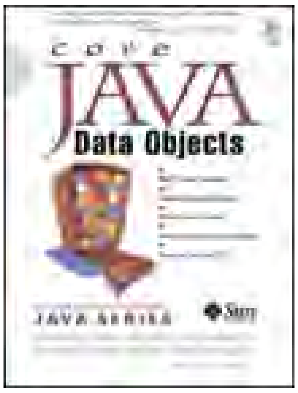


Core Java™ Data Objects

By Sameer Tyagi, Keiron McCammon, Michael Vorburger, Heiko Bobzin

Start Reading

Publisher: Prentice Hall PTR
Pub Date: September 11, 2003
ISBN: 0-13-140731-7
Pages: 576

The experienced Java developer's guide to persistence with JDO!

Master JDO, the breakthrough technology for persistenting Java objects!

Java Data Objects (JDO) streamlines development by providing Java-centric mechanisms for making objects persistent, and standard APIs for connecting application code with underlying persistent datastores. *Core Java Data Objects* is the definitive, comprehensive guide to JDO persistence for every experienced Java developer.

Using realistic code examples, this book's expert authors cover creating, reading, updating, and deleting persistent objects; object lifecycles and state transitions; JDO classes and APIs; queries, architecture, security, and much more. They show how to integrate JDO with EJB, JTA, JCA, and other J2EE technologies; and introduce best practices for using JDO in both standalone programs and J2EE components.

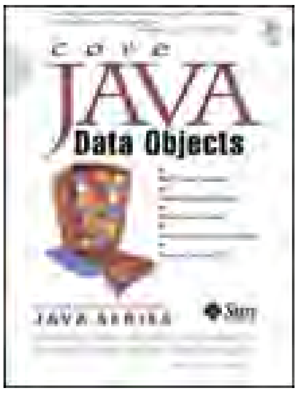
If you want to spend more time solving business problems and less time worrying about persistence, you need JDO. And you need the one book that'll help you make the most of JDO: *Core Java Data Objects*.

Every Core Series book:

- DEMONSTRATES how to write commercial quality code
- FEATURES nontrivial programs and examples--no toy code!
- FOCUSES on the features and functions most important to real developers
- PROVIDES objective, unbiased coverage of cutting-edge technologies - no hype!

Core Java Data Objects delivers:

- Practical examples showing how JDO can be applied to existing applications
- Powerful insights for using JDO to architect new systems more effectively
- Step-by-step guidance for integrating JDO with other J2EE technologies
- Best practices for using JDO in real-world business environments



• Table of Contents

Core Java™ Data Objects

By Sameer Tyagi, Keiron McCammon, Michael Vorburger, Heiko Bobzin

Publisher: Prentice Hall PTR
Pub Date: September 11, 2003
ISBN: 0-13-140731-7
Pages: 576

Copyright

Editorial/production supervision: *Mary Sudul*

Composition: *FASTpages*

Cover design director: *Jerry Votta*

Cover designer: *Anthony Gemmellaro*

Cover illustration: *Karen Strelecki*

Manufacturing manager: *Alexis R. Heydt-Long*

Acquisitions editor: *Gregory G. Doench*

Sun Microsystems Press publisher: *Myrna Rivera*

First Printing

Sun Microsystems Press

A Prentice Hall Title

Praise for *Core JAVA Data objects*

"Core JDO is packed with valuable information and provides a strong foundation for building JDO-based Enterprise applications."

—Dan Malks, Author, Core J2EE Patterns

"All Java developers working on business applications should understand the implications of JDO. This book helps them to do just that."

—Scott W. Ambler, Author, Agile Database Techniques

"Core JDO is a must read book for those interested in Java Data Objects.The authors provide a solid introduction to the JDO standard but then focus on how to use JDO in the real world, reflecting their experiences as members of the spec team as well as application developers."

—Neelan Choksi, President, SolarMetric

"Core JDO probes the innards of JDO and explains everything lucidly, from the fundamentals to its most advanced features (using JDO with EJB, JCA, Security, Transactions, etc). The book stands out as an excellent JDO resource for both Architects and Developers alike. Congratulations and thanks to the Authors, Sun, and Prentice-Hall for bringing out such a great book on JDO."

—Gopalan Suresh Raj, Chief Architect, hywy Software Corporation

"Core JDO is a clear and detailed treatment of JDO. The JDO and EJB chapter is essential reading for anyone involved in Enterprise Java development."

—David Tinker, Hemisphere Technologies

"The Core JDO Book provides the reader exceptional in-depth information about JDO. What I like in particular is that the book doesn't stop by just explaining the technology in a vaccuum. It provides plentiful guidance how to use it in real world application environments such as J2EE, it includes a very helpful comparison with JDBC and helps the reader to make a more informed decision on his application database architecture, and almost all topics are accompanied with extensive code examples that help learning JDO the easy way."

—Dirk Bartels, General Manager, JDOcentral.com

"When JBoss Group looked at the available APIs for orthogonal persistence, JDO emerged as a finally mature standard. It was clear we needed to implement JDO in JBoss. This book is very well written, with a balanced mix between specification coverage and practical tidbits. A must read."

—Marc Fleury, President and Founder, JBoss Group, LLC.

"As a JDO vendor we are very keen on this book.This is a book that Java architects and programmers can use to save weeks when they want to build and deploy critical J2EE transactional applications."

—Eric Samson, President, LIBeLIS

Foreword

When I first heard about Java Data Objects' (JDO) development back in 2000, I was absolutely thrilled. I was thrilled that the Java industry would finally have a standard and easy way to transparently persist object models. I saw JDO not just as a tool to solve the problem of persistence but also as a catalyst that would encourage Java projects to adopt object-oriented development paradigms.

I began my own career in enterprise development in 1998. Prior to that, I read some great books that set my first impressions of how large-scale systems should be designed. Reading books like Peter Coad's *Java Design*, Scott Ambler's *Building Object Applications That Work*, and then being lucky enough to have friends who were from a Smalltalk and Gemstone background, convinced me that building object models was the best and most accepted way to build enterprise systems.

But, when I started actually working on enterprise projects and saw what people were saying on TheServerSide.com, I was shocked. Most J2EE developers were not building object-oriented systems and there wasn't even an easy way for them to do so if they wanted to. The only alternatives were to build your own persistence layer or lock yourself into a proprietary O/R mapping product or object database. Building your own persistence layer was a very complex and poorly understood subject—there was virtually no published material on this subject. Entity Beans were an option if you were working with an application server, but entity beans are a component persistence technology that isn't suitable for persisting complex, fine-grained object models. There really weren't that many options, which is why I was a very strong defender of entity beans because it was the only standard in the java universe that addressed persistence in somewhat of a transparent way.

However, when JDO was announced, I knew that this technology would enable developers to easily write their applications to object-oriented paradigms, allowing them to complete their projects faster, make their applications more maintainable, and overall increase the quality of Java software projects in the industry as a whole.

