be persisted as part of the provider-managed entity life cycle.

One could conceive of having a hierarchy that is composed of an entity that has a transient subclass, which in turn has one or more entity subclasses. While this case is not really a common one, it is nonetheless possible and can be achieved in the rare circumstances when having shared transient state or common behavior is desired. It would normally be more convenient, though, to declare the transient state or behavior in the entity superclass than to create an intermediate transient class. Listing 8-21 shows an entity that inherits from a superclass that defines transient state that is the time an entity was created in memory.

Listing 8-21. *Entity Inheriting from a Transient Superclass*

```
public abstract class CachedEntity {
    private long createTime;

    public CachedEntity() { createTime = System.currentTimeMillis(); }

    public long getCacheAge() { return System.currentTimeMillis() - createTime; }
}

@Entity
public class Employee extends CachedEntity {
    public Employee() { super(); }
    // ...
}
```

In this example we moved the transient state from the entity class into a transient superclass, but the end result is really quite the same. The previous example might have been a little neater without the extra class, but this example allows us to share the transient state and behavior across any number of entities that need only extend `CachedEntity`.

Abstract and Concrete Classes

We have mentioned the notion of abstract versus concrete classes in the context of mapped superclasses, but we didn't go into any more detail about entity and transient classes. Most people, depending upon their philosophy, might expect that all non-leaf classes in an object hierarchy should be abstract, or at the very least that some of them would be. A restriction that entities always be concrete classes would mess this up quite handily, and fortunately this is not the case. It is perfectly acceptable for entities, mapped superclasses, or transient classes to be either abstract or concrete at any level of the inheritance tree. Like mapped superclasses, making transient classes concrete in the hierarchy doesn't really serve any purpose and as a general rule should be avoided.

The case that we have not talked about is the one where an entity is an abstract class. The only difference between an entity that is an abstract class and one that is a concrete class is the Java rule that prohibits abstract classes from being instantiated. They can still define persistent state and behavior that will be inherited by the concrete entity subclasses below them. They can be queried, the result of which will be composed of concrete entity subclass instances. They can also bear the inheritance mapping metadata for the hierarchy.

Our hierarchy in Figure 8-13 had an `Employee` class that was a concrete class. It does not make any sense to instantiate this class, so we would likely want it to be abstract. We would then end up with all of our non-leaf classes being abstract and the leaf classes being persistent.

Inheritance Models

The Java Persistence API provides support for three different data representations. The use of two of them is fairly widespread, while the third is less common and not required to be supported, though it is still fully defined with the intention that it be required to be supported by providers in the future.

When an entity hierarchy exists, it is always rooted at an entity class. Recall that mapped superclasses do not count as levels in the hierarchy since they contribute only to the entities beneath them. The root entity class must signify the inheritance hierarchy by being annotated with the `@Inheritance` annotation. This annotation indicates the strategy that should be used for mapping and must be one of the three strategies described in the following sections.

Every entity in the hierarchy must either define or inherit its identifier, which means that the identifier must be defined either in the root entity or in a mapped superclass above it. A mapped superclass may be higher up in the class hierarchy than where the identifier is defined.

Single-Table Strategy

The most common and performant way of storing the state of multiple classes is to define a single table to contain a superset of all of the possible state in any of the entity classes. This approach is called, not surprisingly, a *single-table* strategy. It has the consequence that for any given table row representing an instance of a concrete class, there may be columns that do not have values because they apply only to a sibling class in the hierarchy.

From Figure 8-13 we see that the `id` is located in the root `Employee` entity class and is shared by the rest of the persistence classes. All of the persistent entities in an inheritance tree must use the same type of identifier. We don't need to think about it very long before we see why this makes sense at both levels. In the object layer it wouldn't be possible to issue a polymorphic `find()` operation on a superclass if there were not a common identifier type that we could pass in. Similarly at the table level we would need multiple primary key columns but without being able to fill them all in on any given insertion of an instance that only made use of one of them.

The table must contain enough columns to store all of the state in all of the classes. An individual row stores the state of an entity instance of a concrete entity type, which would normally imply that there would be some columns left unfilled in every row. Of course this leads to the conclusion that the columns mapped to concrete subclass state should be nullable, which is normally not a big issue but could be a problem for some database administrators.

In general, the single-table approach tends to be more wasteful of database tablespace, but it does offer peak performance for both polymorphic queries and write operations. The SQL that is needed to issue these operations is simple, optimized, and does not require joining.

To specify the single-table strategy for the inheritance hierarchy, the root entity class is annotated with the `@Inheritance` annotation with its strategy set to `SINGLE_TABLE`. In our previous model this would mean annotating the `Employee` class as follows:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Employee { ... }
```

As it turns out, though, the single-table strategy is the default one, so we wouldn't strictly even need to include the strategy element at all. An empty `@Inheritance` annotation, or even no `@Inheritance` annotation at all, would work just just as well.

In Figure 8-14 we see the single-table representation of our `Employee` hierarchy model. In terms of the table structure and schema architecture for the single-table strategy, it makes no difference whether `CompanyEmployee` is a mapped superclass or an entity.

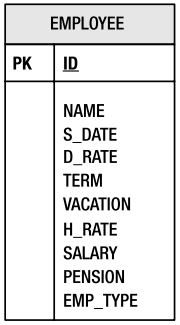

Figure 8-14. *A single-table inheritance data model*

Discriminator Column

You may have noticed an extra column named `EMP_TYPE` in Figure 8-13 that was not mapped to any field in any of the classes. This field has a special purpose and is required when using a single table to model inheritance. It is called a *discriminator column* and is mapped using the `@DiscriminatorColumn` annotation in conjunction with the `@Inheritance` annotation we have already learned about. The `name` element of this annotation specifies the name of the column that should be used as the discriminator column, and if not specified will be defaulted to a column named `DTYPE`.

A `discriminatorType` element dictates the type of the discriminator column. Some applications prefer to use strings to discriminate between the entity types, while others like using integer values to indicate the class. The type of the discriminator column may be one of three predefined discriminator column types: `INTEGER`, `STRING`, or `CHAR`. If the `discriminatorType` element is not specified, then the default type of `STRING` will be assumed.

Discriminator Value

Every row in the table will have a value in the discriminator column called a *discriminator value*, or a *class indicator*, to indicate the type of entity that is stored in that row. Every concrete entity in the inheritance hierarchy, therefore, needs a discriminator value specific to that entity type so that the provider can process or assign the correct entity type when it loads and stores the row. The way this is done is to use an `@DiscriminatorValue` annotation on each concrete entity class. The string value in the annotation specifies the discriminator value that instances of the class will get assigned when they are inserted into the database. This will allow the provider to recognize instances of the class when it issues queries. This value should be of the same type as was specified or defaulted as the `discriminatorType` element in the `@DiscriminatorColumn` annotation.

If no `@DiscriminatorValue` annotation is specified, then the provider will use a provider-specific way of obtaining the value. If the `discriminatorType` was `STRING`, then the provider will just use the entity name as the class indicator string. If the `discriminatorType` is `INTEGER`, then we would either have to specify the discriminator values for every entity class or none of them. If we were to specify some but not others, then we could not guarantee that a provider-generated value would not overlap with one that we specified.

Listing 8-22 shows how our `Employee` hierarchy is mapped to a single-table strategy.

Listing 8-22. *Entity Hierarchy Mapped Using Single-Table Strategy*

```
@Entity
@Table(name="EMP")
@Inheritance
@DiscriminatorColumn(name="EMP_TYPE")
public abstract class Employee { ... }

@Entity
public class ContractEmployee extends Employee { ... }

@MappedSuperclass
public abstract class CompanyEmployee extends Employee { ... }

@Entity
@DiscriminatorValue("FTEmp")
public class FullTimeEmployee extends CompanyEmployee { ... }

@Entity(name="PTEmp")
public class PartTimeEmployee extends CompanyEmployee { ... }
```

The `Employee` class is the root class, so it establishes the inheritance strategy and discriminator column. We have assumed the default strategy of `SINGLE_TABLE` and discriminator type of `STRING`.

Neither the `Employee` nor the `CompanyEmployee` classes have discriminator values, because discriminator values should not be specified for abstract entity classes, mapped superclasses, transient classes, or any abstract classes for that matter. Only concrete entity classes use discriminator values since they are the only ones that actually get stored and retrieved from the database.

The `ContractEmployee` entity does not use an `@DiscriminatorValue` annotation, because the default string "ContractEmployee", which is the default entity name that is given to the class, is just what we want. The `FullTimeEmployee` class explicitly lists its discriminator value to be "FTEmp", so that is what is stored in each row for instances of `FullTimeEmployee`. Meanwhile, the `PartTimeEmployee` class will get "PTEmp" as its discriminator value since it set its entity name to be "PTEmp", and that is the name that gets used as the discriminator value when none is specified.

In Figure 8-15 we can see a sample of some of the data that we might find given the earlier model and settings. We can see from the `EMP_TYPE` discriminator column that there are three different types of concrete entities. We also see null values in the columns that do not apply to an entity instance.

EMPLOYEE

ID	NAME	S_DATE	D_RATE	TERM	VACATION	H_RATE	SALARY	PENSION	EMP_TYPE
1	John	020101	500	12					ContractEmployee
2	Paul	020408	600	24					ContractEmployee
3	Sarah	030610	700	18					ContractEmployee
4	Patrick	040701			15		55000	100000	FTEmp
5	Joan	030909			15		59000	200000	FTEmp
6	Sam	000312			20		60000	450000	FTEmp
7	Mark	041101			15	17.00			PTEmp
8	Ryan	051205			15	16.00			PTEmp
9	Jackie	060103			10	15.00			PTEmp

Figure 8-15. *Sample of single-table inheritance data*

Joined Strategy

From the perspective of a Java developer, a data model that maps each entity to its own table makes a lot of sense. Every entity, whether it is abstract or concrete, will have its state mapped to a different table. Consistent with our earlier description, mapped superclasses do not get mapped to their own tables but are mapped as part of their entity subclasses.

Mapping a table per entity provides the data reuse that a *normalized*[2] data schema offers and is the most efficient way to store data that is shared by multiple subclasses in a hierarchy. The problem is that when it comes time to reassemble an instance of any of the subclasses, the tables of the subclasses must be joined together with the superclass tables. It makes fairly obvious the reason why this strategy is called the *joined* strategy. It is also somewhat more expensive to insert an entity instance, because a row must be inserted in each of its superclass tables along the way.

Recall from the single-table strategy that the identifier must be of the same type for every class in the hierarchy. In a joined approach we will have the same type of primary key in each of the tables, and the primary key of a subclass table also acts as a foreign key that joins to its superclass table. This should ring a bell because of its similarity to the multiple-table case earlier in the chapter where we joined the tables together using the primary keys of the tables and used the `@PrimaryKeyJoinColumn` annotation to indicate it. We use this same annotation in the joined inheritance case since we have multiple tables that each contain the same primary key type and each potentially has a row that contributes to the final combined entity state.

While joined inheritance is both intuitive and efficient in terms of data storage, the joining that it requires makes it somewhat expensive to use when hierarchies are deep or wide. The deeper the hierarchy the more joins it will take to assemble instances of the concrete entity at the bottom. The broader the hierarchy the more joins it will take to query across an entity superclass.

In Figure 8-16 we see our `Employee` example mapped to a joined table architecture. The data for an entity subclass is spread across the tables in the same way that it is spread across the class hierarchy. When using a joined architecture, the decision as to whether `CompanyEmployee`

2. Normalization of data is a database practice that attempts to remove redundantly stored data. For the seminal paper on data normalization, see "A Relational Model of Data for Large Shared Databanks" by E. F. Codd (*Communications of the ACM*, 13(6) June 1970). Also, any database design book or paper should have an overview.

is a mapped superclass or an entity makes a difference, since mapped superclasses do not get mapped to tables. An entity, even if it is an abstract class, always does. Figure 8-13 shows it as a mapped superclass, but if it were an entity then an additional COMPANY_EMP table would exist with ID and VACATION columns in it, and the VACATION column in the FT_EMP and PT_EMP tables would not be present.

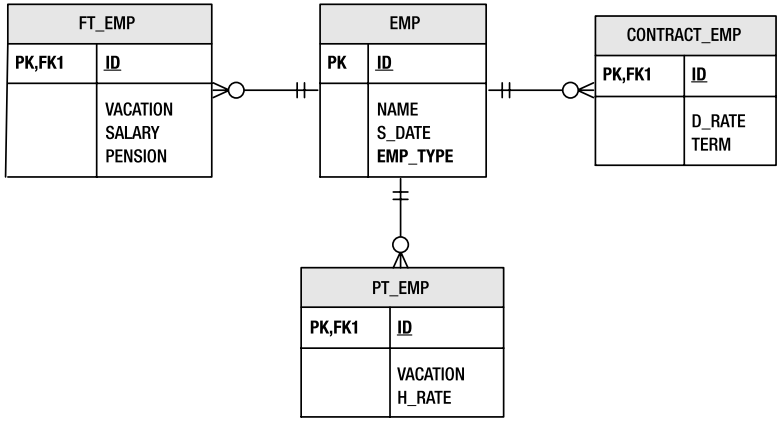

Figure 8-16. *Joined inheritance data model*

To map an entity hierarchy to a joined model, the `@Inheritance` annotation need only specify `JOINED` as the strategy. Like the single-table example, the subclasses will adopt the same strategy that is specified in the root entity superclass.

Even though there are multiple tables to model the hierarchy, the discriminator column is only defined on the root table, so the `@DiscriminatorColumn` annotation is placed on the same class as the `@Inheritance` annotation.

■ **Tip** Some vendors offer implementations of joined inheritance without the use of a discriminator column. Discriminator columns should be used if portability is required.

Our `Employee` hierarchy example can be mapped using the joined approach shown in Listing 8-23. In this example we have used integer discriminator columns instead of the default string type.

Listing 8-23. *Entity Hierarchy Mapped Using the Joined Strategy*

```
@Entity
@Table(name="EMP")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE", discriminatorType=DiscriminatorType.INTEGER)
public abstract class Employee { ... }

@Entity
@Table(name="CONTRACT_EMP")
@DiscriminatorValue("1")
public class ContractEmployee extends Employee { ... }

@MappedSuperclass
public abstract class CompanyEmployee extends Employee { ... }

@Entity
@Table(name="FT_EMP")
@DiscriminatorValue("2")
public class FullTimeEmployee extends CompanyEmployee { ... }

@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("3")
public class PartTimeEmployee extends CompanyEmployee { ... }
```

Table-per-Concrete-Class Strategy

A third approach to mapping an entity hierarchy is to use a strategy where a *table per concrete class* is defined. This data architecture goes in the reverse direction of non-normalization of entity data and maps each concrete entity class and all of its inherited state to a separate table. This has the effect of causing all shared state to be redefined in the tables of all of the concrete entities that inherit it. This strategy is not required to be supported by providers but is included because it is anticipated that it will be required in a future release of the API. We will describe it briefly for completeness.

The negative side of using this strategy is that it makes polymorphic querying across a class hierarchy more expensive than the other strategies. The problem is that it must query across all of the subclass tables using a UNION operation, which is generally regarded as being expensive when lots of data is involved.

The bright side of table-per-concrete-class hierarchies when compared to joined hierarchies is seen in cases of querying over instances of a single concrete entity. In the joined case, every query requires a join, even when querying across a single concrete entity class. In the table-per-concrete-class case, it is kind of like the single-table hierarchy because the query is confined to a single table. Another advantage is that the discriminator column goes away. Every concrete entity has its own separate table, and there is no mixing or sharing of schema, so no class indicator is ever needed.

Mapping our example to this type of hierarchy is a matter of specifying the strategy as TABLE_PER_CLASS and making sure there is a table for each of the concrete classes. If a legacy database is being used, then the inherited columns could be named differently in each of the concrete tables and the @AttributeOverride annotation would come in handy. In this case the CONTRACT_EMP table didn't have the NAME and S_DATE columns but instead had FULLNAME and SDATE for the name and startDate fields defined in Employee.

If the attribute that we wanted to override was an association instead of a simple state mapping, then we could still override the mapping, but we would need to use an @AssociationOverride annotation instead of @AttributeOverride. The @AssociationOverride annotation allows us to override the join columns used to reference the target entity of a many-to-one or one-to-one association defined in a mapped superclass. To show this, we need to add a manager attribute to the CompanyEmployee mapped superclass. The join column is mapped by default in the CompanyEmployee class to the MANAGER column in the two FT_EMP and PT_EMP subclass tables, but in PT_EMP the name of the join column is actually MGR. We override the join column by adding the @AssociationOverride annotation to the PartTimeEmployee entity class and specifying the name of the attribute we are overriding and the join column that we are overriding it to be. Listing 8-24 shows a complete example of the entity mappings, including the overrides.

Listing 8-24. *Entity Hierarchy Mapped Using Table-per-Concrete-Class Strategy*

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
    @Id private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}

@Entity
@Table(name="CONTRACT_EMP")
@AttributeOverrides({
    @AttributeOverride(name="name", column=@Column(name="FULLNAME")),
    @AttributeOverride(name="startDate", column=@Column(name="SDATE"))
})
public class ContractEmployee extends Employee {
    @Column(name="D_RATE")
    private int dailyRate;
    private int term;
    // ...
}
```



```
@MappedSuperclass
public abstract class CompanyEmployee extends Employee {
    private int vacation;
    @ManyToOne
    private Employee manager;
    // ...
}

@Entity @Table(name="FT_EMP")
public class FullTimeEmployee extends CompanyEmployee {
    private long salary;
    @Column(name="PENSION")
    private long pensionContribution;
    // ...
}

@Entity
@Table(name="PT_EMP")
@AssociationOverride(name="manager",
                     joinColumns=@JoinColumn(name="MGR"))
public class PartTimeEmployee extends CompanyEmployee {
    @Column(name="H_RATE")
    private float hourlyRate;
    // ...
}
```

The table organization shows how these columns are mapped to the concrete tables. See Figure 8-17 for a clear picture of what the tables would look like and how the different types of employee instances would be stored.

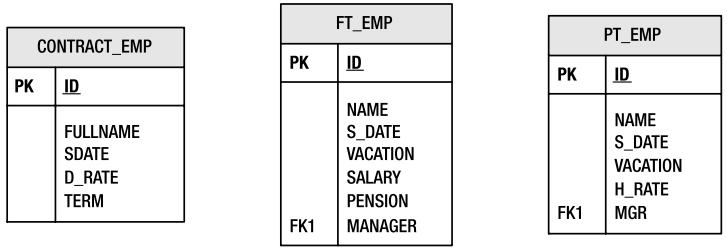

Figure 8-17. *Table-per-concrete-class data model*

Mixed Inheritance

We should begin this section by saying that the practice of mixing inheritance types within a single inheritance hierarchy is currently outside the specification. We are including it because it is both useful and interesting, but we are offering a warning that it may not be portable to rely on such behavior, even if your vendor supports it.

Furthermore, it really makes sense to mix only single-table and joined inheritance types. We will show an example of mixing these two, bearing in mind that support for them is vendor-specific. The intent is that in future releases of the specification, the more useful cases will be standardized and required to be supported by compliant implementations.

The premise for mixing inheritance types is that it is well within the realm of possibilities that a data model includes a combination of single-table and joined-table designs within a single entity hierarchy. This can be illustrated by taking our joined example in Figure 8-16 and storing the `FullTimeEmployee` and `PartTimeEmployee` instances in a single table. This would produce a model that looks like the one shown in Figure 8-18.

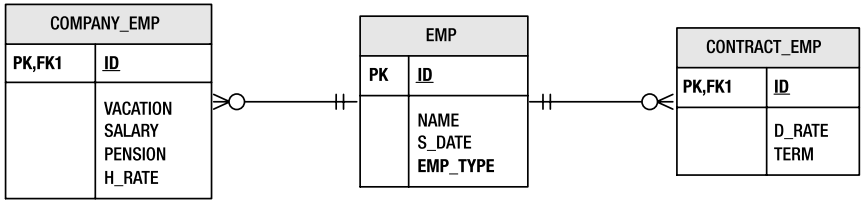

Figure 8-18. *Mixed inheritance data model*

In this example the joined strategy is used for the `Employee` and `ContractEmployee` classes, while the `CompanyEmployee`, `FullTimeEmployee`, and `PartTimeEmployee` classes revert to a single-table model. To make this inheritance strategy switch at the level of the `CompanyEmployee`, we need to make a simple change to the hierarchy. We need to turn `CompanyEmployee` into an abstract entity instead of a mapped superclass so that it can bear the new inheritance metadata. Note that this is simply an annotation change, not making any change to the domain model.

The inheritance strategies can be mapped as shown in Listing 8-25. Notice that we do not need to have a discriminator column for the single-table subhierarchy since we already have one in the superclass `EMP` table.

Listing 8-25. *Entity Hierarchy Mapped Using Mixed Strategies*

```
@Entity
@Table(name="EMP")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="EMP_TYPE")
public abstract class Employee {
    @Id private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private Date startDate;
    // ...
}
```



```
@Entity
@Table(name="CONTRACT_EMP")
public class ContractEmployee extends Employee {
    @Column(name="D_RATE") private int dailyRate;
    private int term;
    // ...
}

@Entity
@Table(name="COMPANY_EMP")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class CompanyEmployee extends Employee {
    private int vacation;
    // ...
}

@Entity
public class FullTimeEmployee extends CompanyEmployee {
    private long salary;
    @Column(name="PENSION")
    private long pensionContribution;
    // ...
}

@Entity
public class PartTimeEmployee extends CompanyEmployee {
    @Column(name="H_RATE")
    private float hourlyRate;
    // ...
}
```

Summary

Entity mapping requirements often go well beyond the simplistic mappings that map a field or a relationship to a named column. In this chapter we addressed some of the more varied and diverse mapping practices that are supported by the Java Persistence API.

We discussed defining secondary objects called embedded objects and storing them into the tables of the entities that own them. We showed how the embedded object class definitions can be reused by multiple entities and how to override the mappings within any given entity.

Identifiers may be composed of multiple columns. We revealed the two approaches for defining and using compound primary keys and demonstrated using them in a way that is compatible with EJB 2.1 primary key classes. We established how other entities can have foreign key references to entities with compound identifiers and explained how multiple join columns can be used in any context when a single join column applies. We also showed an example of mapping an identifier that included a relationship as part of its identity.

We went on to show how to distribute entity state across multiple tables and how to use the secondary tables with relationships. We even saw how an embedded object can map to a secondary table of an entity.

Finally we went into detail about the three different inheritance strategies that can be used to map inheritance hierarchies to tables. We explained mapped superclasses and how they can be used to define shared state and behavior. We went over the data models that differentiate the various approaches and showed how to map an entity hierarchy to the tables in each case. We finished off by illustrating how to mix inheritance types within a single hierarchy.

In the next chapter we will continue our discussion of advanced topics, looking at issues such as SQL queries, optimistic locking, and schema generation.

CHAPTER 9

■ ■ ■

Advanced Topics

Having a chapter called "Advanced Topics" is always a bit dicey because not everybody's definition of "advanced" is going to correspond to the same topics. What one person sees as advanced may be another person's basic bread-and-butter feature. It very much depends upon the background and experience of the developer as well as the complexity of applications that one is involved in developing.

What we can say is that, in large part, the topics in this chapter are those that we intended during development of the specification to be of a more advanced nature or to be used by more advanced developers. There are a few exceptions to this rule, though. For example, we included optimistic locking in this chapter even though most applications do need to be aware of and make use of optimistic locking. Generally, we think that most applications will not use more than a few of the features described in this chapter. With this in mind, let us explore some of the other features of the Java Persistence API.

SQL Queries

With all of the effort that has gone into abstracting away the physical data model, both in terms of object-relational mapping and JPQL, it might be surprising to learn that SQL is alive and well in the Java Persistence API. Although JPQL is the preferred method of querying over entities, SQL cannot be overlooked as a necessary element in many enterprise applications. The sheer size and scope of the SQL features supported by the major database vendors means that a portable language like JPQL will never be able to fully encompass all of their features.

■ **Note** SQL queries are also known as *native* queries. `EntityManager` methods and query annotations related to SQL queries also use this terminology. While this allows other query languages to be supported in the future, any query string in a native query operation is assumed to be SQL.

Before discussing the mechanics of SQL queries, let's first consider some of the reasons a developer using JPQL might want to integrate SQL queries into their application.

First, JPQL, despite the enhancements made by the Java Persistence API, still contains only a subset of the features supported by many database vendors. Inline views (subqueries in the FROM clause), hierarchical queries, access to stored procedures, and function expressions to manipulate date and time values are just some of the features missing from JPQL.

Second, although vendors may provide hints to assist with optimizing a JPQL expression, there are cases where the only way to achieve the performance required by an application is to replace the JPQL query with a hand-optimized SQL version. This may be a simple restructuring of the query that the persistence provider was generating, or it may be a vendor-specific version that leverages query hints and features specific to a particular database.

Of course, just because you can use SQL doesn't mean you should. Persistence providers have become very skilled at generating high-performance queries, and many of the limitations of JPQL can often be worked around in application code. We recommend avoiding SQL initially if possible and then introducing it only when necessary.

In the following sections we will discuss how SQL queries are defined using the Java Persistence API and how their result sets may be mapped back to entities. One of the major benefits of SQL query support is that it uses the same `Query` interface used for JPQL queries. With some small exceptions that will be described later, all of the `Query` interface operations discussed earlier in the chapter apply equally to both JPQL and SQL queries.

Native Queries vs. JDBC

A perfectly valid question for anyone investigating SQL support in the Java Persistence API is whether or not it is needed at all. JDBC has been in use for years, provides a broad feature set, and works well. It's one thing to introduce a persistence API that works on entities, but another thing entirely to introduce a new API for issuing SQL queries.

The main reason to consider using SQL queries with the Java Persistence API is when the result of the query will be converted back into entities. As an example, let's consider a typical use case for SQL in an application that uses the Java Persistence API. Given the employee id for a manager, the application needs to determine all of the employees that report to that manager either directly or indirectly. For example, if the query were for a senior manager, then the results would include all of the managers who report to that senior manager as well as the employees who report to those managers. This type of query cannot be implemented using JPQL, but a database such as Oracle natively supports hierarchical queries for just this purpose. Listing 9-1 demonstrates the typical sequence of JDBC calls to execute this query and transform the results into entities for use by the application.

Listing 9-1. *Querying Entities Using SQL and JDBC*

```
@Stateless
public class OrgStructureBean implements OrgStructure {
    private static final String ORG_QUERY =
        "SELECT emp_id, name, salary " +
        "FROM emp " +
        "START WITH manager_id = ? " +
        "CONNECT BY PRIOR emp_id = manager_id";

    @Resource
    DataSource hrDs;

    public List findEmployeesReportingTo(int managerId) {
        Connection conn = null;
        PreparedStatement sth = null;
```



```
        try {
            conn = hrDs.getConnection();
            sth = conn.prepareStatement(ORG_QUERY);
            sth.setLong(1, managerId);
            ResultSet rs = sth.executeQuery();

            ArrayList<Employee> result = new ArrayList<Employee>();
            while (rs.next()) {
                Employee emp = new Employee();
                emp.setId(rs.getInt(1));
                emp.setName(rs.getString(2));
                emp.setSalary(rs.getLong(3));
                result.add(emp);
            }
            return result;
        } catch (SQLException e) {
            throw new EJBException(e);
        }
    }
}
```

Now consider the alternative syntax supported by the Java Persistence API as shown in Listing 9-2. By simply indicating that the result of the query is the `Employee` entity, the query engine uses the object-relational mapping of the entity to figure out which result columns map to the entity properties and builds the result set accordingly.

Listing 9-2. *Querying Entities Using SQL and the* `Query` *Interface*

```
@Stateless
public class OrgStructureBean implements OrgStructure {
    private static final String ORG_QUERY =
        "SELECT emp_id, name, salary, manager_id, dept_id, address_id " +
        "FROM emp " +
        "START WITH manager_id = ? " +
        "CONNECT BY PRIOR emp_id = manager_id";

    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List findEmployeesReportingTo(int managerId) {
        return em.createNativeQuery(ORG_QUERY, Employee.class)
                 .setParameter(1, managerId)
                 .getResultList();
    }
}
```

Not only is the code much easier to read, but it also makes use of the same `Query` interface used for JPQL queries. This helps to keep application code consistent, as it only needs to concern itself with the `EntityManager` and `Query` interfaces.

Defining and Executing SQL Queries

SQL queries may be defined dynamically at runtime or named in persistence unit metadata, similar to the JPQL query definitions we discussed in Chapter 6. The key difference between defining JPQL and SQL queries lies in the understanding that the query engine should not parse and interpret vendor-specific SQL. In order to execute a SQL query and get entity instances in return, additional mapping information about the query result is required.

The first and simplest form of dynamically defining a SQL query that returns an entity result is to use the `createNativeQuery()` method of the `EntityManager` interface, passing in the query string and the entity type that will be returned. Listing 9-2 in the previous section demonstrated this approach to map the results of an Oracle hierarchical query to the `Employee` entity. The query engine uses the object-relational mapping of the entity to figure out which result column aliases map to which entity properties. As each row is processed, the query engine instantiates a new entity instance and sets the available data into it.

If the column aliases of the query do not match up with the object-relational mapping for an entity, or if the results contain both entity and non-entity results, then SQL result set mapping metadata is required. SQL result set mappings are defined as persistence unit metadata and are referenced by name. When the `createNativeQuery()` method is invoked with a SQL query string and a result set mapping name, the query engine uses this mapping to build the result set. SQL result set mappings are discussed in the next section.

Named SQL queries are defined using the `@NamedNativeQuery` annotation. This annotation may be placed on any entity and defines the name of the query as well as the query text. Like JPQL named queries, the name of the query must be unique within the persistence unit. If the result type is an entity, the `resultClass` element may be used to indicate the entity class. If the result requires a SQL mapping, the `resultSetMapping` element may be used to specify the mapping name. Listing 9-3 shows how the hierarchical query demonstrated earlier would be defined as a named query.

Listing 9-3. *Using an Annotation to Define a Named Native Query*

```
@NamedNativeQuery(
    name="orgStructureReportingTo",
    query="SELECT emp_id, name, salary, manager_id, dept_id, address_id " +
          "FROM emp " +
          "START WITH manager_id = ? " +
          "CONNECT BY PRIOR emp_id = manager_id",
    resultClass=Employee.class
)
```

The advantage of using named SQL queries is that the application can use the `createNamedQuery()` method on the `EntityManager` interface to create and execute the query. The fact that the named query was defined using SQL instead of JPQL is not important. Listing 9-4 demonstrates the reporting structure bean again, this time using a named query. The other advantage of using named queries instead of dynamic queries is that they can be overridden using XML mapping files. A query originally specified in JPQL can be overridden with a SQL version and vice versa. This technique is described in Chapter 10.

Listing 9-4. *Executing a Named SQL Query*

```
@Stateless
public class OrgStructureBean implements OrgStructure {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List findEmployeesReportingTo(int managerId) {
        return em.createNamedQuery("orgStructureReportingTo")
                 .setParameter(1, managerId)
                 .getResultList();
    }
}
```

One thing to be careful of with SQL queries that return entities is that the resulting entity instances become managed by the persistence context, just like the results of a JPQL query. If you modify one of the returned entities, it will be written to the database when the persistence context becomes associated with a transaction. This is normally what you want, but it requires that any time you select data that corresponds to existing entity instances, it is important to ensure that all of the necessary data required to fully construct the entity is part of the query. If you leave out a field from the query, or default it to some value and then modify the resulting entity, you will most likely overwrite the correct version already stored in the database.

There are two benefits to getting managed entities back from a SQL query. The first is that a SQL query can replace an existing JPQL query and that application code should still work without changes. The second benefit is that it allows the developer to use SQL queries as a method of constructing new entity instances from tables that may not have any object-relational mapping. For example, in many database architectures, there is a staging area to hold data that has not yet been verified or requires some kind of transformation before it can be moved to its final location. Using the Java Persistence API, a developer could start a transaction, query the staged data to construct entities, perform any required changes, and then commit. The newly created entities will get written to the tables mapped by the entity, not the staging tables used in the SQL query.

SQL data-manipulation statements (INSERT, UPDATE, and DELETE) are also supported as a convenience so that JDBC calls do not have to be introduced in an application otherwise restricted to the Java Persistence API. To define such a query, use the `createNativeQuery()` method, but without any mapping information. Listing 9-5 demonstrates these types of queries in the form of a session bean that logs messages to a table. Note that the bean methods run in a `REQUIRES_NEW` transaction context to ensure that the message is logged even if an active transaction rolls back.

Listing 9-5. *Using SQL INSERT and DELETE Statements*

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public class LoggerBean implements Logger {
    private static final String INSERT_SQL =
        "INSERT INTO message_log (id, message, log_dttm) " +
        "       VALUES(id_seq.nextval, ?, SYSDATE)";
    private static final String DELETE_SQL =
        "DELETE FROM message_log";

    @PersistenceContext(unitName="Logger")
    EntityManager em;

    public void logMessage(String message) {
        em.createNativeQuery(INSERT_SQL)
          .setParameter(1, message)
          .executeUpdate();
    }

    public void clearMessageLog() {
        em.createNativeQuery(DELETE_SQL)
          .executeUpdate();
    }
}
```

Executing SQL statements that make changes to data in tables mapped by entities is generally discouraged. Doing so may cause cached entities to be inconsistent with the database, as the provider is not able to track changes made to entity state that has been modified by data-manipulation statements.

SQL Result Set Mapping

In the SQL query examples shown so far, the result mapping was straightforward. The column aliases in the SQL string matched up directly with the object-relational column mapping for a single entity. It is not always the case that the names match up, nor is it always the case that only a single entity type is returned. The Java Persistence API provides SQL result set mappings to handle these scenarios.

A SQL result set mapping is defined using the `@SqlResultSetMapping` annotation. It may be placed on an entity class and consists of a name (unique within the persistence unit) and one or more entity and column mappings. The entity result class argument on the `createNativeQuery()` method is really a shortcut to specifying a simple SQL result set mapping. The following mapping is equivalent to specifying `Employee.class` in a call to `createNativeQuery()`:

```
@SqlResultSetMapping(
    name="employeeResult",
    entities=@EntityResult(entityClass=Employee.class)
)
```


Here we have defined a SQL result set mapping called `employeeResult` that may be referenced by any query returning `Employee` entity instances. The mapping consists of a single entity result, specified by the `@EntityResult` annotation, that references the `Employee` entity class. The query must supply values for all columns mapped by the entity, including foreign keys. It is vendor-specific whether the entity is partially constructed or whether an error occurs if any required entity state is missing.

Mapping Foreign Keys

When the query engine attempts to map the query results to an entity, it considers foreign key columns for single-valued associations as well. Let's look at the reporting structure query again:

```
SELECT emp_id, name, salary, manager_id, dept_id, address_id
FROM emp
START WITH manager_id IS NULL
CONNECT BY PRIOR emp_id = manager_id
```

The `MANAGER_ID`, `DEPT_ID`, and `ADDRESS_ID` columns all map to the join columns of associations on the `Employee` entity. An `Employee` instance returned from this query can use the methods `getManager()`, `getDepartment()`, and `getAddress()`, and the results will be as expected. The persistence provider will retrieve the associated entity based on the foreign key value read in from the query. There is no way to populate collection associations from a SQL query. Entity instances constructed from this example are effectively the same as they would have been had they been returned from a JPQL query.

Multiple Result Mappings

A query may return more than one entity at a time. This is most often useful if there is a one-to-one relationship between two entities, otherwise the query will result in duplicate entity instances. Consider the following query:

```
SELECT emp_id, name, salary, manager_id, dept_id, address_id,
       id, street, city, state, zip
FROM emp, address
WHERE address_id = id
```

The SQL result set mapping to return both the `Employee` and `Address` entities out of this query is defined in Listing 9-6. Each entity is listed in an `@EntityResult` annotation, an array of which is assigned to the `entities` element. The order in which the entities are listed is not important. The query engine uses the column names of the query to match against entity mapping data, not column position.

Listing 9-6. *Mapping a SQL Query That Returns Two Entity Types*

```
@SqlResultSetMapping(
    name="EmployeeWithAddress",
    entities={@EntityResult(entityClass=Employee.class),
              @EntityResult(entityClass=Address.class)}
)
```

Mapping Column Aliases

If the column aliases in the SQL statement do not directly match up with the names specified in the column mappings for the entity, then field result mappings are required for the query engine to make the correct association. Consider, for example, if both the EMP and ADDRESS tables listed in the previous example used the column ID for their primary key. The query would have to be altered to alias the ID columns so that they are unique:

```
SELECT emp.id AS emp_id, name, salary, manager_id, dept_id, address_id,
       address.id, street, city, state, zip
FROM emp, address
WHERE address_id = address.id
```

The @FieldResult annotation is used to map column aliases to the entity attributes in situations where the name in the query is not the same as the one used in the column mapping. Listing 9-7 shows the mapping required to convert the EMP_ID alias to the id attribute of the entity. More than one @FieldResult may be specified, but only the mappings that are different need to be specified. This can be a partial list of entity attributes.

Listing 9-7. *Mapping a SQL Query with Unknown Column Aliases*

```
@SqlResultSetMapping(
    name="EmployeeWithAddress",
    entities={@EntityResult(entityClass=Employee.class,
                            fields=@FieldResult(name="id", column="EMP_ID")),
              @EntityResult(entityClass=Address.class)}
)
```

Mapping Scalar Result Columns

SQL queries are not limited to returning only entity results, although it is expected that this will be the primary use case. Consider the following query:

```
SELECT e.name AS emp_name, m.name AS manager_name
FROM emp e,
     emp m
WHERE e.manager_id = m.emp_id (+)
START WITH e.manager_id IS NULL
CONNECT BY PRIOR e.emp_id = e.manager_id
```

Non-entity result types, called scalar result types, are mapped using the @ColumnResult annotation. One or more column mappings may be assigned to the columns attribute of the mapping annotation. The only attribute available for a column mapping is the column name. Listing 9-8 shows the SQL mapping for the employee and manager hierarchical query.

Listing 9-8. *Scalar Column Mappings*

```
@SqlResultSetMapping(
    name="EmployeeAndManager",
    columns={@ColumnResult(name="EMP_NAME"),
             @ColumnResult(name="MANAGER_NAME")}
)
```


Scalar results may also be mixed with entities. In this case the scalar results are typically providing additional information about the entity. Let's look at a more complex example where this would be the case. A report for an application needs to see information about each department, showing the manager, the number of employees, and the average salary. The following JPQL query produces the correct report:

```
SELECT d, m, COUNT(e), AVG(e.salary)
FROM Department d LEFT JOIN e.employees e
                  LEFT JOIN e.employees m
WHERE m IS NULL OR m IN (SELECT de.manager
                         FROM Employee de
                         WHERE de.department = d)
GROUP BY d, m
```

This query is particularly challenging, because there is no direct relationship from Department to the Employee who is the manager of the department. Therefore the employees relationship must be joined twice, once for the employees assigned to the department and once for the employee in that group who is also the manager. This is possible because the subquery reduces the second join of the employees relationship to a single result. We also need to accommodate the fact that there might not be any employees currently assigned to the department and further that a department might not have a manager assigned. This means that each of the joins must be an outer join and that we further have to use an OR condition to allow for the missing manager in the WHERE clause.

Once in production, it is determined that the SQL query generated by the provider is not performing well, so the DBA proposes an alternate query that takes advantage of the inline views possible with the Oracle database. The query to accomplish this result is shown in Listing 9-9.

Listing 9-9. *Department Summary Query*