The possibilities that JDO represented were just too attractive for the community to ignore, which is why JDO has been so popular at the grassroots level. Today, JDO has significant industry support. There are now dozens of thriving commercial products and open source projects that support JDO and transparent persistence. Microsoft decided to include the JDO-like ObjectSpaces technology in the .NET 2004 framework. JBoss 4 is expected to be the first application server with transparent persistence (and JDO) built right into it. JDO has become a viable option to enable transparent persistence and is continuing to grow in importance.

However, the development practices being applied in the industry are still behind. The lack of a standard way to transparently persist object models has been a barrier to the maturity of J2EE development practices. This is why JDO is so important, and why this book is so important.

Core Java Data Objects is a great book and it represents the combined expertise of a group of very qualified authors (consisting of JDO expert group members and JDO users) as well as thousands of J2EE professionals and JDO enthusiasts on TheServerSide.com, where the book's early chapter drafts were publicly reviewed. It is difficult to find such book as this that was carefully crafted and tuned by so many people. For developers looking to understand and apply JDO technology, this book is a must have.

Floyd Marinescu
Director of TheServerSide.com & Symposium
Author, *EJB Design Patterns*

Preface

The need for persistence is a prerequisite for any enterprise application, and the case is no different for applications written in the Java programming language. A chunk of any typical Java application consists of code that is responsible for storing and retrieving the application-specific data to some type of storage facility so that it can be retrieved and used at a later time.

Java is an object-oriented language, and developers inherently work with objects that, at any time, have states represented by member fields. From the developers perspective, persistence focuses on taking the application-specific information assets, which in fact is just state in these objects, and making them available beyond the lifetime of the virtual machine process. In short, data should be available once the Java virtual machine has exited.

The realization of this notion by developers can take many shapes: extracting and storing the state in a file or in a database using JDBC, using Java serialization and storing the binary representation of the entire object, or the more common enterprise approach of leveraging the facilities of an application server via EJBs to name a few. Irrespective of the technique and the varying degree of complexity involved, each of these approaches has one common conceptual characteristic: the dependence on the developer to *extract* state information from the objects when it needs to be saved and *re-create* that state in the object representation when needed by the application.

Now there is a new alternative—Java Data Objects (JDO)—that is well placed to change the way developers have traditionally perceived persistence in their applications. JDO is a standard developed under the auspices of the Java Community Process by a number of industry participants. The JDO specifications were the realization of an effort to satisfy the need for a transparent persistence mechanism in Java that would satisfy two orthogonal visions at the same time. One aimed to provide a standard programming interface between application code and the underlying persistent stores like databases and file systems in a manner similar to JDBC; the other aimed to simplify secure and scalable application development by providing a Java-centric mechanism that would allow the manipulation and traversal of persistent objects. JDO satisfies, in fact excels, at both. With JDO, developers can concentrate on designing applications around the intended purpose—to solve business problems—rather than around the mechanics of persistence infrastructures.

JDO is fast gaining momentum and acceptance in the community as the preferable means to introduce persistence in Java applications. Developers are learning to appreciate the simplicity and power of this technology, and architects are pleased at the reduced development time. In this book, we not only guide the reader in understanding JDO itself, but help readers learn how to apply this technology effectively in real-world applications.

Audience

JDO is a new technology, but it is one that holds great promise. If you are an architect, a developer, or even a manager who is involved with persistence in Java and would like to gain an insight into this technology, then this book is for you.

The driving goal of this author team is to combine their wealth of experience into a single accessible and unified resource and give the community a definitive and authoritative source to turn to for JDO technology. The team strives to provide leading edge and practical examples that illustrate how JDO can be applied to existing applications and new architecture initiatives so that readers can have a clear perspective of this new technology. We hope that professionals using Java in any measure, from simple programs to complex enterprise applications, find this book valuable. Where additional detail is required, this book provides the stepping-stone and points out additional sources of information.

In keeping with the vision of Java itself, this book attempts to stay vendor- and platform-neutral. Possible implementation strategies may be discussed occasionally, but the goal of the author team is not to discuss intricate details, recommend, or push any specific vendor product.

What This Book Is About

This book is about:

- Everything JDO, in that the book provides an in-depth explanation about the technology
- Applying JDO in applications ranging from standalone programs to J2EE components
- The best practices in designing and developing applications that use JDO

Part One—Introduction

Chapter 1: JDO Overview

Java Data Objects is a new standard for persistence in Java applications. JDO is well placed to change the fundamental way in which architects and developers approach the design and development of persistence in Java applications. This chapter gives you an insight into the concepts around which JDO is centered.

Chapter 2: Object Persistence Fundamentals

Chapter 2 looks at what's technically behind object persistence. It explains and compares different ways to implement persistence frameworks. It takes a broad view of the JDBC-based object-relational mapping solutions and outlines the effort of implementing such a framework from scratch.

Part Two—The Details

Chapter 3: Getting Started with JDO

This chapter explains the basics of how JDO works and how to use JDO to create, read, update, and delete persistent objects. It introduces some of the advanced concepts behind JDO and aims to provide sufficient understanding to allow a Java programmer to develop an application using JDO.

Chapter 4: Lifecycles

This chapter discusses the lifecycle of objects and the reasons why state changes can be meaningful to the application developer. A closer look is taken at mandatory and optional states of instances, the methods to gather information about states, and methods that lead to state transitions.

Chapter 5: Developing with JDO

This chapter looks at fundamental JDO concepts and all the classes and APIs defined by the JDO specification. It explains each of the methods for interfaces like the PersistenceManager, the JDOHelper, and I18NHelper classes, overviews JDO exception classes, and the classes and interfaces in the service provider interface package.

Chapter 6: Finding Your Data

This chapter discusses the query facility provided by JDO specification. The techniques surrounding the retrieval and querying of the underlying datatstore using the API and the JDOQL (JDO Query Language) are discussed with detailed examples.

Chapter 7: Architecture Scenarios

This chapter outlines the different types of application architectures that can be used with JDO. It covers when to use JDO versus JDBC, different types of datastores that can be used with JDO, using JDO with multi-tier applications, and using multi-threaded programming with JDO.

Part Three—J2EE

Chapter 8: JDO and the J2EE Connector Architecture

This chapter explains how JDO can be used in conjunction with J2EE via the J2EE Connector Architecture (JCA). After a brief introduction to JCA, this chapter looks at how JCA fits together with JDO and how they can be used together to build J2EE applications.

Chapter 9: JDO and Enterprise JavaBeans (EJB)

This chapter looks at the Enterprise JavaBeans (EJB) component model and its relationship to JDO. It addresses various implementation topics and provides extensive source code examples. The goal of this chapter is not to simply show 'how' to code EJB and JDO together, but also to help you decide if the combination of these two APIs makes sense for you.

Chapter 10: Security

This chapter defines the different levels of security, their requirements, and their objectives. It looks at the Reference Enhancer, and examines the rules and measures defined by JDO to secure an application. Finally, the chapter looks at application-level security and how J2EE and JDO work together regarding security.

Chapter 11: Transactions

This chapter looks at some fundamental concepts around transactions in Java, the transaction capabilities included in JDO, and how they can be used in standalone applications and in conjunction with Java Transaction APIs (JTA) in managed environments. It also outlines design considerations and practices when using JDO in a transactional manner.