```
SELECT d.id, d.name AS dept_name,
       e.emp_id, e.name, e.salary, e.manager_id, e.dept_id, e.address_id,
       s.tot_emp, s.avg_sal
FROM dept d,
     (SELECT *
      FROM emp e
      WHERE EXISTS(SELECT 1 FROM emp WHERE manager_id = e.emp_id)) e,
     (SELECT d.id, COUNT(*) AS tot_emp, AVG(e.salary) AS avg_sal
      FROM dept d, emp e
      WHERE d.id = e.dept_id (+)
      GROUP BY d.id) s
WHERE d.id = e.dept_id (+) AND
      d.id = s.id
```

Fortunately, mapping this query is a lot easier than reading it. The query results consist of a Department entity, an Employee entity, and two scalar results, the number of the employees and the average salary. Listing 9-10 shows the mapping for this query.

Listing 9-10. *Mapping for the Department Query*

```
@SqlResultSetMapping(
    name="DepartmentSummary",
    entities={
        @EntityResult(entityClass=Department.class,
                      fields=@FieldResult(name="name", column="DEPT_NAME")),
        @EntityResult(entityClass=Employee.class)
    },
    columns={@ColumnResult(name="TOT_EMP"),
             @ColumnResult(name="AVG_SAL")}
)
```

Mapping Compound Keys

When a primary or foreign key is comprised of multiple columns that have been aliased to unmapped names, then a special notation must be used in the `@FieldResult` annotations to identify each part of the key. Consider the query shown in Listing 9-11 that returns both the employee and the manager of the employee. The table in this example is the same one we demonstrated in Figure 8-6 of Chapter 8. Because each column is repeated twice, the columns for the manager state have been aliased to new names.

Listing 9-11. *SQL Query Returning Employee and Manager*

```
SELECT e.country, e.emp_id, e.name, e.salary,
       e.manager_country, e.manager_id, m.country AS mgr_country,
       m.emp_id AS mgr_id, m.name AS mgr_name, m.salary AS mgr_salary,
       m.manager_country AS mgr_mgr_country, m.manager_id AS mgr_mgr_id
FROM   emp e,
       emp m
WHERE  e.manager_country = m.country AND
       e.manager_id = m.emp_id
```

The result set mapping for this query depends on the type of primary key class used by the target entity. Listing 9-12 shows the mapping in the case where an id class has been used. For the primary key, each attribute is listed as a separate field result. For the foreign key, each primary key attribute of the target entity (the `Employee` entity again in this example) is suffixed to the name of the relationship attribute.

Listing 9-12. *Mapping for Employee Query Using Id Class*

```
@SqlResultSetMapping(
    name="EmployeeAndManager",
    entities={
        @EntityResult(entityClass=Employee.class),
        @EntityResult(
            entityClass=Employee.class,
```



```
            fields={
                @FieldResult(name="country", column="MGR_COUNTRY"),
                @FieldResult(name="id", column="MGR_ID"),
                @FieldResult(name="name", column="MGR_NAME"),
                @FieldResult(name="salary", column="MGR_SALARY"),
                @FieldResult(name="manager.country", column="MGR_MGR_COUNTRY"),
                @FieldResult(name="manager.id", column="MGR_MGR_ID")
            }
        )
    }
)
```

If `Employee` uses an embedded id class instead of an id class, the notation is slightly different. We have to include the primary key attribute name as well as the individual attributes within the embedded type. Listing 9-13 shows the result set mapping using this notation.

Listing 9-13. *Mapping for Employee Query Using Embedded Id Class*

```
@SqlResultSetMapping(
    name="EmployeeAndManager",
    entities={
        @EntityResult(entityClass=Employee.class),
        @EntityResult(
            entityClass=Employee.class,
            fields={
                @FieldResult(name="id.country", column="MGR_COUNTRY"),
                @FieldResult(name="id.id", column="MGR_ID"),
                @FieldResult(name="name", column="MGR_NAME"),
                @FieldResult(name="salary", column="MGR_SALARY"),
                @FieldResult(name="manager.id.country", column="MGR_MGR_COUNTRY"),
                @FieldResult(name="manager.id.id", column="MGR_MGR_ID")
            }
        )
    }
)
```

Mapping Inheritance

In many respects, polymorphic queries in SQL are no different than regular queries returning a single entity type. All columns must be accounted for, including foreign keys and the discriminator column for single-table and joined inheritance strategies. The key thing to remember is that if the results include more than one entity type, then each of the columns for all of the possible entity types must be represented in the query. The field result mapping techniques we demonstrated earlier may be used to customize columns that use unknown aliases. These columns may be at any level in the inheritance tree. The only special element in the `@EntityResult` annotation for use with inheritance is the `discriminatorColumn` element. This element allows the name of the discriminator column to be specified in the unlikely event that it is different from the mapped version.

Assume that the Employee entity had been mapped to the table shown in Figure 8-14 from Chapter 8. To understand aliasing a discriminator column, consider the following query that returns data from another EMPLOYEE_STAGE table structured to use single-table inheritance:

```
SELECT id, name, start_date, daily_rate, term, vacation,
       hourly_rate, salary, pension, type
FROM employee_stage
```

To convert the data returned from this query to Employee entities, the following result set mapping would be used:

```
@SqlResultSetMapping(
    name="EmployeeStageMapping",
    entities=
        @EntityResult(
            entityClass=Employee.class,
            discriminatorColumn="TYPE",
            fields={
                @FieldResult(name="startDate", column="START_DATE"),
                @FieldResult(name="dailyRate", column="DAILY_RATE"),
                @FieldResult(name="hourlyRate", column="HOURLY_RATE")
            }
        )
)
```

Parameter Binding

SQL queries have traditionally supported only positional parameter binding. The JDBC specification itself did not introduce support for named parameters until version 3.0, and not all database vendors support this syntax. As a result, the Java Persistence API guarantees only the use of positional parameter binding for SQL queries. Check with your vendor to see if the named parameter methods of the Query interface are supported, but understand that using them may make your application non-portable between persistence providers.

Another limitation of parameter support for SQL queries is that entity parameters may not be used. The specification does not define how these parameter types should be treated. Be careful when converting or overriding a named JPQL query with a native SQL query that the parameter values are still interpreted correctly.

Lifecycle Callbacks

Every entity has the potential to go through one or more of a defined set of *lifecycle events*. Depending upon the operations invoked upon an entity, these events may or may not occur for that entity, but there is at least the potential for them to occur. In order to respond to any one or more of the events, an entity class or any of its superclasses may declare one or more methods that will be invoked by the provider when the event gets fired. These methods are called *callback methods*.

Lifecycle Events

The event types that make up the lifecycle fall into four categories: persisting, updating, removing, and loading. These are really data-level events that correspond to the database operations of inserting, updating, deleting, and reading, and except for loading, each has a "Pre" event and a "Post" event. In the load category there is only a PostLoad event, since it would not make any sense for there to be PreLoad on an entity that was not yet built. Thus the full suite of lifecycle events that can occur is composed of: PrePersist, PostPersist, PreUpdate, PostUpdate, PreRemove, PostRemove, and PostLoad.

PrePersist and PostPersist

The PrePersist event notifies an entity when persist() has been successfully invoked on it. PrePersist events may also occur on a merge() call when a new entity has been merged into the persistence context. If the PERSIST cascade option is set on a relationship of an object that is being persisted and the target object is also a new object, then the PrePersist event is triggered on the target object. If multiple entities are cascaded to during the same operation, then the order in which the PrePersist callbacks occur cannot be relied upon.

PostPersist events occur when an entity is inserted, which normally occurs during the transaction completion phase. Firing of a PostPersist event does not indicate that the entity has committed successfully to the database, since the transaction in which it was persisted may be subsequently rolled back.

PreRemove and PostRemove

When a remove() call is invoked on an entity, the PreRemove callback is triggered. This callback implies that an entity is being queued for deletion, and any related entities across relationships that have been configured with the REMOVE cascade option will also get a PreRemove notification. When the SQL for deletion of an entity finally does get sent to the database, the PostRemove event will get fired. As with the PostPersist lifecycle event, the PostRemove event does not guarantee success. The enclosing transaction may still be rolled back.

PreUpdate and PostUpdate

Updates to managed entities may occur at any time, either within a transaction, or in the case of an extended persistence context, outside a transaction. Because there is no explicit method on the EntityManager, the PreUpdate callback is guaranteed to be invoked only at some point before the database update. Some implementations may track changes dynamically and may invoke the callback on each change, while others may wait until the end of the transaction and just invoke the callback once.

Another difference between implementations is whether PreUpdate events get fired on entities that were persisted in a transaction and then modified in the same transaction before being committed. This would be a rather unfortunate choice because unless the writes were done eagerly on each entity call, there would be no symmetric PostUpdate call, since in the usual deferred writing case, a single persist to the database would occur when the transaction ends. The PostUpdate callback occurs right after the database update. The same potential rollback exists after PostUpdate callbacks as exist with PostPersist and PostRemove.

PostLoad

The `PostLoad` callback occurs after the data for an entity is read from the database and the entity instance is constructed. This can get triggered by any operation that causes an entity to be loaded, normally by either a query or traversal of a lazy relationship. It can also happen as a result of a `refresh()` call on the entity manager. When a relationship is set to cascade `REFRESH`, then the entities that get cascaded to will also get loaded. The order of invocation of entities in a single operation, be it a query or a refresh, is not guaranteed to be in any order, so we should not rely upon any observed order in any implementation.

Callback Methods

Callback methods may be defined a few different ways, the most basic of which is to simply define a method on the entity class. Designating the method as a callback method involves two steps: defining the method according to a given signature, and annotating the method with the appropriate lifecycle event annotation.

The required signature definition is very simple. The callback method may have any name but must have a signature that takes no parameters and has a return type of `void`. A method like `public void foo() {}` is an example of a valid method. Final or static methods are not valid callback methods, however.

Checked exceptions may not be thrown from callback methods, because the method definition of a callback method is not permitted to include a throws clause. Runtime exceptions may be thrown, though, and if they are thrown while in a transaction, they will cause the provider to not only abandon invocation of subsequent lifecycle event methods in that transaction but also mark the transaction for rollback.

A method is indicated as being a callback method by being annotated with a lifecycle event annotation. The relevant annotations match the names of the events listed earlier: `@PrePersist`, `@PostPersist`, `@PreUpdate`, `@PostUpdate`, `@PreRemove`, `@PostRemove`, and `@PostLoad`. A method may be annotated with multiple lifecycle event annotations, but only one lifecycle annotation of a given type may be present in an entity class.

Certain types of operations may not be portably performed inside callback methods. For example, invoking on an entity manager or executing queries obtained from an entity manager are not supported, as well as accessing entities other than the one to which the lifecycle event applies. Looking up resources in JNDI or using JDBC and JMS resources are allowed, so looking up and invoking EJB session beans is also allowed.

Now that we know all of the different kinds of lifecycle events that we can handle, let's look at an example that uses them. One common usage of lifecycle events is to maintain non-persistent state inside a persistent entity. If we want the entity to record its cached age or the time it was last synchronized with the database, then we could easily do this right inside the entity using callback methods. We consider that the entity is synchronized with the database each time it is read from or written to the database. Users of this `Employee` could check on the cached age of this object to see if it meets their freshness requirements. The entity is shown in Listing 9-14.

Listing 9-14. *Using Callback Methods on an Entity*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @Transient private long syncTime;

    // ...

    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() {
        syncTime = System.currentTimeMillis();
    }

    public long getCachedAge() {
        return System.currentTimeMillis() - syncTime;
    }

    // ...
}
```

Enterprise Contexts

When a callback method is invoked, the provider will not take any particular action to suspend or establish any different kind of naming, transaction, or security context in the Java EE environment. Callback methods are executed in whatever contexts are active at the time they are invoked.

Remembering this fact is important, because it will most often be a session bean with a container-managed transaction that invokes calls on the entity manager, and it will be that session bean's contexts that will be in effect when the "Pre" calls are invoked. Depending upon where the transaction started and is committed, the "Post" calls will likely be invoked at the end of the transaction and could actually be in an entirely different set of contexts than the "Pre" methods. This is especially true in the case of an extended persistence context where the entities are managed and persisted outside a transaction, yet the next transaction commit will cause the entities that were persisted to be written out.

Entity Listeners

Callback methods in the entity are fine when you don't mind if the event callback logic is included in the entity, but what if you want to pull the event handling behavior out of the entity class into a different class? To do this you can use an *entity listener*. An entity listener is not an entity but is a class on which you can define one or more lifecycle callback methods to be invoked for the lifecycle events of an entity. Like the callback methods on the entity, however, each event type may have only one method annotated to be invoked.

When the callback is invoked on a listener, the listener typically needs to have access to the entity state. For example, if we were to implement the previous example of the cached age of an entity instance then we would want to get passed the entity instance. For this reason, the signature required of callback methods on entity listeners is slightly different than the one required on entities. On an entity listener, a callback method must have a similar signature as on an entity with the exception that it must also have a single defined parameter of a type that is compatible with the entity type, either as the entity class, a superclass (including `Object`) or an interface implemented by the entity. A method with the signature `public void foo(Object o) {}` is an example of a valid callback method on an entity listener. The method must then be annotated with the necessary event annotation(s).

Entity listener classes must be stateless, meaning that they should not declare any fields. A single instance may be shared amongst multiple entity instances and may even be invoked upon concurrently for multiple entity instances. In order for the provider to be able to create instances of the entity listener, every entity listener class must have a public no-argument constructor.

Attaching Entity Listeners to Entities

An entity designates the entity listeners that should be notified of its lifecycle events through the use of the `@EntityListeners` annotation. One or more entity listeners may be listed in the annotation. When a lifecycle event occurs, the provider will iterate through each of the entity listeners in the order in which they were listed and instantiate an instance of the entity listener class that has a method annotated with the annotation for the given event. It will invoke the callback method on the listener, passing in the entity to which the event applies. After it has done this for all of the listed entity listeners, then it will invoke the callback method on the entity if there is one. If any of the listeners throws an exception, then it will abort the callback process, causing the remaining listeners and the callback method on the entity to not be invoked.

Now let's look at our cached entity age example and add some entity listeners into the mix. Because we now have the ability to do multiple tasks in multiple listeners, we can add a listener to do some name validation as well as some extra actions on employee record changes. Listing 9-15 shows the entity with its added listeners.

Listing 9-15. *Using Multiple Entity Listeners*

```
@Entity
@EntityListeners({EmployeeDebugListener.class, NameValidator.class})
public class Employee implements NamedEntity {
    @Id private int id;
    private String name;
    @Transient private long syncTime;

    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() {
        syncTime = System.currentTimeMillis();
    }
```



```
    public long getCachedAge() {
        return System.currentTimeMillis() - syncTime;
    }

    // ...
}

public interface NamedEntity {
    public String getName();
}

public class NameValidator {
    static final int MAX_NAME_LEN = 40;

    @PrePersist
    public void validate(NamedEntity obj) {
        if (obj.getName().length()) > MAX_NAME_LEN)
            throw new ValidationException("Identifier out of range");
    }
}

public class EmployeeDebugListener {
    @PrePersist
    public void prePersist(Employee emp) {
        System.out.println("Persist on employee id: " + emp.getId());
    }

    @PreUpdate
    public void preUpdate(Employee emp) { ... }

    @PreRemove
    public void preRemove(Employee emp) { ... }

    @PostLoad
    public void postLoad(Employee emp) { ... }
}
```

As we can see, different listener callback methods take different types of parameters. The callback methods in the `EmployeeDebugListener` class take `Employee` as a parameter because they are being applied only to `Employee` entities. In the `NameValidator` class, the `validate()` method parameter is of type `NamedEntity`. The `Employee` entity and any number of other entities that have names may implement this interface. The validation logic may be needed because a particular aspect of the system may have a current name length limitation but may change in the future. It is preferable to centralize this logic in a single class than to duplicate the validation logic in each of the class setter methods if there is possibility of an inheritance hierarchy.

Even though entity listeners are convenient, we have decided to leave the cache age logic in the entity, because it is actually modifying the state of the entity and because putting it in a separate class would have required us to relax the access of the private `resetSyncTime()` method.

In general, when a callback method accesses state beyond what should be publicly accessible, then it is best suited to being on the entity and not in an entity listener.

Default Entity Listeners

A listener may be attached to more than one type of entity simply by being listed in the `@EntityListeners` annotation of more than one entity. This can be useful in cases where the listener provides a more general facility or wide-ranging runtime logic.

For even broader usage of an entity listener across all of the entities in a persistence unit, one or more *default entity listeners* may be declared. There is currently no standard annotation target for persistence unit scoped metadata, so this kind of metadata can be declared only in an XML mapping file. See Chapter 10 for the specifics of how to declare default entity listeners.

When a list of default entity listeners is declared, then it will be traversed in the order they were listed in the declaration, and each one that has a method annotated or declared for the current event will be invoked upon. Default entity listeners will always get invoked before any of the entity listeners listed in the `@EntityListeners` annotation for a given entity.

Any entity may opt out of having the default entity listeners applied to it by using the `@ExcludeDefaultListeners` annotation. When an entity is annotated with this annotation, then none of the declared default listeners will get invoked for the lifecycle events for instances of that entity type.

Inheritance and Lifecycle Events

The presence of events with class hierarchies requires that we explore the topic of lifecycle events in a little more depth. What happens when we have multiple entities that each define callback methods or entity listeners or both? Do they all get invoked on a subclass entity or only those that are defined on or in the subclass entity?

These and many other questions arise because of the added complexity of inheritance hierarchies. It follows that there must be rules for defining predictable behavior in the face of potentially complex hierarchies where lifecycle event methods are scattered throughout the hierarchy.

Inheriting Callback Methods

Callback methods may occur on any entity or mapped superclass, be it abstract or concrete. The rule is fairly simple. It is that every callback method for a given event type will be invoked in the order according to its place in the hierarchy, most general classes first. Thus, if in our `Employee` hierarchy that we saw in Figure 8-13 the `Employee` class contains a `PrePersist` callback method named `checkName()` and `FullTimeEmployee` also contains a `PrePersist` callback method named `verifyPension()`, then when the `PrePersist` event occurs, the `checkName()` method will get invoked followed by the `verifyPension()` method.

We could also have a method on the `CompanyEmployee` mapped superclass that we want to apply to all of the entities that subclassed it. If we add a `PrePersist` method named `checkVacation()` that verifies that the vacation carryover is less than a certain amount, then it will be executed after `checkName()` and before `verifyPension()`.

It gets more interesting if we define a `checkVacation()` method on the `PartTimeEmployee` class, because part-time employees don't get as much vacation. Annotating the overridden method with `PrePersist` would cause the `PartTimeEmployee.checkVacation()` method to be invoked instead of the one in `CompanyEmployee`.

Inheriting Entity Listeners

Like callback methods in an entity, the `@EntityListeners` annotation is also valid on entities or mapped superclasses in a hierarchy, whether they are concrete or abstract. Also similar to callback methods, the listeners listed in the entity superclass annotation get invoked before the listeners in the subclass entities. In other words, defining an `@EntityListeners` annotation on an entity is additive in that it only adds listeners; it does not redefine them or their order of invocation. Any listener defined on a superclass of the entity will be invoked before the listeners defined on the entity.

To redefine which entity listeners get invoked and their order of invocation, an entity or mapped superclass should be annotated with `@ExcludeSuperclassListeners`. This will cause the listeners defined in all of the superclasses to not be invoked for any of the lifecycle events of the annotated entity subclass. If we want a subset of the listeners to still be invoked, then they must be listed in the `@EntityListeners` annotation on the overriding entity and in the order that is appropriate.

Lifecycle Event Invocation Order

The rules for lifecycle event invocation are now a little more complex, so they warrant being laid out more carefully. Perhaps the best way to describe it is to outline the process that the provider must follow to invoke the event methods. If a given lifecycle event X occurs for entity A, the provider will do the following:

1. Check whether any default entity listeners exist (see Chapter 10). If they do, then iterate through them in the order they are defined and look for methods that are annotated with the lifecycle event X annotation. Invoke the lifecycle method on the listener if a method was found.
2. Check on the highest mapped superclass or entity in the hierarchy for classes that have an `@EntityListeners` annotation. Iterate through the entity listener classes that are listed in the annotation and look for methods that are annotated with the lifecycle event X annotation. Invoke the lifecycle method on the listener if a method was found.
3. Repeat step 2 going down the hierarchy on entities and mapped superclasses until entity A is reached, and then repeat it for entity A.
4. Check on the highest mapped superclass or entity in the hierarchy for methods that are annotated with the lifecycle event X annotation. Invoke the callback method on the entity if a method was found and the method is not also defined in entity A with the lifecycle event X annotation on it.
5. Repeat step 2 going down the hierarchy on entities and mapped superclasses until entity A is reached.
6. Invoke any methods that are defined on A and annotated with the lifecycle event X annotation.

This process might be easier to follow if we have code that includes these cases and we go through the order in which they are executed. Listing 9-16 shows our entity hierarchy with a number of listeners and callback methods on it.

Listing 9-16. *Using Entity Listeners and Callback Methods in a Hierarchy*

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@EntityListeners(NameValidator.class)
public class Employee implements NamedEntity {
    @Id private int id;
    private String name;
    @Transient private long syncTime;

    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() { syncTime = System.currentTimeMillis(); }
    // ...
}

public interface NamedEntity {
    public String getName();
}

@Entity
@ExcludeSuperclassListeners
@EntityListeners(LongNameValidator.class)
public class ContractEmployee extends Employee {
    private int dailyRate;
    private int term;

    @PrePersist
    public void verifyTerm() { ... }
    // ...
}

@MappedSuperclass
@EntityListeners(EmployeeAudit.class)
public abstract class CompanyEmployee extends Employee {
    protected int vacation;
    // ...

    @PrePersist
    @PreUpdate
    public void verifyVacation() { ... }
}
```



```
@Entity
public class FullTimeEmployee extends CompanyEmployee {
    private long salary;
    private long pension;
    // ...
}

@Entity
@EntityListeners({})
public class PartTimeEmployee extends CompanyEmployee {
    private float hourlyRate;
    // ...

    @PrePersist
    @PreUpdate
    public void verifyVacation() { ... }
}

public class EmployeeAudit {
    @PostPersist
    public void auditNewHire(CompanyEmployee emp) { ... }
}

public class NameValidator {
    @PrePersist
    public void validateName(NamedEntity obj) { ... }
}

public class LongNameValidator {
    @PrePersist
    public void validateLongName(NamedEntity obj) { ... }
}

public class EmployeeDebugListener {
    @PrePersist
    public void prePersist(Employee emp) {
        System.out.println("Persist called on: " + emp);
    }

    @PreUpdate
    public void preUpdate(Employee emp) { ... }

    @PreRemove
    public void preRemove(Employee emp) { ... }

    @PostLoad
    public void postLoad(Employee emp) { ... }
}
```

We have a pretty complex example here to study, and the easiest way to make use of it is to say what happens when a given event occurs for a specific entity. We will assume that the `EmployeeDebugListener` class has been set in the XML mapping file as a default entity listener for all entities.

Let's see what happens when we create a new instance of `PartTimeEmployee` and pass it to `em.persist()`. Since the first step is always to invoke the default listeners and our default listener does indeed have a `PrePersist` method on it, the `EmployeeDebugListener.prePersist()` method will be invoked first.

The next step would be to traverse down the hierarchy looking for entity listeners. The first class we find is the `Employee` class, which defines a `NameValidator` entity listener. The `NameValidator` class does define a `PrePersist` method, so the next method to get executed would be `NameValidator.validateName()`. The next class we hit moving down the hierarchy is the `CompanyEmployee` class. This class defines an `EmployeeAudit` listener that does not happen to have a `PrePersist` method on it, so we skip past it.

Next we get to the `PartTimeEmployee` class that has an `@EntityListeners` annotation but does not define any listeners. This is essentially a false alarm that does not really override anything, but is simply a no-op in terms of adding listeners (probably a leftover of a listener that was once there but has since been removed).

The next phase in the process is to start looking for callback methods on entities and mapped superclasses. Once again we start at the top of the hierarchy and look at the `Employee` class to see if a `PrePersist` method exists, but none does. We have `PostPersist` and others, but no `PrePersist`. We continue on down to `CompanyEmployee` and see a `PrePersist` method called `verifyVacation()`, but looking down on the `PartTimeEmployee` entity we find that the method has been overridden by a `verifyVacation()` method there that also has an `@PrePersist` annotation on it. This is a case of overriding the callback method and will result in the `PartTimeEmployee.verifyVacation()` method being called instead of the `CompanyEmployee.verifyVacation()` method. We are finally done, and the entity will be persisted.

The next event might then be a `PostPersist` event on the same entity at commit time. This will bypass the default listener since there is no `PostPersist` method in `EmployeeDebugListener` and also bypass the `NameValidator` since there is no `PostPersist` event method there either. The next listener that it tries will be the `EmployeeAudit` listener class, which does include a `PostPersist` method called `auditNewHire()`, which will then get invoked. There are no more listeners to examine, so we move on to the callback methods and find the `resetSyncTime()` method in `Employee`. This one gets called, and since we find no more `PostPersist` callback methods in the hierarchy, we are done.

The next thing we can try is persisting a `ContractEmployee`. This is a simple persistence structure with only the `Employee` and `ContractEmployee` entities in it. When we create a `ContractEmployee` and the `PrePersist` event gets triggered, we first get our default `EmployeeDebugListener.prePersist()` callback and then move on to processing the entity listeners. The curve is that the `@ExcludeSuperclassListeners` annotation is present on the `ContractEmployee`, so the `NameValidator.validateName()` method that would otherwise have been invoked will not be considered. We instead go right to the `@EntityListeners` annotation on the `ContractEmployee` class and find that we need to look at `LongNameValidator`. When we do, we find that it has a `validateLongName()` method on it that we execute and then go on to executing the callback methods. There are callback methods in both classes in the hierarchy, and the `Employee.resetSyncTime()` method gets invoked first, followed by the `ContractEmployee.verifyTerm()` method.

Concurrency

The concurrency of entity access and entity operations is not heavily specified, but there are a few rules that dictate what we can and can't expect. We will go over these and leave the rest to the vendors to explain in the documentation for their respective implementations.

Entity Operations

A managed entity belongs to a single persistence context and should not be managed by more than one persistence context at any given time. This is an application responsibility, however, and may not necessarily be enforced by the persistence provider. Merging the same entity into two different open persistence contexts could produce undefined results.

Entity managers and the persistence contexts that they manage are not intended to be accessed by more than one concurrently executing thread. The application cannot expect it to be synchronized and is responsible for ensuring that it stays within the thread that obtained it.

Entity Access

Applications may not access an entity directly from multiple threads while it is managed by a persistence context. An application may choose, however, to allow entities to be accessed concurrently when they are detached. If it chooses to do so, the synchronization must be controlled through the methods coded on the entity. Concurrent entity state access is not recommended, however, since the entity model does not lend itself well to concurrent patterns. It would be preferable to simply copy the entity and pass the copied entity to other threads for access and then merge any changes back into a persistence context when they need to be persisted.

Refreshing Entity State

The `refresh()` method of the `EntityManager` interface can be useful in situations when we know or suspect that there are changes in the database that we do not have in our managed entity. The refresh operation applies only when an entity is managed, since when we are detached we typically only need to issue a query to get an updated version of the entity from the database.

Refreshing makes more sense the longer the duration of the persistence context that contains it. Refreshing is especially relevant when using an extended or application-managed persistence context, since it prolongs the interval of time that an entity is effectively cached in the persistence context in isolation from the database.

To refresh a managed entity, we simply call `refresh()` on the entity manager. If the entity that we try to refresh is not managed, then an `IllegalArgumentException` exception will be thrown. To clarify some of the issues around the refresh operation, we will use the example session bean shown in Listing 9-17.

Listing 9-17. *Periodic Refresh of a Managed Entity*

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class EmployeeServiceBean implements EmployeeService {
    public static final long REFRESH_THRESHOLD = 300000;

    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Employee emp;
    long loadTime;

    public void loadEmployee (int id) {
        emp = em.find(Employee.class, id);
        if (emp == null)
            throw new IllegalArgumentException("Unknown employee id: " + id);
        loadTime = System.currentTimeMillis();
    }

    public void deductEmployeeVacation(int days) {
        refreshEmployeeIfNeeded();
        emp.setVacationDays(emp.getVacationDays() - days);
    }

    public void adjustEmployeeSalary(long salary) {
        refreshEmployeeIfNeeded();
        emp.setSalary(salary);
    }

    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void finished() {}

    private void refreshEmployeeIfNeeded() {
        if ((System.currentTimeMillis() Ð loadTime) > REFRESH_THRESHOLD) {
            em.refresh(emp);
            loadTime = System.currentTimeMillis();
        }
    }

    // ...
}
```

The stateful session bean in Listing 9-17 uses an extended persistence context in order to keep an `Employee` instance managed while various operations are applied to it via the business methods of the session bean. It might allow a number of modifying operations on it before it commits the changes, but we need to include only a couple of operations for this example.

Let's look at this bean in detail. The first thing to notice is that the default transaction attribute has been changed from `REQUIRED` to `NOT_SUPPORTED`. This means that as the `Employee` instance is changed by the various business methods of the bean, those changes will not be written to the database. This will occur only when the `finished()` method is invoked, which has a transaction attribute of `REQUIRED`. This is the only method on the bean that will associate the extended persistence context with a transaction and cause it to be synchronized with the database.

The second interesting thing about this bean is that it stores the time the `Employee` instance was last accessed from the database. Since the stateful session bean instance may exist for a long time, the business methods use the `refreshEmployeeIfNeeded()` method to see if it has been too long since the `Employee` instance was last refreshed. If the refresh threshold has been reached, the `refresh()` method is used to update the `Employee` state from the database.

Unfortunately, the refresh operation does not behave as the author of the session bean expected. When refresh is invoked, it will overwrite the managed entity with the state in the database, causing any changes that have been made to the entity to be lost. For example, if the salary is adjusted and five minutes later the vacation is adjusted, then the employee will get refreshed, causing the previous change to the salary to be lost. It turns out that although the example in Listing 9-17 does indeed do a periodic refresh of the managed entity, the result is not only an inappropriate use of `refresh()` but also a detrimental result to the application.

So when is refreshing valid for objects that we are modifying? The answer is, not as often as you think. One of the primary use cases is to "undo" or discard changes made in the current transaction, reverting them back to their original value. It may also be used in long-lived persistence contexts where read-only managed entities are being cached. In these scenarios, the `refresh()` operation can safely restore an entity to its currently recorded state in the database. This would have the effect of picking up changes made in the database since the entity had been last loaded into the persistence context. The stipulation is that the entity should be read-only or be guaranteed to not contain changes.

Recall our editing session in Listing 5-33. Using `refresh()`, we can add the ability to revert an entity when the user decides to cancel their changes to an `Employee` editing session. Listing 9-18 shows the bean with its additional `revertEmployee()` method.

Listing 9-18. *Employee Editing Session with Revert*

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class EmployeeEditBean implements EmployeeEdit {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
    Employee emp;

    public void begin(int id) {
        emp = em.find(Employee.class, id);
        if (emp == null) {
            throw new IllegalArgumentException("Unknown employee id: " + id);
        }
    }
```

```
    public Employee getEmployee() { return emp; }

    public Employee revertEmployee() {
        em.refresh(emp);
        return emp;
    }

    @Remove
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void save() {}

    @Remove
    public void cancel() {}
}
```

Refresh operations may also be cascaded across relationships. This is done on the relationship annotation by setting the `cascade` element to include the `REFRESH` value. If the `REFRESH` value is not present in the `cascade` element, then the refresh will stop at the source entity. Listing 9-19 demonstrates how to set the `REFRESH` cascade operation for a many-to-one relationship.

Listing 9-19. *Cascading a Refresh Operation*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    @ManyToOne(cascade={CascadeType.REFRESH})
    private Employee manager;
    // ...
}
```

Locking

Locking surfaces at many different levels is intrinsic to the Java Persistence API. It is used and assumed at various points throughout the API and the specification. Whether your application is simple or complex, chances are that you will make use of locking somewhere along the way.

Optimistic Locking

When we talk about locking we are generally referring to optimistic locking,[1] which is essentially just what its name implies, that is, a model that takes an optimistic approach to locking the entity. The optimistic locking model subscribes to the philosophy that there is a good

1. Pessimistic locking means to eagerly obtain a lock on the resource before operating on it. This is typically very resource-restrictive and results in significant performance degradation. The Java Persistence API does not currently support a portable mechanism of pessimistic locking.

chance that the transaction in which changes are made to an entity will be the only one that actually changes the entity during that interval. This translates into the decision to not acquire a lock on the entity until the change is actually made to the database, usually at the end of the transaction.

When the data actually does get sent to the database to get updated at flush time or at the end of the transaction, then the entity lock is acquired and a check is made on the data in the database. The flushing transaction must see whether any other transaction has committed a change to the entity in the intervening time since this transaction read it in and changed it. If a change occurred, then it means that the flushing transaction has data that does not include those changes and should not write its own changes to the database lest it overwrite the changes from the intervening transaction. At this stage it must roll back the transaction and throw a special exception called `OptimisticLockException`. The example in Listing 9-20 shows how this could happen.

Listing 9-20. *Method That Adjusts Vacation Balance*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public void deductEmployeeVacation(int id, int days) {
        Employee emp = em.find(Employee.class, id);
        int currentDays = emp.getVacationDays();
        // Do some other stuff like notify HR system, etc.
        // ...
        emp.setVacationDays(currentDays Ð days);
    }
}
```

While this method may seem harmless enough, it is really just an accident waiting to happen. The problem is as follows. Imagine that two HR data-entry operators, Frank and Betty, were charged with entering a backlog of vacation adjustments into the system and they both happened to be entering an adjustment for the employee with id 14 at the same time. Frank is supposed to deduct 1 day from employee 14, while Betty is deducting 12 days. Frank's console calls `deductEmployeeVacation()` first, which immediately reads employee 14 in from the database, finds that employee 14 has 20 days, and then proceeds into the HR notification step. Meanwhile, Betty starts to enter her data on her console, which also calls `deductEmployeeVacation()`. It also reads employee 14 in from the database and finds that the employee has 20 vacation days, but Betty happens to have a much faster connection to the HR system. As a result Betty gets past the HR notification before Frank does and proceeds to set the vacation day count to 8 before committing her transaction and going on to the next item. Frank finally gets past the HR system notification and deducts 1 day from the 20, and then commits his transaction. If Frank commits, then he has overwritten Betty's deduction and employee 14 gets an extra 12 days of vacation.

Instead of committing Frank's transaction, though, an optimistic locking strategy would find out when it was time to commit that someone else had changed the vacation count. When Frank attempted to commit his transaction, an `OptimisticLockException` would have been

thrown, and his transaction would have been rolled back instead. The result is that Frank would have to reenter his change and try again, which is far superior to getting an incorrect result for employee 14.

Versioning

The question that you might have been asking is how the provider can know if somebody made changes in the intervening time since the committing transaction read the entity? The answer is that the provider maintains a versioning system for the entity. In order for it to do this, the entity must have a dedicated persistent field or property declared in it to store the version number of the entity that was obtained in the transaction. The version number must also be stored in the database. When going back to the database to update the entity, the provider can check the version of the entity in the database to see if it matches the version that it obtained previously. If the version in the database is the same, then the change can be applied and everything goes on without any problems. If the version was greater, then somebody else changed the entity since it was obtained in the transaction, and an exception should be thrown. The version field will get updated both in the entity and in the database whenever an update to the entity is sent to the database.

Version fields are not required, but we recommend that version fields be in every entity that has any chance of being concurrently modified by more than one process. A version column is an absolute necessity whenever an entity gets modified as a detached entity and merged back into a persistence context again afterwards. The longer an entity stays in memory, the higher the chance that it will be changed in the database by another process, rendering the in-memory copy invalid. Version fields are at the core of optimistic locking and provide the best and most performant protection for infrequent concurrent entity modification.

Version fields are defined simply by annotating the field or property on the entity with an `@Version` annotation. In Listing 9-21 is an `Employee` entity annotated to have a version field.

Listing 9-21. *Using a Version Field*

```
@Entity
public class Employee {
    @Id private int id;
    @Version private int version;
    // ...
}
```

Version-locking fields defined on the entity may be of type `int`, `Integer`, `short`, `Short`, `long`, `Long`, or `java.sql.Timestamp`. The most common practice is just to use `int` or one of the numeric types, but some legacy databases use timestamps.

Like the identifier, the application should not set or change the version field once the entity has been created. It may access it, though, for its own purposes if it wants to make use of the version number for some application-dependent reason.

■ **Tip** Some vendors do not require that the version field be defined and stored in the entity. Variations of storing it in the entity are storing it in a vendor-specific cache, or not storing anything at all but instead using field comparison. For example, a popular option is to compare some application-specified combination of the entity state in the database with the entity state being written and then use the results as criteria to decide whether state has been changed.

A final note about version fields is that they are not guaranteed to be updated, either in the managed entities or the database, as part of a bulk update operation. In fact, there should not be any managed entities before a bulk update. Some vendors offer support for automatic updating of the version field during bulk updates, but this cannot be portably relied upon. For those vendors that do not support automatic version updates, the entity version may be manually updated as part of the UPDATE statement, as exhibited by the following query:

```
UPDATE Employee e
SET e.salary = e.salary + 1000, e.version = e.version + 1
WHERE EXISTS (SELECT p
              FROM e.projects p
              WHERE p.name = 'Release2')
```

Additional Locking Strategies

By default, the Java Persistence API assumes what is defined in the ANSI/ISO SQL specification and known in transaction isolation parlance as *Read Committed* isolation. This standard isolation level simply guarantees that any changes made inside a transaction will not be visible to other transactions until the changing transaction has been committed. Optimistic locking works with Read Committed isolation to provide additional data-consistency checks in the face of interleaved writes. Satisfying tighter locking constraints than what optimistic locking offers requires that an additional locking strategy be used. To be portable, these strategies may be used only on entities with version fields.

Read Locking

The next level of transaction isolation is termed *Repeatable Read* and prevents the so-called non-repeatable read anomaly. This anomaly can be described a few different ways, but perhaps the simplest is to say that when a transaction queries for the same data twice in the same transaction, the second query returns a different version of the data than was returned the first time because another transaction modified it in the intervening time. Put another way, Repeatable Read isolation level means that once a transaction has accessed data and another transaction modifies that data, then at least one of the transactions must be prevented from committing. A *read lock* in the Java Persistence API provides this level of isolation.

To read-lock an entity, the `EntityManager.lock()` method must be invoked, passing the entity to lock and a lock mode of `LockModeType.READ`. Obviously this may be invoked only within a transaction, but in addition, the entity that is passed into the call must already be managed. The resulting lock will guarantee that both the transaction that obtains the entity read lock and any other that tries to change that entity instance will not both succeed. At least one will fail, but like the database isolation levels, which one fails depends upon the implementation.

The way read locking is implemented is entirely up to the provider. It may choose to be heavy-handed and obtain an eager write lock on the entity, in which case any other transaction that tries to change the entity will fail or block until the locking transaction completes. Often the provider will optimistically read-lock the object instead. This means that the provider will not actually go to the database for a lock when the `lock()` method is called. It will instead wait until the end of the transaction, and at commit time it will reread the entity to see if the entity has been changed since it was last read in the transaction. If it has not changed, then the read lock was honored, but if the entity has changed, then the gamble was lost and the transaction will be rolled back.

A corollary to this optimistic form of read-locking implementation is that it doesn't matter at which point `lock()` is actually invoked during the transaction. It may be invoked even right up until just before the commit, and the exact same results will be produced. All the `lock()` method does is flag the entity for being reread at commit time. It doesn't really matter when, during the transaction, the entity gets added to this list since the actual read operation will not occur until the end of the transaction. You can kind of think of the `lock()` call as being retroactive to the point at which the entity was read into the transaction to begin with since that is the point at which the version is read and recorded in the managed entity.

The quintessential case for using this kind of lock is when an entity has an intrinsic dependency on one or more other entities for consistency. There is often a relationship between the entities, but not always. To demonstrate this, think of a `Department` that has employees where we want to generate a salary report for a given set of departments and have the report indicate the salary expenditures of each department. We have a method called `generateDepartmentsSalaryReport()` that will iterate through the set of departments and use an internal method to find the total salary for each one. The method defaults to having a transaction attribute of `REQUIRED`, so it will be executed entirely within the context of a transaction. The code is in Listing 9-22.

Listing 9-22. *Department Salaries Report*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    // ...

    public SalaryReport generateDepartmentsSalaryReport(List<Integer> deptIds) {
        SalaryReport report = new SalaryReport();
        long total = 0;
        for (Integer deptId : deptIds) {
            long deptTotal = totalSalaryInDepartment(deptId);
            report.addDeptSalaryLine(deptId, deptTotal);
            total += deptTotal;
        }
        report.addSummarySalaryLine(total);
        return report;
    }

    protected long totalSalaryInDepartment(int deptId) {
        long total = 0;
        Department dept = em.find(Department.class, deptId);
        for (Employee emp : dept.getEmployees())
            total += emp.getSalary();
        return total;
    }

    public void changeEmployeeDepartment(int deptId, int empId) {
        Employee emp = em.find(Employee.class, empId);
        emp.getDepartment().removeEmployee(emp);
        Department dept = em.find(Department.class, deptId);
        dept.addEmployee(emp);
        emp.setDepartment(dept);
    }
    // ...
}
```


The report will get generated fine, but is it correct? What happens if an employee gets moved from one department to another during the time we are computing the total salary? For example, we make a request for a report on departments 10, 11, and 12. The request starts to generate the report for department 10. It finishes department 10 and moves on to department 11. As it is iterating through all of the employees in department 11, employee with id 50 in department 10 gets changed to be in department 12. Somewhere a manager invokes the `changeEmployeeDepartment()` method, the transaction commits, and employee 50 is changed to be in department 12. Meanwhile the report generator has finished department 11 and is now going on to generate a salary total for department 12. When it iterates through the employees it will find employee 50 even though it already counted that employee in department 10, so employee 50 will be counted twice. We did everything in transactions but we still got an inconsistent view of the employee data. Why?

The problem was in the fact that we did not lock any of the employee objects from being modified during our operation. We issued multiple queries and were vulnerable to viewing the same object with different state in it, which is the non-repeatable read phenomenon. We could fix it in a number of ways, one of which would be to set the database isolation to Repeatable Read. Since we are explaining the `lock()` method, we will use it to lock each of the employees so that either they could not change while our transaction was active, or if one did, then our transaction would fail. Listing 9-23 shows the updated method that does the locking.

Listing 9-23. *Using a Read Lock*

```
protected long totalSalaryInDepartment(int deptId) {
    long total = 0;
    Department dept = em.find(Department.class, deptId);
    for (Employee emp : dept.getEmployees()) {
        em.lock(emp, LockModeType.READ);
        total += emp.getSalary();
    }
    return total;
}
```

We mentioned that the implementation is permitted to lock eagerly or defer acquisition of the locks until the end of the transaction. Most major implementations defer the locking until commit time and by doing so provide far superior performance and scalability without sacrificing any of the semantics.

Write Locking

The other level of locking is called a *write lock*, which by virtue of its name hints correctly that we are actually locking the object for writing. The write lock guarantees all that the read lock does but in addition pledges to increment the version field in the transaction regardless of whether a user updated the entity or not. This provides a promise of an optimistic lock failure if another transaction also tries to modify the same entity before this one commits. This is equivalent to making a forced update to the entity in order to trigger the version number to be augmented. The obvious conclusion is that if the entity is being updated or removed by the application, then it never needs to be write-locked and that write-locking it anyway would be redundant at best and at worst could lead to an additional update, depending upon the implementation.

The common case for using write locks is to guarantee consistency across entity relationship changes (often they are one-to-many relationships with target foreign keys) when in the object model the entity relationship pointers change but in the data model no columns in the entity table change.

For example, let's say an employee has a set of assigned uniforms that were given to him, and his company has a cheap cleaning service that bills him automatically through payroll deduction. So `Employee` has a one-to-many relationship to `Uniform`, and `Employee` has a `cleaningCost` field that contains the amount that will get deducted from his paycheck at the end of the month. If there are two different stateful session beans that have extended persistence contexts, one for managing employees (`EmployeeManagement`) and another that manages the cleaning fees (`CleaningFeeManagement`) for the company, then if the `Employee` exists in both of the persistence contexts, there is a possibility of inconsistency.

Both copies of the `Employee` entity start out the same, but let's say that an operator records that the employee has received an additional brand new uniform. This implies creation of a new `Uniform` entity and adding it to the one-to-many collection of the `Employee`. The transaction is committed and everything is fine, except that now the `EmployeeManagement` persistence context has a different version of the `Employee` than the `CleaningFeeManagement` persistence context has. The operator has done the first maintenance task and now goes on to computing the cleaning charge for clients. The `CleaningFeeManagement` session computes the cleaning charges

based on the one-to-many relationship that it knows about (without the extra uniform) and writes out a new version of the `Employee` with the employee's cleaning charge based on one less uniform. The transaction commits successfully even though the first transaction had already committed and though the changes to the uniform relationship had already committed to the database. Now we have an inconsistency between the number of uniforms and the cost of cleaning them, and the `CleaningFeeManagement` persistence context could go on with its stale copy of the `Employee` without even knowing about the new uniform and never get a lock conflict.

The reason the change was not seen and no lock exception occurred for the second operation was because in the first operation no writes to the `Employee` actually occurred and thus the version column was not updated. The only changes to the `Employee` were to its relationship, and because it was owned by the uniform side there was no reason to make any updates to the `Employee`. Unfortunately for the company (though the employee may not be so unfortunate) this means they will be out a cleaning fee for the uniform.

The solution is to use the write lock, as shown in Listing 9-24, and force an update to the `Employee` when the relationship changed in the first operation. This will cause any updates in any other persistence contexts to fail if they make changes without knowing about the relationship update.

Listing 9-24. *Using a Write Lock*

```
@Stateful
public class EmployeeManagementBean implements EmployeeManagement {
    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;

    public void addUniform(int id, Uniform uniform) {
        Employee emp = em.find(Employee.class, id);
        em.lock(emp, LockModeType.WRITE);
        emp.addUniform(uniform);
        uniform.setEmployee(emp);
    }

    // ...
}

@Stateful
public class CleaningFeeManagementBean implements CleaningFeeManagement {
    static final Float UNIFORM_COST = 4.7f;

    @PersistenceContext(unitName="EmployeeService",
                        type=PersistenceContextType.EXTENDED)
    EntityManager em;
```

```
    public void calculateCleaningCost(int id) {
        Employee emp = em.find(Employee.class, id);
        Float cost = emp.getUniforms().size() * UNIFORM_COST;
        emp.setCost(emp.getCost() + cost);
    }

    // ...
}
```

Recovering from Optimistic Failures

An optimistic failure means that one or more of the entities that were modified were not *fresh* enough to be allowed to record their changes. The version of the entity that was modified was *stale*, and the entity had since been changed in the database, hence an `OptimisticLockException` was thrown. There is not always an easy solution to recovering, and depending upon the application architecture, it may or may not even be possible, but if and when appropriate, one solution may be to get a fresh copy of the entity and then re-apply the changes. In other cases it may only be possible to give the client (such as a web browser) an indication that the changes were in conflict with another transaction and must be reentered. The harsh reality of it is that in the majority of cases it is neither practical nor feasible to handle optimistic lock problems other than to simply retry the operation at a convenient transactional demarcation point.

The first problem you might encounter when an `OptimisticLockException` is thrown could be the one you never see. Depending on what your settings are, for example whether the calling bean is container-managed or bean-managed, and whether the interface is remote or local, you may only get a container-initiated `EJBException`. This exception will not necessarily even wrap the `OptimisticLockException`, since all that is formally required of the container is to log it before throwing the exception.

Listing 9-25 shows how this could happen when invoking a method on a session bean that initiates a new transaction.

Listing 9-25. *BMT Session Bean Client*

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class EmpServiceClientBean implements EmpServiceClient {
    @EJB EmployeeService empService;

    public void adjustVacation(int id, int days) {
        try {
            empService.deductEmployeeVacation(id, days);
        } catch (EJBException ejbEx) {
            System.out.println("Something went wrong, but I have no idea what!");
        } catch (OptimisticLockException olEx) {
            System.out.println("This exception would be nice, but I will ➥
```



```
probably never get it!");
        }
    }
}
```

The problem is that when an optimistic exception occurs down in the bowels of the persistence layer, it will get passed back to the `EmployeeService` session bean and get handled according to the rules of runtime exception handling by the container. Since the `EmpServiceClientBean` uses bean-managed transactions and does not start a transaction, and `EmployeeServiceBean` defaults to container-managed transactions with a `REQUIRED` attribute, a transaction will be initiated when the call to `deductVacationBalance()` occurs.

Once the method has completed and the changes have been made, the container will attempt to commit the transaction. In the process of doing this, the persistence provider will get a transaction synchronization notification from the transaction manager to flush its persistence context to the database. As the provider attempts its writes, it finds during its version number check that one of the objects has been modified by another process since being read by this one, so it throws an `OptimisticLockException`. The problem is that the container treats this exception the same way as any other runtime exception. The exception simply gets logged and the container throws an `EJBException`.

The solution to this problem is to perform a `flush()` operation from inside the container-managed transaction at the moment just before we are ready to complete the method. This forces a write to the database and locks the resources only at the end of the method so the effects on concurrency are minimized. It also allows us to handle an optimistic failure while we are in control, without the container interfering and potentially swallowing the exception. If we do get an exception from the `flush()` call, then we can throw an application exception that the caller can recognize. This is shown in Listing 9-26.

Listing 9-26. *Catching and Converting* `OptimisticLockException`

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public void deductEmployeeVacation(int id, int days) {
        Employee emp = em.find(Employee.class, id);
        emp.setVacationDays(emp.getVacationDays() - days);
        // ...
        flushChanges();
    }

    public void adjustEmployeeSalary(int id, long salary) {
        Employee emp = em.find(Employee.class, id);
        emp.setSalary(salary);
        // ...
        flushChanges();
    }
```

```
    protected void flushChanges() {
        try {
            em.flush();
        } catch (OptimisticLockException optLockEx) {
            throw new ChangeCollisionException();
        }
    }
    // ...
}

@ApplicationException
public class ChangeCollisionException extends RuntimeException {
    public ChangeCollisionException() { super(); }
}
```

The `OptimisticLockException` may contain the object that caused the exception, but it is not guaranteed to. We have only one object that we know is the `Employee`, so we are not passing it on or looking at it. To access this object, we would have invoked `getObject()` on the exception that we caught to see whether the object was included.

We factor out the flushing as every method must flush and catch the exception and then rethrow a domain-specific application exception. The `ChangeCollisionException` class is annotated with `@ApplicationException`, which is an EJB 3.0 container annotation in the `javax.ejb` package to indicate to the container that the exception is not really a system-level exception but should be thrown back to the client as is. Normally, defining an application exception will cause the container to not roll back the transaction, but this is an EJB 3.0 container notion. The persistence provider that threw the `OptimisticLockException` does not know about the special semantics of designated application exceptions and seeing a runtime exception will go ahead and mark the transaction for rollback.

The client code that we saw earlier can now receive and handle the application exception and potentially do something about it. At the very least it is aware of the fact that the failure was a result of a data collision instead of some other more fatal error. The client bean is shown in Listing 9-27.

Listing 9-27. *Handling* `OptimisticLockException`

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class EmpServiceClientBean implements EmpServiceClient {
    @EJB EmployeeService empService;

    public void adjustVacation(int id, int days) {
        try {
            empService.deductEmployeeVacation(id, days);
        } catch (ChangeCollisionException ccEx) {
            System.out.println("Collision with other change Ð RetryingÉ");
            empService.deductEmployeeVacation(id, days);
        }
    }
}
```


When an `OptimisticLockException` occurs in this context, the easy answer is to retry. This was really quite a trivial case, so the decision to retry was not hard to make. If we are in an extended persistence context, however, we may have a much harder job of it since all of the entities in the extended persistence context become detached when a transaction rolls back. Essentially we would need to reenlist all of our objects after having reread them and then replay all of the changes that we had applied in the previous failed transaction. Not a very easy thing to do in most cases.

In general it is quite difficult to code for the optimistic exception case. When running in a server environment, chances are that any `OptimisticLockException` will be wrapped by an EJB exception or server exception. The best approach is to simply treat all transaction failures equally and retry the transaction from the beginning or to indicate to the browser client that they must restart and retry.

Schema Generation

When we touched on schema generation in Chapter 4 we promised to go over the mapping annotation elements that are considered when schema generation occurs. In this section we will make good on that pledge and explain which elements get applied to the generated schema for those vendors that support schema generation.[2]

A couple of comments are in order before we start into them, though. First, the elements that contain the schema-dependent properties are, with few exceptions, in the physical annotations. This is to try to keep them separate from the logical non-schema related metadata. Second, these annotations are ignored, for the most part,[3] if the schema is not being generated. This is one reason why using them is a little out of place in the usual case, since schema information about the database is of little use once the schema has been created and is being used.

One of the complaints around schema generation is that you can't specify everything that you need to be able to finely tune the schema. This was done on purpose. There are too many differences between databases and too many different settings to try to put in options for every database type. If every database-tuning option were exposed through the Java Persistence API then we would end up duplicating the features of Data Definition Language (DDL) in an API that was not meant to be a database schema generation facility. As we mentioned earlier, the majority of applications find themselves in a meet-in-the-middle mapping scenario in any case, and when they do have control over the schema, then the final schema will typically be tuned by a database administrator or someone with the appropriate level of database experience.

Unique Constraints

A unique constraint can be created on a generated column or join column by using the `unique` element in the `@Column` or `@JoinColumn` annotations. There are not actually very many cases where this will be necessary since most vendors will generate a unique constraint when it is appropriate, such as on the join column of one-to-one relationships. Otherwise the value of

2. Most vendors supporting the Java Persistence API support some kind of schema generation either in the runtime or in a tool.
3. The exception to this rule may be the `optional` element of the mapping annotations, which may result in a NON NULL constraint but which may also be used in memory to indicate that the value is or isn't allowed to be set to null.

the unique element defaults to false. Listing 9-28 shows an entity with a unique constraint defined for the STR column.

Listing 9-28. *Including Unique Constraints*

```
@Entity
public class Employee {
    @Id private int id;
    @Column(unique=true)
    private String name;
    // ...
}
```

Note that the unique element is unnecessary on the identifier column since a primary key constraint will always be generated for the primary key.

A second way of adding a unique constraint is to embed one or more @UniqueConstraint annotations in a uniqueConstraints element in the @Table or @SecondaryTable annotations. Any number of unique constraints may be added to the table definition, including compound constraints. The value passed to the @UniqueConstraint annotation is an array of one or more strings listing the column names that make up the constraint. Listing 9-29 demonstrates how to define a unique constraint as part of a table.

Listing 9-29. *Unique Constraints Specified in Table Definition*

```
@Entity
@Table(name="EMP",
       uniqueConstraints=@UniqueConstraint(columnNames={"NAME"}))
public class Employee {
    @Id private int id;
    private String name;
    // ...
}
```

Null Constraints

Constraints on a column may also be in the form of null constraints. A null constraint just indicates that the column may or may not be null. It is defined when the column is declared as part of the table.

Null constraints are defined on a column by using the nullable element in the @Column or @JoinColumn annotations. A column allows null values by default, so this element really needs to be used only when a value for the field or property is required. Listing 9-30 demonstrates how to set the nullable element of basic and relationship mappings.

Listing 9-30. *Null Constraints Specified in Column Definitions*

```
@Entity
public class Employee {
    @Id private int id;
    @Column(nullable=false)
    private String name;
    @ManyToOne
    @JoinColumn(nullable=false)
    private Address address;
    // ...
}
```

String-Based Columns

When no length is specified for a column that is being generated to store string values, then the length will be defaulted to 255. When a column is generated for a basic mapping of a field or property of type String, char[], or Character[], its length should be explicitly listed in the length element of the @Column annotation if 255 is not the desired maximum length. Listing 9-31 shows an entity with explicitly specified lengths for strings.

Listing 9-31. *Specifying the Length of Character-Based Column Types*

```
@Entity
public class Employee {
    @Id
    @Column(length=40)
    private String name;
    @ManyToOne
    @JoinColumn(name="MGR")
    private Employee manager;
    // ...
}
```

We can see from the previous example that there is no similar length element in the @JoinColumn annotation. When primary keys are string-based, the provider may set the join column length to the same length as the primary key column in the table that is being joined to. This is not required to be supported, however.

It is not defined for length to be used for large objects; some databases do not require or even allow the length of lobs to be specified.

Floating Point Columns

Columns containing floating point types have a precision and scale associated with them. The precision is just the number of digits that are used to represent the value, and the scale is the number of digits after the decimal point. These two values may be specified as precision and scale elements in the @Column annotation when mapping a floating point type. Like other

schema generation elements, they have no effect on the entity at runtime. Listing 9-32 demonstrates how to set these values.

Listing 9-32. *Specifying the Precision and Scale of Floating Point Column Types*

```
@Entity
public class PartTimeEmployee {
    // ...
    @Column(precision=8, scale=2)
    private float hourlyRate;
    // ...
}
```

Tip Precision may be defined differently for different databases. In some databases and for some floating point types it is the number of binary digits, while for others it is the number of decimal digits.

Defining the Column

There may be a time when you are happy with all of the generated columns except for one. It isn't what you want it to be, and you don't want to go through the trouble of manually generating the schema for the sake of one column. This is one instance when the `columnDefinition` element comes in handy. By hand-rolling the DDL for the column, we can include it as the column definition and let the provider use it to define the column.

The `columnDefinition` element is available in all of the column-oriented annotation types, including `@Column`, `@JoinColumn`, `@PrimaryKeyJoinColumn`, and `@DiscriminatorColumn`. Whenever a column is to be generated, the `columnDefinition` element may be used to indicate the DDL string that should be used to generate the type (not including the trailing comma). This gives the user complete control over what is generated in the table for the column being mapped. It also allows a database-specific type or format to be used that may supercede the generated type offered by the provider for the database being used.[4] Listing 9-33 shows some definitions specified for two columns and a join column.

Listing 9-33. *Using a Column Definition to Control DDL Generation*

```
@Entity
public class Employee {
    @Id
    @Column(columnDefinition="NVARCHAR2(40)")
    private String name;
    @Column(name="START_DATE", columnDefinition="DATE DEFAULT SYSDATE")
    private java.sql.Date startDate;
    @ManyToOne
    @JoinColumn(name="MGR", columnDefinition="NVARCHAR2(40)")
    private Employee manager;
    // ...
}
```

4. The resulting column must be supported by the provider runtime to enable reading from and writing to the column.

In this example we are using a Unicode character field for the primary key and then also for the join column that refers to the primary key. We also define the date to be assigned the default current date at the time the record was inserted (in case it was not specified).

Specifying the column definition is quite a powerful schema generation practice that allows overriding of the generated column to an application-defined custom column definition. But the power is accompanied by some risk as well. When a column definition is included, then other accompanying column-specific generation metadata is ignored. Specifying the precision, scale, or length in the same annotation as a column definition would be both unnecessary and confusing.

Not only does using `columnDefinition` in your code bind you to a particular schema, but it also binds you to a particular database since the DDL tends to be database-specific. This is just a flexibility-portability trade-off, and you have to decide whether it is appropriate for your application.

Summary

Over the course of this chapter we have covered a wide range of diverse topics, from SQL queries to schema generation. Not everything we have described will be immediately usable in a new application, but some features such as optimistic locking are likely to play a prominent role in many enterprise applications.

We began the chapter with a look at SQL queries. We looked at the role of SQL in applications that also use JPQL and the specialized situations where only SQL can be used. To bridge the gap between native SQL and entities, we described the result set mapping process in detail, showing a wide range of queries and how they translate back into the application domain model.

The Lifecycle Callbacks section introduced the life cycle of an entity and showed the points at which an application can monitor events that are fired as an entity moves through different stages of its life cycle. We looked at two different approaches to implementing callback methods, on the entity class and as part of a separate listener class.

In our discussion of locking and versioning, we introduced optimistic locking and described the vital role it plays in many applications, particularly those that use detached entities. We also looked at read and write locks for entities and how they correspond to isolation levels in the database. We described the difficulties of recovering from optimistic lock failures and when it is appropriate to refresh the state of a managed entity.

Finally we looked at schema generation and how to specify schema properties using different elements of the mapping annotations.

In the next chapter we will look at the XML mapping file, showing how to use XML with, or instead of, annotations and how annotation metadata can be overridden.

CHAPTER 10

XML Mapping Files

Since the release of Java SE 5 there has been a quiet, and sometimes not-so-quiet, ongoing debate about whether annotations are better or worse than XML. The defenders of annotations vigorously proclaim how annotations are so much simpler and provide in-lined metadata that is co-located with the code that it is describing. They claim that this avoids the need to replicate the source code context of where the metadata applies. The XML proponents then retort that annotations unnecessarily couple the metadata to the code and that changes to metadata should not require changes to the source code.

The truth is that both sides are right and that there are appropriate times for using annotation metadata and others for using XML. When the metadata really is coupled to the code, then it does make sense to use annotations since the metadata is just another aspect of the program. For example, specification of the identifier field of an entity is not only a relevant piece of information to the provider but also a necessary detail known and assumed by the referencing application code. Other kinds of metadata, such as which column a field is mapped to, can be safely changed without needing to change the code. This metadata is akin to configuration metadata and might be better expressed in XML, where it can be configured according to the usage pattern or execution environment.

These arguments also tend to unfairly compartmentalize the issue, because in reality it goes deeper than simply deciding when it might make sense to use one type of metadata or the other. In many talks and forums leading up to the release of the specification, we asked people whether they planned on using annotations or XML, and we consistently saw that there was a split. The reason is that there are other factors that have nothing to do with which is better, such as existing development processes, source control systems, developer experience, and so forth. Reasons for using XML go beyond whether or not it is a better or worse technique for specifying metadata.

With this controversy as a backdrop, it is easy to see that it was not by accident that mapping metadata was allowed to be specified in either format. In fact, XML mapping usage is defined in such a way as to allow annotations to be used and then overridden by XML. This provides the ability to use annotations for some things and XML for others, or to use annotations for an expected configuration but then supply an overriding XML file to suit a particular execution environment. The XML file may be sparse and supply only the information that is required to be overridden, and we will see later on in this chapter that the granularity with which this metadata may be specified offers a good deal of object-relational mapping flexibility.

Over the course of this chapter we will describe the structure and content of the mapping file and how it relates to the metadata annotations. We will also discuss how XML mapping metadata may combine with and override annotation metadata. We have tried to structure the chapter in a format that will allow it to be used as both a source of information and a reference for the mapping file format.

The Metadata Puzzle

The rules of XML and annotation usage and overriding can be a little confusing to say the least, especially given the permutation space of mixing annotations with XML. The trick to understanding the semantics and being able to properly specify metadata the way that you would like it to be specified is to understand the metadata collection process. Once you have a solid understanding of what the metadata processor does, you will be well on your way to understanding what you need to do to achieve a specific result.

The provider may choose to perform the metadata gathering process in any way it chooses, but the result is that it must honor the requirements of the specification. Developers understand algorithms, so we decided that it will be easier to understand if we present the logical functionality as an algorithm, even though the implementation may not actually implement it this way. The following algorithm can be considered as the simplified logic for obtaining the metadata for the persistence unit:

1. **Process the annotations.** The set of entities, mapped superclasses, and embedded objects is discovered (we'll call this set E) by looking for the `@Entity`, `@MappedSuperclass`, and `@Embeddable` annotations. The class and method annotations in all of the classes in E are processed, and the resulting metadata is stored in the set C. Any missing metadata that was not explicitly specified in the annotations is left empty.

2. **Add the classes defined in XML.** Look for all of the entities, mapped superclasses, and embedded objects that are defined in the mapping files and add them to E. If we find that one of the classes already exists in E, then we apply the overriding rules for class-level metadata that we found in the mapping file. Add or adjust the class-level metadata in C according to the overriding rules.

3. **Add the attribute mappings defined in XML.** For each class in E, look at the fields or properties in the mapping file and try to add the method metadata to C. If the field or property already exists there, then apply the overriding rules for attribute-level mapping metadata.

4. **Apply defaults.** Determine all default values according to the scoping rules and where defaults may have been defined (see the following for description of default rules). The classes, attribute mappings, and other settings that have not yet been filled in are assigned values and put in C.

Some of the following cases may cause this algorithm to be modified slightly, but in general this is what will logically happen when the provider needs to obtain the mapping metadata.

We already learned in the mapping chapters that annotations may be sparse and that not annotating a persistent attribute will cause it to default to being mapped as a basic mapping. Other mapping defaults were also explained, and we saw how much easier they made configuring and mapping entities. We notice in our algorithm that the defaults are applied at the end, so the same defaults that we saw for annotations will be applied when using mapping files as well. It should be of some comfort to XML users that mapping files may be sparsely specified in the same ways as annotations. They also have the same requirements for what needs to be specified; for example, an identifier must be specified, a relationship mapping must have at least its cardinality specified, and so forth.

The Mapping File

By this point you are well aware that if you don't want to use XML for mapping, then you don't need to use XML. In fact, as we will see in Chapter 11, any number of mapping files or none may be included in a persistence unit. If you do use one, however, each mapping file that is supplied must conform and be valid against the `orm_1_0.xsd` schema located at `http://java.sun.com/xml/ns/persistence/orm_1_0.xsd`. This schema defines a namespace called `http://java.sun.com/xml/ns/persistence/orm` that includes all of the ORM elements that can be used in a mapping file. A typical XML header for a mapping file is shown in Listing 10-1.

Listing 10-1. *XML Header for Mapping File*

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                        http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    version="1.0">
```

The root element of the mapping file is called `entity-mappings`. All object-relational XML metadata is contained within this element, and as seen in the example, the header information is also specified as attributes in this element. The subelements of `entity-mappings` can be categorized into four main scoping and functional groups: persistence unit defaults, mapping files defaults, queries and generators, and managed classes and mappings. There is also a special setting that determines whether annotations should be considered in the metadata for the persistence unit. These groups are discussed in the following sections. For the sake of brevity we won't include the header information in the XML examples in these sections.

Disabling Annotations

For those who are perfectly happy with XML and don't feel the need for annotations, there are ways to skip the annotation processing phase (step 1 in the previous algorithm). The `xml-mapping-metadata-complete` element and `metadata-complete` attribute provide a convenient way to reduce the overhead that is required to discover and process all of the annotations on the classes in the persistence unit. It is also a way to effectively disable any annotations that do exist. These options will cause the processor to completely ignore them as if they did not exist at all.

xml-mapping-metadata-complete

When the `xml-mapping-metadata-complete` element is specified, all annotations in the entire persistence unit will be ignored, and only the mapping files in the persistence unit will be considered as the total set of provided metadata. Only entities, mapped superclasses, and embedded objects that have entries in a mapping file will be added to the persistence unit.

The `xml-mapping-metadata-complete` element needs to be in only one of the mapping files if there are multiple mapping files in the persistence unit. It is specified as an empty subelement of the `persistence-unit-metadata` element, which is the first[1] subelement of `entity-mappings`. An example of using this setting is in Listing 10-2.

Listing 10-2. *Disabling Annotation Metadata for the Persistence Unit*

```
<entity-mappings>
    <persistence-unit-metadata>
        <xml-mapping-metadata-complete/>
    </persistence-unit-metadata>
    ...
</entity-mappings>
```

If enabled, there is no way to portably override this setting. It will apply globally to the persistence unit, regardless of whether any `metadata-complete` attribute is set to false in an entity.

metadata-complete

The `metadata-complete` attribute is an attribute on the `entity`, `mapped-superclass`, and `embeddable` elements. If specified, all annotations on the specified class and on any fields or properties in the class will be ignored, and only the metadata in the mapping file will be considered as the set of metadata for the class.

■ **Caution** Annotations defining queries, generators, or result set mappings are ignored if they are defined on a class that is marked as `metadata-complete` in an XML mapping file.

When `metadata-complete` is enabled, the same rules that apply to annotated entities will also apply to the XML-mapped entity. For example, the identifier must be mapped, and all relationships must be specified with their corresponding cardinality mappings inside the `entity` element.

An example of using the `metadata-complete` attribute is in Listing 10-3. The entity mappings in the annotated class are disabled by the `metadata-complete` attribute, and because the fields are not mapped in the mapping file, the default mapping values will be used. The `name` and `salary` fields will be mapped to the `NAME` and `SALARY` columns, respectively.

1. Technically there is a `description` element in many of the elements, just as there are in most of the standard schemas in Java EE, but these have little functional value and will not be mentioned here. They may be of some use to tools that parse XML schemas and use the descriptions for tooltips, and similar actions.

Listing 10-3. *Disabling Annotations for a Managed Class*

```
@Entity
public class Employee {
    @Id private int id;
    @Column(name="EMP_NAME")
    private String name;
    @Column(name="SAL")
    private long salary;
    // ...
}

<entity-mappings>
    ...
    <entity class="examples.model.Employee"
            metadata-complete="true">
        <attributes>
            <id name="id"/>
        </attributes>
    </entity>
    ...
</entity-mappings>
```

Persistence Unit Defaults

One of the conditions for using annotation metadata is that we need to have something to annotate. If we want to define metadata for a persistence unit, then we are in the unfortunate position of not having anything to annotate, since a persistence unit is just a logical grouping of Java classes, basically a configuration. This brings us back to the discussion that we had earlier when we decided that if metadata is not coupled to code, then maybe it shouldn't really be in the code. These are the reasons why persistence unit metadata may be specified only in an XML mapping file.

In general, a persistence unit default means that whenever a value for that setting is not specified at a more local scope, the persistence unit default value will apply. It is a convenient way to set default values that will apply to all entities, mapped superclasses, and embedded objects in the entire persistence unit, be they in any of the mapping files or annotated classes. The default values will not be applied if a value is present at any level below the persistence unit. This value may be in the form of a mapping file default value, some value in an entity element, or an annotation on one of the managed classes or persistent fields or properties.

The element that encloses all of the persistence unit level defaults is the aptly named `persistence-unit-defaults` element. It is the other subelement of the `persistence-unit-metadata` element (after `xml-mapping-metadata-complete`). If more than one mapping file exists in a persistence unit, then only one of the files should contain these elements.

There are five settings that can be configured to have default values for the persistence unit. They are specified using the `schema`, `catalog`, `access`, `cascade-persist`, and `entity-listeners` elements.

schema

The schema element is useful if you don't want to have to specify a schema in every @Table, @SecondaryTable, @JoinTable, or @TableGenerator annotation or table, secondary-table, join-table, or table-generator XML element in the persistence unit. When set here, it will apply to all tables in the persistence unit, whether they were actually defined or defaulted by the provider. The value of this element may be overridden by any of the following:

- schema element defined in the mapping file defaults (see the Mapping File Defaults section)
- schema attribute on any table, secondary-table, join-table, or table-generator element in a mapping file
- schema defined within an @Table, @SecondaryTable, @JoinTable, or @TableGenerator annotation or in an @TableGenerator annotation (unless xml-mapping-metadata-complete is set)

Listing 10-4 shows an example of how to set the schema for all of the tables in the persistence unit that do not already have their schema set.

Listing 10-4. *Setting the Default Persistence Unit Schema*

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <schema>HR</schema>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    ...
</entity-mappings>
```

catalog

The catalog element is exactly analogous to the schema element except that it is for databases that support catalogs. It has the same behavior as schema and is overridden in exactly the same ways. The exact same rules may be applied to the catalog mapping file default as described in the preceding schema section.

access

The access element that is defined in the persistence-unit-defaults section is used to set the access type for all of the managed classes in the persistence unit that have XML entries but are not annotated. There is no corresponding annotation for this element. Its value may be either "FIELD" or "PROPERTY", indicating how the provider should access the persistent state.

The access setting is a subtly different default that does not affect any of the managed classes that have annotated fields or properties. It is a convenience for when XML is used and obviates having to specify the access for all of the entities listed in all of the XML mapping files.

This element affects only the managed classes defined in the mapping files, because a class with annotated fields or properties is considered to have overridden the access mode by virtue of its having annotations placed on its fields or properties. If the xml-mapping-metadata-complete element is enabled, then the persistence unit access default will be applied to these annotated classes that have entries in XML. Put another way, the annotations that would have otherwise overridden the access mode would no longer be considered, and the XML defaults, including the default access mode, would be applied.

The value of this element may be overridden only by one or more of the following:

- access element defined in the mapping file defaults (see the Mapping File Defaults section)
- access attribute on any entity, mapped-superclass, or embeddable element in a mapping file
- An annotated field or property in an entity, mapped superclass, or embedded object

In Listing 10-5 we show an example of setting the access mode to "PROPERTY" for all of the managed classes in the persistence unit that do not have annotated fields.

Listing 10-5. *Setting the Default Access Mode for the Persistence Unit*

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <access>PROPERTY</access>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    ...
</entity-mappings>
```

cascade-persist

The cascade-persist element is unique in a different way. When the empty cascade-persist element is specified, it is analogous to adding the PERSIST cascade option to all of the relationships in the persistence unit. See Chapter 5 for a discussion about the cascade options on relationships.

The term *persistence-by-reachability* is often used to signify that when an object is persisted, all of the objects that are reachable from that object are also automatically persisted. The cascade-persist element provides the persistence-by-reachability semantics that some people are used to having. This setting cannot currently be overridden, but the intent is that it be overridable in future releases. The assumption is that when somebody is accustomed to persistence-by-reachability semantics, they don't normally want to be turning it off. If more fine-grained control over cascading of the persist operation is needed, then this element should not be specified, and the relationships should have their PERSIST cascade option specified explicitly.

An example of using the cascade-persist element is shown in Listing 10-6.

Listing 10-6. *Configuring for Persistence-by-Reachability Semantics*

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <cascade-persist/>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    ...
</entity-mappings>
```

entity-listeners

This is the only place where a list of *default entity listeners* can be specified. A default entity listener is a listener that will be applied to every entity in the persistence unit. They will be invoked in the order that they are listed in this element, before any other listener or callback method is invoked on the entity. It is the logical equivalent of adding the listeners in this list to the front of the `@EntityListeners` list in the root superclass. We discussed entity listeners in the last chapter, so take a look at Chapter 9 to review the order of invocation if you need to. A description of how to specify an entity listener is given in the Entity Listeners section.

The `entity-listeners` element is composed of zero or more `entity-listener` elements that each defines an entity listener. They can be overridden or disabled in either of the following two ways:

- `exclude-default-listeners` element in an `entity` or `mapped-superclass` mapping file element
- `@ExcludeDefaultListeners` annotation on an entity or mapped superclass (unless `xml-mapping-metadata-complete` is set)

Mapping File Defaults

The next level of defaults after the ones defined for the entire persistence unit are those that pertain only to the entities, mapped superclasses, and embedded objects that are contained in a particular mapping file. In general, if there is a persistence unit default defined for the same setting, then this value will override the persistence unit default for the managed classes in the mapping file. Unlike the persistence unit defaults, the mapping file defaults do not affect managed classes that are annotated and not defined in the mapping file.

The mapping file defaults consist of four subelements of the `entity-mappings` element. They are `package`, `schema`, `catalog`, and `access`.

package

The `package` element is intended to be used by developers who don't want to have to repeat the fully qualified class name in all of the mapping file metadata. It may be overridden in the mapping file by fully qualifying a class name in any element or attribute where a class name is expected. These are the following:

- `class` attribute of `id-class`, `entity-listener`, `entity`, `mapped-superclass`, or `embeddable` elements
- `target-entity` attribute of `many-to-one`, `one-to-one`, `one-to-many`, and `many-to-many` elements
- `result-class` attribute of `named-native-query` element
- `entity-class` attribute of `entity-result` element

An example of using this element is shown in Listing 10-7. We set the default mapping file package name to `examples.model` for the entire mapping file and can just use the unqualified `Employee` and `EmployeePK` class names throughout the file. The package name will not be applied to `OtherClass`, though, as it is already fully specified.

Listing 10-7. *Using the* `package` *Element*

```
<entity-mappings>
    <package>examples.model</package>
    ...
    <entity class="Employee">
        <id-class class="EmployeePK"/>
        ...
    </entity>
    <entity class="examples.tools.OtherClass">
        ...
    </entity>
    ...
</entity-mappings>
```

schema

The `schema` element will set a default schema to be assumed for every table, secondary table, join table, or table generator defined or defaulted within the mapping file. This element may be overridden by the specification of the `schema` attribute on any `table`, `secondary-table`, `join-table`, or `table-generator` element in the mapping file.

Listing 10-8 shows the mapping file schema default set to "HR" so the `EMP` table that `Employee` is mapped to is assumed to be in the `HR` schema.

Listing 10-8. *Using the* `schema` *Element*

```
<entity-mappings>
    <package>examples.model</package>
    <schema>HR</schema>
    ...
    <entity class="Employee">
        <table name="EMP"/>
        ...
    </entity>
    ...
</entity-mappings>
```

The mapping file schema default will also affect `@Table`, `@SecondaryTable`, `@JoinTable`, and `@TableGenerator` annotations on classes that have entries in the mapping file. For example, since `Employee` is listed in the mapping file, it becomes part of the set of classes to which the default applies. If there was an `@TableGenerator(name="EmpGen", table="IDGEN")` annotation on `Employee`, then the mapping file default will be applied to it, and the `IDGEN` table will be assumed to be in the `HR` schema.

catalog

The `catalog` element is again exactly analogous to the `schema` element except it is for databases that support catalogs. It has the same behavior as `schema` at the mapping file default level and is overridden in exactly the same ways. As we mentioned in the persistence unit section, the exact same rules may be applied to the `catalog` mapping file default as described in the `schema` mapping file default section.

access

Setting a particular access mode as the mapping file default value affects only the managed classes that are defined in the mapping file. It is done through the use of the `access` element. As with annotations, it is not portable to use different access types for different classes in an entity hierarchy, although some vendors do allow it. Classes that have entries in the mapping file in addition to annotations must have matching access types if specified in XML. It is not portable to mix access types for the same entity.

Queries and Generators

Some persistence artifacts, such as id generators and queries, are defined as annotations on a class even though they are actually global to the persistence unit in scope. The reason for this is because they are annotations and there is no other place to put them other than on a class. In XML this global metadata does not need to be placed arbitrarily within a class but can be defined at the level of subelements of the `entity-mappings` element.

The global metadata elements are made up of generator and query elements that include `sequence-generator`, `table-generator`, `named-query`, `named-native-query`, and `sql-result-set-mapping`. These elements may appear in different contexts, but they are nevertheless still scoped to the persistence unit. There are three different persistence unit namespaces, one for queries, one for generators, and one for result set mappings that are used for native queries. When any of the elements that we just listed are defined in the mapping file, the artifacts they define will be added into the persistence unit namespace to which they apply. The namespaces will already contain all of the existing persistence unit artifacts that may have been defined in annotations or in another mapping file. Since these artifacts share the same global persistence unit namespace type, when one of the artifacts that is defined in XML shares the same name as one that already exists in the namespace of the same type, it is viewed as an override. The artifact that is defined in XML overrides the one that was defined by the annotation. There is no concept of overriding queries, generators, or result set mappings within the same or different mapping files. If one or more mapping files contains one of these objects defined with the same name, then it is undefined which overrides the other since the order that they are processed in is not specified.[2]

sequence-generator

The `sequence-generator` element is used to define a generator that uses a database sequence to generate identifiers. It corresponds to the `@SequenceGenerator` annotation (see Chapter 4) and may be used to define a new generator or override a generator of the same name that is defined by an `@SequenceGenerator` annotation in any class in the persistence unit. It may be specified either at the global level as a subelement of `entity-mappings`, at the entity level as a subelement of `entity`, or at the field or property level as a subelement of the `id` mapping element.

The attributes of `sequence-generator` line up exactly with the elements in the `@SequenceGenerator` annotation. Listing 10-9 shows an example of defining a sequence generator.

Listing 10-9. *Defining a Sequence Generator*

```
<entity-mappings>
    ...
    <sequence-generator name="empGen" sequence-name="empSeq"/>
    ...
</entity-mappings>
```

table-generator

The `table-generator` element defines a generator that uses a table to generate identifiers. Its annotation equivalent is the `@TableGenerator` annotation (see Chapter 4). This element may define a new generator, or it may be overriding a generator defined by an `@TableGenerator` annotation. Like the `sequence-generator` element, it may be defined within any of `entity-mappings`, `entity`, or `id` elements.

The attributes of `table-generator` also match the `@TableGenerator` annotation elements. Listing 10-10 shows an example of defining a sequence generator in annotation form but overriding it to be a table generator in XML.

Listing 10-10. *Overriding a Sequence Generator with a Table Generator*

```
@Entity
public class Employee {
    @SequenceGenerator(name="empGen")
    @Id @GeneratedValue(generator="empGen")
    private int id;
    // ...
}
```

2. It is possible, and even probable, that vendors will process the mapping files in the order that they are listed, but this is neither required nor standardized.