Part Four—The Conclusion

Chapter 12—JDO and Java Database Connectivity (JDBC)

This chapter reviews the JDBC API, gives a brief overview of relational databases, looks at the history of JDBC, and provides an introduction to the new JDBC 3.0 API. The second part of the chapter compares JDBC with JDO, discusses features and differences of each, and highlights common misconceptions.

Chapter 13: Best Practices

This chapter provides some suggestions, usage guidelines, and best practices for using JDO successfully in typical real-world enterprise business applications.

Chapter 14: Current Status and the Road Ahead

This chapter provides the reader with a possible roadmap and the direction that JDO is likely to take. Features covered in this chapter are in no way guaranteed to make it into JDO 2.0, nor do they represent the limit of new features that may be considered. Rather, this chapter aims to give some insight into what may appear in future revisions of the JDO specification.

Chapter 15: Case Study

This chapter discusses a complete working example of the "Core JDO Library" to put come context around material covered in this book and to give the reader an insight into how applications can be developed with JDO. The "Core JDO Library" application is an implementation of a couple of use cases to explain basic and advanced JDO features. It is also meant to explain concepts mentioned in the previous chapters such as two-tier and n-tier applications. The chapter is divided into smaller sections that can be used as cook-book style recipes.

Appendix A: JDO States

Appendix B: JDO Metadata XML DTD

Appendix C: JDOQL BNF

Appendix D: PersistenceManagerFactory Quick Reference

Appendix E: JDO Vendors

What This Book Is Not About

This book is not about:

- Learning Java: This book is Java-focused and assumes that the reader is familiar with the Java programming language. This book is not a tutorial on Java or data structures in Java, nor is it a guide on how to implement different programming algorithms in Java.
- Object databases: This book is about JDO, and though the specifications may be drawn from the wealth of experience and knowledge in the Object Data Management Group, this book is not about the Object Data Management Group (ODMG) specifications or about using object databases.
- Development processes and methodology: Although UML notation is used where appropriate, this book does not suggest any particular programming technique or methodology to follow in Java projects.

Related Information

The J2EE technologies cited in this book are described in their specifications:

- Java™ Data Objects Specification, Version 1.0. Copyright 2001, 2002 Sun Microsystems, Inc. Available at http://www.jcp.org/jsr/detail/12.jsp.
- Java™ 2 Platform, Enterprise Edition Specification, Version 1.2 (J2EE specification). Copyright 1999, Sun Microsystems, Inc. Available at http://java.sun.com/j2ee/download.html.
- Java™ 2 Platform, Standard Edition, Version 1.2.2 (J2SE specification). Copyright 1993-99, Sun Microsystems, Inc. Available at http://java.sun.com/products/jdk/1.2/docs/api/index.html.
- Java™ Servlet Specification, Version 2.2 (Servlet specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/servlet.
- JavaServer Pages™ Specification, Version 2.1 (JSP specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/jsp.
- Enterprise JavaBeans™ Specification, Version 2.0 (EJB specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/ejb.
- JDBC™ 2.0 API (JDBC specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/jdbc.
- JDBC™ 2.0 Standard Extension API (JDBC extension specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/jdbc.
- Java™ Transaction API, Version 1.0.1 (JTA specification). Copyright 1998,1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/jta.
- Java™ Transaction Service, Version 0.95 (JTS specification). Copyright 1997-1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/jts.
- Java Naming and Directory Interface™, Version 1.2 (JNDI specification).Copyright 1998, 1999, Sun Microsystems, Inc. Available at http://java.sun.com/products/jndi.
- Java IDL. Copyright 1993-99, Sun Microsystems, Inc. Available at http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html.

Additionally, the following resources may be helpful to the readers:

- JDO homepage: http://access1.sun.com/jdo
- JDO Community homepage: http://www.jdocentral.com
- JDO-INTEREST mailing list: http://archives.java.sun.com
- Homepage for this book http://www.corejdo.com

Conventions Used

This book has four distinct authors. In an attempt to remain consistent throughout the book, the authors have chosen to adopt the following conventions regarding terminology:

- **Persistence capable class:** This term is used to indicate a class that implements the PersistenceCapable interface specified in the JDO API.
- **Persistent object:** This term indicates an instance of a persistence capable class that is persistent.
- **Java object:** This term indicates the general concept of a Java object instance. The object may or may not follow the standard JavaBeans conventions.
- **Transient instance:** This term is indicates an instance of any class.
- **Field:** The term is used to refer to the instance variables of a class. Rather than calling them *members, member variables,* or *attributes,* they are generically referred to as *fields.*
- **Datastore:** This term is used to refer to the persistent mechanism used by a JDO implementation. The terms *datastore* and *underlying datastore* are used interchangeably. The term *database* is used only in the context of talking specifically about a relational database or object database.
- **JDO implementation:** Rather than *JDO runtime* or *JDO vendor*, the term *JDO implementation* refers to an implementation of the JDO specification.

About the Authors

Sameer Tyagi

Sameer Tyagi is employed as an Enterprise Java Architect for Sun Microsystems and works at the Java Center in Burlington, MA. He has eight years of experience in information technology and has co-authored many books in different areas of Java technology. He also writes regularly for *Java Pro*, *Java World*, and *Java Developers Journal*. Sameer can be reached at s.t@sun.com.

Keiron McCammon

Keiron is the CTO for Versant Corporation. He has worked in the IT industry for more than ten years, principally applying object-oriented technologies and techniques for solving business problems. He has hands-on experience in C/C++/Java as well as Corba/J2EE. As the CTO of Versant, Keiron is responsible for defining the product strategy and direction. He has worked with strategic customers in the financial and telecommunications arenas, aiding in the development of distributed object solutions utilizing Java, J2EE, and Versant technologies. Keiron is a member of the JDO expert group (JSR 12) and presents regularly at Java and J2EE conferences and seminars on JDO and J2EE application architecture and development.

Michael Vorburger

Michael currently holds the title of Chief Software Architect at Vertical*I, an enterprise software company focusing on providing solutions to the life sciences industries. Following early drafts of the JDO standard, he was inspired to implement a similarly architected in-house persistence service, which is currently being re-implemented based on standard JDO. He has been giving feedback and exchanging questions on the JDO expert mailing list.

Heiko Bobzin

Heiko Bobzin is a member of the JDO expert group (JSR 12). He led the JDO and ODMG 3.0 implementation at Poet Software, a vendor of object-oriented databases. Prior to joining Poet, he developed a universal communications library including Java, Delphi, VB, and C++ bindings, a TCP stack, ISDN drivers, fax server, and ported the GNU C++ compiler to Microware's OS-9. He presented at Java One 2000, ICJD San Jose, New York in 2001, and many other conferences in the United States and Germany. You can reach him at corejdo@bobzin.com.

About the Reviewers

The contents of the Core JDO book were made available to the general public for the purposes of review. All material was posted at http://www.theserverside.com to allow the community to participate in the review and editing of the content. Many comments and suggestions have made it into the final manuscript. In other words, this book has followed a sort of "open source approach" for an open technology.

The authors would like to thank all the people who helped in the reviewing process and provided their valuable feedback, in particular Ken Egervari, Chin Nelson, Christian Romberg, David Jencks, Jakimovski Robert, Justin Walsh, John Broglio, James McGovern, Neelan Choksi, and Jean-Louis Marechaux.

The authors would also like to thank all the people at theserverside.com, in particular Nitin Bharti and Floyd Marinescu for hosting the chapters and facilitating the review process.

Acknowledgments

Sameer Tyagi

I would like to express my appreciation and gratitude to my parents and family for their love and continuing support, without which none of this would have been possible in the first place.

Keiron McCammon

I would like to acknowledge my co-workers who helped in reviewing my early drafts, Sven Erik Knop, Gerhard Klein, Martin Wessel, Per Bergman, Nelson Chin and Patrick Guiltot. Thank you to Ken Egervari and David Jencks for your reviews and thank you to Floyd Marinescu of theServerSide.com for hosting the book chapters for public review and thank you to all theServerSide.com members who contributed their feedback. Most importantly, thank you to my wife, Kerry, for supporting me in everything I do.

Michael Vorburger

Michael most importantly wishes to thank his wife Divvya, for her love, support and inspiration. He dedicates this book to their son Dév Raj-Joe, who was born shortly before these pages went to press. He also wishes to thank his parents, Hannah and Joseph, for making him who he is. Finally he is also very grateful to Geeta Rajagopalan for all her help.

Heiko Bobzin

My appreciation goes to Dirk Bartels, Andreas Renner, Michael Damkier, and the Java team at Poet. Dirk Bartels, co-founder and former president, encouraged me to join the Java Data Objects Group in 1999 and to write this book. Andreas and Michael spent many hours working with me on JDO concepts and e-mails. Finally, thanks to my wife for her patience and support.

Part 1: Introduction

Chapter 1. JDO Overview

"It is the beginning of wisdom to be able to say 'I do not know.'"

— Chinese proverb

Java Data Objects is not just another specification that has been put together by the community process: It is a new standard for persistence in Java applications. JDO is well placed to change the fundamental way in which architects and developers have approached the design and development of persistence in Java applications. This chapter gives you an insight into the concepts around which JDO is centered.

1.1 JDO Background

The wheels for an object persistence specification were put into motion in June 1999 under the Java Community Process (JCP) with both individuals and corporations contributing in the development. The first version of the JDO specification, the toolkit, and a reference implementation were released in April 2002.

Already, a dozen or so vendor implementations of the JDO standard are available, including a few open-source releases. A brief summary and comparison of these products can be found in Appendix A. The developer community is growing with more and more developers gravitating to the simplicity and ease of use that JDO has exposed. A good place to explore and exchange ideas is http://www.jdocentral.com (and the alias jdo-interest@java.sun.com).

1.2 Domain Object Model

A term used frequently when describing JDO is *domain object model.* A domain model is an abstraction that identifies "real-world" entities and their relationships in the context of the business. Architects use the domain model to capture the subject matter expert's or domain expert's perspective of the system. For example, a broker or stock trader actually using a trading system identifies the logical structure in a trading system.

A domain model is not intended for developers and, as a result, is not supposed to include any implementation-specific details (for example, logging or security context).

This can be confusing because the static structure of a domain model is usually represented using UML class models. However, the domain model would normally never have a source-code representation. An entity from a domain model generally has multiple representations in different parts of a system. For example, a domain model for a trading system might show an account and a customer with a relationship between them, but the domain model represents the true real-world concepts of *account* and *customer.* In the working system, many manifestations of these real-world entities might exist; for example, a customer widget object might exist on the client side, a customer EJB or business object might exist in the middle tier, and a database record might be in the resource tier. A variety of other analysis, design, and implementation models may reflect how these domain entities are represented and manipulated by the system, but there is only one domain model.

Figure 1-1(a) shows a simple domain model in UML notation for a stock trading system used by a broker. Figure 1-1(b) shows another domain model for a system used by agents to process returned orders in UML notation with members expressed in the Java programming language. In general, the abstractions represented by the domain model are turned into one or more *object models* that can be expressed in a programming language and used by developers.

Figure 1-1a. A simple stock trading domain model.

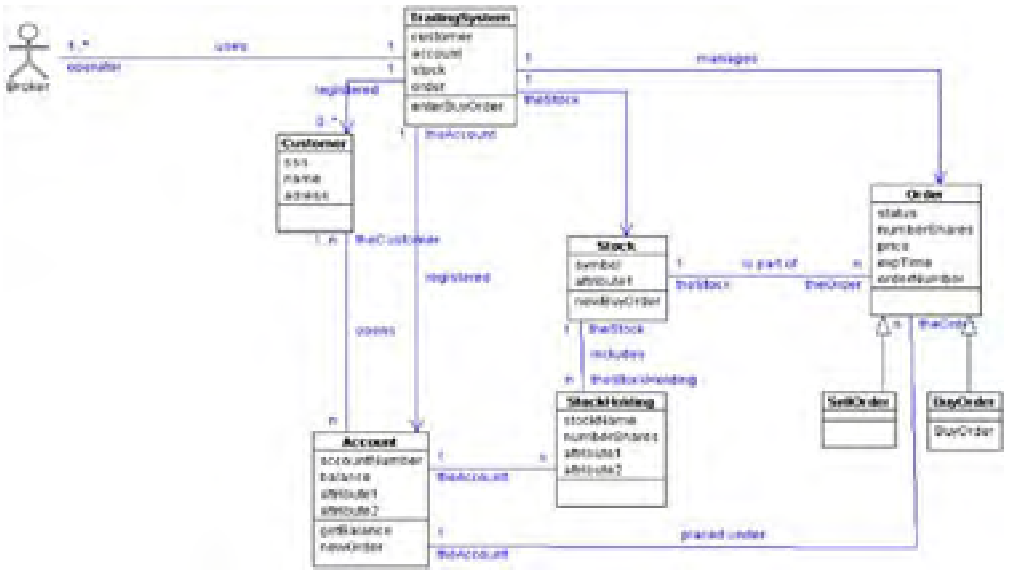
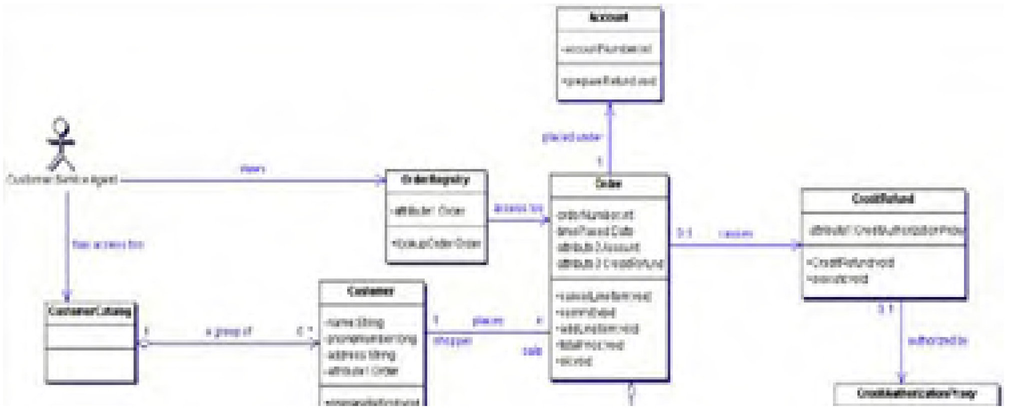
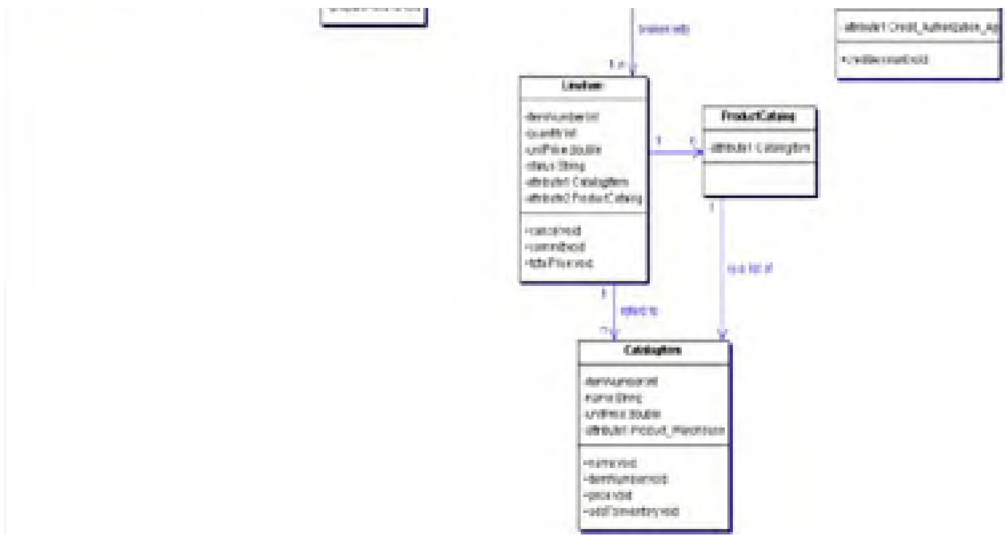