```
<entity-mappings>
    ...
    <table-generator name="empGen" table="ID_GEN" pk-column-value="EmpId"/>
    ...
</entity-mappings>
```

named-query

Static or named queries may be defined both in annotation form using `@NamedQuery` (see Chapter 6) or in a mapping file using the `named-query` element. A `named-query` element in the mapping file may also override an existing query of the same name but that was defined as an annotation. It makes sense, of course, when overriding a query to override it only with a query that has the same result type, be it an entity, data, or projection of data. Otherwise all of the code that executes the query and processes the results stands a pretty good chance of breaking.

A `named-query` element may appear as a subelement of `entity-mappings` or as a subelement of `entity`. Regardless of where it is defined, it will be keyed by its name in the persistence unit query namespace.

The name of the query is specified as an attribute of the `named-query` element, while the query string goes in a `query` subelement within it. Any number of query hints may also be provided as `hint` subelements.

In Listing 10-11 we see an example of two named queries, one of which uses a vendor-specific hint that bypasses the cache.

Listing 10-11. *Named Query in a Mapping File*

```
<entity-mappings>
    ...
    <named-query name="findEmpsWithName">
        <query>SELECT e FROM Employee e WHERE e.name LIKE :empName</query>
        <hint name="toplink.cache-usage" value="DoNotCheckCache"/>
    </named-query>
    <named-query name="findEmpsWithHigherSalary">
        <query><![CDATA[SELECT e FROM Employee e WHERE e.salary > :salary]]></query>
    </named-query>
    ...
</entity-mappings>
```

Query strings may also be expressed as CDATA within the `query` element. We can see in Listing 10-9 that this is helpful in cases when the query includes XML characters such as ">" that would otherwise need to be escaped.

named-native-query

Native SQL may also be used for named queries by defining an `@NamedNativeQuery` annotation (see Chapter 9) or by specifying a `named-native-query` element in a mapping file. Both named queries and native queries share the same query namespace in the persistence unit, so using either the `named-query` or `named-native-query` element will cause that query to override any query of the same name defined in annotation form.

Native queries are the same as named queries in that the `native-named-query` element may appear as a subelement of `entity-mappings` or as a subelement of `entity`. The name is specified using the `name` attribute, and the query string uses a `query` subelement. The hints are also specified in the same way. The only difference is that two additional attributes have been added to `named-native-query` to supply the result class or the result set mapping.

One use case for overriding queries is when the DBA comes to you and demands that your query run a certain way on a certain database. You can leave the query as generic JPQL for the other databases, but it turns out that the Oracle database can do this one particular thing very well using native syntax. By putting this query in a DB-specific XML file, it will be much easier to manage in the future. In Listing 10-12 we see an example of a vanilla named query in JPQL that is being overridden by a native SQL query.

Listing 10-12. *Overriding a JPQL Query with SQL*

```
@NamedQuery(name="findAllManagers"
            query="SELECT e FROM Employee e WHERE e.directs IS NOT EMPTY")
@Entity
public class Employee { ... }

<entity-mappings>
    ...
    <named-native-query name="findAllManagers"
                        result-class="examples.model.Employee">
        <query>
            SELECT /*+ FULL(m) */ e.id, e.name, e.salary,
                   e.manager_id, e.dept_id, e.address_id
            FROM   emp e,
                   (SELECT DISTINCT manager_id AS id FROM emp) m
            WHERE  e.id = m.id
        </query>
    </named-native-query>
    ...
</entity-mappings>
```

sql-result-set-mapping

A result set mapping is used by native queries to instruct the persistence provider how to map the results. The `sql-result-set-mapping` element corresponds to the `@SqlResultSetMapping` annotation. The name of the result set mapping is specified in the `name` attribute of the `sql-result-set-mapping` element. The result may be mapped as one or more entity types, projection data, or a combination of the two. Just as `@SqlResultSetMapping` encloses arrays of `@EntityResult` or `@ColumnResult` or both, so also can the `sql-result-set-mapping` element contain multiple `entity-result` and `column-result` elements. And similarly, as each `@EntityResult` contains an array of `@FieldResult`, the `entity-result` element may contain multiple `field-result` elements. The other `entityClass` and `discriminatorColumn` elements of the `@EntityResult` annotation map directly to the `entity-class` and `discriminator-column` attributes of the `entity-result` element.

Each sql-result-set-mapping may define a new mapping or override an existing one of the same name that was defined by an annotation. It is not possible to override only a part of the result set mapping. If you're overriding an annotation, then the entire annotation will be overridden, and the components of the result set mapping defined by the sql-result-set-mapping element will apply.

Having said all this about overriding, there is really not that much use in overriding an @SqlResultSetMapping since they are used to structure the result format from a static native query. As we mentioned earlier, queries tend to be executed with a certain expectation of the result that is being returned. Result set mappings are typically defined in a mapping file because that is also where the native query that it is defining the result is defined.

Listing 10-13 shows the "DepartmentSummary" result set mapping that we defined in Chapter 9 and its equivalent XML mapping file form.

Listing 10-13. *Specifying a Result Set Mapping*

```
@SqlResultSetMapping(
    name="DepartmentSummary",
    entities={
        @EntityResult(entityClass=Department.class,
                      fields=@FieldResult(name="name", column="DEPT_NAME")),
        @EntityResult(entityClass=Employee.class)
    },
    columns={@ColumnResult(name="TOT_EMP"),
             @ColumnResult(name="AVG_SAL")}
)

<entity-mappings>
    ...
    <sql-result-set-mapping name="DepartmentSummary">
        <entity-result entity-class="examples.model.Department">
            <field-result name="name" column="DEPT_NAME"/>
        </entity-result>
        <entity-result entity-class="examples.model.Employee"/>
        <column-result name="TOT_EMP"/>
        <column-result name="AVG_SAL"/>
    </sql-result-set-mapping>
    ...
</entity-mappings>
```

Managed Classes and Mappings

The main portion of every mapping file is typically going to be the managed classes in the persistence unit that are the entity, mapped-superclass, and embeddable elements and their state and relationship mappings. Each of these has their class specified as a class attribute of the element and their access type specified in an access attribute. The access attribute is required only when there are no annotations on the managed class or when metadata-complete (or xml-mapping-metadata-complete) has been specified for the class. If neither of these conditions

apply and annotations do exist on the class, then the access attribute setting should match the access used by the annotations.

Queries and generators may be specified within an entity element. Generators may also be defined inside an id element in an entity or mapped superclass. These have already been described in the preceding Queries and Generators section.

Attributes

Unfortunately, the word *attribute* is grossly overloaded. It can be a general term for a field or property in a class, it can be a specific part of an XML element that can be inlined in the element tag, or it can be a generic term referring to a characteristic. Throughout these sections we have usually referred to it in the context of the second meaning because we have been talking a lot about XML elements. In this section, however, it refers to the first definition of a state attribute in the form of a field or property.

The attributes element is a subelement of the entity, mapped-superclass, and embeddable elements. It is an enclosing element that groups all of the mapping subelements for the fields or properties of the managed class. Because it is only a grouping element, it does not have an analogous annotation. It dictates which mappings are allowed for each type of managed class.

In the entity and mapped-superclass elements, there are a number of mapping subelements that may be specified. For identifiers, either multiple id subelements or a single embedded-id subelement may be included. The simple basic, version, and transient mapping subelements may also be specified, as well as the many-to-one, one-to-one, one-to-many, and many-to-many association subelements. The mapping mix is rounded out with the embedded subelement. An embeddable element is limited to containing only basic and transient mapping subelements. These elements will all be discussed separately in their own sections later, but each element has one thing in common. They each have a name attribute (in the XML attribute sense) that is required to indicate the name of the attribute (in this case we mean field or property) that it is mapping.

A first general comment about overriding that applies to all of these elements as attribute mappings is that overriding of XML over annotations is done at the level of the attribute (field or property) name. Our algorithm will apply to these mappings as they are keyed by attribute name, and XML overrides will be applied by attribute name alone. All of the annotated mapping information for the attribute will be overridden as soon as a mapping element for that attribute name is defined in XML.

The type of mapping that was defined in annotation form and the type that it is being overridden to are not really relevant to the provider at the time of overriding. The provider is responsible only for implementing the overriding rules. This leads us to our second comment about overriding, which is that when overriding annotations, we should use the correct and compatible XML mapping. There are some cases where it might be valid to actually map an attribute differently in XML, but these cases are few and far between and primarily for exceptional types of testing or debugging.

For example, one could imagine overriding a field mapped in annotation form as a basic mapping with a transient mapping in XML. This would be completely legal but not necessarily a good idea. At some point a client of the entity may actually be trying to access that state, and if it is not being persisted, then the client might get quite confused and fail in curious ways that are difficult to debug. Overriding an address association property that is mapped as a many-to-one mapping could conceivably be overridden to be stored serially as a blob, but this could not

only break client access but also spill over to break other areas like JPQL queries that traverse the address.

The rule of thumb is that mappings should be overridden primarily to change the data-level mapping information. This would normally need to be done in the case, for example, where an application is developed on one database but deployed to another or must deploy to multiple different databases in production. In these cases the XML mappings would likely be `xml-mapping-metadata-complete` anyway, and the XML metadata would be used in its entirety rather than cobbling together bits of annotations and bits of XML and trying to keep it all straight across multiple database XML mapping configurations.

Tables

Specifying tables in XML works pretty much the same way as it does in annotation form. The same defaults are applied in both cases. There are two elements for specifying table information for a managed class: `table` and `secondary-table`.

table

A `table` element may occur as a subelement of `entity` and describes the table that the entity is mapped to. It corresponds to the `@Table` annotation (see Chapter 4) and has `name`, `catalog`, and `schema` attributes. One or more `unique-constraint` subelements may be included if unique column constraints are to be created in the table during schema generation.

If an `@Table` annotation exists on the entity, then the `table` element will override the table defined by the annotation. Overriding a table is usually accompanied also by the overridden mappings of the persistent state to the overridden table. In Listing 10-14 is an example that shows how an entity can be mapped to a different table than what it is mapped to by an annotation.

Listing 10-14. *Overriding a Table*

```
@Entity
@Table(name="EMP", schema="HR")
public class Employee { ... }

<entity class="examples.model.Employee">
    <table name="EMP_REC" schema="HR"/>
    ...
</entity>
```

secondary-table

Any number of secondary tables can be added to the entity by adding one or more `secondary-table` subelements to the `entity` element. This element corresponds to the `@SecondaryTable` annotation (see Chapter 8), and if it is present in an `entity` element, it will override any and all secondary tables that are defined in annotations on the entity class. The `name` attribute is required, just as the name is required in the annotation. The `schema` and `catalog` attributes and the `unique-constraint` subelements may be included just as with the `table` element.

Every secondary table needs to be joined to the primary table through a primary key join column (see Chapter 8). The `primary-key-join-column` element is a subelement of the `secondary-table` element and corresponds to the `@PrimaryKeyJoinColumn` annotation. As with the annotation, this is required only if the primary key column of the secondary table is different from that of the primary table. If the primary key happens to be a compound primary key, then multiple `primary-key-join-column` elements may be specified.

Listing 10-15 compares the specification of secondary tables in annotation and XML form.

Listing 10-15. *Specifying Secondary Tables*

```
@Entity
@Table(name="EMP")
@SecondaryTables({
    @SecondaryTable(name="EMP_INFO"),
    @SecondaryTable(name="EMP_HIST",
                    pkJoinColumns=@PrimaryKeyJoinColumn(name="EMP_ID"))
 })
public class Employee {
    @Id private int id;
    // ...
}

<entity class="examples.model.Employee">
    <table name="EMP"/>
    <secondary-table name="EMP_INFO"/>
    <secondary-table name="EMP_HIST">
        <primary-key-join-column name="EMP_ID"/>
    </secondary-table>
    ...
</entity>
```

Identifier Mappings

The three different types of identifier mappings may also be specified in XML. Overriding applies to the configuration information within a given identifier type, but the identifier type of a managed class should almost never be changed.

id

The `id` element is the most common method used to indicate the identifier for an entity. It corresponds to the `@Id` annotation but also encapsulates metadata that is relevant to identifiers. This includes a number of subelements, the first of which is the `column` subelement. It corresponds to the `@Column` annotation that might accompany an `@Id` annotation on the field or property. When not specified, the default column name will be assumed even if an `@Column` annotation exists on the field or property. As we discussed in the Attributes section previously, this is because the XML mapping of the attribute overrides the entire group of mapping metadata on the field or property.

A generated-value element corresponding to the @GeneratedValue annotation may also be included in the id element. This is used to indicate that the identifier will have its value automatically generated by the provider (see Chapter 4). This generated-value element has strategy and generator attributes that match those on the annotation. The named generator may be defined anywhere in the persistence unit. Sequence and table generators may also be defined within the id element. These were discussed in the Queries and Generators section.

An example of overriding an id mapping is to change the generator for a given database. This is shown in Listing 10-16.

Listing 10-16. *Overriding an Id Generator*

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.TABLE, generator="empTab")
    @TableGenerator(name="empTab", table="ID_GEN")
    private long id;
    // ...
}

<entity class="examples.model.Employee">
    ...
    <attributes>
        <id name="id">
            <generated-value strategy="SEQUENCE" generator="empSeq"/>
            <sequence-generator name="empSeq" sequence-name="mySeq"/>
        </id>
        ...
    </attributes>
</entity>
```

embedded-id

An embedded-id element is used when a compound primary key class is used as the identifier (see Chapter 8). It corresponds to the @EmbeddedId annotation and is really just mapping an embedded class as the identifier. All of the state is actually mapped within the embedded object, so there are only attribute overrides available within the embedded-id element. As we will discuss in the Embedded Object Mappings section, attribute overrides allow mapping of the same embedded object in multiple entities. The zero or more attribute-override elements in the property or field mapping of the entity provide the local overrides that apply to the entity table. Listing 10-17 shows how to specify an embedded identifier in annotation and XML form.

Listing 10-17. *Specifying an Embedded Id*

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



```
<entity class="examples.model.Employee">
    ...
    <attributes>
        <embedded-id name="id"/>
        ...
    </attributes>
</entity>
```

id-class

An id class is one strategy that can be used for a compound primary key (see Chapter 8). The id-class subelement of an entity or mapped-superclass element corresponds to the @IdClass annotation, and when it is specified in XML, it will override any @IdClass annotation on the class. Overriding the id class should not normally be done in practice since code that uses the entities will typically assume a particular identifier class.

The name of the class is indicated as the value of the class attribute of the id-class element as shown in Listing 10-18.

Listing 10-18. *Specifying an Id Class*

```
@Entity
@IdClass(EmployeePK.class)
public class Employee { ... }

<entity class="examples.model.Employee">
    ...
    <id-class="examples.model.EmployeePK"/>
    ...
</entity>
```

Simple Mappings

A simple mapping takes an attribute and maps it to a single column in a table. The majority of persistent state mapped by an entity will be composed of simple mappings. In this section we will discuss basic mappings and also cover the metadata for versioning and transient attributes.

basic

Basic mappings were discussed in detail in the early part of the book; they map a simple state field or property to a column in the table. The basic element provides this same ability in XML and corresponds to the @Basic annotation. Unlike the @Basic annotation that we described in Chapter 4 and which is rarely used, the basic element is required when mapping persistent state to a specific column. Just as with annotations, when a field or property is not mapped, it will be assumed to be a basic mapping and will be defaulted as such. This will occur if the field or property is not annotated or has no named entry in the attributes element.

In addition to a name, the basic element has fetch and optional attributes that can be used for lazy loading and optionality. These are not required and not very useful at the level of a field or property.

The most important and useful subelement of `basic` is the `column` element. Three other subelements may optionally be included inside the `basic` element. These are used to indicate the type to use when communicating with the JDBC driver to the database column. The first is an empty `lob` element that corresponds to the `@Lob` annotation. This is used when the target column is a large object type. Whether it is a character or binary object depends upon the type of the field or property.

The next is the `temporal` element that contains as its content one of "`DATE`", "`TIME`", or "`TIMESTAMP`". It corresponds to the `@Temporal` annotation and is used for fields of type `java.util.Date` or `java.util.Calendar`.

Finally, if the field or property is an enumerated type and the enumerated values are to be mapped using strings instead of ordinals, then the `enumerated` element should be used. It corresponds to the `@Enumerated` annotation and contains either `ORDINAL` or `STRING` as its content.

Listing 10-19 shows some examples of basic mappings. By not specifying the column in the `basic` element mapping for the `name` field, the column is overridden from using the annotated `EMP_NAME` column to being defaulted to `NAME`. The `comments` field, however, is overridden from using the default to being mapped to the `COMM` column. It is also stored in a CLOB (character large object) column due to the `lob` element being present and the fact that the field is a `String`. The `type` field is overridden to be mapped to the `STR_TYPE` column, and the enumerated type of `STRING` is specified to indicate that the values should be stored as strings. The `salary` field does not have any metadata either in annotation or XML form and continues to be mapped to the default column name of `SALARY`.

Listing 10-19. *Overriding Basic Mappings*

```
@Entity
public class Employee {
    // ...
    @Column(name="EMP_NAME")
    private String name;
    private String comments;
    private EmployeeType type;
    private long salary;
    // ...
}

<entity class="examples.model.Employee">
    ...
    <attributes>
        ...
        <basic name="name"/>
        <basic name="comments">
            <column name="COMM"/>
            <lob/>
        </basic>
        <basic name="type">
            <column name="STR_TYPE"/>
            <enumerated>STRING</enumerated>
        </basic>
        ...
    </attributes>
</entity>
```


transient

A `transient` element marks a field or property as being nonpersistent. It is equivalent to the `@Transient` annotation or having a transient qualifier on the field or property. Listing 10-20 shows an example of how to set a field to be transient.

Listing 10-20. *Setting a Transient Field in a Mapping File*

```
<entity-mappings>
    <entity class="examples.model.Employee">
        <attributes>
            <transient name="cacheAge"/>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

version

The `version` element is used to map the version number field in the entity. It corresponds to the `@Version` annotation and is normally mapped to an integral field for the provider to increment when it makes persistent changes to the entity (see Chapter 9). The `column` subelement specifies the column that stores the version data. Only one version field should exist for each entity. Listing 10-21 shows how a version field is specified in annotations and XML.

Listing 10-21. *Specifying the Version*

```
@Entity
public class Employee {
    // ...
    @Version
    private int version;
    // ...
}

<entity-mappings>
    <entity class="examples.model.Employee">
        <attributes>
            ...
            <version name="version"/>
            ...
        </attributes>
    </entity>
    ...
</entity-mappings>
```

Relationship Mappings

Like their annotation counterparts, the XML relationship elements are used to map the associations between entities. The following sections discuss each of the relationship mapping types that exist in XML.

many-to-one

To create a many-to-one mapping for a field or property, the `many-to-one` element may be specified. This element corresponds to the `@ManyToOne` annotation and, like the basic mapping, has `fetch` and `optional` attributes. Normally the target entity is known by the provider since the field or property is almost always of the target entity type. The `target-entity` attribute may also be specified.

A `join-column` element may be specified as a subelement of the `many-to-one` element when the column name is different than the default. If the association is to an entity with a compound primary key, then multiple `join-column` elements will be required. Mapping an attribute using a `many-to-one` element causes the mapping annotations that may have been present on that attribute to be ignored. All of the mapping information for the relationship, including the join column information, must be specified or defaulted within the `many-to-one` XML element.

Instead of a join column, it is possible to have a many-to-one or one-to-many relationship that uses a join table. It is for this case that a `join-table` element may be specified as a subelement of the `many-to-one` element. The `join-table` element corresponds to the `@JoinTable` annotation and contains a collection of `join-column` elements that join to the owning entity, which is normally the many-to-one side. A second set of join columns joins the join table to the inverse side of the relationship. These are called `inverse-join-column` elements. In the absence of one or both of these, the default values will be applied.

Unique to relationships is the ability to cascade operations across them. The cascade settings for a relationship dictate which operations are cascaded to the target entity of the many-to-one mapping. To specify how cascading should occur, a cascade element should be included as a subelement of the `many-to-one` element. Within the `cascade` element, we can include our choice of empty `cascade-all`, `cascade-persist`, `cascade-merge`, `cascade-remove`, or `cascade-refresh` subelements that dictate that the given operations be cascaded. Of course, specifying cascade elements in addition to the `cascade-all` element is simply redundant.

Now we come to an exception to our rule that we gave earlier when we said that overriding of mappings will typically be for physical data overrides. When it comes to relationships, there are times where you will want to test the performance of a given operation and would like to be able to set certain relationships to load eagerly or lazily. You will not want to go through the code and have to keep changing these settings back and forth, however. It would be more practical to have the mappings that you are tuning in XML and just change them according to your whim.[3] Listing 10-22 shows overriding two many-to-one relationships to be lazily loaded.

3. Some have argued that these kinds of tuning exercises are precisely some of the reasons why XML should be used to begin with.

Listing 10-22. *Overriding Fetch Mode*

```
@Entity
public class Employee {
    // ...
    @ManyToOne
    private Address address;
    @ManyToOne
    @JoinColumn(name="MGR")
    private Employee manager;
    // ...
}
```

```
<entity class="examples.model.Employee">
    ...
    <attributes>
        ...
        <many-to-one name="address" fetch="LAZY"/>
        <many-to-one name="manager" fetch="LAZY">
            <join-column name="MGR"/>
        </many-to-one>
        ...
    </attributes>
</entity>
```

one-to-many

A one-to-many mapping is created by using a `one-to-many` element. This element corresponds to the `@OneToMany` annotation and has the same optional `target-entity` and `fetch` attributes that were described in the many-to-one mapping. It has an additional attribute called `mapped-by`, which indicates the field or property of the owning entity (see Chapter 4).

A one-to-many mapping is a collection-valued association, and the collection may be a `List`, `Map`, `Set`, or `Collection`. If it is a `List`, then the elements may be populated in a specific order by specifying an `order-by` subelement. This element corresponds to the `@OrderBy` annotation and will cause the contents of the list to be ordered by the specific field or property name that is specified in the element content.

If the collection is a `Map`, then an optional `map-key` subelement may be specified to indicate the name of the field or property to use as the key for the `Map`. This element corresponds to the `@MapKey` annotation and will default to the primary key field or property when none is specified.

A join table is used to map a unidirectional one-to-many association that does not store a join column in the target entity. To make use of this kind of mapping, the `mapped-by` attribute is omitted and the `join-table` element is included.

Finally, cascading across the relationship is specified through an optional `cascade` element. Listing 10-23 shows a bidirectional one-to-many mapping both in annotations and XML.

Listing 10-23. *Specifying a One-to-Many Mapping*

```
@Entity
public class Employee {
    // ...
    @OneToMany(mappedBy="manager")
    @OrderBy
    private List<Employee> directs;
    @ManyToOne
    private Employee manager;
    // ...
}

<entity class="examples.model.Employee">
    ...
    <attributes>
        ...
        <one-to-many name="directs" mapped-by="manager">
            <order-by/>
        </one-to-many>
        <many-to-one name="manager"/>
        ...
    </attributes>
</entity>
```

one-to-one

To map a one-to-one association, the `one-to-one` element must be used. This element corresponds to the `@OneToOne` annotation that we described in Chapter 4 and has the same `target-entity`, `fetch`, and `optional` attributes that the `many-to-one` element has. It also has the `mapped-by` attribute that we saw in the one-to-many mapping to refer to the owning entity.

A `one-to-one` element may contain a `join-column` element if it is the owner of the relationship, or it may have multiple `join-column` elements if the association is to an entity with a compound primary key.

When the one-to-one association is joined using the primary keys of the two entity tables, then the `one-to-one` element will contain a `primary-key-join-column` element, which corresponds to the `@PrimaryKeyJoinColumn` annotation. When it has a compound primary key, multiple `primary-key-join-column` elements will be present. Either of `primary-key-join-column` or `join-column` elements may be present, but not both.

The annotated classes and XML mapping file equivalents for a one-to-one mapping using a primary key join column are shown in Listing 10-24.

Listing 10-24. *One-to-One Primary Key Association*

```
@Entity
public class Employee {
    // ...
    @OneToOne(mappedBy="employee")
    private ParkingSpace parkingSpace;
    // ...
}

@Entity
public class ParkingSpace {
    // ...
    @OneToOne
    @PrimaryKeyJoinColumn
    private Employee employee;
    // ...
}

<entity-mappings>
    <entity class="examples.model.Employee">
        <attributes>
            ...
            <one-to-one name="parkingSpace" mapped-by="employee"/>
            ...
        </attributes>
    </entity>
    <entity class="examples.model.ParkingSpace">
        <attributes>
            ...
            <one-to-one name="employee">
                <primary-key-join-column/>
            </one-to-one>
            ...
        </attributes>
    </entity>
</entity-mappings>
```

many-to-many

Creating a many-to-many association is done through the use of a `many-to-many` element. This element corresponds to the `@ManyToMany` annotation (see Chapter 4) and has the same optional `target-entity`, `fetch`, and `mapped-by` attributes that were described in the one-to-many mapping.

Also, being a collection-valued association like the one-to-many mapping, it supports the same `order-by`, `map-key`, `join-table`, and `cascade` subelements as the one-to-many mapping. Listing 10-25 shows the entity classes and equivalent XML.

Listing 10-25. *Many-to-Many Mapping Annotations and XML*

```
@Entity
public class Employee {
    // ...
    @ManyToMany
    @MapKey(name="name")
    @JoinTable(name="EMP_PROJ",
          joinColumns=@JoinColumn(name="EMP_ID"),
          inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
    private Map<String, Project> projects;
    // ...
}

@Entity
public class Project {
    // ...
    private String name;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    // ...
}

<entity-mappings>
    <entity class="examples.model.Employee">
        <attributes>
            ...
            <many-to-many name="projects">
                <map-key name="name"/>
                <join-table name="EMP_PRJ">
                    <join-column name="EMP_ID"/>
                    <inverse-join-column name="PROJ_ID"/>
                </join-table>
            </many-to-many>
            ...
        </attributes>
    </entity>
    <entity class="examples.model.Project">
        <attributes>
            ...
            <many-to-many name="employee" mapped-by="projects"/>
            ...
        </attributes>
    </entity>
</entity-mappings>
```


Embedded Object Mappings

An embedded object is a class that depends upon its parent entity for its identity. Embedded objects are specified in XML using the `embedded` element and are customized using the `attribute-override` element.

embedded

An `embedded` element is used for mapping an embedded object contained within a field or property (see Chapter 8). It corresponds to the `@Embedded` annotation. Since the persistent state is mapped within the embedded object, only the `attribute-override` subelement is allowed within the `embedded` element.

There must be an `embeddable` class entry in a mapping file for the embedded object or it must be annotated as `@Embeddable`. An example of overriding an embedded `Address` is shown in Listing 12-26.

Listing 12-26. *Embedded Mappings in Annotations and XML*

```
@Entity
public class Employee {
    // ...
    @Embedded
    private Address address;
    // ...
}

@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    private String zip;
    // ...
}

<entity-mappings>
    <entity class="examples.model.Employee">
        <attributes>
            ...
            <embedded name="address"/>
            ...
        </attributes>
    </entity>
    <embeddable class="examples.model.Address"/>
</entity-mappings>
```

attribute-override

When an embedded object is used by multiple entity types, it is likely that some of the basic mappings in the embedded object will need to be remapped by one or more of the entities (see Chapter 8). The `attribute-override` element may be specified as a subelement of `embedded` and `embedded-id` elements to accommodate this case.

The annotation that corresponds to the `attribute-override` element is the `@AttributeOverride` annotation. This annotation may be on the entity class or on a field or property that stores an embedded object or embedded id. When an `@AttributeOverride` annotation is present in the entity, it will be overridden only by an `attribute-override` element in the entity mapping file entry that specifies the same named field or property. Our earlier algorithm still holds if we think of the attribute overrides as keyed by the name of the field or property that they are overriding. All of the annotation overrides for an entity are gathered, then all of the XML overrides for the class are applied on top of the annotation overrides. If there is an override in XML for the same named field or property, it will overwrite the annotated one. The remaining non-overlapping overrides from annotations and XML will also be applied.

The `attribute-override` element stores the name of the field or property in its `name` attribute and the column that the field or property maps to as a `column` subelement. Listing 10-27 revisits Listing 10-26 and overrides the `state` and `zip` fields of the embedded address.

Listing 10-27. *Using Attribute Overrides*

```
@Entity
public class Employee {
    // ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROV")),
        @AttributeOverride(name="zip", column=@Column(name="PCODE"))})
    private Address address;
    // ...
}

<entity class="examples.model.Employee">
    <attributes>
        ...
        <embedded name="address">
            <attribute-override name="state">
                <column name="PROV"/>
            </attribute-override>
            <attribute-override name="zip">
                <column name="PCODE"/>
            </attribute-override>
        </embedded>
        ...
    </attributes>
</entity>
```


Inheritance Mappings

An entity inheritance hierarchy is mapped using the `inheritance`, `discriminator-column`, and `discriminator-value` elements. If the inheritance strategy is changed, then it must be overridden for the entire entity hierarchy.

inheritance

The `inheritance` element is specified to indicate the root of an inheritance hierarchy. It corresponds to the `@Inheritance` annotation and indicates the inheritance mapping strategy that is to be used. When it is included in the `entity` element, it will override any inheritance strategy that is defined or defaulted in the `@Inheritance` annotation on the entity class.

Changing the inheritance strategy may cause repercussions that spill out into the other areas. For example, changing a strategy from single-table to joined will likely require adding a table to each of the entities below it. The example in Listing 10-28 overrides an entity hierarchy from using a single table to using a joined strategy.

Listing 10-28. *Overriding an Inheritance Strategy*

```
@Entity
@Table(name="EMP")
@Inheritance
@DiscriminatorColumn(name="TYPE")
public abstract class Employee { ... }

@Entity
@DiscriminatorValue("FT")
public class FullTimeEmployee { ... }

@Entity
@DiscriminatorValue("PT")
public class PartTimeEmployee { ... }

<entity-mappings>
    <entity class="examples.model.Employee">
        <table name="EMP"/>
        <inheritance strategy="JOINED"/>
        ...
    </entity>
    <entity class="examples.model.FullTimeEmployee">
        <table name="FT_EMP"/>
        ...
    </entity>
    <entity class="examples.model.PartTimeEmployee">
        <table name="PT_EMP"/>
        ...
    </entity>
</entity-mappings>
```

discriminator-column

Discriminator columns store values that differentiate between concrete entity subclasses in an inheritance hierarchy (see Chapter 8). The `discriminator-column` element is a subelement of the `entity` or `entity-result` elements and is used to define or override the discriminator column. It corresponds to and overrides the `@DiscriminatorColumn` annotation and has attributes that include the `name`, `discriminator-type`, `columnDefinition`, and `length`. It is an empty element that has no subelements.

The `discriminator-column` element is not typically used to override a column on its own but in conjunction with other inheritance and table overrides. Listing 10-29 demonstrates specifying a discriminator column.

Listing 10-29. *Specifying a Discriminator Column*

```
@Entity
@Inheritance
@DiscriminatorColumn(name="TYPE")
public abstract class Employee { ... }

<entity class="examples.model.Employee">
    <inheritance/>
    <discriminator-column name="TYPE"/>
    ...
</entity >
```

discriminator-value

A `discriminator-value` element is used to declare the value that identifies the concrete entity subclass that is stored in a database row (see Chapter 8). It exists only as a subelement of the `entity` element. The discriminator value is indicated by the content of the element. It has no attributes or subelements.

The `discriminator-value` element corresponds to the `@DiscriminatorValue` annotation and overrides it when it exists on the entity class. As with the other inheritance overrides, it is seldom used as an override. Even when a hierarchy is remapped to a different database or set of tables, it will not be normally be necessary to override the value. Listing 10-30 shows how to specify a discriminator value in annotation and XML form.

Listing 10-30. *Specifying a Discriminator Column*

```
@Entity
@DiscriminatorValue("FT")
public class FullTimeEmployee extends Employee { ... }

<entity class="examples.model.FullTimeEmployee">
    <discriminator-value>FT</discriminator-value>
    ...
</entity >
```


association-override

Association overrides are similar to attribute overrides except that they are used to override single-valued associations instead of simple persistent state. The `association-override` element is a subelement of `entity` and corresponds to the `@AssociationOverride` annotation (see Chapter 8). The XML override rules are the same as those for `attribute-override` elements described in the Embedded Object Mappings section and are based on the name of the association field or property. Association overrides do not apply to embedded objects, however, because embedded objects may not portably have relationships within them.

Simple mappings and associations may be overridden through the use of attribute overrides and association overrides, but only in the case of an entity that is the subclass of a mapped superclass. Simple persistent state or association state that is inherited from an entity superclass may not portably be overridden.

There is one other difference that distinguishes the `association-override` element from its `attribute-override` counterpart. Single-valued relationships have one or more join columns instead of a column, which requires that they contain one or more `join-column` elements instead of a single `column` element. There may be multiples because foreign keys may be compound and require more than one join column.

An example of overriding two simple `name` and `salary` persistent field mappings, and a `manager` association with a compound primary key, is shown in Listing 10-31.

Listing 10-31. *Using Attribute and Association Overrides*

```
@MappedSuperclass
@IdClass(EmployeePK.class)
public abstract class Employee {
    @Id private String name;
    @Id private java.sql.Date dob;
    private long salary;
    @ManyToOne
    private Employee manager;
    // ...
}

@Entity
@Table(name="PT_EMP")
@AttributeOverrides({
    @AttributeOverride(name="name", column=@Column(name="EMP_NAME")),
    @AttributeOverride(name="salary", column=@Column(name="SAL"))})
@AssociationOverride(name="manager",
    joinColumns={
        @JoinColumn(name="MGR_NAME", referencedName="EMP_NAME"),
        @JoinColumn(name="MGR_DOB", referencedName="DOB")})
public class PartTimeEmployee extends Employee { ... }
```

```
<entity class="examples.model.PartTimeEmployee">
    ...
    <attribute-override name="name">
        <column name="EMP_NAME"/>
    </attribute-override>
    <attribute-override name="salary">
        <column name="SAL"/>
    </attribute-override>
    <association-override name="manager">
        <join-column name="MGR_NAME"  referenced-column-name="EMP_NAME"/>
        <join-column name="MGR_DOB" referenced-column-name="DOB"/>
    </association-override>
    ...
</entity>
```

Lifecycle Events

All of the lifecycle events that can be associated with a method in an entity listener can also be associated directly with a method in an entity or mapped superclass (see Chapter 9). The `pre-persist`, `post-persist`, `pre-update`, `post-update`, `pre-remove`, `post-remove`, and `post-load` methods are all valid subelements of the `entity` or `mapped-superclass` elements. Each of these may occur only once in each class. Each lifecycle event element will override any entity callback method of the same event type that may be annotated in the entity class.

Before anyone goes out and overrides all their annotated callback methods with XML overrides, we should mention that the use case for doing such a thing borders on, if not completely falls off into, the non-existent. An example of specifying an entity callback method in annotations and in XML is shown in Listing 10-32.

Listing 10-32. *Specifying Lifecycle Callback Methods*

```
@Entity
public class Employee {
    // ...
    @PrePersist
    @PostLoad
    public void initTransientState() { ... }
    // ...
}
```

```
<entity class="examples.model.Employee">
    ...
    <pre-persist method-name="initTransientState"/>
    <post-load method-name="initTransientState"/>
    ...
</entity>
```


Entity Listeners

Lifecycle callback methods defined on a class other than the entity class are called entity listeners. The following sections describe how to configure entity listeners in XML using the `entity-listeners` element and how to exclude inherited and default listeners.

entity-listeners

One or more ordered entity listener classes may be defined in an `@EntityListeners` annotation on an entity or mapped superclass (see Chapter 9). When a lifecycle event fires, the listeners that have methods for the event will get invoked in the order in which they are listed. The `entity-listeners` element may be specified as a subelement of an `entity` or `mapped-superclass` element to accomplish exactly the same thing. It will also have the effect of overriding the entity listeners defined in an `@EntityListeners` annotation with the ones defined in the `entity-listeners` element.

An `entity-listeners` element includes a list of ordered `entity-listener` subelements, each of which defines an entity-listener class in its `class` attribute. For each listener, the methods corresponding to lifecycle events must be indicated as subelement events. The events may be one or more of `pre-persist`, `post-persist`, `pre-update`, `post-update`, `pre-remove`, `post-remove`, and `post-load`, which correspond to the `@PrePersist`, `@PostPersist`, `@PreUpdate`, `@PostUpdate`, `@PreRemove`, `@PostRemove`, and `@PostLoad` annotations, respectively. Each of the event subelements has a `method-name` attribute that names the method to be invoked when its lifecycle event is triggered. The same method may be supplied for multiple events, but no more than one event of the same type may be specified on a single listener class.

The `entity-listeners` element can be used to disable all of the entity listeners defined on a class or just add an additional listener. Disabling listeners is not recommended, of course, since listeners defined on a class tend to be fairly coupled to the class itself, and disabling them might introduce bugs into either the class or the system as a whole.

In Listing 10-33 we see that the XML mapping file is overriding the entity listeners on the `Employee` class. It is keeping the existing ones but also adding one more at the end of the order to notify the IT department to remove an employee's user accounts when he or she leaves the company.

Listing 10-33. *Overriding Entity Listeners*