Figure 1-1b. An order entry domain model with Java notation.





Based on the domain model, business requirements, and their own object models, developers agree that some of the entities represented in their *application layer* require *persistence,* in the sense that the data represented by these classes needs to be available beyond the life of the physical software process (like the JVM) in which they are being used. For example, the order, account, and customer information needs to be saved so that it can be accessed at a later time when the user checks the status. At the same application layer, other programming constructs might also require persistence – for example, log messages, security audit trails, user sessions, and so on.

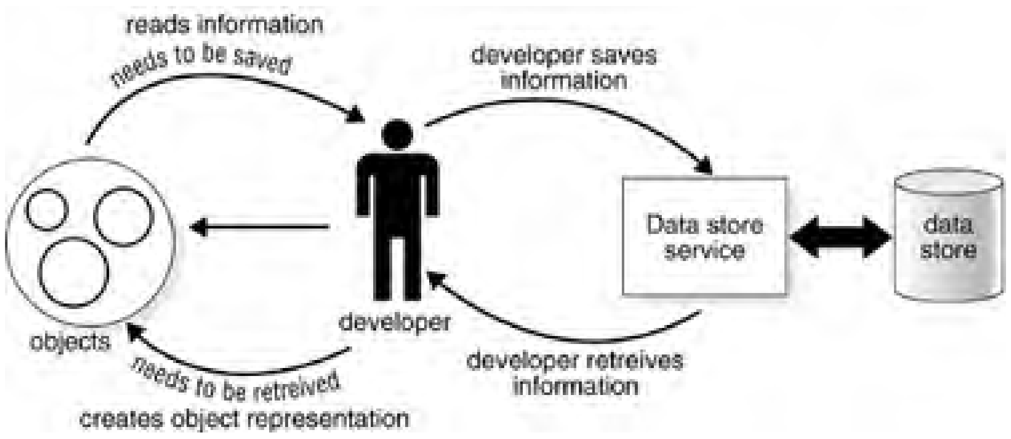
This is where JDO comes in. It provides developers with a mechanism for *transparent persistence.*

1.3 Orthogonal Persistence

Java is an object-oriented language, and developers inherently work with objects, which at any time have a state represented by member variables. From the Java developer's perspective, persistence focuses on the fact that the state in these objects should be available after the Java virtual machine has exited. For example, an application passing and using an instance of an Order class should be capable of recreating the Order instance after a machine reboot, or a year later when the account is audited.

The traditional programming approach has been to acquire the relevant state information and store it externally in some form – for example, using a relational database. When required, a new object instance is created and populated with this previously stored state. It is the developer's responsibility to map the information in the underlying datastore to its programmatic representation, as illustrated in Figure 1-2:

Figure 1-2. Traditional persistence deals with information.



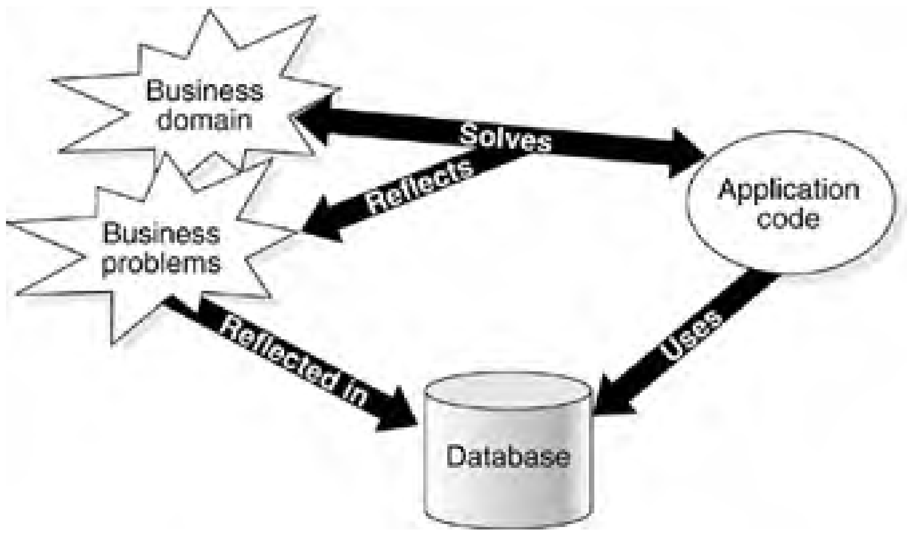
For example, to save an Order object from Figure 1-1(b), developers write code to insert order attributes (date, amount, etc.) and dependent attributes like those in the LineItem in some datastore. When needed, the datastore is queried for some field (like an orderid) that uniquely identifies the collective information set about that order and a new Order instance is created and populated with that information.