```
@Entity
@EntityListeners({ EmployeeAuditListener.class, NameValidator.class })
public class Employee { ... }

public class EmployeeAuditListener {
    @PostPersist
    public void employeeCreated(Employee emp) { ... }
    @PostUpdate
    public void employeeUpdated(Employee emp) { ... }
    @PostRemove
    public void employeeRemoved(Employee emp) { ... }
}
```

```
public class NameValidator {
    @PrePersist
    public void validateName(Employee emp) { ... }
}
public class EmployeeExitListener {
    public void notifyIT(Employee emp) { ... }
}

<entity class="examples.model.Employee">
    ...
    <entity-listeners>
        <entity-listener class="examples.listeners.EmployeeAuditListener">
            <post-persist method-name="employeeCreated"/>
            <post-update method-name="employeeUpdated"/>
            <post-remove method-name="employeeRemoved"/>
        </entity-listener>
        <entity-listener class="examples.listeners.NameValidator">
            <pre-persist method-name="validateName"/>
        </entity-listener>
        <entity-listener class="examples.listeners.EmployeeExitListener">
            <post-remove method-name="notifyIT"/>
        </entity-listener>
    </entity-listeners>
    ...
</entity>
```

Note that we have fully specified each of the entity callback listeners in XML. Some vendors will find the lifecycle event annotations on the `EmployeeAuditListener` and `NameValidator` entity listener classes, but this is not required behavior. To be portable, the lifecycle event methods should be specified in each of the `entity-listener` elements.

exclude-default-listeners

The set of default entity listeners that applies to all entities is defined in the `entity-listeners` subelement of the `persistence-unit-defaults` element (see the `entity-listeners` section). These listeners can be turned off or disabled for a particular entity or hierarchy of entities by specifying an empty `exclude-default-listeners` element within the `entity` or `mapped-superclass` element. This is equivalent to the `@ExcludeDefaultListeners` annotation, and if either one is specified for a class, then default listeners are disabled for that class. Note that `exclude-default-listeners` is an empty element, not a `Boolean`. If default entity listeners are disabled for a class by an `@ExcludeDefaultListeners` annotation, then there is currently no way to re-enable them through XML.

exclude-superclass-listeners

Entity listeners defined on the superclass of an entity will normally be fired before the entity listeners defined on the entity class itself are fired (see Chapter 9). To disable the listeners defined on an entity superclass or mapped superclass, an empty `exclude-superclass-listeners` element may be supplied inside an `entity` or `mapped-superclass` element. This will disable the superclass listeners for the managed class and all of its subclasses.

The `exclude-superclass-listeners` element corresponds to the `@ExcludeSuperclassListeners` annotation and, like the `exclude-default-listeners`/`@ExcludeDefaultListeners` pair, either one of the two may be specified in order to disable the superclass listeners for the entity or mapped superclass and its subclasses.

Summary

With all of the XML mapping information under your belt, you should now be able to map entities using annotations, XML, or a combination of the two. In this chapter we went over all of the elements in the mapping file and compared them to their corresponding annotations. We discussed how each of the elements is used, what they override, and how they are overridden. We also used them in some short examples.

Defaults may be specified in the mapping files at different levels, from the global persistence unit level to the mapping file level. We covered what each of the defaulting scopes was and how they were applied.

In the next chapter we will look at how to package and deploy applications that use the Java Persistence API. We will also look at how XML mapping files are referenced as part of a persistence unit configuration.

CHAPTER 11

Packaging and Deployment

Configuring a persistence application involves specifying the bits of additional information that the execution environment or persistence platform may require in order for the code to function as a runtime application. Packaging means putting all of the pieces together in a way that makes sense and can be correctly interpreted and used by the infrastructure when the application is deployed into an application server or run in a stand-alone JVM. Deployment is the process of getting the application into an execution environment and running it.

One could view the mapping metadata as part of the overall configuration of an application, but that has already been discussed in previous chapters. In this chapter we will be discussing the primary runtime persistence configuration file, `persistence.xml`, which defines persistence units. We will go into detail about how to specify the different elements of this file, when they are required, and what the values should be.

Once the persistence unit has been configured, we will package a persistence unit with a few of the more common deployment units, such as EJB archives, web archives, and the application archives in a Java EE server. The resulting package will then be deployable into a compliant application server. We will also step through the packaging and deployment rules for Java SE applications.

Configuring Persistence Units

The persistence unit is the primary unit of runtime configuration. It defines the various pieces of information that the provider needs to know in order to manage the persistent classes during program execution and is configured within a `persistence.xml` file. There may be one or more `persistence.xml` files in an application, and each `persistence.xml` file may define multiple persistence units. There will normally be only one, though. Since there is one `EntityManagerFactory` for each persistence unit, you can think of the configuration of the persistence unit as the configuration of the factory for that persistence unit.

A common configuration file goes a long way to standardizing the runtime configuration, and the `persistence.xml` file offers exactly that. While some providers might still require an additional provider-specific configuration file, most will also support their properties being specified within the properties section (described in the Adding Vendor Properties section) of the `persistence.xml` file.

The `persistence.xml` file is the first step to configuring a persistence unit. All of the information required for the persistence unit should be specified in the `persistence.xml` file. Once a packaging strategy has been chosen, the `persistence.xml` file should be placed in the `META-INF` directory of the chosen archive.

Each persistence unit is defined by a `persistence-unit` element in the `persistence.xml` file. All of the information for that persistence unit is enclosed within that element. The following sections describe the metadata that a persistence unit may define when deploying to a Java EE server.

Persistence Unit Name

Every persistence unit must have a name that uniquely identifies it within the scope of its packaging. We will be discussing the different packaging options later, but in general, if a persistence unit is defined within a Java EE module, then there must not be any other persistence unit of the same name in that module. For example, if a persistence unit named "EmployeeService" is defined in an EJB JAR named `emp_ejb.jar`, then there should not be any other persistence units named "EmployeeService" in `emp_ejb.jar`. There may be persistence units named "EmployeeService" in a web module or even in another EJB module within the application though.

We have seen in some of the examples in previous chapters that the name of the persistence unit is just an attribute of the `persistence-unit` element, as in:

```
<persistence-unit name="EmployeeService"/>
```

This empty `persistence-unit` element is the minimal persistence unit definition. It may be all that is needed if the server defaults the remaining information, but not all servers will do this. Some may require other persistence unit metadata to be present, such as the data source to be accessed.

Transaction Type

The factory that is used to create entity managers for a given persistence unit will generate entity managers to be of a specific transactional type. We went into detail in Chapter 5 about the different types of entity managers, and one of the things that we saw was that every entity manager must either use JTA or resource-local transactions. Normally, when running in a managed server environment, the JTA transaction mechanism is used. It is the default transaction type that a server will assume when none is specified for a persistence unit and generally the only one that most applications will ever need, so in practice the transaction type will never need to be specified.

If the data source is required by the server, as it often will be, then a JTA-enabled data source should be supplied (see section Data Source). Specifying a data source that is not JTA-enabled may actually work in some cases, but the database operations will not be participating in the global JTA transaction or necessarily be atomic with respect to that transaction.

In situations like those described in Chapter 5, when you want to use resource-local transactions instead of JTA, the `transaction-type` attribute of the `persistence-unit` element is used to explicitly declare the transaction type of `RESOURCE_LOCAL` or `JTA`, as in the following example:

```
<persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL"/>
```

Here, we are overriding the default JTA transaction type to be resource-local, so all of the entity managers created in the "EmployeeService" persistence unit must use the `EntityTransaction` interface to control transactions.

Persistence Provider

The Java Persistence API has a pluggable Service Provider Interface (SPI) that allows any compliant Java EE server to communicate with any compliant persistence provider implementation. Servers normally have a default provider, though, that is native to the server, meaning that it is implemented by the same vendor or is shipped with the server. In most cases, this default provider will be used by the server, and no special metadata will be necessary to explicitly specify it.

In order to switch to a different provider, the provider-supplied class that implements the `javax.persistence.spi.PersistenceProvider` interface must be listed in the `provider` element. Listing 11-1 shows a simple persistence unit that explicitly defines the TopLink Essentials provider class. The only requirement is that the provider JARs be on the server or application classpath and accessible to the running application at deployment time.

Listing 11-1. *Specifying a Persistence Provider*

```
<persistence-unit name="EmployeeService">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
</persistence-unit>
```

Data Source

A fundamental part of the persistence unit metadata is the description of where the provider should obtain database connections from in order to read and write entity data. The target database is specified in terms of the name of a JDBC data source that is usually in the server JNDI space. This data source must be globally accessible since the provider accesses it when the persistence application is deployed.

The typical case is that JTA transactions are used, so it is in the `jta-data-source` element that the name of the JTA data source should be specified. Similarly, if the transaction type of the persistence unit is resource-local, then the `non-jta-data-source` element should be used.

There is no standard format for specifying the name of the data source; it is totally vendor-specific. However, the *de facto* standard way that a data source is accessed is from JNDI. Normally, a data source is made available in JNDI by being configured in a server-specific configuration file or management console. Even though the name is not officially portable, at least the place where it can be specified is standard, and in practice the names will usually be of the form "jdbc/myDataSource". Listing 11-2 shows how a data source would normally be specified. This example assumes that the provider is being defaulted.

Listing 11-2. *Specifying JTA Data Source*

```
<persistence-unit name="EmployeeService">
    <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
</persistence-unit>
```

Some servers actually provide a default data source at the deployed Java EE application level, and if the provider is a native implementation for the server, then it may make use of this default. In other cases the data source will need to be specified.

Some servers also offer high-performance reading through database connections that are not associated with the current JTA transaction. The query results are then returned and made

conformant with the contents of persistence context. This improves the scalability of the application because the database connection does not get enlisted in the JTA transaction until later on when it absolutely needs to be, usually at commit time. To enable these types of scalable reads, the `non-jta-data-source` element value would be supplied in addition to the `jta-data-source` element. An example of specifying these two is in Listing 11-3.

Listing 11-3. *Specifying JTA and Non-JTA Data Sources*

```
<persistence-unit name="EmployeeService">
    <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
    <non-jta-data-source>jdbc/NonTxEmployeeServiceDS</non-jta-data-source>
</persistence-unit>
```

Note that the "EmployeeServiceDS" is a regularly configured data source that accesses the employee database but that "NonTxEmployeeServiceDS" is a separate data source configured to access the same employee database but not be enlisted in JTA transactions.

Mapping Files

In Chapter 10 we used XML mapping files to supply mapping metadata. Part, or all of the mapping metadata for the persistence unit may be specified in mapping files. The union of all of the mapping files (and the annotations in the absence of `xml-mapping-metadata-complete`) will be the metadata that is applied to the persistence unit.

Some might wonder why multiple mapping files might be useful. There are actually numerous cases for using more than one mapping file in a single persistence unit, but it really comes down to preference and process. For example, you might want to define all of the persistence-unit-level artifacts in one file and then all of the entity metadata in another file. In another case it may make sense for you to group all of the queries together in a separate file to isolate them from the rest of the physical database mappings. Perhaps it suits the development process to even have a file for each entity, either to decouple them from each other or to reduce conflicts resulting from the version control and configuration management system. This can be a popular choice for a team that is working on different entities within the same persistence unit. Each may want to change the mappings for a particular entity without getting in the way of other team members who are modifying other entities. Of course this must be negotiated carefully when there really are dependencies across the entities such as relationships or embedded objects. It makes sense to group entity metadata together when the relationships between them are not static or when the object model may change. As a general rule, if there is strong coupling in the object model, then the coupling should be considered in the mapping configuration model.

Some might just prefer to have a single mapping file with all of the metadata contained within it. This is certainly a simpler deployment model and makes for easier packaging. There is built-in support available to those who are happy limiting their metadata to a single file and willing to name it "orm.xml". If a mapping file named "orm.xml" exists in a `META-INF` directory on the classpath, for example beside the `persistence.xml` file, then it does not need to be explicitly listed. The provider will automatically search for such a file and use it if one exists. Mapping files that are named differently or are in a different location must be listed in the `mapping-file` elements in the `persistence.xml` file.

Mapping files listed in the `mapping-file` elements are loaded as Java resources (using methods such as `ClassLoader.getResource()`, for example) from the classpath, so they should be specified in the same manner as any other Java resource that was intended to be loaded as such. The directory location component followed by the file name of the mapping file will cause it to be found, loaded, and processed at deployment time. For example, if we put all of our persistence-unit-level metadata in `META-INF/orm.xml`, all of our queries in `META-INF/employee_service_queries.xml`, and all of our entities in `META-INF/employee_service_entities.xml`, then we should end up with the persistence unit definition shown in Listing 11-4. Remember, we don't need to specify the `META-INF/orm.xml` file, because it will be found and processed by default. The other mapping files could be in any directory, not necessarily just the `META-INF` directory. We put them in `META-INF` just to keep them together with the `orm.xml` file.

Listing 11-4. *Specifying Mapping Files*

```
<persistence-unit name="EmployeeService">
    <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
    <mapping-file>META-INF/employee_service_queries.xml</mapping-file>
    <mapping-file>META-INF/employee_service_entities.xml</mapping-file>
</persistence-unit>
```

Managed Classes

Typical deployments will put all of the entities and other managed classes in a single JAR, with the `persistence.xml` file in the `META-INF` directory and one or more mapping files also tossed in when XML mapping is used. The deployment process is optimized for these kinds of deployment scenarios to minimize the amount of metadata that a deployer has to specify.

The set of entities, mapped superclasses, and embedded objects that will be managed in a particular persistence unit is determined by the provider when it processes the persistence unit. At deployment time it may obtain managed classes from any of four sources. A managed class will be included if it is among the following:

1. **Local Classes:** the annotated classes in the deployment unit in which its `persistence.xml` file was packaged.
2. **Classes in Mapping Files:** the classes that have mapping entries in an XML mapping file.
3. **Explicitly Listed Classes:** the classes that are listed as `class` elements in the `persistence.xml` file.
4. **Additional JARs of Managed Classes:** the annotated classes in a named `JAR` listed in a `jar-file` element in the `persistence.xml` file.

As a deployer you may choose to use any one or a combination of these mechanisms to cause your managed classes to be included in the persistence unit. We will discuss each of them in turn.

Local Classes

The first category of classes that get included is the one that is the easiest and will likely be used the most often. We call these classes local classes because they are local to the deployment unit. When a JAR is deployed with a `persistence.xml` file in the `META-INF` directory, then that JAR will be searched for all of the classes that are annotated with `@Entity`, `@MappedSuperclass`, or `@Embeddable`. This will hold true for various types of deployment units that we will describe in more detail later in the chapter.

This method is clearly the simplest way to cause a class to be included because all that has to be done is to put the annotated classes into a JAR and add the `persistence.xml` file in the `META-INF` directory of the JAR. The provider will take care of going through the classes and finding the entities. Other classes may also be placed in the JAR with the entities and will have no effect on the finding process other than perhaps potentially slowing down the finding process if there are many such classes.

Classes in Mapping Files

Any class that has an entry in a mapping file is also going to be considered a managed class in the persistence unit. It need only be named in an `entity`, `mapped-superclass`, or `embeddable` element in one of the mapping files. The set of all of the classes from all of the listed mapping files (including the implicitly processed `orm.xml` file) will be added to the set of managed classes in the persistence unit. Nothing special has to be done apart from ensuring that the classes named in a mapping file are on the classpath of the unit being deployed. If they are in the deployed component archive, then they will already be on the classpath. But if they aren't, then they must be explicitly included in the classpath just as the explicitly listed ones are (see the following Explicitly Listed Classes section).

Explicitly Listed Classes

When the persistence unit is small or when there are not a large number of entities, then you may want to list classes explicitly in `class` elements in the `persistence.xml` file. This will cause the listed classes to be added to the persistence unit.

Since a class that is local to the deployment unit will already be included, we don't need to list them in the `class` elements. Explicitly listing the classes is really useful only in three main cases.

The first is when there are additional classes that are not local to the deployment unit JAR. For example, there is an embedded object class in a different JAR that you want to use in an entity in your persistence unit. You would list the fully qualified class in the `class` element in the `persistence.xml` file. You will also need to ensure that the JAR or directory that contains the class is on the classpath of the deployed component, for example, by adding it to the manifest classpath of the deployment JAR.

In the second case, we want to exclude one or more classes that may be annotated as an entity. Even though the class may be annotated with `@Entity`, we don't want it to be treated as an entity in this particular deployed context. For example, it may be used as a transfer object and need to be part of the deployment unit. In this case we need to make use of a special element called `exclude-unlisted-classes` in the `persistence.xml` file, which disables local classes from being added to the persistence unit. When `exclude-unlisted-classes` is used, then none of the classes in the local classes category described earlier will be included.

The third case is when we expect to be running the application in a Java SE environment and when we list the classes explicitly because that is the only portable way to do so in Java SE. We will explain deployment to the Java SE non-server environment later in the chapter.

Additional JARs of Managed Classes

The last way to get managed classes included in the persistence unit is to add them to another JAR and specify the name in a `jar-file` element in the `persistence.xml`. The `jar-file` element is used to indicate to the provider a JAR that may contain annotated classes. The provider will then treat the named JAR as if it were a deployment JAR, and it will look for any annotated classes and add them to the persistence unit. It will even search for an `orm.xml` file in the `META-INF` directory in the JAR and process it just as if it were an additionally listed mapping file.

Any JAR listed in a `jar-file` entry must be on the classpath of the deployment unit. You must do this manually, though, since the server will not automatically do it for you. Again this may be done by adding the JAR to the manifest classpath of the deployment unit or by some other vendor-specific means.

When listing a JAR in a `jar-file` element, it must be listed relative to the parent of the JAR file in which the `persistence.xml` file is located. This matches what you would put in the classpath entry in the manifest. For example, assume the enterprise archive (EAR), that we will call `emp.ear`, is structured as shown in Listing 11-5.

Listing 11-5. *Using Entities in an External JAR*

```
emp.ear
    emp-ejb.jar
        META-INF/persistence.xml
    employee/emp-classes.jar
        examples/model/Employee.class
```

The contents of the `persistence.xml` file should be as shown in Listing 11-6, with the `jar-file` element containing "employee/emp-classes.jar" to reference the `emp-classes.jar` in the employee directory in the EAR file. This would cause the provider to add the annotated classes it found in `emp-classes.jar` (`Employee.class`) to the persistence unit.

Listing 11-6. *Contents of* `persistence.xml`

```
<persistence-unit name="EmployeeService">
    <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
    <jar-file>employee/emp-classes.jar</jar-file>
</persistence-unit>
```

Adding Vendor Properties

The last section in the `persistence.xml` file is the properties section. The `properties` element gives a deployer the chance to supply provider-specific settings for the persistence unit. To guarantee runtime compatibility, a provider must ignore properties it does not understand. While it is helpful to be able to use the same `persistence.xml` file across different providers,

it also makes it easy to mistakenly type a property incorrectly and have it unintentionally and silently ignored. An example of adding some vendor properties is shown in Listing 11-7.

Listing 11-7. *Using Provider Properties*

```
<persistence-unit name="EmployeeService">
    ...
    <properties>
        <property name="toplink.logging.level" value="FINE"/>
        <property name="toplink.cache.size.default" value="500"/>
    </properties>
</persistence-unit>
```

Building and Deploying

One of the big wins that a standard persistence API brings is not only a portable runtime API but also a common way to compose, assemble, and configure an application that makes use of persistence. In this section we will describe some of the popular and practical choices that are used to deploy persistence-enabled applications.

Deployment Classpath

In some of the previous sections we say that a class or a JAR must be on the deployment classpath. When we say this we mean that the JAR must be accessible to the EJB JAR, the web archive (WAR), or the enterprise application archive (EAR). This may be achieved in a few ways.

The first is by putting the JAR in the manifest classpath of the EJB JAR or WAR. This is done by adding a classpath entry to the `META-INF/MANIFEST.MF` file in the JAR or WAR. One or more directories or JARs may be specified, as long as they are separated by spaces. For example, the following manifest file classpath entry will add the `employee/emp-classes.jar` and the `employee/classes` directory to the classpath of the JAR that contains the manifest file:

```
Class-Path: employee/emp-classes.jar employee/classes
```

Another way to get a JAR into the deployment unit classpath is to place the JAR in the library directory of the EAR. When a JAR is in the library directory, then it will be on the application classpath and accessible by all of the modules deployed within the EAR. By default this would just be the `lib` directory of the EAR, although it may be configured to be any directory in the EAR using the `library-directory` element in the `application.xml` deployment descriptor. The `application.xml` file would look something like the skeletal one shown in Listing 11-8.

Listing 11-8. *Setting the Application Library Directory*

```
<application ... >
    ...
    <library-directory>myDir/jars</library-directory>
</application>
```


Vendors usually provide their own vendor-specific way for deployers to add classes or JARs to the deployment classpath. This is usually offered at the application level and not at the level of a JAR or WAR; however some may provide both.

Packaging Options

A primary focus of the Java Persistence API is its integration with the Java EE platform. Not only has it been integrated in fine-grained ways, such as allowing injection of entity managers into Java EE components, but it also has special status in Java EE application packaging. Java EE allows for persistence to be supported in a variety of packaging configurations that offer flexibility and choice. We will divide them up into the different module types that the application might be deployed to: EJB modules, web modules, and persistence archives.

EJB JAR

Modularized business logic typically ends up in session bean components, which is why session beans have always been the primary clients of persistence. It is not only fitting but also an essential part of the integration of the Java Persistence API with Java EE. Because session beans provide such a natural home for code that operates on entities, the best supported way to access and package entities will be with session beans in an EJB JAR. We assume that the reader is familiar with packaging and deploying EJB components in an EJB JAR, but if not, there is a host of books and resources available to learn about it.

In EJB 3.0 you no longer need to have an `ejb-jar.xml` deployment descriptor, but if you choose to use one, then it must be in the `META-INF` directory. When defining a persistence unit in an EJB JAR, the `persistence.xml` file is not optional. It must be created and placed in the `META-INF` directory of the JAR alongside the `ejb-jar.xml` deployment descriptor, if it exists. Although the existence of `persistence.xml` is required, the contents may be very sparse indeed, in some cases including only the name of the persistence unit.

The only real work in defining a persistence unit is to decide where we want our entities and managed classes to reside. As we saw in the preceding sections, we have a number of options available to us. The simplest approach is to simply dump our managed classes into the EJB JAR along with the EJB components. As long as they are correctly annotated, the entities will be automatically discovered by the provider at deployment time and added to the persistence unit. Listing 11-9 shows a sample enterprise application archive file that does this.

Listing 11-9. *Packaging Entities in an EJB JAR*

```
emp.ear
    emp-ejb.jar
        META-INF/persistence.xml
        META-INF/orm.xml
        examples/ejb/EmployeeService.class
        examples/ejb/EmployeeServiceBean.class
        examples/model/Employee.class
        examples/model/Phone.class
        examples/model/Address.class
        examples/model/Department.class
        examples/model/Project.class
```

In this case the orm.xml file contains any mapping information that we might have at the persistence-unit level, for example, setting the schema for the persistence unit. In the persistence.xml file we would need to specify only the name of the persistence unit and the data source. Listing 11-10 shows the corresponding persistence.xml file in its entirety.

Listing 11-10. Persistence.xml *File for Entities Packaged in an EJB JAR*

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
            http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
        version="1.0">
    <persistence-unit name="EmployeeService">
        <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
    </persistence-unit>
</persistence>
```

If we wanted to separate the entities from the EJB components, then we could put them in a different JAR and reference that JAR in a jar-file entry in the persistence.xml file. We showed a simple example of doing this in the Additional JARs of Managed Classes section, but we will show one again here with an additional orm.xml file and emp-mappings.xml mapping file. Listing 11-11 shows what the structure and contents of the EAR would look like.

Listing 11-11. *Packaging Entities in a Separate JAR*

```
emp.ear
    emp-ejb.jar
        META-INF/persistence.xml
        examples/ejb/EmployeeService.class
        examples/ejb/EmployeeServiceBean.class
    emp-classes.jar
        META-INF/orm.xml
        META-INF/emp-mappings.xml
        examples/model/Employee.class
        examples/model/Phone.class
        examples/model/Address.class
        examples/model/Department.class
        examples/model/Project.class
```

The emp-classes.jar file containing the entities would need to be on the classpath as described in the Deployment Classpath section. In addition to processing the entities found in the emp-classes.jar file, the orm.xml file in the META-INF directory will also be detected and processed automatically. We need to explicitly list the additional emp_mappings.xml mapping file in a mapping-file element, though, in order for the provider to find it as a resource. The persistence unit portion of the persistence.xml file is shown in Listing 11-12.

Listing 11-12. Persistence.xml *File for Entities Packaged in a Separate JAR*

```
<persistence-unit name="EmployeeService">
    <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
    <mapping-file>META-INF/emp-mappings.xml</mapping-file>
    <jar-file>emp-classes.jar</jar-file>
</persistence-unit>
```

Web Archive

We have not been shy about the fact that we believe that session beans are the best way to access entities. Regardless of what we say, though, there will be people who for one reason or another don't want or are not able to use session beans. Operating on entities directly from the web tier is still a valid option and may continue to be popular for a subset of web developers.

The WAR is a little more complex than the EJB JAR is, and learning to package persistence units in web archives requires understanding the relevance of the persistence.xml file location. The location of the persistence.xml file determines the *persistence unit root.* The root of the persistence unit is defined as the JAR or directory that contains the META-INF directory where the persistence.xml file is located. For example, in an EJB JAR the persistence.xml file is located in the META-INF directory of the root of the JAR, so the root of the persistence unit is always the root of the EJB JAR file itself. In a WAR the persistence unit root is the WEB-INF/classes directory, so the persistence.xml file should be placed in the WEB-INF/classes/META-INF directory. Any annotated managed classes rooted in the WEB-INF/classes directory will be detected and added to the persistence unit. Similarly, if an orm.xml file is located in WEB-INF/classes/META-INF, then it will be processed. An example of packaging a persistence unit in the WEB-INF/classes directory is shown in Listing 11-13.

Listing 11-13. *Packaging Entities in the* WEB-INF/classes *Directory*

```
emp.ear
    emp.war
        WEB-INF/web.xml
        WEB-INF/classes/META-INF/persistence.xml
        WEB-INF/classes/META-INF/orm.xml
        WEB-INF/classes/examples/web/EmployeeServlet.class
        WEB-INF/classes/examples/model/Employee.class
        WEB-INF/classes/examples/model/Phone.class
        WEB-INF/classes/examples/model/Address.class
        WEB-INF/classes/examples/model/Department.class
        WEB-INF/classes/examples/model/Project.class
```

The persistence.xml file would be specified in exactly the same way as is shown in Listing 11-10. If we need to add another mapping file then we can put it anywhere on the deployment unit classpath. We just need to add a mapping-file element to the persistence.xml file. If, for example, we put emp-mapping.xml in the WEB-INF/classes/mapping directory, then we would add the following element to the persistence.xml file:

```
<mapping-file>mapping/emp-mapping.xml</mapping-file>
```

Since the WEB-INF/classes directory is automatically on the classpath of the WAR, the mapping file is specified relative to that directory.

Persistence Archive

If we want to allow a persistence unit to be shared or accessible by multiple components in different Java EE modules, then we should use a persistence archive. We saw a simple persistence archive back in Chapter 2 when we were first getting started and observed how it housed the persistence.xml file and the managed classes that were part of the persistence unit defined within it. By placing a persistence archive in the EAR, we can make it available to any component that needs to operate on the entities defined by its contained persistence unit.

The persistence archive is simple to create and easy to deploy. It is simply a JAR that contains a persistence.xml in its META-INF directory and the managed classes for the persistence unit defined by the persistence.xml file. In fact, an EJB JAR is really doubling as a persistence archive when it contains a META-INF/persistence.xml and the managed classes of the persistence unit. Now, with a persistence archive, we can define the persistence unit outside the EJB JAR but still use it from within the EJB JAR.

Listing 11-14 shows the contents of the simple persistence archive that defines the persistence unit that we have been using in the previous examples.

Listing 11-14. *Packaging Entities in a Persistence Archive*

```
emp.ear
    emp-persistence.jar
        META-INF/persistence.xml
        META-INF/orm.xml
        examples/model/Employee.class
        examples/model/Phone.class
        examples/model/Address.class
        examples/model/Department.class
        examples/model/Project.class
```

If this archive looks familiar it is because it is virtually the same as the structure that we defined in Listing 11-9 except that it is a persistence archive JAR instead of an EJB JAR. We just changed the name of the JAR and took out the session bean classes. The contents of the persistence.xml file are exactly the same as what is shown in Listing 11-10. Just like with the other archive types, the orm.xml file in the META-INF directory will be automatically detected and processed, and other XML mapping files may be placed within the JAR and referenced by the persistence.xml file as a mapping-file entry.

Once created, the persistence archive may be placed in either the root or the application library directory of the EAR. The advantage of placing it in the library directory is that it will automatically be on the application classpath and shared by all of the application components. If it is in the root directory of the EAR, then it will need to be incorporated into the application classpath in some vendor-specific way. In either case, though, the entity classes in the persistence archive will be loaded by the application class loader. This will enable the same class definition to be accessible to all of the components in the application.

As an alternative to putting the classes loose inside the web archive, a persistence archive may also be placed in the WEB-INF/lib directory of a WAR. This will make the persistence unit accessible only to the classes inside the WAR, but it allows the definition of the persistence unit to be decoupled from the web archive itself.

Managed classes may also be stored in a separate JAR external to the persistence archive, just as they could be in other packaging archive configurations. The external JAR would be referenced by the persistence.xml file as a jar-file entry with the same rules for specification as were described in the other cases. This is neither recommended nor useful, though, since the persistence archive itself is already separated from the other component classes. Seldom will there be a reason to create yet another JAR to store the managed classes, but there may be a case when the other JAR is pre-existing, and you need to reference it because you can't or don't want to put the persistence.xml file in the pre-existing JAR.

Persistence archives are actually a very tidy way of packaging a persistence unit. By keeping them self-contained (if they do not reference external JARs of classes using jar-file entries), they do not depend upon any other components of the application but can sit as a layer underneath those components to be used by them.

Persistence Unit Scope

For simplicity we have talked about a persistence unit in the singular. The truth is that any number of persistence units may be defined in the same persistence.xml file and be used in the scope within which they were defined. We saw in the preceding sections, when we discussed how managed classes get included in the persistence unit, that local classes in the same archive will be processed by default. If multiple persistence units are defined in the same persistence.xml file and exclude-unlisted-classes is not used on either one, then the same classes will be added to all of the defined persistence units. This may be a convenient way to import and transform data from one data source to another simply by reading in entities through one persistence unit and performing the transformation on them before writing them out through another persistence unit.

Now that we have defined and packaged our persistence units, we should outline the rules and ways to use them. There are only a few, but they are important to know.

The first rule is that persistence units are accessible only within the scope of their definition. We have already mentioned this in passing a couple of times, and we hinted at it again in the Persistence Archive section. We said that the persistence unit defined within a persistence archive at the EAR level was accessible to all of the components in the EAR, and that a persistence unit defined in a persistence archive in a WAR is accessible only to the components defined within that WAR. In fact, in general a persistence unit defined from an EJB JAR is seen by EJB components defined by that EJB JAR, and a persistence unit defined in a WAR will be seen only by the components defined within that WAR. Persistence units defined in a persistence archive that lives in the EAR will be seen by all of the components in the application.

The next part is that the names of persistence units must be unique within their scope. For example, there may be only one persistence unit of a given name within the same EJB JAR. Likewise there may be only one persistence unit of a given name in the same WAR, as well as only one persistence unit of the same name in all of the persistence archives at the EAR level. There may be a named persistence unit name in one EJB JAR and another that shares its name in another EJB JAR, or there may even be a persistence unit with the same name in an EJB JAR as there is in a persistence archive. It just means that whenever a persistence unit is referenced either within an @PersistenceContext, an @PersistenceUnit annotation, or a createEntityManagerFactory() method, then the most locally scoped one will get used.

A final note about naming is appropriate at this point. Just because it's possible to have multiple persistence units with the same name in different component archive namespaces doesn't mean that it is a good idea. As a general rule, you should always give persistence units unique names within the application.

Outside the Server

There are some obvious differences between deploying in a Java EE server and deploying to a Java SE runtime environment. For example, some of the Java EE container services are not going to be present, and this spills out into the runtime configuration information for a persistence unit. In this section we will outline the differences to consider when packaging and deploying to a Java SE environment.

Configuring the Persistence Unit

As before, the place to start is the configuration of the persistence unit, which is chiefly in the creation of the `persistence.xml` file. We will outline the differences between creating a `persistence.xml` file for a Java SE application and creating one for a Java EE application.

Transaction Type

When running in a server environment, the `transaction-type` attribute in the persistence unit defaults to being JTA. The JTA transaction layer was designed for use within the Java EE server and is intended to be fully integrated and coupled to the server components. There are currently no plans to make it pluggable, either, so the chances of getting a fully compliant JTA transaction manager to run in a Java SE environment are not only thin but also getting slimmer all the time. Given this fact, the Java Persistence API does not even provide support for using JTA outside the server. Some providers may offer this support, but it cannot be portably relied upon, and of course it relies upon the JTA component being present.

The transaction type does not normally need to be specified when deploying to Java SE. It will just default to being `RESOURCE_LOCAL` but may be specified explicitly to make the programming contract more clear.

Data Source

When we described configuration in the server, we illustrated how the `jta-data-source` element denotes the JNDI location of the data source that will be used to obtain connections. We also saw that some servers might even default the data source.

The `non-jta-data-source` element is used in the server to specify where resource-local connections can be obtained in JNDI. It may also be used by providers that do optimized reading through non-JTA connections.

When configuring for outside the server, not only can we not rely upon JTA, as we described in the Transaction Type section, but we cannot rely upon JNDI at all. We therefore cannot portably rely upon either of the data source elements in Java SE configurations.

When using resource-local transactions outside the server, the provider obtains database connections directly vended out by the JDBC driver. In order for it to get these connections it must obtain the driver-specific information, which typically includes the name of the driver

class, the URL that the driver uses to connect to the database, and the user and password authentication that the driver also passes to the database. This metadata may be specified in whichever way the provider prefers it to be specified, but the most common method is to use the vendor-specific properties section. Listing 11-15 shows an example of using the TopLink Essentials properties that we use in the code example to connect to the Derby database through the Derby driver.

Listing 11-15. *Specifiying Resource-Level JDBC Properties*

```
<persistence-unit name="EmployeeService">
    ...
    <properties>
        <property name="toplink.jdbc.driver"
                  value="org.apache.derby.jdbc.ClientDriver"/>
        <property name="toplink.jdbc.url"
                  value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
        <property name="toplink.jdbc.user" value="APP"/>
        <property name="toplink.jdbc.password" value="APP"/>
    </properties>
</persistence-unit>
```

Providers

Many servers are going to have a default or native provider that they will use when the provider is not specified. It will automatically call into that provider to create an `EntityManagerFactory` at deployment time.

When not in a server, the factory is created programmatically using the `Persistence` class. When the `createEntityManagerFactory()` method is invoked, the `Persistence` class will begin a built-in pluggability protocol that goes out and finds the provider that is specified in the persistence unit configuration. If none was specified, then the first one that it finds will be used. Providers export themselves through a service that exists in the provider JAR that must be on the classpath. The net result is that the provider element is not required.

In the majority of cases when only one provider will be on the classpath, then the provider will be detected and used by the `Persistence` class to create an `EntityManagerFactory` for a given persistence unit. If you are ever in a situation where you have two providers on the classpath and you want a particular one to be used, then you should specify the provider class in the `provider` element. To prevent runtime and deployment errors, the `provider` element should be used if the application has a code dependency on a specific provider.

Listing the Entities

One of the benefits of deploying inside the server is that it is a highly controlled and structured environment. Because of this, the server can support the deployment process in ways that cannot be achieved by a simple Java SE runtime. The server already has to process all of the deployment units in an application and can do things like detecting all of the managed persistence classes in an EJB JAR or a persistence archive. This kind of class detection makes persistence archives a very convenient way to bundle a persistence unit.

The problem with this kind of detection outside the server is that the Java SE environment permits all kinds of different class resources to be added to the classpath, including network URLs or any other kind of resource that is acceptable to a classloader. This makes it difficult for the Java Persistence API to require providers to support doing automatic detection of the managed classes inside a persistence archive. The official position of the API is that for an application to be portable across all vendors it must explicitly list all of the managed classes in the persistence unit using `class` elements. When a persistence unit is large and includes a large number of classes, this task can become rather onerous.

In practice, however, most of the time the classes are sitting in a regular persistence archive JAR on the filesystem, and the provider runtime really can do the detection that the server would do in Java EE. For this reason all the major providers actually do support detecting the classes outside the server. This is really kind of an essential usability issue since the maintenance of a class list would be so cumbersome as to be a productivity bottleneck unless you had a tool manage the list for you.

A corollary to the official portability guideline to use `class` elements to enumerate the list of managed classes is that the `exclude-unlisted-classes` element is not guaranteed to have any impact in Java SE persistence units. Some providers may allow this element to be used outside the server, but it is not really very useful in the SE environment anyway given the flexibility of the classpath and packaging allowances in that environment.

Specifying Properties at Runtime

One of the benefits of running outside the server is the ability to specify provider properties at runtime. This is available because of the overloaded `createEntityManagerFactory()` method that accepts a `Map` of properties in addition to the name of the persistence unit. The properties passed to this method are combined with those already specified, normally in the `persistence.xml` file. They may be additional properties or they may override the value of a property that was already specified. This may not seem very useful to some applications, since putting runtime configuration information in code is not normally viewed as being better than isolating it in an XML file. However, one can imagine this being a convenient way to set properties obtained from a program input, such as the command line, as an even more dynamic configuration mechanism. In Listing 11-16 is an example of taking the user and password properties from the command line and passing them to the provider when creating the `EntityManagerFactory`.

Listing 11-16. *Using Command-Line Persistence Properties*

```
public class EmployeeService {
    public static void main(String[] args) {
        Map props = new HashMap();
        props.put("toplink.jdbc.user", args[0]);
        props.put("toplink.jdbc.password", args[1]);
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("EmployeeService", props);
        // ...
        emf.close();
    }
}
```


System Classpath

In some ways configuring a persistence unit in a Java SE application is actually easier than configuring in the server because the classpath is simply the system classpath. Adding classes or jars on the system classpath is a trivial exercise. In the server we may have to manipulate the manifest classpath or add some vendor-specific application classpath configuration.

Summary

It is a simple exercise to package and deploy persistence applications using the Java Persistence API. In most cases it is just a matter of adding a very short `persistence.xml` file to the JAR containing the entity classes.

In this chapter we described how to configure the persistence unit in the Java EE server environment using the `persistence.xml` file and how in some cases the name may be the only setting required. We then explained when to apply and how to specify the transaction type, the persistence provider, and the data source. We showed how to use and specify the default `orm.xml` mapping file and then went on to use additional mapping files within the same persistence unit. We also discussed the various ways that classes may be included in the persistence unit and how to customize the persistence unit using vendor-specific properties.

We looked at the ways that persistence units may be packaged and deployed to a Java EE application as part of an EJB archive, a web archive, or a persistence archive that is accessible to all of the components in the application. We examined how persistence units may exist within different scopes of a deployed Java EE application and what the name-scoping rules were.

Finally we compared the configuration and deployment practices to deploying an application to a Java SE environment.

In the next chapter we will consider the accepted and best practices for testing applications that use persistence.

CHAPTER 12

Testing

One of the major selling points of the Java Persistence API and EJB 3.0 has been the drive towards better testability. The use of plain Java classes where possible as well as the ability to use persistence outside of the application server has made enterprise applications much easier to test. This chapter will cover unit testing and integration testing with entities, with a mix of modern and traditional test techniques.

Testing Enterprise Applications

If we accept that testing is a good thing, then how exactly should we go about it? Almost all enterprise applications are hosted in some kind of server environment, whether it is a servlet container like Apache Tomcat or a full Java EE application server. Once deployed to such an environment, the developer is effectively cut off from the application. At this point it can only be tested using the public interface of the application, such as a browser using HTTP, RMI, or a messaging interface.

This presents an issue for developers, because we want to be able to focus on the components of an application in isolation. An elaborate sequence of operations through a web site may resolve to only a single method of a session bean that implements a particular business service. For example, to view an `Employee` record, a test client might have to log in using a user name and password, traverse through several menu options, execute a search, and then finally access the record. Afterwards the HTML output of the report must be verified to ensure that the operation completed as expected. In some applications this may be short-circuited by directly accessing the URL that retrieves a particular record. But with more and more information cached in HTTP session state, URLs are beginning to look like random sequences of letters and numbers. Direct access to a particular feature of an application may not be an easy process.

Java SE clients (so-called "fat" clients) that communicate with databases and other resources suffer from the same problem despite the ability to execute the program without the need for an application server. The user interface of a Java SE client may well be a Swing application requiring special tools to drive the user interface in order to do any kind of test automation. The application itself is still just a black box without any obvious way to get inside.

Numerous attempts have been made in recent years to expose the internals of an application to testing while deployed on a server. The Cactus[1] framework, for example, allows developers to write tests using JUnit, which are then deployed to the server along with the

1. Visit `http://jakarta.apache.org/cactus/` for more information.

application and executed via a web interface provided by Cactus. Other frameworks have adopted a similar approach using RMI instead of a web interface to remotely control the tests.

Though effective, the downside to these approaches is that the application server still has to be up and running before we can attempt any kind of testing. For developers who use the test-driven development (TDD) methodology, where tests are written before code and the full unit test suite is executed after every development iteration (which can be as small as a change to a single method), any kind of interaction with the application server is a problem. Even for developers who practice a more traditional testing methodology, frequent test execution is hampered by the need to keep the application server running, with a packaging and deployment step before every test run.

Clearly, for developers who wish to break a Java EE application into its component parts and test those components in isolation, there is a need for tools that will let us directly execute portions of the application outside of the server environment in which it is normally hosted.

Terminology

Not everyone agrees about exactly what constitutes a unit test or an integration test. In fact, it is quite likely that any survey of a group of developers will yield a wide variety of results, some similar in nature with others venturing into completely different areas of testing. Therefore we feel it is important to define our terminology for testing so that you can translate it into whatever terms you are comfortable with.

We see tests falling into the following four categories:

1. **Unit Tests.** Unit tests are written by developers and focus on isolated components of an application. Depending on your approach, this may be a single class or a collection of classes. The only key defining element is that the unit test is not coupled to any server resources (these are typically stubbed out as part of the test process) and execute very quickly. It must be possible to execute an entire suite of unit tests from within an IDE and get the results in a matter of seconds. Unit test execution can be automated and is often configured to happen automatically as part of every merge to a configuration management system.

2. **Integration Tests.** Integration tests are also written by developers and focus on use cases within an application. They are still decoupled from the application server, but the difference between a unit test and an integration test is that the integration test makes full use of external resources such as a database. In effect, an integration test takes a component from an application and runs in isolation as if it were still inside the application server. Running the test locally makes it much faster than a test hosted in an application server but still slower than a unit test. Integration tests are also automated and often run at least daily to ensure that there are no regressions introduced by developers.

3. **Functional Tests.** Functional tests are the black box tests written and automated by quality engineers instead of developers. Quality engineers look at the functional specification for a product and its user interface and seek to automate tests that can verify product behavior without the understanding of how the application is implemented. Functional tests are a critical part of the application development process, but it is unrealistic to execute these tests as part of the day-to-day work done by a developer. Automated execution of these tests often happens at a different schedule relative to the regular development process.

4. **Acceptance Tests.** Acceptance tests are customer-driven. These tests, usually conducted manually, are carried out directly by customers or representatives who play the role of the customer. The goal of an acceptance test is to verify that the requirements set out by the customer are reflected in the user interface and behavior of the application.

In this chapter we will focus only on unit tests and integration tests. These tests are written by developers for the benefit of developers and constitute what is called white box testing. These tests are written with the full understanding of how the application is implemented and what it will take to test not only the successful path through an application but also how to trigger failure scenarios.

Testing Outside the Server

The common element between unit tests and integration tests is that they are executed without the need for an application server. Unfortunately for Java EE developers, this has traditionally been a very difficult task to accomplish. Applications developed before the Java EE 5 release are tightly coupled to the application server, often making it difficult and counterproductive to attempt replicating the required container services in a stand-alone environment.

To put this in perspective, let's look at Enterprise JavaBeans as they existed in EJB 2.1. On paper, testing a session bean class should be little more than a case of instantiating the bean class and invoking the business method. For trivial business methods, this is indeed the case, but things start to go downhill quickly once dependencies get involved. For example, let's consider a business method that needs to invoke another business method from a different session bean.

Dependency lookup was the only option in EJB 2.1, so if the business method has to access JNDI to obtain a reference to the other session bean, then either JNDI must be worked around or the bean class must be refactored so that the lookup code can be replaced with a test-specific version. If the code uses the Service Locator[2] pattern, then we have a bigger problem because a singleton static method is used to obtain the bean reference. The only solution for testing beans that use Service Locators outside the container is to refactor the bean classes so that the locator logic can be overridden in a test case.

2. Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition.* Upper Saddle River, N.J.: Prentice Hall PTR, 2003, p. 315.

Next we have the problem of the dependent bean itself. The bean class does not implement the business interface, so it cannot simply be instantiated and made available to the bean we are trying to test. Instead, it will have to be subclassed to implement the business interface, and stubs for a number of low-level EJB methods will have to be provided since the business interface in EJB 2.1 actually extends an interface that is implemented internally by the application server.

Even if we get that to work, what happens if we encounter a container-managed entity bean? Not only do we have the same issues with respect to the interfaces involved, but the bean class is abstract, with all of the persistent state properties unimplemented. We could implement these, but our test framework would rapidly start to outgrow the application code. We can't even just run them against the database like we can with JDBC code because so much of the entity bean logic, relationship maintenance and other persistence operations are only available inside of an EJB container.

The dirty secret of many applications written using older versions of Java EE is that there is little to no developer testing at all. Developers write, package, and deploy applications; test them manually through the user interface; and then hope that the quality assurance group can write a functional test that verifies each feature. It's just too much work to test individual components outside of the application server.

This is where EJB 3.0 and the Java Persistence API come in. Starting with EJB 3.0, a session bean class is a simple Java class that implements a regular Java interface. No special EJB interfaces need to be extended or implemented. To unit test a session bean, we can just implement it and execute it. If the bean depends on another bean, we can instantiate that bean and manually inject it into the bean being tested. The EJB 3.0 release was designed to encourage testing by breaking the hard dependencies between application code and the application server.

Likewise entities are a world apart from container-managed entity beans. If your session bean uses an entity, you can just instantiate it and use it like any other class. If you are testing code that uses the entity manager and want to verify that it is interacting with the database the way you expect it to, just bootstrap the entity manager in Java SE and make full use of the entity manager outside of the application server.

Over the course of this chapter, we will demonstrate how to take a session bean and Java Persistence API code from a Java EE application and run it outside the container, using unit testing and integration testing approaches. If you have worked with older versions of EJB and experienced the pain of developer testing, prepare yourself for a completely different look at testing enterprise applications.

Test Frameworks

In recent years, the JUnit test framework has become the de facto standard for testing Java applications. JUnit is a simple unit testing framework that allows tests to be written as Java classes. These Java classes are then bundled together and run in suites using a test runner that is itself a simple Java class. Out of this simple design, a whole community has emerged to provide extensions to JUnit and integrate it into all major development environments.

Despite its name, unit testing is now only one of the many things that JUnit can be used for. It has been extended to support testing of web sites, automatic stubbing of interfaces for testing, concurrency testing, and performance testing. Many quality assurance groups now use JUnit as part of the automation mechanism to run whole suites of end-to-end functional tests.

For our purposes we will look at JUnit in the context of its unit testing roots, and also at strategies that allow it to be used as an effective integration test framework. Collectively we look at these two approaches simply as developer tests, because they are written by developers to assist with the overall quality and development of an application.

In addition to the test framework itself, there are other libraries that can assist with the testing of Java EE components. There have been announcements for EJB 3.0 containers that can be used outside of the application server, providing developers with the full services of dependency injection even for isolated session bean testing. Frameworks like Spring also offer sophisticated dependency injection support even in the Java SE environment, allowing dependent classes to be woven together. Even though it may not directly support EJB 3.0 annotations, the fact that session beans are simple Java classes makes them usable with any lightweight container framework. As always, before writing these kinds of frameworks for testing, check to see that the problem hasn't already been solved. If nothing else, the Java community has shown a remarkable willingness to share solutions to problems, both in the open source community and even from the commercial vendors.

We will assume that you are familiar with JUnit at this point. Introductory articles and tutorials can be found on the JUnit website at `http://www.junit.org`. There are also a large number of books and other online resources that cover testing with JUnit in extensive detail.

Unit Testing

It might seem counterintuitive at first, but one of the most interesting things about entities is that they can participate in tests without requiring a running application server or live database. For years enterprise developers have been frustrated with container-managed entity beans because they were effectively untestable without a live application server. The component and home interfaces could conceivably be used in unit tests, but only if the developer provided implementations of those interfaces, duplicating effort already invested in writing the real bean classes and potentially introducing new bugs in the process. Because entities are plain Java classes, they can be used directly in tests without any additional effort required.

In the following sections we will look both at testing entity classes directly and using entities as part of tests for Java EE components. We will also discuss how to leverage dependency injection in unit tests and how to deal with the presence of Java Persistence API interfaces.

Testing Entities

Entities themselves are unlikely to be extensively tested in isolation. Most methods on entities are simple getters or setters that relate to the persistent state of the entity or to its relationships. Business methods may also appear on entities but are less common. In many applications, entities are little more than basic JavaBeans.

As a rule, property methods do not generally require explicit tests. Verifying that a setter assigns a value to a field and the corresponding getter retrieves the same value is not testing the application so much as the compiler. Unless there is a side effect in one or both of the methods, getters and setters are too simple to break and therefore too simple to warrant testing.

Key things to look for in determining whether or not an entity warrants individual testing are side effects from a getter or setter method (such as data transformation or validation rules) and the presence of business methods. The entity shown in Listing 12-1 contains non-trivial logic that warrants specific testing.

Listing 12-1. *An Entity That Validates and Transforms Data*

```
@Entity
public class Department {
    @Id private String id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;

    public String getId() { return id; }
    public void setId(String id) {
        if (id.length() != 4) {
            throw new IllegalArgumentException("Department identifiers must ➥
be four characters in length");
        }
        this.id = id.toUpperCase();
    }

    // ...
}
```

The `setId()` method both validates the format of the department identifier and transforms the string to uppercase. This type of logic and the fact that setting the identifier can actually cause an exception to be thrown suggests that tests would be worthwhile. Testing this behavior is simply a matter of instantiating the entity and invoking the setter with different values. Listing 12-2 shows one possible set of tests.

Listing 12-2. *Testing a Setter Method for Side Effects*

```
public class DepartmentTest extends TestCase {

    public void testValidDepartmentId() throws Exception {
        Department dept = new Department();
        dept.setId("NA65");
        assertEquals("NA65", dept.getId());
    }

    public void testDepartmentIdInvalidLength() throws Exception {
        Department dept = new Department();
        try {
            dept.setId("NA6");
            fail("Department identifiers must be four characters");
        } catch (IllegalArgumentException e) {
        }
    }
```



```
    public void testDepartmentIdCase() throws Exception {
        Department dept = new Department();
        dept.setId("na65");
        assertEquals("NA65", dept.getId());
    }
}
```

Testing Entities in Components

The most likely test scenario for entities is not the entities themselves but the application code that uses the entities as part of its business logic. For many applications this means testing session beans and other Java EE components. Just as with the entity test shown in Listing 12-1, these types of tests are made easy in the sense that the entity class can simply be instantiated, populated with entity data and set into the bean class for testing. When used as a domain object in application code, an entity is no different than any other Java class. You can effectively pretend that it's not an entity at all.

Of course, there is more to unit testing a session bean than simply instantiating entities to be used with a business method. We also need to concern ourselves with the dependencies that the session bean has in order to implement its business logic. These dependencies are usually manifested as fields on the bean class that are populated using a form of dependency injection or dependency lookup.

When writing unit tests, our goal is to introduce the minimum set of dependencies required to implement a particular test. If we are testing a business method that needs to invoke a method on the `EJBContext` interface, then we should worry only about providing a stubbed version of the interface. If the bean uses a data source but is not relevant to our testing, then ideally we would like to ignore it entirely.

Dependency injection is the key to effective unit testing. By removing the JNDI API from session bean code and eliminating the need for the Service Locator pattern, the bean class has few dependencies on the application server. We need only instantiate the bean instance and manually inject the required resources, the majority of which will either be other beans from the application or test-specific implementations of a standard interface.

As we discussed in Chapter 3, the setter injection form of dependency injection is the easiest to use in unit tests. Because the setter methods are almost always public, they can be invoked directly by the test case to assign a dependency to the bean class. Field injection is still easy to deal with so long as the field uses package scope since the convention for unit tests is to use the same package name as the class that is being tested.

When the dependency is another session bean, you must make a choice as to whether all of the dependencies of the required bean class must be met or whether a test-specific version of the business interface should be used instead. If the business method from the dependent business interface does not affect the outcome of the test, then it may not be worth the effort to establish the full dependency. As an example, consider the session bean shown in Listing 12-3. We have shown a single method for calculating years of service for an employee that retrieves an `Employee` instance using the `EmployeeService` session bean.

Listing 12-3. *Using the* `EmployeeService` *Bean in a Different Business Method*

```
@Stateless
public class VacationBean implements Vacation {
    public static final long MILLIS_PER_YEAR = 1000 * 60 * 60 * 24 * 365;
    @EJB EmployeeService empService;

    public int getYearsOfService(int empId) {
        Employee emp = empService.findEmployee(empId);
        long current = System.currentTimeMillis();
        long start = emp.getStartDate().getTime();
        return (int)((current - start) / MILLIS_PER_YEAR);
    }

    // ...
}
```

Since the only thing necessary to verify the `getYearsOfService()` method is a single `Employee` instance with a start date value, there is no need to use the real `EmployeeService` bean. An implementation of the `EmployeeService` interface that returns an entity instance preconfigured for the test is more than sufficient. In fact, the ability to specify a well-known return value from the `findEmployee()` method makes the overall test much easier to implement. Listing 12-4 demonstrates using a test-specific implementation of a session bean interface. Implementing an interface specifically for a test is called *mocking* the interface, and the instantiated instance is referred to as a *mock object.*

Listing 12-4. *Creating a Test-specific Version of a Business Interface*

```
public class VacationBeanTest extends TestCase {
    public void testYearsOfService() throws Exception {
        VacationBean bean = new VacationBean();
        bean.empService = new EmployeeService() {
            public Employee findEmployee(int id) {
                Employee emp = new Employee();
                emp.setStartDate(new Time(System.currentTimeMillis() -
                                          VacationBean.MILLIS_PER_YEAR * 5));
                return emp;
            }

            // ...
        };
        int yearsOfService = bean.getYearsOfService(0);
        assertEquals(5, yearsOfService);
    }

    // ...
}
```


The Entity Manager in Unit Tests

The `EntityManager` and `Query` interfaces present a challenge to developers writing unit tests. Code that interacts with the entity manager can vary from the simple (persisting an object) to the complex (issuing an JPQL query and obtaining the results). There are two basic approaches to dealing with the presence of standard interfaces:

- Introduce a subclass that replaces methods containing entity manager or query operations with test-specific versions that do not interact with the Java Persistence API.
- Provide custom implementations of standard interfaces that may be predictably used for testing.

Before covering these strategies in detail, consider the session bean implementation shown in Listing 12-5 that provides a simple authentication service. For such a simple class, it is surprisingly challenging to unit test. The entity manager operations are embedded directly within the `authenticate()` method, coupling the implementation to the Java Persistence API.

Listing 12-5. *Session Bean That Performs Basic Authentication*

```
@Stateless
public class UserServiceBean implements UserService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public User authenticate(String userId, String password) {
        User user = em.find(User.class, userId);
        if (user != null) {
            if (password.equals(user.getPassword())) {
                return user;
            }
        }
        return null;
    }
}
```

The first technique we will demonstrate to make this class testable is to introduce a subclass that eliminates entity manager calls. For the `UserServiceBean` example shown in Listing 12-5, entity manager access must first be isolated to a separate method before it can be tested. Listing 12-6 demonstrates such a refactoring.

Listing 12-6. *Isolating Entity Manager Operations for Testing*

```
@Stateless
public class UserServiceBean implements UserService {
    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public User authenticate(String userId, String password) {
        User user = findUser(userId);
        // ...
    }

    User findUser(String userId) {
        return em.find(User.class, userId);
    }
}
```

With this refactoring complete, the `authenticate()` method no longer has any direct dependency on the entity manager. The `UserServiceBean` class can now be subclassed for testing, replacing the `findUser()` method with a test-specific version that returns a well-known result. Listing 12-7 demonstrates a complete test case using this technique.

Listing 12-7. *Using a Subclass to Eliminate Entity Manager Dependencies*

```
public class UserServiceTest extends TestCase {
    private static final String USER_ID = "test_id";
    private static final String PASSWORD = "test_password";
    private static final String INVALID_USER_ID = "test_user";

    public void testAuthenticateValidUser() throws Exception {
        TestUserService service = new TestUserService();
        User user = service.authenticate(USER_ID, PASSWORD);
        assertNotNull(user);
        assertEquals(USER_ID, user.getName());
        assertEquals(PASSWORD, user.getPassword());
    }

    public void testAuthenticateInvalidUser() throws Exception {
        TestUserService service = new TestUserService();
        User user = service.authenticate(INVALID_USER_ID, PASSWORD);
        assertNull(user);
    }

    class TestUserService extends UserServiceBean {
        private User user;

        public TestUserService() {
            user = new User();
            user.setName(USER_ID);
            user.setPassword(PASSWORD);
        }

        User findUser(String userId) {
            if (userId.equals(user.getName())) {
                return user;
            }
            return null;
        }
    }
}
```


This test case has the advantage of leaving the original `authenticate()` method implementation intact, only overriding the `findUser()` method for the test. This works well for classes that have been refactored to isolate persistence operations, but these changes cannot always be made. The alternative is to mock the `EntityManager` interface. Listing 12-8 demonstrates this approach.

Listing 12-8. *Using a Mock Entity Manager in a Unit Test*

```
public class UserServiceTest2 extends TestCase {
    private static final String USER_ID = "test_id";
    private static final String PASSWORD = "test_password";
    private static final String INVALID_USER_ID = "test_user";

    public void testAuthenticateValidUser() throws Exception {
        UserServiceBean service = new UserServiceBean();
        service.em = new TestEntityManager(USER_ID, PASSWORD);
        User user = service.authenticate(USER_ID, PASSWORD);
        assertNotNull(user);
        assertEquals(USER_ID, user.getName());
        assertEquals(PASSWORD, user.getPassword());
    }

    public void testAuthenticateInvalidUser() throws Exception {
        UserServiceBean service = new UserServiceBean();
        service.em = new TestEntityManager(USER_ID, PASSWORD);
        User user = service.authenticate(INVALID_USER_ID, PASSWORD);
        assertNull(user);
    }
```

```
    class TestEntityManager extends MockEntityManager {
        private User user;

        public TestEntityManager(String user, String password) {
            this.user = new User();
            this.user.setName(user);
            this.user.setPassword(password);
        }

        public <T> T find(Class<T> entityClass, Object pk) {
            if (entityClass == User.class && ((String)pk).equals(user.getName())) {
                return (T) user;
            }
            return null;
        }
    }
}
```

The advantage of this approach over subclassing is that it leaves the original bean class unchanged while allowing it to be unit tested. The `MockEntityManager` class referenced in the test is a concrete implementation of the `EntityManager` interface with empty method definitions. All methods that return a value return null or an equivalent instead. By defining it separately, it can be reused for other test cases. Many unit test suites contain a small set of mocked interfaces that can be reused across multiple tests.

■ **Tip** Check out `http://www.mockobjects.com` for further information on mock object techniques and open source tools to assist with mock object creation.

Integration Testing

Integration testing, for our purposes, is an extension of unit testing that takes components of a Java EE application and executes them outside of an application server. Unlike unit testing, where we went to great lengths to avoid the entity manager, in integration testing we embrace it and leverage the fact that it can be used in Java SE.

The following sections explore using the Java Persistence API outside of an application server in order to test application logic with a live database but without starting the application server.

Using the Entity Manager

In Listing 12-5 we demonstrated a session bean that performed basic authentication against a `User` object retrieved from the database. To unit test this class, a number of techniques were presented to replace or mock the entity manager operation. The downside to this approach is that the test code required to work around external dependencies in the application code can quickly reach a point where it is difficult to maintain and is a potential source of bugs.

Instead of mocking the entity manager, a resource-local, application-managed entity manager may be used to perform tests against a live database. Listing 12-9 demonstrates a functional test version of the `UserServiceBean` test cases.

Listing 12-9. *Integration Test for* `UserServiceBean`

```
public class UserServiceTest3 extends TestCase {
    private static final String USER_ID = "test_id";
    private static final String PASSWORD = "test_password";
    private static final String INVALID_USER_ID = "test_user";

    private EntityManagerFactory emf;
    private EntityManager em;

    public void setUp() {
        emf = Persistence.createEntityManagerFactory("hr");
        em = emf.createEntityManager();
        createTestData();
    }

    public void tearDown() {
        if (em != null) {
            removeTestData();
            em.close();
        }
        if (emf != null) {
            emf.close();
        }
    }

    private void createTestData() {
        User user = new User();
        user.setName(USER_ID);
        user.setPassword(PASSWORD);
        em.getTransaction().begin();
        em.persist(user);
        em.getTransaction().commit();
    }

    private void removeTestData() {
        em.getTransaction().begin();
        User user = em.find(User.class, USER_ID);
        if (user != null) {
            em.remove(user);
        }
        em.getTransaction().commit();
    }
```

```
    public void testAuthenticateValidUser() throws Exception {
        UserServiceBean service = new UserServiceBean();
        service.em = em;
        User user = service.authenticate(USER_ID, PASSWORD);
        assertNotNull(user);
        assertEquals(USER_ID, user.getName());
        assertEquals(PASSWORD, user.getPassword());
    }

    public void testAuthenticateInvalidUser() throws Exception {
        UserServiceBean service = new UserServiceBean();
        service.em = em;
        User user = service.authenticate(INVALID_USER_ID, PASSWORD);
        assertNull(user);
    }
}
```

This test case uses the fixture methods `setUp()` and `tearDown()` to create `EntityManagerFactory` and `EntityManager` instances using the Java SE bootstrap API and then closes them when the test completes. The test case also uses these methods to seed the database with test data and remove it when the test completes. The `tearDown()` method is guaranteed to be called even if a test fails due to an exception. Like any Java Persistence API application in the Java SE environment, a `persistence.xml` file will need to be on the classpath in order for the `Persistence` class to bootstrap an entity manager factory. This example demonstrates the basic pattern for all integration tests that use an entity manager.

The advantage of this style of test versus a unit test is that no effort was required to mock up persistence interfaces. Emulating the entity manager and query engine in order to test code that interacts directly with these interfaces suffers from diminishing returns as more and more effort is put into preparing a test environment instead of writing tests. In the worst-case scenario, incorrect test results occur because of bugs in the test harness, not in the application code. Given the ease with which the Java Persistence API can be used outside the application server, this type of effort may be better spent establishing a simple database test environment and writing automated functional tests.

However, despite the opportunity that testing outside the application server presents, care must be taken to ensure that such testing truly adds value. Quite often, developers fall into the trap of writing tests that do little more than test vendor functionality as opposed to true application logic. An example of this mistake is seeding a database, executing a query and verifying that the desired results are returned. It sounds valid at first, but all that it tests is the developer's understanding of how to write a query. Unless there is a bug in the database or the persistence provider, the test will never fail. A more valid variation of this test would be to start the scenario further up the application stack, by executing a business method on a session façade that initiates a query and then validating that the resulting transfer objects are formed correctly for later presentation by a JSP page.

Test Setup and Teardown

Many tests involving persistence require some kind of test data in the database before the test can be executed. If the business operation itself does not create and verify the result of a persistence

operation, the database must already contain data that can be read and used by the test. Since tests should ideally be able to set and reset their own test data before and after each test, we must have a way to seed the database appropriately.

This sounds pretty straightforward; use JDBC to seed the database during `setUp()` and again during `tearDown()` to reset it. But there is a danger here. Most persistence providers employ some kind of data or object caching. Any time data changes in the database without the persistence provider knowing about it, its cache will get out of sync with the database. In the worst-case scenario, this could cause entity manager operations to return entities that have since been removed or that have stale data.

It's worth reiterating that this is not a problem with the persistence provider. Caching is a good thing and the reason that Java Persistence API solutions often significantly outperform direct JDBC access in read-mostly applications. The Reference Implementation, for example, uses a sophisticated shared-cache mechanism that is scoped to the entire persistence unit. When operations are completed in a particular persistence context, the results are merged back into the shared cache so that they can be used by other persistence contexts. This happens whether the entity manager and persistence context are created in Java SE or Java EE. Therefore you can't assume that closing an entity manager clears test data from the cache.

There are several approaches we can use to keep the cache consistent with our test database. The first, and easiest, is to create and remove test data using the entity manager. Any entity persisted or removed using the entity manager will always be kept consistent with the cache. For small data sets, this is very easy to accomplish. This is the approach we used in Listing 12-9.

For larger data sets, however, it can be cumbersome to create and manage test data using entities. JUnit extensions such as DbUnit[3] allow seed data to be defined in XML files and then loaded in bulk to the database before each test begins. So given that the persistence provider won't know about this data, how can we still make use of it? The first strategy is to establish a set of test data that is read-only. So long as the data is never changed, it doesn't matter if the entity exists in the provider cache or not. The second strategy is to either use special data sets for operations that need to modify test data without creating it, or to ensure that these changes are never permanently committed. If the transaction to update the database is rolled back, then the database and cache state will both remain consistent.

The last thing to consider is explicit cache invalidation. This is vendor-specific, but every vendor that supports a shared cache will also provide some mechanism to clear the cache. TopLink Essentials provides a number of options for controlling the cache. The following method demonstrates how to invalidate the entire shared cache in TopLink Essentials given any `EntityManager` instance:

```
public static void clearCache(EntityManager em) {
    em.clear();
    oracle.toplink.essentials.ejb.cmp3.EntityManager tlem =
        (oracle.toplink.essentials.ejb.cmp3.EntityManager) em;
    tlem.getActiveSession()
        .getIdentityMapAccessor()
        .initializeAllIdentityMaps();
}
```

3. Visit `http://dbunit.sourceforge.net/` for more information.

Note that we cleared the current persistence context as well as invalidated the cache. As we have discussed before, the persistence context is a localized set of transactional changes. It uses data from the shared cache but is actually a separate and distinct data structure.

Switching Configurations for Testing

One of the great advantages of the Java Persistence API is that metadata specified in annotation form may be overridden or replaced by metadata specified in XML form. This affords us a unique opportunity to develop an application targeting the production database platform and then provide an alternate set of mappings (even query definitions) targeted to a test environment. In the context of testing, the Java SE bootstrap mechanism will use the `persistence.xml` file located in the `META-INF` directory on the classpath. So long as the persistence unit definition inside this file has the same name as the one the application was written to, the test version can retarget it as necessary to suit the needs of the integration test.

There are two main uses for this approach. The first is to specify properties in the `persistence.xml` file that are specific to testing. For many developers, this will mean providing JDBC connection information to a local database so that tests do not collide with other developers on a shared database.

The second major use of a custom `persistence.xml` file is to customize the database mappings for deployment on a completely different database platform. For example, if Oracle is your production database and you don't wish to run the full database[4] on your local machine, you can adjust the mapping information to target an embedded database such as Apache Derby.

As an example of when this would be necessary, consider an application that uses the native sequencing of the Oracle database. Derby does not have an equivalent, so table generators must be used instead. First, let's consider an example entity that uses a native sequence generator:

```
@Entity
public class Phone {
    @SequenceGenerator(name="Phone_Gen", sequenceName="PHONE_SEQ")
    @Id @GeneratedValue(generator="Phone_Gen")
    private int id;
    // ...
}
```

The first step to getting this entity working on Derby is to create an XML mapping file that overrides the definition of the "Phone_Gen" generator to use a table generator. The following fragment of a mapping file demonstrates how to replace the sequence generator with a table generator:

```
<entity-mappings>
    ...
    <table-generator name="Phone_Gen" table="ID_GEN" pk-column-value="PhoneId"/>
    ...
</entity-mappings>
```

4. At the risk of sounding somewhat biased, might we humbly suggest Oracle XE. It represents the power of the Oracle database conveniently sized to an individual machine at no cost. All of the examples in this book (including the advanced SQL query examples in Chapter 9) were developed on Oracle XE.

This is the same technique we applied in Chapter 10 when we discussed overriding a sequence generator.

Finally we need to create a new `persistence.xml` file that references this mapping file. If the overrides were placed in a mapping file called `derby-overrides.xml`, then the following persistence unit configuration would apply the mapping overrides:

```
<persistence>
    <persistence-unit name="hr">
        ...
        <mapping-file>derby-overrides.xml</mapping-file>
        ...
    </persistence-unit>
</persistence>
```

Unlike the mapping file, which sparsely defines overrides, all of the information that was present in the production `persistence.xml` file must be copied into the test-specific version. The only exception to this is the JDBC connection properties, which will now have to be customized for the embedded Derby instance.

Minimizing Database Connections

Integration tests execute slower than unit tests due to the nature of the database interaction, but what may not be obvious from the test case shown in Listing 12-9 is that two separate connections are made to the database, one each for the `testAuthenticateValidUser()` and `testAuthenticateInvalidUser()` tests. JUnit actually instantiates a new instance of the test case class each time it runs a test method, running `setUp()` and `tearDown()` each time as well. The reason for this behavior is to minimize the chance of data stored in fields from one test case interfering with the execution of another.

While this works well for unit tests, it may lead to unacceptable performance for integration tests. To work around this limitation, an extension to JUnit called `TestSetup` may be used to create a fixture that runs `setUp()` and `tearDown()` only once for an entire test suite. Listing 12-10 demonstrates a test suite that uses this feature.

Listing 12-10. *One-time Database Setup for Integration Tests*

```
public class DatabaseTest {
    public static EntityManagerFactory emf;
    public static EntityManager em;

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(UserServiceTest3.class);

        TestSetup wrapper = new TestSetup(suite) {
```

```
            protected void setUp() throws Exception {
                emf = Persistence.createEntityManagerFactory("hr");
                em = emf.createEntityManager();
            }

            protected void tearDown() throws Exception {
                if (em != null) {
                    em.close();
                }
                if (emf != null) {
                    emf.close();
                }
            }
        };

        return wrapper;
    }
}
```

Using this test suite as a starting point, all child test cases or test suites that execute in the context of the `TestSetup` wrapper have access to the correctly populated `EntityManager` and `EntityManagerFactory` static fields on the `DatabaseTest` class. The `setUp()` method of each test case now only needs to reference this class to obtain the objects instead of creating them each time. The following example demonstrates the change required for the `UnitServiceTest3` test case:

```
public void setUp() {
    emf = DatabaseTest.emf;
    em = DatabaseTest.em;
    createTestData();
}
```

This is a useful technique to minimize the cost of acquiring expensive resources, but care must be taken to ensure that side effects from one test do not accidentally interfere with the execution of other tests. Because all tests share the same entity manager instance, data may be cached or settings may be changed that have an unexpected impact later on. Just as it is necessary to keep the database tables clean between tests, any changes to the entity manager itself (including flushing the persistence context) must be reverted when the test ends, regardless of whether the outcome is a success or a failure.

Components and Persistence

More often than not, session beans in an integration test are no different than session beans in a unit test. You instantiate the bean, supply any necessary dependencies, and execute the test. Where we start to diverge is when we start to take into account issues such as transaction management and multiple session bean instances collaborating together to implement a single use case. In the following sections we will discuss techniques to handle more complex session bean scenarios when testing outside of the container.

Transaction Management

Transactions lie at the heart of every enterprise application. We made this statement back in Chapter 3 and tried to drive this point home in Chapter 5, demonstrating all of the different ways in which entity managers and persistence contexts can intersect with different transaction models. It might come as a surprise then to learn that when it comes to writing integration tests, we can often sidestep the stringent transactional requirements of the application to easily develop tests outside the container. The following sections will delve into when transactions are really required and how to translate the container-managed and bean-managed transaction models of the Java EE server into your test environment.

When to Use Transactions

Except for resource-local application-managed entity managers, which are rarely used in the Java EE environment, transaction management is the purview of session beans and other components that use the Java Persistence API. We will focus specifically on session beans, but the topics we cover apply equally to transactional persistence operations hosted by message-driven beans or servlets.

The transaction demarcation for a session bean method needs to be considered carefully when writing tests. Despite the default assumption that transactions are used everywhere in the application server, only a select number of methods actually require transaction management for the purpose of testing. Since we are focused on testing persistence, the situation we are concerned with is when the entity manager is being used to persist, merge, or remove entity instances. We also need to determine if these entities actually need to be persisted to the database.

In a test environment, we are using resource-local application-managed entity managers. Recall from Chapter 5 that an application-managed entity manager can perform all of its operations without an active transaction. In effect, invoking `persist()` queues up the entity to be persisted the next time a transaction starts and is committed. Furthermore, we also know that once an entity is managed, it can typically be located using the `find()` operation without the need to go to the database. Given these facts, we generally need a transacted entity manager only if the business method creates or modifies entities and executes a query that should include the results.

Although not required to satisfy business logic, a transaction may also be required if you wish the results of the operation to be persisted so that they can be analyzed using something other than the active entity manager. For example, the results of the operation may be read from the database using JDBC and compared to a known value using a test tool.

Overall, the main thing we want to stress here before we look into how to implement transactions for session bean tests is that more often than not, you don't really need them at all. Look at the sequence of operations you are testing and consider whether or not the outcome will be impacted one way or the other, first if the data must be written to the database and later if it truly must be committed as part of the test. Given the complexity that manual transaction management can sometimes require, use them only when they are necessary.

Container-Managed Transactions

One of the most important benefits of container-managed transactions is that they are configured for session bean methods entirely using metadata. There is no programming interface invoked by the session bean to control the transaction other than the `setRollbackOnly()` method on the `EJBContext` interface, and even this occurs only in certain circumstances.

Therefore, once we make a determination that a particular bean method requires a transaction to be active, we need only start a transaction at the start of the test and commit or roll back the results when the test ends.

Listing 12-11 shows a bean method that will require an open transaction during a test. The `assignEmployeeToDepartment()` method assigns an employee to a given department and then returns the list of employees currently assigned to the department by executing a query. Because the data modification and query occur in the same transaction, our test case will also require a transaction.

Listing 12-11. *Business Method Requiring a Transaction*

```
@Stateless
public class DepartmentServiceBean implements DepartmentService {
    private static final String QUERY =
        "SELECT e " +
        "FROM Employee e " +
        "WHERE e.department = ?1 ORDER BY e.name";

    @PersistenceContext(unitName="EmployeeService")
    EntityManager em;

    public List assignEmployeeToDepartment(int deptId, int empId) {
        Department dept = em.find(Department.class, deptId);
        Employee emp = em.find(Employee.class, empId);
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
        return em.createQuery(QUERY)
                 .setParameter(1, dept)
                 .getResultList();
    }

    // ...
}
```

Because we are using a resource-local entity manager, we will be simulating container-managed transactions with `EntityTransaction` transactions managed by the test case. Listing 12-12 shows the test case for the `assignEmployeeToDepartment()` method. We have followed the same template as in Listing 12-9, so the `setUp()` and `tearDown()` methods are not shown. Before the session bean method is invoked, we create a new transaction. When the test is complete, we roll back the changes since it isn't necessary to persist them in the database.

Listing 12-12. *Testing a Business Method That Requires a Transaction*

```
public class DepartmentServiceBeanTest extends TestCase {
    // ...

    private void createTestData() {
        Employee emp = new Employee(500, "Scott");
        em.persist(emp);
        emp = new Employee(600, "John");
        em.persist(emp);
        Department dept = new Department(700, "TEST");
        dept.getEmployees().add(emp);
        emp.setDepartment(dept);
        em.persist(dept);
    }

    public void testAssignEmployeeToDepartment() throws Exception {
        DepartmentServiceBean bean = new DepartmentServiceBean();
        bean.em = em;
        em.getTransaction().begin();
        List result = bean.assignEmployeeToDepartment(700, 500);
        em.getTransaction().rollback();
        assertEquals(2, result.size());
        assertEquals("John", ((Employee)result.get(0)).getName());
        assertEquals("Scott", ((Employee)result.get(1)).getName());
    }

    // ...
}
```

Bean-Managed Transactions

For a session bean that uses bean-managed transactions, the key issue we need to contend with is the `UserTransaction` interface. It may or may not be present in any given bean method and may be used for a number of purposes, from checking the transaction status to marking the current transaction for rollback, to committing and rolling back transactions. Fortunately, almost all of the `UserTransaction` methods have a direct correlation to one of the `EntityTransaction` methods. Since our test strategy involves a single entity manager instance for a test, we need to adapt its `EntityTransaction` implementation to the `UserTransaction` interface.

Listing 12-13 shows an implementation of the `UserTransaction` interface that delegates to the `EntityTransaction` interface of an `EntityManager` instance. Exception handling has been added to convert the unchecked exceptions thrown by `EntityTransaction` operations into the checked exceptions that clients of the `UserTransaction` interface will be expecting.