In general Java applications have followed variations of the following development steps:

- Identify the object model.
- Identify persistent entities in the model.
- Write database schemas to map the persistent entities information.
- Write code to store that information in a form that is understood by the database (e.g., access attributes in objects and insert corresponding rows in a database table), and write code to access that information in a form that can be used by the application (e.g., read rows in a table and return Java objects).
- Write application code that uses these classes and performs the business task.

As shown in Figure 1-3, developers now have to define, build, and use two models that programmatically reflect the same business domain – for example, business objects in the application code and their relational representation in the database. They also have to ensure that the view and behavior of these two models remains consistent because the slightest variation in either will result in an impedance mismatch between the models and the real-world business.

Figure 1-3. Traditional persistence leads to two models.



The idea of *orthogonal persistence* is one in which the application design is decoupled and independent of the entire underlying persistence infrastructure and has been around for a while.[1] An application is said to exhibit orthogonal persistence when the following three principles[2] that have been established after many years of research into persistence are met:

[1] Atkinson, M. P.; Bailey, P. J.; Chisholm, K. J.; Cockshott, W. P.; and Morrison, R. (1983). "An Approach to Persistent Programming." *Computer Journal* 26: 4.

[2] Atkinson, M. P.; Daynes, L.; Jordan, M. J.; Printezis, T.; and Spence, S. (1996). "An orthogonally persistent Java." *ACM SIGMOD Record*, December, pp. 2-3.

- The principle of *data type orthogonality* requires that all data, irrespective of its type information, should have the ability to be persisted. There are no special cases in which objects are not allowed to be *persistent* or are not allowed to be *transient*.
- The principle of *persistence independence* requires the application code to be indistinguishable whether it is manipulating persistent data (i.e., data originating from the stable store and outliving the program) or transient data (i.e., data that is created in and will only exist during the lifetime of the Java program execution).
- The principle of *persistence by reachability* requires that the lifetime of objects be determined by their reachability. This is the same principle that is applied to objects on the Java heap. When an object is marked for deletion, the entire tree of objects that it references can be garbage collected.

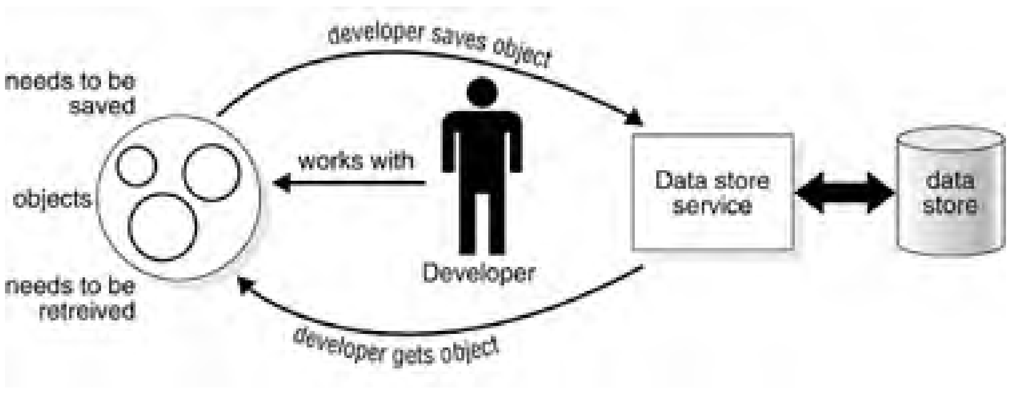
The first attempts at building orthogonal persistence using these principles in Java were initiated by Sun Laboratories in July 1995 as a research project called "Forest" and resulted in a prototype design called PJama.[3] However, the design was based around a modified virtual machine, and a second research project called JSPIN[4] tried another approach with the use of a preprocessor based on earlier research on ADA83. Although JDO is unrelated to these efforts, the principles outlined in all these works JDO share a conceptual similarity. In practical terms, JDO fulfills the needs of the community with a much simpler architecture.

[3] The Forest project, Sun Laboratories, available at http://research.sun.com/forest/index.html.

[4] Our SPIN on Persistent Java: The JavaSPIN Approach Jack C. Wileden, Alan Kaplan, Geir A Myrestrand, John V.E. Ridgway, available at http://research.sun.com/forest/UK.Ac.Gla.Dcs.PJW1.Jack_Wileden2_pdf.pdf. First International Workshop on Persistence and Java (1996).

An architecture utilizing any form of orthogonal persistence is quite different from the traditional persistence approach that we looked at earlier, in that this model deals with objects directly rather than the information contained therein. Such a persistence service abstracts away all the semantics relating to persistence from the application. Rather than working with information and mapping that information back and forth between objects, developers work with *persistent objects* directly in their code just like any other application objects. The difference is that some of these objects now represent persistent information in a datastore, as illustrated in Figure 1-4. The details about how they are persisted and recreated are *transparent* to the developer; this is often referred to as *transparent persistence.*

Figure 1-4. Transparent persistence deals directly with objects.



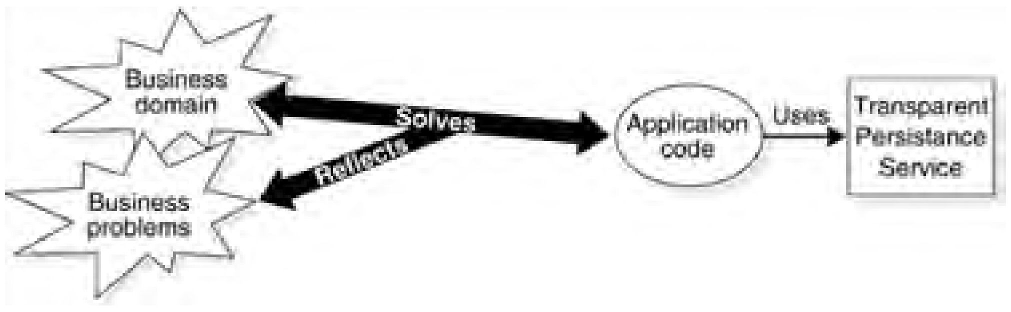
For example, to save an Order object from Figure 1-1(b), developers would use a transparent persistence service and give it that particular Order instance. When needed by the application at a later time, developers would query the service for that unique Order, by using, for example, an attribute "give me the order object with the orderid of 2828." The previously stored object matching that criterion is returned by the service. JDO is the standard that provides transparent persistence services to Java applications.

With JDO, architects and developers can concentrate on the application at hand, rather than the information contained therein. In comparison to the previously listed steps, development with JDO would involve the following steps:

- Identify the object model.
- Identify persistent entities in the model.
- Write application code that uses these objects and performs the business task.
- Use the rather straightforward JDO API in application code to persist and retrieve the objects when needed.

As shown in Figure 1-5, design with transparent persistence is much more simplified. Developers need to write and maintain only a single model or programmatic representation, e.g., business objects in the application code, when needed entities from that model can be persisted using the transparent persistence service.

Figure 1-5. The transparent persistence model.



Of course, JDO is not the single solution to all persistence-related issues, and some situations in which this might not be appropriate are discussed in Chapter 13. However, one thing is clear: JDO greatly simplifies (and possibly speeds up) the development approach.

JDO and Java Serialization

Developers will notice the similarities between JDO and Java Serialization. We look at Serialization, which also is a form of transparent persistence, and other persistence-related technologies in Chapter 2.

1.3.1 Transient and persistent objects

Java applications instantiate objects using the new operator and the constructor. For example:

```
Order myorder= new Order(2828);
myorder.addItem( new LineItem("Core JDO","$29.99"));
```

When instantiated, the virtual machine allocates space on the heap in memory, creates the object, and returns a reference. These objects do not represent any persistent data, meaning that their state and the state of objects referenced by them is present only in the memory of the virtual machine in which they are created. Such objects are called *transient* because they live only as long as the virtual machine process is running, or unless they are garbage collected.

A persistent object is no different from any other object, in that it is created using the same new operator and constructor. However, a persistent object is an instance of a class that has been marked as *persistence capable* and represents data in some persistent store. The lifespan of persistent objects is not tied to the life of the virtual machine in which they are created.

A persistent object becomes the representation of persistent data as a result of an explicit invocation of JDO API. For example:

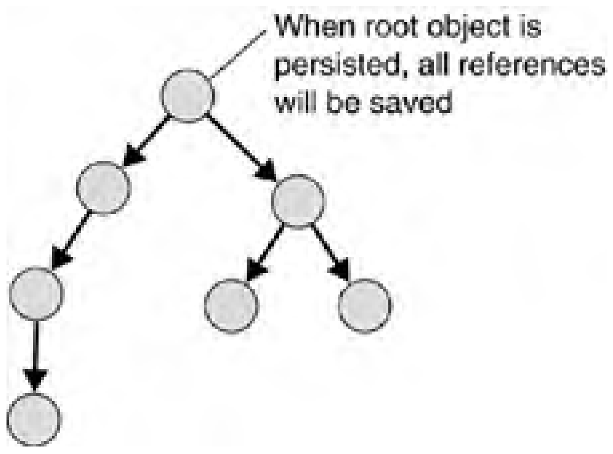
```
PersistenceManager mgr=pmf.getPersistenceManager();// JDO API
mgr.currentTransaction().begin()/ JDO API
Order myorder= new Order(2828); // Transient object
mgr.makePersistent(myorder); // myorder marked as persistent
mgr.currentTransaction().commit() // JDO API instance myorder
                             //is written to datastore
```

A transient object can also become persistent if it is referenced as a field in a persistent object. This is known as *persistence by reachability or transitive persistence* (in ODMG terms), and it ensures that when an object is stored in a datastore, every item or the graph of items that the object references and collectively represent that object's state are also preserved. This is necessary to recreate the object instance correctly when required. Persistence by reachability is the aspect that results in the appearance of transparent persistence to the developer. The code extract below explains this further:

```
PersistenceManager mgr=pmf.getPersistenceManager();// JDO API
mgr.currentTransaction().begin();// JDO API
Order myorder= new Order(2828); // Transient object
mgr.makePersistent(myorder); // myorder marked as persistent
// other application code here
myorder.addItem( new LineItem("Core JDO","$29.99"));
mgr.currentTransaction().commit();// JDO API myorder
                //including the LineItem is
                           //also written to datastore
```

In the above code segment, unlike the Order object, the LineItem object was never explicitly persisted. However, because it is referenced by an object that is persisted, it too will be saved, as depicted here in Figure 1-6:

Figure 1-6. Persistence by reachability.



A persistent object can be changed back to a transient object by an explicit call to the JDO API. When a persistent object is marked as transient, any change made to it is not reflected in the datastore. For example:

```
PersistenceManager mgr=pmf.getPersistenceManager();// JDO API
mgr.currentTransaction().begin(); // JDO API
// other application code here
mgr.makeTransient(myorder) // JDO API
// other application code here
```

A persistent object is also automatically changed to a transient object after it has been persisted. For example, look again at the code shown earlier:

```
PersistenceManager mgr=pmf.getPersistenceManager();// JDO API
mgr.currentTransaction().begin()/ JDO API
Order myorder= new Order(2828); // Transient object
mgr.makePersistent(myorder); // myorder marked as persistent
mgr.currentTransaction().commit() // JDO API instance myorder
                              //is written to datastore
myorder.doSomething(); // Transient object
```

Both transient and persistent instances go through different state changes that we discuss in Chapter 4.

Another strategy employed in object persistence is termed *persistence by inheritance*, in which object instances of a subclass of a persistence-capable class can be persisted. JDO is very complete in its support for inheritance. A class can be persistence capable even if its parent is not, a parent can be persistence capable and sub classes may not be, and even selective individual classes may be persistence capable in a large inheritance tree. It is up to the requirements specified by the developer.

The key architectural points surrounding JDO persistence are as follows:

- Persistence is orthogonal to the class hierarchy. All user-defined objects of any level and of any type may be persisted except those whose state depends on the operating environment (for example, system classes like File, Sockets, Thread, InputStream, etc.). This is understandable because objects with such native state depend on or are a reflection off the state of the operating system in which they are executing, upon which no guarantees can be placed.
- Persistence is transparent to clients. The logic relating to object persistence is abstracted away. Application code does not move objects to and from stable storage, nor does it deal directly with the underlying datastore.
- Persistence is per-object based. Objects of a particular class may be persistent or transient, and in fact may be in general objects of both kinds for a given class.
- Persistence is based on method invocation. An object is promoted to a persistent state by making use of a `makePersistent()` method.
- Persistence is based on reachability and ensures that all observable properties of a persistent object are preserved even though they may refer to other objects.
- Persistence does not alter object type information and guarantees type safety.

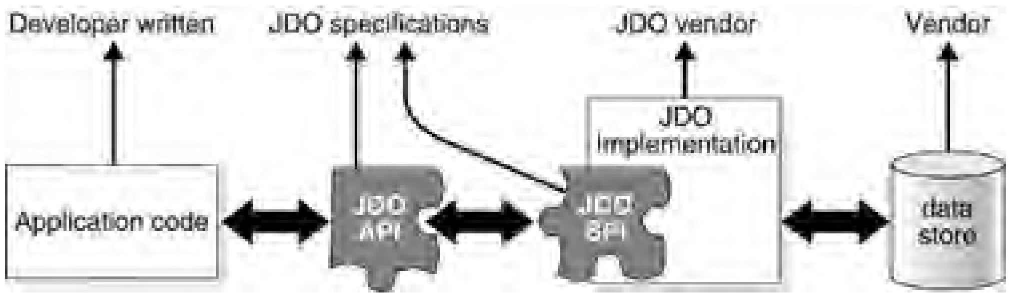
1.4 Non-Managed and Managed Environments

The previous section discussed the need for "*someone*" to call the datastore and persistence services. Figure 1-4 showed this *someone* as being the developer, and this needs a bit of elaboration. With the arrival of J2EE, Java applications environments can be broadly grouped into two categories—namely, non-managed and managed environments.