Listing 12-13. *Emulating* `UserTransaction` *Using* `EntityTransaction`

```
public class EntityUserTransaction implements UserTransaction {
    private EntityManager em;

    public EntityUserTransaction(EntityManager em) {
        this.em = em;
    }

    public void begin() throws NotSupportedException {
        if (em.getTransaction().isActive()) {
            throw new NotSupportedException();
        }
        em.getTransaction().begin();
    }

    public void commit() throws RollbackException {
        try {
            em.getTransaction().commit();
        } catch (javax.persistence.RollbackException e) {
            throw new RollbackException(e.getMessage());
        }
    }

    public void rollback() throws SystemException {.
        try {
            em.getTransaction().rollback();
        } catch (PersistenceException e) {
            throw new SystemException(e.getMessage());
        }
    }

    public void setRollbackOnly() {
        em.getTransaction().setRollbackOnly();
    }

    public int getStatus() {
        if (em.getTransaction().isActive()) {
            return Status.STATUS_ACTIVE;
        } else {
            return Status.STATUS_NO_TRANSACTION;
        }
    }

    public void setTransactionTimeout(int timeout) {
        throw new UnsupportedOperationException();
    }
}
```


Note that we have implemented `setTransactionTimeout()` to throw an exception, but this does not necessarily have to be the case. If the transaction timeout is set simply to prevent processes from taking too long to complete, it might be safe to ignore the setting in an integration test.

To demonstrate this wrapper, first consider Listing 12-14, which demonstrates a variation of the example from Listing 12-11 that uses bean-managed transactions instead of container-managed transactions.

Listing 12-14. *Using Bean-Managed Transactions*

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class DepartmentServiceBean implements DepartmentService {
    // ...
    @Resource UserTransaction tx;

    public List assignEmployeeToDepartment(int deptId, int empId) {
        try {
            tx.begin();
            Department dept = em.find(Department.class, deptId);
            Employee emp = em.find(Employee.class, empId);
            dept.getEmployees().add(emp);
            emp.setDepartment(dept);
            tx.commit();
            return em.createQuery(QUERY)
                     .setParameter(1, dept)
                     .getResultList();
        } catch (Exception e) {
            // handle transaction exceptions
            // ...
        }
    }

    // ...
}
```

Using the `UserTransaction` wrapper is simply a matter of injecting it into a session bean that has declared a dependency on `UserTransaction`. Since the wrapper holds onto an entity manager instance, it can begin and end `EntityTransaction` transactions as required from within the application code being tested. Listing 12-15 shows the revised test case from Listing 12-12 using this wrapper to emulate bean-managed transactions.

Listing 12-15. *Executing a Test with Emulated Bean-Managed Transactions*

```
public class DepartmentServiceBeanTest extends TestCase {
    // ...

    public void testAssignEmployeeToDepartment() throws Exception {
        DepartmentServiceBean2 bean = new DepartmentServiceBean2();
        bean.em = em;
        bean.tx = new EntityUserTransaction(em);
        List result = bean.assignEmployeeToDepartment(700, 500);
        assertEquals(2, result.size());
        assertEquals("John", ((Employee)result.get(0)).getName());
        assertEquals("Scott", ((Employee)result.get(1)).getName());
    }

    // ...
}
```

Note that just because the `UserTransaction` interface is used doesn't mean it's actually necessary for any particular test. If the transaction state doesn't affect the outcome of the test, consider using an implementation of the `UserTransaction` interface that doesn't actually do anything. For example, the implementation of `UserTransaction` shown in Listing 12-16 is fine for any case where transaction demarcation is declared but unnecessary.

Listing 12-16. *A Stubbed* `UserTransaction`

```
public class NullUserTransaction implements UserTransaction {
    public void begin() {}
    public void commit() {}
    public void rollback() {}
    public void setRollbackOnly() {}
    public int getStatus() {
        return Status.STATUS_NO_TRANSACTION;
    }
    public void setTransactionTimeout(int timeout) {}
}
```

The test case shown in Listing 12-12 could also have tested the bean from Listing 12-14 if the empty `UserTransaction` wrapper from Listing 12-16 was also injected into the bean instance. This would disable the bean-managed transactions of the actual business method, allowing the transactions of the test case to be used instead.

Container-Managed Entity Managers

The default entity manager type for a session bean is container-managed and transaction-scoped. Extended entity managers are an option only for stateful session beans. In either case, the goal of testing outside the container is to map the application-managed entity manager used by the test to one of these entity manager types.

The good news for testing code that uses the extended entity manager is that the application-managed entity manager offers nearly the exact same feature set. It can usually be injected into a stateful session bean instance in place of an extended entity manager, and the business logic should function without change in most cases.

Likewise, most of the time the transaction-scoped entity manager works just fine when an application-managed entity manager is used in its place. The only issue we need to deal with in the case of transaction-scoped entity managers is detachment. When a transaction ends, any managed entities become detached. In terms of a test, that just means that we need to ensure that `clear()` is invoked on the transaction boundary for our test entity manager.

We may also need to deal with the issue of propagation. In some respects, propagation is easy in a test environment. If you inject the same application-managed entity manager instance into two session bean instances, the beans share the same persistence context as if the entity manager were propagated with the transaction. In fact, it is far more likely that you will need to inject multiple entity managers to simulate the intentional lack of propagation (such as a bean that invokes a `REQUIRES_NEW` method on another bean) than that you will have to do anything special for propagation.

Let's look at a concrete example of transaction propagation using the examples we first introduced in Chapter 5. Listing 12-17 shows the implementation for the `AuditService` session bean that performs audit logging. We have used setter injection in this example to contrast it against the version from Chapter 5.

Listing 12-17. `AuditService` *Session Bean with Setter Injection*

```
@Stateless
public class AuditServiceBean implements AuditService {
    private EntityManager em;

    @PersistenceContext(unitName="hr")
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    public void logTransaction(int empNo, String action) {
        // verify employee number is valid
        if (em.find(Employee.class, empNo) == null) {
            throw new IllegalArgumentException("Unknown employee id");
        }
        LogRecord lr = new LogRecord(empNo, action);
        em.persist(lr);
    }
}
```

Likewise, Listing 12-18 shows a fragment from the `EmployeeService` session bean that uses the `AuditService` session bean to record when a new `Employee` instance has been persisted. Because both the `createEmployee()` and `logTransaction()` methods are invoked in the same transaction without a commit in between, the persistence context must be propagated from one to the other. Again we have used setter injection instead of field injection to make the bean easier to test.

Listing 12-18. EmployeeService *Session Bean with Setter Injection*

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    EntityManager em;
    AuditService audit;

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    @EJB
    public void setAuditService(AuditService audit) {
        this.audit = audit;
    }

    public void createEmployee(Employee emp) {
        em.persist(emp);
        audit.logTransaction(emp.getId(), "created employee");
    }

    // ...
}
```

Using the previous two session beans as an example, Listing 12-19 demonstrates how to emulate propagation between two transaction-scoped container-managed entity managers. The first step to make this testable is to instantiate each session bean. The AuditService bean is then injected into the EmployeeService bean, and the test entity manager instance is injected into both session beans. The injection of the same EntityManager instance effectively propagates any changes from the EmployeeService bean to the AuditService bean. Note that we have also used the entity manager in the test to locate and verify the results of the business method.

Listing 12-19. *Simulating Container-Managed Transaction Propagation*

```
public class TestEmployeeServiceBean extends TestCase {
    // ...

    public void testCreateEmployee() throws Exception {
        EmployeeServiceBean bean = new EmployeeServiceBean();
        AuditServiceBean auditBean = new AuditServiceBean();
        bean.setEntityManager(em);
        bean.setAuditService(auditBean);
        auditBean.setEntityManager(em);
        Employee emp = new Employee();
        emp.setId(99);
        emp.setName("Wayne");
        bean.createEmployee(emp);
        emp = em.find(Employee.class, 99);
        assertNotNull(emp);
        assertEquals(99, emp.getId());
        assertEquals("Wayne", emp.getName());
    }

    // ...
}
```


Other Services

There is more to a session bean than just dependency injection and transaction management. For example, as we saw in Chapter 3, session beans can also take advantage of life cycle methods. Other services that are beyond the scope of this book include security management and interceptors.

The general rule is that in a test environment, you need to manually perform the work that would have otherwise been done automatically by the container. In the case of life cycle methods, for example, you will have to explicitly invoke these methods if they are required for a particular test. Given this requirement, it is a good idea to use package or protected scope methods so that they can be manually invoked by test cases.

That being said, be aggressive in determining the true number of things that have to occur in order for a test to succeed. Just because security roles have been declared for a session bean method doesn't mean that it actually has any effect on the test outcome. If it doesn't have to be invoked prior to the test, don't waste time setting up the test environment to make it happen.

Using Spring for Integration Testing

When multiple session beans collaborate together to implement a particular application use case, there can be a lot of scaffolding code required to get things up and running. If multiple test cases share similar graphs of session beans, then some or all of this code may have to be duplicated across multiple test cases. Ideally, we would like a framework to assist with issues like dependency injection in our test environment.

At this point in time there are no embeddable EJB 3.0 containers that we can use in an integration test, but there are several frameworks that can perform dependency injection without the need for an application server. One of these is the Spring framework, which maintains a stand-alone dependency-injection mechanism that can be integrated into any application. To demonstrate how to use Spring for integration testing with session beans and the entity manager, we will revisit the propagation test case from the preceding Container-Managed Entity Managers section and convert it to use Spring for dependency management.

■ **Tip** This section is intended for readers already familiar with the Spring framework. Visit http://www.springframework.org to learn more.

Spring is designed to work with POJO service classes, and in most cases EJB 3.0 session beans are fully compatible with the format that it is expecting. Spring supports several different injection styles, but the one that is easiest to adapt to session beans is setter injection. So long as the bean class uses setter injection to acquire resources, no changes are required to use the bean with Spring.

One issue that we have to contend with is that by default Spring treats all managed beans as singletons. Therefore there is only one instance per named class type. How we configure the entity manager in the bean configuration file will determine whether or not it is shared between entity instances.

Before we can change our test case to use Spring, we need to establish a helper class that will let Spring create and manage entity manager instances. Listing 12-20 shows a bean class that bootstraps an `EntityManagerFactory` instance using the `Persistence` class and uses it to create and return new application-managed entity managers from the `createEntityManager()` method. The `unitName` field will be injected into this class using a value we configure in the XML bean configuration file. We will also configure the "em-factory" bean to invoke the `destroy()` method when the bean factory is destroyed so that the `EntityManagerFactory` instance will be closed correctly. Closing the factory will also close any entity managers that were created from it.

Listing 12-20. *Helper Class to Create Entity Manager Instances*

```
public class EntityManagerFactoryBean {
    String unitName;
    EntityManagerFactory emf;

    public void setUnitName(String unitName) {
        this.unitName = unitName;
    }

    public EntityManager createEntityManager() {
        if (emf == null) {
            emf = Persistence.createEntityManagerFactory(unitName);
        }
        return emf.createEntityManager();
    }

    public void destroy() {
        if (emf != null) {
            emf.close();
        }
    }
}
```

Next we need to configure this class in the bean configuration file and add two other dynamic bean definitions, one for shared (propagated) entity managers and one for private (non-propagated) entity managers. Listing 12-21 shows this configuration. Note that the "private-entity-manager" bean definition sets the `singleton` attribute to `false`, meaning that a new entity manager will be created every time a bean with this id is requested.

Listing 12-21. *Bean Configuration for Entity Managers*

```
<beans>
    <bean id="em-factory"
          class="examples.session.EntityManagerFactoryBean"
          destroy-method="destroy">
        <property name="unitName">
            <value>hr</value>
        </property>
    </bean>
    <bean id="shared-entity-manager"
          factory-bean="em-factory"
          factory-method="createEntityManager"/>
    <bean id="private-entity-manager"
          factory-bean="em-factory"
          factory-method="createEntityManager"
          singleton="false"/>
    ...
</beans>
```

So far we have the ability to create and manage entity managers using the Spring bean factory. We can access these objects directly in our test case or reference them from other beans to have the values injected automatically. The only step left before we get to our test case is to add in the configuration for our EJB 3.0 session beans. Listing 12-22 shows the remaining bean definitions for this test.

Listing 12-22. *Configuring Session Beans for Use with Spring*

```
<beans>
    ...
    <bean id="employee-service"
          class="examples.session.EmployeeServiceBean">
        <property name="entityManager">
            <ref bean="shared-entity-manager"/>
        </property>
        <property name="auditService">
            <ref bean="audit-service"/>
        </property>
    </bean>
    <bean id="audit-service"
          class="example.session.AuditServiceBean">
        <property name="entityManager">
            <ref bean="shared-entity-manager"/>
        </property>
    </bean>
</beans>
```

Because we are emulating propagation, both of the session beans in this test will use the same shared entity manager instance. To make this possible, we configure each bean to reference the "shared-entity-manager" bean for the "entityManager" property.

Finally we can modify the test case to use the dependency-injection features of Spring. Listing 12-23 shows the complete test case revised to use a dependency-injection framework. The propagated entity manager and `EmployeeService` session bean are both obtained from the `XmlBeanFactory` instance. Spring handles all of the dependency injection and ensures that both beans are sharing the correct entity manager instance. When the test is complete, we explicitly destroy the singleton objects in the `XmlBeanFactory` instance in order to ensure that the `EntityManagerFactory` will be closed before the next test.

Listing 12-23. *Test Case Using the Spring* `BeanFactory`

```
public class TestEmployeeServiceBean extends TestCase {
    XmlBeanFactory factory;

    public void setUp() {
        ClassPathResource resource =
            new ClassPathResource("test-employee-service-bean.xml");
        factory = new XmlBeanFactory(resource);
    }

    public void tearDown() {
        factory.destroySingletons();
    }

    public void testCreateEmployee() throws Exception {
        EmployeeService bean =
            (EmployeeService) factory.getBean("employee-service");
        Employee emp = new Employee();
        emp.setId(99);
        emp.setName("Wayne");
        bean.createEmployee(emp);
        EntityManager em =
            (EntityManager) factory.getBean("shared-entity-manager");
        emp = em.find(Employee.class, 99);
        assertNotNull(emp);
        assertEquals(99, emp.getId());
        assertEquals("Wayne", emp.getName());
    }

    // ...
}
```


For two session beans this approach is arguably overkill compared to the same test case shown in Listing 12-19. But it should be easy to see even from this small example how complex bean relationships can be modeled and realized using a dependency-injection framework. In this example we have only focused on session bean classes, but many other resource types from the Java EE environment can be mocked or extended for testing and automatically managed by a dependency-injection framework for integration testing.

Best Practices

A full discussion of developer testing strategies is beyond the scope of this chapter, but to make testing of application code that uses entities easier, consider adopting the following best practices:

- **Avoid using the entity manager from within entity classes.** This creates a tight coupling between the domain object and the persistence API, making testing difficult. Queries related to an entity but not part of its object-relational mapping are better executed within a session façade or data access object.
- **Prefer dependency injection to JNDI lookups in session beans.** Dependency injection is a key technology for simplifying tests. Instead of mocking the JNDI interfaces to provide runtime support for testing, the required values can be directly assigned to the object using a setter method or field access. Note that accessing private fields from a test case is bad form. Either use package private fields as the target for injected objects or provide a setter method.
- **Isolate persistence operations.** Keeping `EntityManager` and `Query` operations separate in their own methods makes replacing them easier during unit testing.
- **Decouple with interfaces.** Just as the Java Persistence API uses interfaces to minimize dependencies on the persistence provider, loosely coupled objects with interfaces can help manage complex dependencies.
- **Refactor when necessary.** Don't be afraid to refactor application code to make it more test-friendly so long as the refactoring benefits the application as a whole. Method extraction, parameter introduction, and other refactoring techniques can help break down complex application logic into testable chunks, improving the overall readability and maintainability of the application in the process.

Note that everything we have described so far is just the beginning of a complete strategy for testing Java EE applications that use the Java Persistence API. As more developers gain experience with the Java Persistence API and learn how to take advantage of the simplifications brought about by EJB 3.0, we expect much more to be written on this subject.

Summary

In this chapter we started with an exploration of testing enterprise applications and the challenges that have traditionally faced developers. We also looked at the different types of testing performed by developers, quality engineers, and customers, and we refined our focus to look specifically at developer tests for EJB 3.0 and Java Persistence API applications.

In the section on unit testing, we looked at how to test entity classes and then pulled back to look at how to test session beans in combination with entities in a unit test environment. We introduced the concept of mock objects and explored how to test code that depends on the entity manager without actually using a real entity manager.

In our discussion of integration testing, we discussed how to get the entity manager up and running in JUnit tests in the Java SE environment and the situations where it makes sense to use this technique. We covered a number of issues related to the entity manager, including how to safely seed a database for testing, how to use multiple mapping files for different database configurations, and how to minimize the number of database connections required for a test suite.

We looked at how to use session beans in integration tests and how to deal with dependency-injection and transaction-management issues. For transaction management, we looked at how to emulate container-managed and bean-managed transactions, as well as how to simulate persistence context propagation in a test environment. We concluded with a summary of some best practices to consider when building Java EE applications using the Java Persistence API.

In the next chapter we will look at how to migrate existing EJB 2.1 and JDBC applications to the Java Persistence API.

CHAPTER 13

■ ■ ■

Migration

Now that the Java Persistence API has been explained in detail, the challenge is deciding how and when to adopt the new persistence model. For new applications, this is not an issue, but what about existing applications? In this chapter we will look at the challenges facing developers wishing to integrate the Java Persistence API into legacy applications and offer some solutions to help ease the transition.

Migrating from CMP Entity Beans

Until the publication of the EJB 3.0 specification and the Java Persistence API, the only persistence technology officially part of the Java EE platform was container-managed persistence using EJB entity beans. Ever since they were first required to be supported in EJB 1.1, entity beans have been criticized as being both too complex and lacking in features to handle the persistence requirements of real-world applications. But standards matter in the enterprise, so despite the availability of proven object-relational mapping solutions, both commercial and open source, companies have always found a way to work around the entity bean shortcomings and get the job done. As a result, there is a large installed base of applications based on CMP entity beans, and bringing them forward into the next generation of Java EE standards may be a task worth pursuing.

The complexity of entity beans lies not in the concept but in the implementation. Like session beans, entity beans are true EJB components, with separate classes for the bean implementation, home interface, and business interfaces. Entity beans also require a verbose XML deployment descriptor that describes the persistent properties of the bean, container-managed relationships between entities, and the EJB QL queries used to access the entities. Finally, many of the entity bean details require vendor-specific configuration to deploy and run. In response to these issues, the Java Persistence API offers a programming model that is easier to use, while offering a larger feature set with less vendor-specific configuration.

Although the Java Persistence API is the standard persistence model moving forward, the good news for companies that have made an investment in CMP entity beans is that the EJB 3.0 specification still fully supports container-managed persistence. Existing applications will work out of the box without changes and can expect to do so for years to come. The EJB 3.0 specification is only now deprecating version 1.1 of the entity bean model. All Java EE 5–compliant application servers must support EJB 2.0 and 2.1 CMP entity beans.

That's good news for applications that aren't likely to require much development going forward, but what about applications that are planning revisions? Is it feasible to move away from CMP and take advantage of the Java Persistence API? In many cases it will depend upon

the design of your application. Only you can decide the most appropriate plan of action for your application. The following sections will lay out the issues and discuss potential strategies for migrating CMP applications to help you make your own decision.

■ **Note** This chapter assumes that you are familiar with EJB 2.1 container-managed entity bean implementation and configuration.

Scoping the Challenge

The challenge in moving from entity beans to entities is not the entity beans themselves. However complex they are to implement, they are relatively straightforward to use. The problem with entity beans is that the public API they expose is tightly coupled to the component model on which they are based. The principal issue facing any migration is the extent and manner in which application code interacts with entity bean interfaces. The more code that uses entity beans, the harder it is to migrate.

There are also some entity bean features that are not reflected in the Java Persistence API. Some of these features, such as container-managed relationships, can be worked around, while others are difficult if not impossible to replace.

The primary showstopper scenario is the use of remote entity bean interfaces. There is simply no equivalent to remote objects in the Java Persistence API. Entities are plain Java classes, not interface-based components that can be compiled down into RMI or Common Object Request Broker Architecture (CORBA) stubs. Entities are mobile in the sense that they can be serialized and transferred between client and server, but they are not network-aware. Ever since the EJB 2.0 specification introduced local interfaces, developers have been warned not to use remote interfaces on entity beans due to the overhead of the network infrastructure they require. If your application is one of the few, it is very unlikely that a migration would be possible until the application was refactored to use local interfaces.

■ **Tip** Often remote interfaces are used on entities only to facilitate transporting data off to a remote tier for presentation. Consider introducing the Transfer Object pattern (described later in this chapter) to remove remote interfaces in these cases. Transfer objects share a strong symmetry with serializable entities, making them good starting points for migration.

Applications that have isolated their persistence code, most likely through the use of one or more design patterns, present the least amount of effort to convert. Conversely, applications that sprinkle entity bean access across all tiers and are tightly coupled to the entity bean API present the greatest challenge. Refactoring to decouple business and presentation logic from persistence code is often a worthwhile exercise before attempting to migrate to the Java Persistence API.

Two levels of application migration are discussed next. The first, documented in Entity Bean Conversion, details the process of mapping an existing entity bean to a new entity. From there the developer can begin refactoring the application to introduce the entity manager and remove entity bean usage. The second level builds on the first by identifying business tier design patterns that present an opportunity to make a switch in persistence technologies with minimal impact to existing application code. Design patterns are discussed in the Leveraging Design Patterns section.

Entity Bean Conversion

When planning any conversion between entity beans and entities, it is useful to use the existing bean as a template for the new entity. The bean class, interfaces, and XML deployment descriptor describe the persistent fields used by the entity, the queries used by the application to find entity instances, and the container-managed relationships between entities. The following sections describe the process to convert an entity bean into an entity. Later sections will describe how to integrate these new entities into an existing application.

Converting the Business Interface

Entity beans are defined using a bean class, business interface, and home interface. When creating the initial entity version, the business interface or bean class can be used as a template. The business interface is often the best place to start as it defines the set of operations directly available on the entity as opposed to the bean class, which also includes home and finder methods specific to the home interface.

Migrating Properties

To demonstrate the process of migrating an entity bean to the Java Persistence API, we will look at converting an entity bean that stores information about a department. The business interface for the `Department` entity bean is shown in Listing 13-1.

Listing 13-1. *Department Business Interface*

```
public interface Department extends EJBLocalObject {
    public int getId();

    public String getName();
    public void setName(String name);

    public Collection getEmployees();
    public void setEmployees(Collection employees);

    public Employee getManager();
}
```

To begin converting this interface into an entity, a concrete implementation of the interface must be provided, removing the dependency on `EJBLocalObject` and providing a field to implement each of the persistent properties. The properties `id`, `name`, and `employees` all map to either persistent fields or relationships. The `getManager()` method is actually a non-persistent business method that searches for and returns the manager for the department. Therefore while the business interface is a good starting point, the bean implementation or the XML descriptor,

which lists the persistent fields, must be consulted to determine the true meaning for each business method.

With the set of persistent properties identified, the next step is to determine how they map to the database. Unfortunately, this mapping was not standardized by the EJB specification, so vendor-specific XML descriptors will have to be checked. For this example, assume that the entity bean maps to the table `DEPT`, which has columns `ID` and `NAME`. Setting aside the `getManager()`, `getEmployees()`, and `setEmployees()` methods for now, the entity implementation with basic mappings is shown in Listing 13-2. Because the entity name and table name are different, the `@Table` annotation is required to override the default table name of the entity.

Listing 13-2. *Department Entity with Basic Mappings*

```
@Entity
@Table(name="DEPT")
public class Department {
    @Id
    private int id;
    private String name;

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

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

Migrating Business Methods

Non-persistent business methods may be a source of problems during entity bean conversion. Many business methods simply perform operations using the persistent state of the entity (using the persistent getter methods to obtain data) and these may be copied to the new entity as is. However, the EJB specification also allows for business methods to invoke select methods in order to issue queries and operate on the results. Listing 13-3 shows a fragment from the `DepartmentBean` class, which defines the implementation of the `getManager()` method.

Listing 13-3. *Business Method That Uses a Select Method*

```
public abstract class DepartmentBean implements EntityBean {
    // ...

    public abstract Employee ejbSelectManagerForDept(int deptId);

    public Employee getManager() {
        return ejbSelectManagerForDept(getId());
    }

    // ...
}
```


Select methods, which begin with the prefix "ejbSelect", are container-provided implementations of EJB QL queries. They may be called by home methods (described later) and business methods. Business methods that invoke "ejbSelect" methods pose a problem in entity bean conversion, as the entity manager required to execute the query is not typically available to entity bean instances. In this example, the select method issues the following query, which was defined in the XML descriptor:

```
SELECT OBJECT(e)
FROM Employee e
WHERE e.department.id = ?1 AND e.manager.department.id <> ?1
```

To execute these queries from within the entity class, the entity manager must be made available to the entity instance. Since the entity manager is not part of the persistent state of the entity, it is strongly discouraged that you store a reference to it. Instead, consider the Service Locator pattern so that the entity manager can be obtained from within the entity even though it is not part of the entity. The following implementation of `getManager()` uses this approach:

```
public Employee getManager() {
    EntityManager em =
        ServiceLocator.getInstance().getEntityManager("EmployeeService");
    return (Employee) em.createNamedQuery("Department.managerForDept")
                        .setParameter(1, getId())
                        .getSingleResult();
}
```

The `ServiceLocator` class looks up the entity manager from JNDI using the current environment naming context. The downside to this approach is that entities tend to get used in a lot of different components, each with its own set of environment references. To ensure portability, the same entity manager reference name must be used consistently in all components, or some vendor-specific approach must be used to acquire the entity manager independent of context.

Entity manager operations within an entity class are generally considered bad style as it introduces a dependency on the persistence runtime directly into the entity. This tightly couples the entity implementation to a particular persistence mechanism (the entity is no longer a plain Java object) and makes testing more difficult. Generally speaking, we recommend moving the business method to a session façade or other business-focused component instead of embedding entity manager operations within the entity class. The only consequence of moving the method to another class is that the entity needs to be passed as an argument to the method in its new location.

Migrating Container-Managed Relationships

CMP entity beans may make use of container-managed relationships. These relationships are called managed because the developer is required to update only one side of the relationship and the server will ensure that the other side of the relationship is updated automatically. Although there is no direct equivalent to container-managed relationships in the Java Persistence API, the XML descriptor for these relationships can guide the definition of entity relationships for object-relational mapping.

The Department entity bean has a one-to-many relationship with the Employee entity bean. Listing 13-4 shows the XML definition of the container-managed relationship between these two entity beans.

Listing 13-4. *XML Definition of a Container-Managed Relationship*

```
<ejb-relation>
    <ejb-relation-name>Dept-Emps</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>Dept-has-Emps</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>DepartmentBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
           <cmr-field-name>employees</cmr-field-name>
           <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>Emps-have-Dept</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
            <ejb-name>EmployeeBean</ejb-name>
        </relationship-role-source>
        <cmr-field><cmr-field-name>department</cmr-field-name></cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

Each side of the relationship is defined using the ejb-relationship-role element. The relationship-role-source and cmr-field elements define the entity bean and relationship property being mapped. The multiplicity element defines the cardinality of that side of the relationship. There is a direct mapping between each ejb-relationship-role element and a relationship annotation, the choice of which is determined by the multiplicity elements from each end of the relationship.

Applying this pattern, the previous relationship descriptor maps to an @OneToMany annotation on the employees attribute of the Department entity and an @ManyToOne annotation on the department attribute of the Employee entity. Since the relationship is bi-directional, Employee will be the owner and Department the inverse, so the mappedBy element of the @OneToMany annotation is set to the name of the owning attribute, in this case department.

We can now complete our mapping for the Department entity by adding the relationships. Listing 13-5 shows the complete entity class.

Listing 13-5. Department *Entity with Relationship Mappings*

```
@Entity
@Table(name="DEPT")
public class Department {
    @Id
    private int id;
    private String name;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees = new ArrayList<Employee>();

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

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

    public Collection<Employee> getEmployees() { return employees; }
}
```

Clients that used to use the relationship properties of the entity bean business interface require special attention when converted to use entities. Relationships that were previously managed by the container now require explicit maintenance to both sides of the relationship whenever a change occurs. In most cases, this amounts to one extra line of code. For example, adding an Employee entity bean to the employees property of the Department entity bean with container-managed relationships used to look like this:

```
dept.getEmployees().add(emp);
```

Without container-managed relationships, an extra step is required:

```
dept.getEmployees().add(emp);
emp.setDepartment(dept);
```

Rather than adding these statements directly throughout application code, a best practice to consider is the use of helper methods on entities to manage relationships. The following example demonstrates these same operations as they would be implemented on the Department entity:

```
public void addEmployee(Employee emp) {
    getEmployees().add(emp);
    emp.setDepartment(this);
}
```

Converting the Home Interface

Creating an entity out of the entity bean business interface is often only the first step in conversion. Application code relies on the home interface to create new entity beans, find existing entity beans, and handle business methods that are related to an entity but not specific to any one entity bean instance.

The first choice to be made regarding the home interface is whether or not application code will be rewritten to work directly with the entity manager. Doing so obsoletes most of the home interface operations, but it may be challenging to implement depending on how tightly coupled the entity bean API is to the application code. Business methods on the home interface also must be accommodated.

If the home interface is still required, a stateless session bean may be used to provide equivalent methods to the home interface operations. The following sections continue the `Department` entity example by implementing a session façade for its business methods and finder operations.

Migrating Queries

EJB QL queries for CMP entity beans are defined in the deployment descriptor. Listing 13-6 shows two query definitions for the `Department` entity bean.

Listing 13-6. *EJB QL Query Definitions*

```
<query>
    <query-method>
        <method-name>findAll</method-name>
        <method-params/>
    </query-method>
    <ejb-ql>SELECT OBJECT(d) From Department d</ejb-ql>
</query>
<query>
    <query-method>
        <method-name>findByName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>SELECT OBJECT(d) FROM Department d WHERE d.name = ?1</ejb-ql>
</query>
```

To reuse these same queries with the converted entity bean, it is necessary to define named queries on the entity. Recall from Chapter 7 that every EJB QL query is a legal JPQL query; therefore existing EJB QL entity bean queries can be migrated without change to the Java Persistence API. The only thing we need to do is define a name for the query that will be unique across the persistence unit. To facilitate this we will prepend the query name with the name of the entity. The following `@NamedQuery` annotations mirror the XML versions:

```
@NamedQueries({
    @NamedQuery(name="Department.findAll",
                query="SELECT d FROM Department d"),
    @NamedQuery(name="Department.findByName",
                query="SELECT d FROM Department d WHERE d.name = ?1")
})
```


Migrating Home Methods

A home method is any method on the home interface that is not a finder (starts with "findBy") or a create method (starts with "create"). They are also typically the easiest to integrate into a session façade, because their implementation often relies only on select methods. Home methods are implemented on the bean class in methods prefixed with "ejbHome". Listing 13-7 shows a fragment of the `DepartmentBean` demonstrating a home method and the select method that it uses.

Listing 13-7. *Entity Bean Home Method*

```
public abstract class DepartmentBean implements EntityBean {
// ...

    public abstract Collection ejbSelectEmployeesWithNoDepartment()
        throws FinderException;

    public Collection ejbHomeUnallocatedEmployees() throws FinderException {
        return ejbSelectEmployeesWithNoDepartment();
    }

    // ...
}
```

Assuming that the entity manager has been injected into the session bean, we can use the EJB QL query definition from the XML descriptor to re-implement this method:

```
public Collection unallocatedEmployees() throws FinderException {
    try {
        return em.createQuery("SELECT e FROM Employee e WHERE e.dept IS NULL")
                 .getResultList();
    } catch (PersistenceException e) {
        throw new FinderException(e.getMessage());
    }
}
```

Creating the Façade

With queries mapped and home methods ready for conversion, creating the façade is straightforward. The advantage of a session bean is that it may be looked up from JNDI just as the entity home was previously and can use a similar interface in order to minimize application code changes. Listing 13-8 shows the home interface for the `Department` entity bean.

Listing 13-8. *The* `DepartmentHome` *Interface*

```
public interface DepartmentHome extends EJBLocalHome {
    public Department create(int id) throws CreateException;
    public Department findByPrimaryKey(int id) throws FinderException;
    public Collection findAll() throws FinderException;
    public Department findByName(String name) throws FinderException;
    public Collection unallocatedEmployees() throws FinderException;
}
```

The first step in this refactoring is to modify the home interface so that it does not extend `EJBLocalHome`. With this dependency removed, the interface is now suitable for use as a stateless session bean business interface. Listing 13-9 shows the converted interface.

Listing 13-9. *The* `DepartmentHome` *Business Interface*

```
public interface DepartmentHome {
    public Department create(int id) throws CreateException;
    public Department findByPrimaryKey(int id) throws FinderException;
    public Collection findAll() throws FinderException;
    public Department findByName(String name) throws FinderException;
    public Collection unallocatedEmployees() throws FinderException;
    public void remove (Object pk) throws RemoveException;
    public void remove (Department dept) throws RemoveException;
}
```

Note the addition of the `remove()` methods. The first is the standard `remove()` method that is part of the `EJBLocalHome` interface, and the second is a convenience method that does not require the user to extract the primary key from the entity. Since entities do not implement `EJBLocalObject`, application code will no longer be able to invoke the `remove()` method directly on the entity bean. Invoking these methods is a compromise that allows application code to avoid directly using the entity manager while maintaining the ability to remove an entity instance. Application code will need to be refactored to change all invocations of `remove()` on the entity bean to one of these new methods on the session bean home façade.

The next step is to create a session bean façade that implements the entity home interface. Using the techniques we have discussed so far, Listing 13-10 shows the complete stateless session bean implementation of the `DepartmentHome` interface. Note the use of checked exceptions on the bean methods. Until existing code is refactored to use the runtime exception model supported by the Java Persistence API, there may be client code that expects `CreateException` or `FinderException` exceptions to be thrown. We have also specified the `name` element for the `@PersistenceContext` annotation. This allows business methods such as the `getManager()` method we described earlier in the section Migrating Business Methods to access the entity manager from the "java:comp/env/EmployeeService" JNDI location.

Listing 13-10. *The* `DepartmentHome` *Session Bean*

```
@Stateless
public class DepartmentHomeBean implements DepartmentHome {
    @PersistenceContext(name="EmployeeService", unitName="EmployeeService")
    EntityManager em;

    public Department create(int id) throws CreateException {
        Department dept = new Department();
        dept.setId(id);
        try {
            em.persist(dept);
        } catch (PersistenceException e) {
            throw new CreateException(e.getMessage());
        } catch (IllegalArgumentException e) {
            throw new CreateException(e.getMessage());
        }
        return dept;
    }

    public Department findByPrimaryKey(int id) throws FinderException {
        try {
            return em.find(Department.class, id);
        } catch (PersistenceException e) {
            throw new FinderException(e.getMessage());
        }
    }

    public Collection findAll() throws FinderException {
        try {
            return em.createNamedQuery("Department.findAll")
                     .getResultList();
        } catch (PersistenceException e) {
            throw new FinderException(e.getMessage());
        }
    }

    public Department findByName(String name) throws FinderException {
        try {
            return (Department)
                em.createNamedQuery("Department.findByDepartmentName")
                  .setParameter(1, name)
                  .getSingleResult();
        } catch (PersistenceException e) {
            throw new FinderException(e.getMessage());
        }
    }

    public Collection unallocatedEmployees() throws FinderException {
        try {
            return em.createNamedQuery("Department.empsWithNoDepartment")
                     .getResultList();
        } catch (PersistenceException e) {
            throw new FinderException(e.getMessage());
        }
    }

    public void remove (Object pk) throws RemoveException {
        Department d = em.find(Department.class, pk);
        if (d == null) {
            throw new RemoveException("Unable to find entity with pk: " + pk);
        }
        em.remove(d);
    }
```

```
    public void remove(Department dept) throws RemoveException {
        Department d = em.find(Department.class, dept.getId());
        if (d == null) {
            throw new RemoveException("Unable to find entity with pk: " +
                                      dept.getId());
        }
        em.remove(d);
    }
}
```

Migrating from JDBC

The oldest and most basic form of relational persistence with Java is JDBC. A thin layer over the programming interfaces required for communicating with a database, JDBC operations are defined primarily in terms of SQL statements. Applications that make heavy use of JDBC may be more difficult to migrate to the Java Persistence API than applications that depend on entity beans.

As with entity beans, the complexity of migration depends on how tightly coupled the business logic is to the JDBC API. There are two basic issues that we need to be concerned with. The first is the amount of code that depends on the `ResultSet` or `RowSet` interfaces. The second is the amount of SQL and the role it plays in the application.

The `ResultSet` and `RowSet` interfaces are a concern because there is no logical equivalent to these structures in the Java Persistence API. Results from JPQL and SQL queries executed through the `Query` interface are basic collections. Even though we can iterate over a collection, which is semantically similar to the row position operations of the JDBC API, each element in the collection is an object or an array of objects. There is no equivalent to the column index operations of the `ResultSet` interface.

Emulating the `ResultSet` interface over the top of a collection is unlikely to be a worthwhile venture. Although some operations could be mapped directly, there is no generic way to map the attributes of an entity to the column positions needed by the application code. There is also no guarantee of consistency in how the column positions are determined; it may be different between two queries that achieve the same goal but have ordered the SELECT clause differently. Even when column names are used, the application code is referring to the column aliases of the query, not necessarily the true column names.

In light of these issues, our goal in planning any migration from JDBC is to isolate the JDBC operations so that they can be replaced as a group as opposed to accommodating business logic that depends on JDBC interfaces. Refactoring the existing application to break its dependencies on the JDBC interfaces is the easiest path forward.

With regards to the SQL usage in a JDBC application, we want to caution that though the Java Persistence API supports SQL queries, it is unlikely that this will be a major benefit to migration of an existing application. There are a number of reasons for this, but the first to consider is that most SQL queries in a JDBC application are unlikely to return results that map directly to the domain model of a Java Persistence API application. As we learned in Chapter 9, to construct an entity from a SQL query requires all of the data and foreign key columns to be returned, regardless of what will eventually be required by the application code at that point in time.

If the majority of SQL queries need to be expanded to add columns necessary to satisfy the requirements of the Java Persistence API and if they then need to be mapped before they can be used, then rewriting the queries in JPQL is probably a better investment of time. The syntax of a JPQL query is easier to read, easier to construct, and directly maps to the domain model you want to introduce to the application. The entities have already been mapped to the database, and the provider knows how to construct efficient queries to obtain the data you need. SQL queries have a role, but they should be the exception, not the rule.

There are many Java EE design patterns that can help in this exercise. We will be exploring several of these in detail later in the chapter, but it is worth mentioning at least a few now in the context of JDBC applications specifically. The first and most important pattern to consider is the Data Access Object pattern. This cleanly isolates the JDBC operations for a specific use case behind a single interface that we can migrate forward. Next consider the Transfer Object pattern as a way of introducing an abstraction of the row and column semantics of JDBC into a more natural object model. When an operation returns a collection of values, don't return the `ResultSet` to the client. Construct Transfer Objects and build a new collection similar to the results of the `Query` operations in the Java Persistence API. These steps can go a long way to creating boundary points where the Java Persistence API can be introduced without having a major impact on the application logic.

Migrating from Other ORM Solutions

Since the very beginnings of the Java programming language, object-relational mapping solutions have been available in one form or another, provided first by commercial vendors and later by a number of open source solutions. Transparent persistence was also standardized for Java as part of the Java Data Objects (JDO) specification, although object-relational mapping was not explicitly defined by JDO until version 2.0 of the specification. It was the growing popularity of the various proprietary solutions that pushed the EJB expert group to create the Java Persistence API and release it as part of the Java EE specification.

Fortunately, representatives from all major object-relational mapping providers (commercial and open source) contributed to the definition of the Java Persistence API specification. As a result, the Java Persistence API standardizes a decade of object-relational mapping techniques that are well understood and in production today. As participants in the process, existing object-relational mapping providers are also providing ways to move from their solution to the new standard while preserving features in their products outside the scope of the standard. It would be impractical for this book to describe the migration process for each product. Instead we invite you to contact your vendor for instructions on how to take advantage of the Java Persistence API.

Leveraging Design Patterns

For many years now, design patterns have been heavily promoted to help developers build better Java applications. For enterprise development in particular, using the proven solutions documented in design patterns has helped to manage the complexity of the Java EE platform. Ironically, patterns designed to help integrate entity beans and JDBC into enterprise applications are also the key to eliminating those technologies and introducing the Java Persistence API. This is because enterprise design patterns almost always point to solutions that isolate persistence code from the rest of the application. An enterprise application that has embraced

common Java EE design patterns typically interacts only with session façades, data access objects, and transfer objects—perfect boundary points to safely make a switch in persistence technologies.

The following sections describe the design patterns that offer the greatest potential to replace container-managed entity beans and JDBC code with the lightweight entities of the Java Persistence API.

Transfer Object

The *Transfer Object*[1] pattern, also called the *Data Transfer Object* pattern, encapsulates the results of persistence operations in simple objects that are decoupled from the particular persistence implementation. Implemented in this way, transfer objects may be shared between application tiers without having dependencies on the entity bean API or requiring the use of remote entity beans. Although originally designed as a solution to avoid the poor performance of network calls for remote entity beans, they are widely used even in applications that do not have remote tiers in order to isolate business logic from the persistence API.

Fine-Grained Transfer Objects

When used with entity beans, transfer objects are typically implemented as a mirror of the entity data that is to be transported. For every persistent attribute on the entity bean, the same property is implemented on the transfer object. Listing 13-11 shows the business interface for the `Address` entity bean. It consists entirely of getter and setter methods to manage the persistent state of the entity bean.

Listing 13-11. *The Address Business Interface*

```
public interface Address extends EJBLocalObject {
    public int getId();
    public void setId(int id);

    public String getStreet();
    public void setStreet(String street);

    public String getCity();
    public void setCity(String city);

    public String getState();
    public void setState(String state);

    public String getZip();
    public void setZip(String zip);
}
```

1. Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition.* Upper Saddle River, N.J.: Prentice Hall PTR, 2003.

Using this business interface as a template, the transfer object corresponding to the `Address` business interface is shown in Listing 13-12. It is implemented using one field corresponding to each persistent property of the entity bean. This particular transfer object can also be used as a template for new entity bean instances or to capture changes that can be merged into an entity bean instance for updating. Transfer objects are often implemented as immutable objects if the client has no need to make changes and return the objects to the server for processing.

Listing 13-12. *The Address Transfer Object*

```
public class AddressTO implements Serializable {
    private int id;
    private String street;
    private String city;
    private String state;
    private String zip;

    public AddressTO() {}

    public AddressTO(int id, String street, String city,
                     String state, String zip) {
        this.id = id;
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

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

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

    public String getState() { return state; }
    public void setState(String state) { this.state = state; }

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    public String getZip() { return zip; }
    public void setZip(String zip) { this.zip = zip; }
}
```

This style of transfer object implementation, containing a one-to-one mapping of the persistent attributes from an entity bean, is considered *fine-grained*. The entire entity bean model used by the application is reflected in an identical, non-persistent model made up of transfer objects. When an application uses a fine-grained transfer object model, one option for migrating to the Java Persistence API is to convert the transfer objects into entities, by applying

an object-relational mapping of the transfer objects to the database tables originally mapped by the entity beans. Even better, this can be accomplished with minimal impact to the business logic code that interacts with the transfer objects.

Compare the previous transfer object to the entity that follows:

```
@Entity
public class AddressTO implements Serializable {
    @Id
    private int id;
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;

    // getter and setter methods
    // ...
}
```

The only difference between the original transfer object and this modified version is the addition of object-relational mapping annotations. Modified in this way, the transfer object, now an entity, is ready to be used with the entity manager to implement the persistence requirements of the application. If the Session Façade or Data Access Object patterns have been used, then this transformation is trivial. Instead of retrieving an entity and converting it into a transfer object, the entity manager and query interfaces can be used to retrieve the transfer object directly from the database.

Coarse-Grained Transfer Objects

A second type of transfer object is sometimes used that does not have a one-to-one correspondence with a particular entity bean. Instead, these transfer objects either collect data from multiple entities into a single transfer object or present a summarized view of overall entity state. This style of transfer object is *coarse-grained* and is sometimes called a view object since it presents a particular view of entity data that does not directly correspond to the entity bean implementation.

Listing 13-13 shows an example of this type of transfer object. Designed for distribution to the web tier for presentation on a web page, the transfer object stores summary information about the manager of a department. The `managerName` property is copied from the `Employee` entity bean, but the `employeeCount` and `avgSalary` properties are aggregate values computed by running summary queries.

Listing 13-13. *A Course-Grained Transfer Object*

```
public class ManagerStats {
    private String managerName;
    private int employeeCount;
    private double avgSalary;
```



```
    public ManagerStats(String managerName, int employeeCount, double avgSalary) {
        this.managerName = managerName;
        this.employeeCount = employeeCount;
        this.avgSalary = avgSalary;
    }

    public String getManagerName() { return managerName; }
    public int getEmployeeCount() { return employeeCount; }
    public double getAverageSalary() { return avgSalary; }
}
```

Fortunately, the Java Persistence API can often accommodate this style of transfer object through the constructor expressions in JPQL queries. The following query populates the transfer object shown previously:

```
SELECT NEW examples.ManagerStats(e.name, COUNT(d), AVG(d.salary))
FROM Employee e JOIN e.directs d
GROUP BY e.name
```

Constructor expression queries are also useful for composite transfer objects that simply combine the data from multiple entities into a single object. This style is sometimes used for entities that have a one-to-one relationship with other entities. The resulting transfer object flattens the object graph so that all reachable persistent fields become properties.

Despite the flexibility of JPQL expressions and native SQL query result set mapping, there will still be situations where transfer objects need to be manually constructed. However, the simplicity of working with entity classes can reduce the amount of code required to build transfer objects and reduce overall complexity as a result.

Session Façade

The *Session Façade*[2] pattern encapsulates business object access behind a session bean façade, typically implemented using stateless session beans. This business interface for the façade presents a coarse-grained view of the operations required on the business data, which may be implemented using entity beans, JDBC, or any other persistence technology.

Originally intended to define coarse-grained boundary operations for access by remote clients, the Session Façade pattern has evolved into a more general service façade, where remote access is no longer the driving factor. Decoupling enterprise applications into sets of collaborating services is a well-established best practice. Each service façade provides the business operations necessary to help realize one or more application use cases.

A key aspect of the Session Façade pattern that makes it appealing for introducing the Java Persistence API is the tendency to isolate persistence operations entirely behind the façade. The original use of transfer objects in the pattern stemmed from the need to prevent entity beans from being used remotely. Today, however, transfer objects are still widely used even for local services as a mechanism to abstract away the particular mechanics of persistence in the application.

---

2. Ibid.

Existing Java EE applications often use entity beans in the implementation of the Session Façade. Listing 13-14 shows a typical EJB 2.1 façade that provides business operations related to the management of `Project` entity beans.

Listing 13-14. *Session Façade with Entity Beans*

```
public class ProjectServiceBean implements SessionBean {
    private SessionContext context;
    private ProjectHome projectHome;
    private EmployeeHome empHome;

    public void setSessionContext(SessionContext context) {
        this.context = context;
    }

    public void ejbCreate() throws CreateException {
        try {
            Context ctx = new InitialContext();
            projectHome = (ProjectHome)
                ctx.lookup("java:comp/env/ejb/ProjectHome");
            empHome = (EmployeeHome)
                ctx.lookup("java:comp/env/ejb/EmployeeHome");
        } catch (NamingException e) {
            throw new CreateException(e.getMessage());
        }
    }

    public void addEmployeeToProject(int projectId, int empId)
        throws ApplicationException {
        try {
            Project project = projectHome.findByPrimaryKey(projectId);
            Employee emp = empHome.findByPrimaryKey(empId);
            project.getEmployees().add(emp);
        } catch (FinderException e) {
            throw new ApplicationException(e);
        }
    }

    // ...
}
```

Relying only on the primary key values as arguments, a service such as the one shown in Listing 13-14 would typically be invoked from a servlet, where the primary keys would have been obtained as part of an earlier display operation using transfer objects. With entity bean access isolated to the bean implementation, introducing entities is relatively straightforward. Listing 13-15 shows the service bean updated for EJB 3.0 and converted to use entities instead of entity beans. No change to existing clients of the service is necessary.

Listing 13-15. *Session Façade with Entities*

```
@Stateless
public class ProjectServiceBean {
    @PersistenceContext(name="EmployeeService")
    private EntityManager em;

    public void addEmployeeToProject(int projectId, int empId)
        throws ApplicationException {
        Project project = em.find(Project.class, projectId);
        if (project == null)
            throw new ApplicationException("Unknown project id: " + projectId);
        Employee emp = em.find(Employee.class, empId);
        if (emp == null)
            throw new ApplicationException("Unknown employee id: " + empId);
        project.getEmployees().add(emp);
        emp.getProjects().add(project);
    }

    // ...
}
```

Data Access Object

The *Data Access Object*[3] pattern, better known simply as the DAO pattern, presents a good opportunity to introduce the Java Persistence API into an existing application. Indeed, the pattern itself was designed on the premise that directly exposing persistence APIs to other application tiers was something to be avoided. Therefore a well-designed data access object implements a simple persistence manager interface by delegating to a particular persistence technology.

The most common form of DAO delegates directly to JDBC, although other persistence technologies are sometimes encountered. Data access objects are typically plain Java objects, although other component types are sometimes used. When implemented as a session bean, particularly when using entity beans, the lines between the data access object pattern and the session façade pattern start to blur.

■ **Note** Many DAO implementations use JDBC directly because it is often considered the "optimal" performance implementation by developers. As the Java Persistence API offers many benefits over direct JDBC, including the potential for performance increases dues to caching, this pattern presents an opportunity to introduce entities and see how they compare to traditional JDBC.

---

3. Ibid.

DAO implementations often use transfer objects to return results. For JDBC implementations, the use of transfer objects gives the illusion of an object-oriented domain model. Listing 13-16 shows a fragment of a DAO that uses JDBC for persistence and returns transfer objects to the client.

Listing 13-16. *DAO Using JDBC for Persistence*

```
public class AddressDAO {
    private static final String INSERT_SQL =
        "INSERT INTO address (id,street,city,state,zip) VALUES (?,?,?,?,?)";
    private static final String UPDATE_SQL =
        "UPDATE address SET street=?,city=?,state=?,zip=? WHERE id=?";
    private static final String DELETE_SQL =
        "DELETE FROM address WHERE id=?";
    private static final String FIND_SQL =
        "SELECT street,city,state,zip FROM address WHERE id=?";

    private DataSource ds;

    public AddressDAO(DataSource ds) {
        this.ds = ds;
    }

    public void create(AddressTO address) {
        Connection conn = null;
        PreparedStatement sth = null;
        try {
            conn = ds.getConnection();
            sth = conn.prepareStatement(INSERT_SQL);
            sth.setInt(1, address.getId());
            sth.setString(2, address.getStreet());
            sth.setString(3, address.getCity());
            sth.setString(4, address.getState());
            sth.setString(5, address.getZip());
            sth.execute();
        } catch (SQLException e) {
            throw new DAOException(e);
        } finally {
            if (sth != null) {
                try { sth.close(); } catch (SQLException e) {}
            }
            if (conn != null) {
                try { conn.close(); } catch (SQLException e) {}
            }
        }
    }
```



```
    // ...

    public AddressTO find(int id) {
        Connection conn = null;
        PreparedStatement sth = null;
        try {
            conn = ds.getConnection();
            sth = conn.prepareStatement(FIND_SQL);
            sth.setInt(1, id);
            ResultSet rs = sth.executeQuery();
            if (rs.next()) {
                AddressTO address = new AddressTO();
                address.setId(id);
                address.setStreet(rs.getString(1));
                address.setCity(rs.getString(2));
                address.setState(rs.getString(3));
                address.setZip(rs.getString(4));
                return address;
            } else {
                return null;
            }
        } catch (SQLException e) {
            throw new DAOException(e);
        } finally {
            if (sth != null) {
                try { sth.close(); } catch (SQLException e) {}
            }
            if (conn != null) {
                try { conn.close(); } catch (SQLException e) {}
            }
        }
    }
}
```

One approach to conversion is to leave the transfer object as a non-persistent class while introducing a separate entity model. The DAO then converts back and forth between the two. Ideally the transfer object is replaced with the entity (see the following for an example of this approach), but preserving the transfer object allows developers to experiment with entities without disrupting the application in any way. Listing 13-17 demonstrates replacing the JDBC operations of a DAO with entities and an application-managed entity manager. Note the use of `joinTransaction()` in this example to ensure that the application-managed entity manager of the DAO class synchronizes itself with the active JTA transaction.

Listing 13-17. *DAO Using the Entity Manager for Persistence*

```
public class AddressDAO {
    private EntityManager em;

    public AddressDAO(EntityManager em) {
        this.em = em;
    }

    public void create(AddressTO address) {
        Address entity = createEntity(address);
        em.joinTransaction();
        em.persist(entity);
    }

    public void update(AddressTO address) {
        em.joinTransaction();
        em.merge(createEntity(address));
    }

    public void remove(int id) {
        em.joinTransaction();
        Address entity = em.find(Address.class, id);
        if (entity != null) {
            em.remove(entity);
        } else {
            throw new DAOException("No such address id: " + id);
        }
    }

    public AddressTO find(int id) {
        Address entity = em.find(Address.class, id);
        if (entity != null) {
            return createTO(entity);
        } else {
            return null;
        }
    }

    private Address createEntity(AddressTO address) {
        Address entity = new Address();
        entity.setId(address.getId());
        entity.setStreet(address.getStreet());
        entity.setCity(address.getCity());
        entity.setState(address.getState());
        entity.setZip(address.getZip());
        return entity;
    }
```



```
    private AddressTO createTO(Address entity) {
        AddressTO address = new AddressTO();
        address.setId(entity.getId());
        address.setStreet(entity.getStreet());
        address.setCity(entity.getCity());
        address.setState(entity.getState());
        address.setZip(entity.getZip());
        return address;
    }
}
```

The symmetry between the transfer object and entity operations suggests a simpler implementation. If the transfer object has been migrated to be an entity, then this data access object can be simplified one further time. Listing 13-18 shows the final result.

Listing 13-18. *DAO Returning Entities*

```
public class AddressDAO {
    private EntityManager em;

    public AddressDAO(EntityManager em) {
        this.em = em;
    }

    public void create(Address address) {
        em.joinTransaction();
        em.persist(address);
    }

    public void update(Address address) {
        em.joinTransaction();
        em.merge(address);
    }

    public void remove(int id) {
        em.joinTransaction();
        Address entity = em.find(Address.class, id);
        if (entity != null) {
            em.remove(entity);
        } else {
            throw new DAOException("No such address id: " + id);
        }
    }
```

```
    public Address find(int id) {
        return em.find(Address.class, id);
    }
}
```

Business Object

The *Business Object*[4] pattern describes application object models that are conceptual rather than physical in nature. If the physical domain model is too fine-grained, a more abstract domain model is sometimes introduced that more closely represents the object model derived from use case modeling. This secondary model reflects the conceptual business objects of the system rather than the domain objects of the system and delegates to the physical domain model in its implementation. Application code typically interacts only with the business objects.

It is this delegation to the physical domain model that makes business objects candidates for migration to the Java Persistence API. Business objects are not directly persistent; instead they persist state using entity beans, Data Access Objects, or other persistence mechanisms. The choice of persistence mechanism is hidden from clients and therefore potentially replaceable.

There are several strategies for dealing with business objects. If the business object depends on a pattern such as Data Access Object, the business object can be ignored, and the application is migrated by virtue of tackling the underlying persistence pattern. If the business object directly uses a persistence mechanism such as entity beans, then an opportunity exists to change the persistence mechanism and rethink the physical domain model.

The advanced object-relational mapping features of the Java Persistence API may make it possible to map the business objects directly to the database, effectively turning the business objects into entities. Caution must be used in these situations, as business objects tend to contain more business logic and focus more on business use cases than persistence. This is not to say that it cannot be done, but the resulting entities are unlikely to be as lightweight as would normally be expected with the Java Persistence API.

Fast Lane Reader

The *Fast Lane Reader*[5] pattern uses JDBC directly instead of using entity beans to query large amounts of data for presentation. The theory behind this pattern is that entity beans are too expensive to create if the only purpose for retrieving the entity is to read some value from it and then discard the instance.

The Fast Lane Reader pattern is more a combination of two other existing patterns than a unique pattern of its own. The DAO pattern is used to collect the data for presentation, and the Transfer Object pattern is used to present the results in a format suitable for rendering in the presentation layer. We mention it distinct from other patterns only because it is one of the few cases where both DAO and entity bean implementations exist in the same application returning the same set of transfer objects.

4. Ibid.
5. See `http://java.sun.com/blueprints/patterns/FastLaneReader.html` for more information.

The result of a query with the Fast Lane Reader is either a set of transfer objects or basic collections containing entity fields. Therefore we can take advantage of the Fast Lane Reader pattern both at the DAO and transfer object levels. If the DAO is returning fine-grained transfer objects, then we can apply the techniques we described earlier in the chapter to change the implementation of the DAO to use the Java Persistence API and ideally return entities instead of transfer objects. Likewise, if the Fast Lane Reader is returning basic collections, we can use projection queries to produce the same results with no additional effort required to translate the JDBC result set.

Active Record

The *Active Record*[6] pattern describes classes that manage their own persistence, typically implemented using JDBC. The advantage of this approach is that the classes are not outwardly tied to any persistence implementation. However, the internal coupling presents difficulties, as a Service Locator must be used to access the data source, and testing may be difficult if the persistence occurs automatically as a side effect of mutating operations.

At first glance, migrating active record classes sounds easy—just map them as entities and get to work. Unfortunately, to be useful, entities require application code to work with the entity manager for all persistence operations. This requires the entity manager to be available in all cases where persistence of the active record needs to occur.

The amount of refactoring required to introduce the entity manager depends on the number of places where persistence operations occur as a side effect of public method calls on the active record object. This may be obvious if the active record exposes insert, update, and delete methods, or more subtle if a store occurs after every setter invocation. Before attempting to convert these classes to entities, ensure that the application persistence strategy is well understood, refactoring to simplify it before conversion if necessary.

Summary

Migrating the persistence layer of an application from one technology to another is rarely a trivial task. The differences between EJB container-managed entity beans and the lightweight entities of the Java Persistence API could make the task of migration tricky. And yet, despite these challenges, it is possible not only to extract enough information out of entity beans to bootstrap an object-oriented domain model but also to leverage the same design patterns that made entity beans easier to work with as the very tool to replace them.

In our discussion of entity beans, we looked at how to use the existing bean as a template for the new entity, using the business interface, bean class, and XML descriptor of the entity bean in the process. We also looked at the home interface and how we can introduce stateless session beans to emulate the functions of the home interface with minimal impact to application code.

6. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2002.

We then touched on the migration of ORM and JDBC technologies to the Java Persistence API. While existing ORM migrations will largely depend on the support provided by the vendor, existing JDBC applications can be tackled by refactoring to existing Java EE design patterns before making the switch to the Java Persistence API.

Finally we looked at a catalog of Java EE design patterns related to persistence. Though not an exhaustive list, we looked at many of the major design patterns in use today and how they can be leveraged to safely introduce the Java Persistence API while minimizing the overall impact to the existing application.

APPENDIX

■ ■ ■

Quick Reference

This appendix serves as a reference to those who may not have access to online Javadoc or who just can't otherwise tear themselves away from this book. The mapping annotations and XML mapping elements are included in this reference to further assist using one or the other, or both. We have also included the complete set of interfaces defined by the API.

Metadata Reference

The annotation definitions are useful in order to see which Java program elements may be annotated by a given annotation and to see the names and types of the annotation elements. Some cursory XML information is included to indicate the attributes and subelements of the XML elements.

`@AttributeOverride`

Description: Overrides an inherited or embedded basic mapping

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AttributeOverride {
    String name();
    Column column();
}

@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AttributeOverrides {
    AttributeOverride[] value();
}
```

XML Element: `annotation-override`
XML Attributes: `name`
XML Subelements: `column`

@AssociationOverride

Description: Overrides an association mapping inherited from a mapped superclass

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AssociationOverride {
    String name();
    JoinColumn[] joinColumns();
}

@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface AssociationOverrides {
    AssociationOverride[] value();
}
```

XML Element: `association-override`
XML Attributes: `name`
XML Subelements: `join-column`

@Basic

Description: Simple mapped persistent attribute

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

XML Element: `basic`
XML Attributes: `name`, `fetch`, `optional`
XML Subelements: `column`, `lob`, `temporal`, `enumerated`

@Column

Description: Column to which entity attribute is mapped

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0;
    int scale() default 0;
}
```


XML Element: `column`
XML Attributes: `name`, `unique`, `nullable`, `insertable`, `updatable`, `column-definition`, `table`, `length`, `precision`, `scale`
XML Subelements: None

@ColumnResult

Description: Nested in `@SqlResultSetMapping` for mapping data projections

```
@Target({}) @Retention(RUNTIME)
public @interface ColumnResult {
    String name();
}
```

XML Element: `column-result`
XML Attributes: `name`
XML Subelements: None

@DiscriminatorColumn

Description: Column to store the entity type when a hierarchy of classes shares at least some of the same table storage

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface DiscriminatorColumn {
    String name() default "DTYPE";
    DiscriminatorType discriminatorType() default STRING;
    String columnDefinition() default "";
    int length() default 31;
}
```

XML Element: `discriminator-column`
XML Attributes: `name`, `discriminator-type`, `column-definition`, `length`
XML Subelements: None

@DiscriminatorValue

Description: The value used to represent a particular concrete entity class

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface DiscriminatorValue {
    String value();
}
```

XML Element: `discriminator-value`
XML Attributes: None
XML Subelements: None

@Embeddable

Description: Denotes an object that may be embedded in an entity

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Embeddable {}
```

XML Element: embeddable
XML Attributes: class, access, metadata-complete
XML Subelements: attributes

@Embedded

Description: Embedded object that is stored in the same table as the embedding entity

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Embedded {}
```

XML Element: embedded
XML Attributes: name
XML Subelements: attribute-override

@EmbeddedId

Description: Identifier attribute for compound primary key type stored in a single entity attribute

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface EmbeddedId {}
```

XML Element: embedded-id
XML Attributes: name
XML Subelements: attribute-override

@Entity

Description: Denotes object as being an entity

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}
```

XML Element: entity
XML Attributes: name, class, access, metadata-complete
XML Subelements: table, secondary-table, primary-key-join-column, id-class, inheritance, discriminator-value, discriminator-column, sequence-generator, table-generator, named-query, named-native-query, sql-result-set-mapping, exclude-default-listeners, exclude-superclass-listeners, entity-listeners, pre-persist, post-persist, pre-remove, post-remove, pre-update, post-update, post-load, attribute-override, association-override, attributes

@EntityListeners

Description: List of ordered entity listeners to be invoked on entity lifecycle events

```
@Target(TYPE) @Retention(RUNTIME)
public @interface EntityListeners {
    Class[] value();
}
```

XML Element: entity-listeners
XML Attributes: None
XML Subelements: entity-listener

@EntityResult

Description: Nested in @SqlResultSetMapping for mapping SQL results sets to entities

```
@Target({}) @Retention(RUNTIME)
public @interface EntityResult {
    Class entityClass();
    FieldResult[] fields() default {};
    String discriminatorColumn() default "";
}
```

XML Element: entity-result
XML Attributes: entity-class, discriminator-column
XML Subelements: field-result

@Enumerated

Description: Simple attribute that is an enumerated type

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Enumerated {
    EnumType value() default ORDINAL;
}
```

XML Element: enumerated
XML Attributes: None
XML Subelements: None

@ExcludeDefaultListeners

Description: Cause default listeners to not be invoked on specifying entity

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface ExcludeDefaultListeners {}
```

XML Element: exclude-default-listeners
XML Attributes: None
XML Subelements: None

@ExcludeSuperclassListeners

Description: Cause listener callbacks defined in superclasses to not be invoked on specifying entity

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface ExcludeSuperclassListeners {}
```

XML Element: `exclude-superclass-listeners`
XML Attributes: None
XML Subelements: None

@FieldResult

Description: Nested in `@EntityResult` for mapping SQL results sets to entity fields

```
@Target({}) @Retention(RUNTIME)
public @interface FieldResult {
    String name();
    String column();
}
```

XML Element: `field-result`
XML Attributes: `name`, `column`
XML Subelements: None

@GeneratedValue

Description: Identifier attribute that is generated automatically

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface GeneratedValue {
    GenerationType strategy() default AUTO;
    String generator() default "";
}
```

XML Element: `generated-value`
XML Attributes: `strategy`, `generator`
XML Subelements: None

@Id

Description: Identifier attribute for simple primary key type

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id {}
```

XML Element: `id`
XML Attributes: `name`
XML Subelements: `column`, `generated-value`, `temporal`, `table-generator`, `sequence-generator`

@IdClass

Description: Compound primary key class spread across multiple entity attributes

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface IdClass {
    Class value();
}
```

XML Element: `id-class`
XML Attributes: `class`
XML Subelements: None

@Inheritance

Description: Denotes an entity inheritance hierarchy

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Inheritance {
    InheritanceType strategy() default SINGLE_TABLE;
}
```

XML Element: `inheritance`
XML Attributes: `strategy`
XML Subelements: None

@JoinColumn

Description: Foreign key column that references the primary key of another entity

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
}

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinColumns {
    JoinColumn[] value();
}
```

XML Element: `join-column`
XML Attributes: `name`, `referenced-column-name`, `unique`, `nullable`, `insertable`, `updatable`, `column-definition`, `table`
XML Subelements: None

@JoinTable

Description: Association table to join two entity types in a many-valued relationship

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}
```

XML Element: `join-table`
XML Attributes: `name`, `catalog`, `schema`
XML Subelements: `join-column`, `inverse-join-column`, `unique-constraint`

@Lob

Description: Simple attribute that is mapped to a large object (LOB) column

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Lob {}
```

XML Element: `lob`
XML Attributes: None
XML Subelements: None

@ManyToMany

Description: Many-to-many association to another entity type

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

XML Element: `many-to-many`
XML Attributes: `name`, `target-entity`, `fetch`, `mapped-by`
XML Subelements: `order-by`, `map-key`, `join-table`, `cascade`

@ManyToOne

Description: Many-to-one association to another entity type


```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

XML Element: `many-to-one`
XML Attributes: `name`, `target-entity`, `fetch`, `mapped-by`
XML Subelements: `join-column`, `join-table`, `cascade`

@MapKey

Description: Entity attribute to act as the key value when storing target entities in a `Map`

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface MapKey {
    String name() default "";
}
```

XML Element: `map-key`
XML Attributes: `name`
XML Subelements: None

@MappedSuperclass

Description: Denotes an entity superclass that may contain mapped persistent state

```
@Target(TYPE) @Retention(RUNTIME)
public @interface MappedSuperclass {}
```

XML Element: `mapped-superclass`
XML Attributes: `class`, `access`, `metadata-complete`
XML Subelements: `id-class`, `exclude-default-listeners`, `exclude-superclass-listeners`, `entity-listeners`, `pre-persist`, `post-persist`, `pre-remove`, `post-remove`, `pre-update`, `post-update`, `post-load`, `attributes`

@NamedNativeQuery

Description: Defines a static query that uses SQL query criteria

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedNativeQuery {
    String name();
    String query();
    QueryHint[] hints() default {};
    Class resultClass() default void.class;
    String resultSetMapping() default "";
}
```

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedNativeQueries {
    NamedNativeQuery[] value ();
}
```

XML Element: named-native-query
XML Attributes: name, result-class, result-set-mapping
XML Subelements: query, hint

@NamedQuery

Description: Defines a static query that uses JPQL query criteria

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedQuery {
    String name();
    String query();
    QueryHint[] hints() default {};
}

@Target({TYPE}) @Retention(RUNTIME)
public @interface NamedQueries {
    NamedQuery[] value ();
}
```

XML Element: named-query
XML Attributes: name
XML Subelements: query, hint

@OneToMany

Description: One-to-many association to another entity type

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToMany {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

XML Element: one-to-many
XML Attributes: name, target-entity, fetch, mapped-by
XML Subelements: order-by, map-key, join-table, join-column, cascade

@OneToOne

Description: One-to-one association to another entity type

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

XML Element: one-to-one
XML Attributes: name, target-entity, fetch, mapped-by
XML Subelements: order-by, map-key, join-table, join-column, cascade

@OrderBy

Description: Entity attribute or attributes to order by when storing target entities in a `List`

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OrderBy {
    String value() default "";
}
```

XML Element: order-by
XML Attributes: None
XML Subelements: None

@PostLoad

Description: Method invoked by provider after loading an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostLoad {}
```

XML Element: post-load
XML Attributes: method-name
XML Subelements: None

@PostPersist

Description: Method invoked by provider after persisting an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostPersist {}
```

XML Element: post-persist
XML Attributes: method-name
XML Subelements: None

@PostRemove

Description: Method invoked by provider after removing an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostRemove {}
```

XML Element: post-remove
XML Attributes: method-name
XML Subelements: None

@PostUpdate

Description: Method invoked by provider after updating an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PostUpdate {}
```

XML Element: post-update
XML Attributes: method-name
XML Subelements: None

@PrePersist

Description: Method invoked by provider before persisting an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PrePersist {}
```

XML Element: pre-persist
XML Attributes: method-name
XML Subelements: None

@PreRemove

Description: Method invoked by provider before removing an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PreRemove {}
```

XML Element: pre-remove
XML Attributes: method-name
XML Subelements: None

@PreUpdate

Description: Method invoked by provider before updating an entity

```
@Target({METHOD}) @Retention(RUNTIME)
public @interface PreUpdate {}
```

XML Element: pre-update
XML Attributes: method-name
XML Subelements: None

@PrimaryKeyJoinColumn

Description: Foreign key column that is also a primary key

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PrimaryKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    String columnDefinition() default "";
}

@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface PrimaryKeyJoinColumns {
    PrimaryKeyJoinColumn[] value();
}
```

XML Element: primary-key-join-column
XML Attributes: name, referenced-column-name, column-definition
XML Subelements: None

@QueryHint

Description: Nested in @NamedQuery and @NamedNativeQuery for vendor-specific behavior

```
@Target({}) @Retention(RUNTIME)
public @interface QueryHint {
    String name();
    String value();
}
```

XML Element: query-hint
XML Attributes: name, value
XML Subelements: None

@SecondaryTable

Description: Additional table or tables in which to store part of the entity state

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTable {
    String name();
    String catalog() default "";
    String schema() default "";
    PrimaryKeyJoinColumn[] pkJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
}

@Target({TYPE}) @Retention(RUNTIME)
public @interface SecondaryTables {
    SecondaryTable[] value();
}
```

XML Element: secondary-table
XML Attributes: name, catalog, schema
XML Subelements: primary-key-join-column, unique-constraint

@SequenceGenerator

Description: Specifies a database sequence used for primary key generation

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface SequenceGenerator {
    String name();
    String sequenceName() default "";
    int initialValue() default 1;
    int allocationSize() default 50;
}
```

XML Element: sequence-generator
XML Attributes: name, sequence-name, initial-value, allocation-size
XML Subelements: None

@SqlResultSetMapping

Description: Defines a mapping from a JDBC result set

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface SqlResultSetMapping {
    String name();
    EntityResult[] entities() default {};
    ColumnResult[] columns() default {};
}

@Target({TYPE}) @Retention(RUNTIME)
public @interface SqlResultSetMappings {
    SqlResultSetMapping[] value();
}
```

XML Element: sql-result-set-mapping
XML Attributes: name
XML Subelements: entity-result, column-result

@Table

Description: Primary table in which state of a given entity type is stored

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
}
```


XML Element: table
XML Attributes: name, catalog, schema
XML Subelements: unique-constraint

@TableGenerator

Description: Specifies a table used for primary key generation

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface TableGenerator {
    String name();
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default {};
}
```

XML Element: table-generator
XML Attributes: name, table, catalog, schema, pk-column-name, value-column-name, pk-column-value, initial-value, allocation-size
XML Subelements: unique-constraint

@Temporal

Description: Simple attribute that is a time-based type

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Temporal {
    TemporalType value();
}
```

XML Element: temporal
XML Attributes: None
XML Subelements: None

@Transient

Description: Non-persistent attribute

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Transient {}
```

XML Element: transient
XML Attributes: name
XML Subelements: None

@UniqueConstraint

Description: Nested in `@Table` and `@SecondaryTable` to specify columns having uniqueness constraints

```
@Target({}) @Retention(RUNTIME)
public @interface UniqueConstraint {
    String[] columnNames();
}
```

XML Element: `unique-constraint`
XML Attributes: None
XML Subelements: `column-name`

@Version

Description: Simple attribute that stores optimistic locking version

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Version {}
```

XML Element: `version`
XML Attributes: `name`
XML Subelements: `column`, `temporal`

Enumerated Types

The following enumerated types are used in the annotations and API.

```
public enum GenerationType { TABLE, SEQUENCE, IDENTITY, AUTO };

public enum FetchType { LAZY, EAGER };

public enum TemporalType { DATE, TIME, TIMESTAMP };

public enum EnumType { ORDINAL, STRING };

public enum CascadeType { ALL, PERSIST, MERGE, REMOVE, REFRESH };

public enum InheritanceType { SINGLE_TABLE, JOINED, TABLE_PER_CLASS };

public enum DiscriminatorType { STRING, CHAR, INTEGER };
```

Mapping File-Level Metadata Reference

The XML elements in this section are default settings that apply only to the mapping file in which they are specified as subelements of the `entity-mappings` element. There are no corresponding annotations that apply to the same mapping file scope.

package

Description: Default package name for classes named in the XML mapping file
XML Attributes: None
XML Subelements: None

schema

Description: Default schema for tables of entities listed in the mapping file
XML Attributes: None
XML Subelements: None

catalog

Description: Default catalog for tables of entities listed in the mapping file
XML Attributes: None
XML Subelements: None

access

Description: Default access mode for managed classes in the mapping file for which there are no annotations specified
XML Attributes: None
XML Subelements: None

Persistence-Unit-Level Metadata Reference

The XML elements in this section are default settings that apply to the entire persistence unit. There are no corresponding annotations that apply to the same persistence-unit-level scope.

persistence-unit-metadata

Description: Top-level element for specifying mapping configuration for the entire persistence unit
XML Attributes: None
XML Subelements: `xml-mapping-metadata-complete`, `persistence-unit-defaults`

xml-mapping-metadata-complete

Description: Setting to indicate that all annotation metadata in the persistence unit is to be ignored
XML Attributes: None
XML Subelements: None

persistence-unit-defaults

Description: Parent element for default persistence unit settings
XML Attributes: None
XML Subelements: `schema`, `catalog`, `access`, `cascade-persist`, `entity-listeners`

schema

Description: Default schema for tables in the persistence unit for which there is no schema specified
XML Attributes: None
XML Subelements: None

catalog

Description: Default catalog for tables in the persistence unit for which there is no catalog specified
XML Attributes: None
XML Subelements: None

access

Description: Default access mode for managed classes in the persistence unit for which there are no annotations specified
XML Attributes: None
XML Subelements: None

cascade-persist

Description: Adds `PERSIST` cascade option to all relationships in the persistence unit
XML Attributes: None
XML Subelements: None

entity-listeners

Description: List of entity listeners to be invoked on every entity in the persistence unit
XML Attributes: None
XML Subelements: `entity-listener`

EntityManager Interface

The `EntityManager` interface is implemented by the persistence provider and proxied by the container in the server. Concrete instances are obtained from an `EntityManagerFactory` or from the container through injection or JNDI lookup.

```
public interface javax.persistence.EntityManager {

    public void persist(Object entity);

    public <T> T merge(T entity);

    public void remove(Object entity);

    public <T> T find(Class<T> entityClass,
                      Object primaryKey);

    public <T> T getReference(Class<T> entityClass,
                              Object primaryKey);

    public void flush();

    public void setFlushMode(FlushModeType flushMode);

    public FlushModeType getFlushMode();

    public void lock(Object entity, LockModeType lockMode);

    public void refresh(Object entity);

    public void clear();

    public boolean contains(Object entity);

    public Query createQuery(String qlString);

    public Query createNamedQuery(String name);

    public Query createNativeQuery(String sqlString);

    public Query createNativeQuery(String sqlString,
                                   Class resultClass);

    public Query createNativeQuery(String sqlString,
                                   String resultSetMapping);

    public void joinTransaction();

    public Object getDelegate();

    public void close();

    public boolean isOpen();

    public EntityTransaction getTransaction();
}
```

Query Interface

The Query interface is implemented by the persistence provider. Concrete instances are obtained from the EntityManager query factory methods.

```
public interface javax.persistence.Query {

    public List getResultList();

    public Object getSingleResult();

    public int executeUpdate();

    public Query setMaxResults(int maxResult);

    public Query setFirstResult(int startPosition);

    public Query setHint(String hintName, Object value);

    public Query setParameter(String name, Object value);

    public Query setParameter(String name,
                              Date value,
                              TemporalType temporalType);

    public Query setParameter(String name,
                              Calendar value,
                              TemporalType temporalType);

    public Query setParameter(int position, Object value);

    public Query setParameter(int position,
                              Date value,
                              TemporalType temporalType);

    public Query setParameter(int position,
                              Calendar value,
                              TemporalType temporalType);

    public Query setFlushMode(FlushModeType flushMode);
}
```


EntityManagerFactory Interface

The EntityManagerFactory interface is implemented by the persistence provider. Concrete instances are obtained from the Persistence bootstrap class or from the container through injection or JNDI lookup.

```
public interface javax.persistence.EntityManagerFactory {

    public EntityManager createEntityManager();

    public EntityManager createEntityManager(Map map);

    public void close();

    public boolean isOpen();
}
```

EntityTransaction Interface

The EntityTransaction interface is implemented by the persistence provider. Concrete instances are obtained from the EntityManager.getTransaction() method.

```
public interface javax.persistence.EntityTransaction {

    public void begin();

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public boolean getRollbackOnly();

    public boolean isActive();
}
```

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W