1.4.1 Non-managed environments

A non-managed environment defines an operational environment for a two-tier application in which an application client connects directly to the resources that it needs. Applications do not depend on any J2EE containers and are typically standalone or based around client-server architectures. In such non-managed environments, the application developer and the application are responsible for all the interactions with the underlying persistence service. This includes configuring the service and its environment and invoking it at appropriate times in the applications lifetime so that data can be persisted. An application that would use JDO in this manner is shown in Figure 1-7:

Figure 1-7. JDO in a non-managed environment.



1.4.2 Managed environments

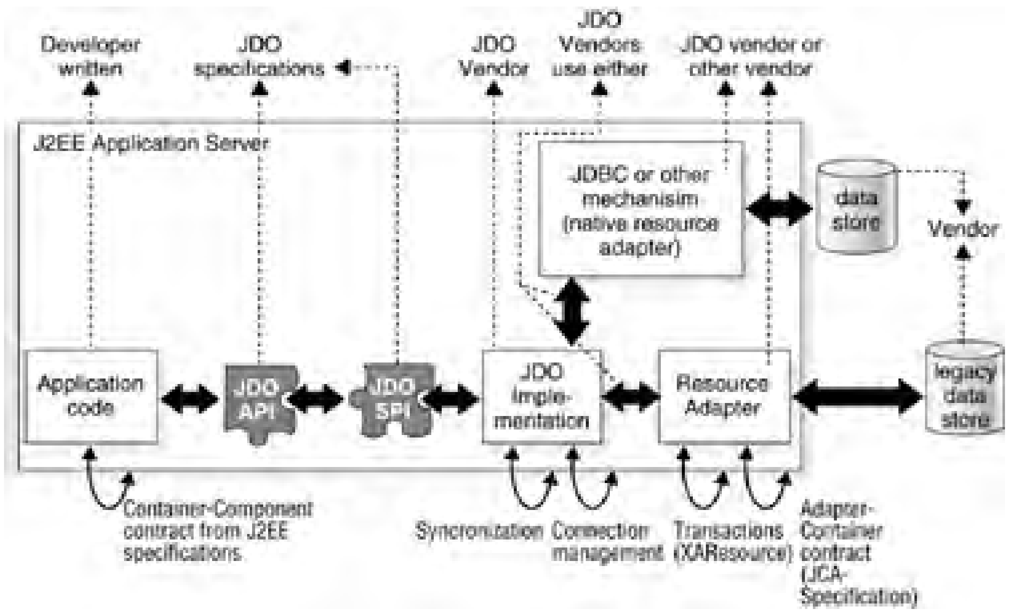
A managed environment defines an operational environment for a J2EE-based, multi-tier, web-enabled application that accesses Enterprise Information Systems (EISs). The application consists of one or more application components (e.g., EJBs, JSPs, and Servlet) that are deployed in *containers*. These containers can be any of the following:

- Web containers that host JSP, Servlets, and static HTML pages
- EJB containers that host EJB components
- Application client containers that host standalone application clients

In a managed environment, the J2EE *container* takes responsibility for configuring the service managing distributed transactions, and for providing security services and other similar system level functions that would be the responsibility of the application in a non-managed environment. As shown in Figure 1-8, in a managed environment, an application component still uses the JDO API; however, the JDO implementation now must support other features so that the application components that are being managed declaratively by the container can work correctly. For example:

- Pooling of objects implementing the transparent persistence service that the application components, e.g., the JDO PersistenceManager.
- The JDO vendor implementation must implement interfaces so that the transactions work as planned and get flushed from the application component to the underlying resources ; for example, the implementation must expose the javax.transaction.Synchronization interface.
- If the JDO implementation uses EIS resources, it is required to access those stores using resource adaptors in a managed environment. The J2EE Connector Architecture (JCA) defines the standard manner in which J2EE applications can communicate with legacy databases and transactional systems. This is covered in Chapter 8.

Figure 1-8. JDO in a managed environment.



Typically, and as most JDO vendors use today, a relational database or an object database is the underlying datastore. In this case, the JDO implementation is said to be using a *native* resource adaptor.

1.5 Roles and Responsibilities

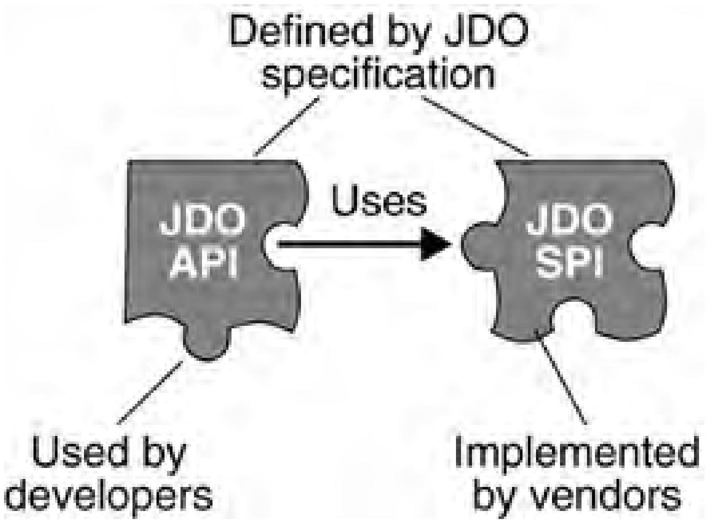
From the preceding discussion, it is clear that there are different roles (developer, JDO speciation, vendor, etc) and each has responsibilities that it must fulfill in order for the application to work as designed. This section looks at these in detail.

1.5.1 JDO specifications

The JDO specifications, besides being a community standard, have five concrete responsibilities:

1. They define the standard for building object persistence.
2. The specifications define the standard API that developers can use in their applications for transparent persistence.
3. The specifications also define a second API – called the Service Provider Interface (SPI) and illustrated in Figure 1-9 – that JDO vendors must implement, which constitutes the JDO environment in a virtual machine. This is a typical design that many Java standards have adopted. For example, JDBC defines the API that developers use and another that the JDBC driver provider must implement; JMS defines an API that developers use and another that the vendor providing the messaging server must implement. JNDI and EJB are other examples that follow the same SPI design.

Figure 1-9. JDO API.



4. The specifications provide a Reference Implementation of the JDO standard for developers to use and develop applications in.
5. The specifications provide a Technology Compatibility Kit (TCK) that can be used by vendors to test compliance with the JDO standards.

1.5.2 The developer's responsibilities

A developer is responsible for the following:

1. Determining which objects in the object model need to be persisted and writing their Java classes.
2. Marking these classes as *persistence capable* or having the ability to be stored by JDO. The developer can do this
    - Explicitly by having the class implement an interface defined by the JDO specification—the `javax.jdo.PersistenceCapable`
    - Implicitly by using a tool provided by the vendor that takes the compiled Java class code and marks it as being persistence capable. This tool is called a *byte code enhancer,* and we see more about this in Chapter 5. Keep in mind that the developer has the option to *not* implement any special API or

persistence-related logic in the objects themselves and to write the usual Java code. The concept of a byte code enhancer or preprocessor is not new to JDO. As mentioned earlier, this was proposed in the 1980s and brought to the forefront for Java by the JSPIN research project.[5]

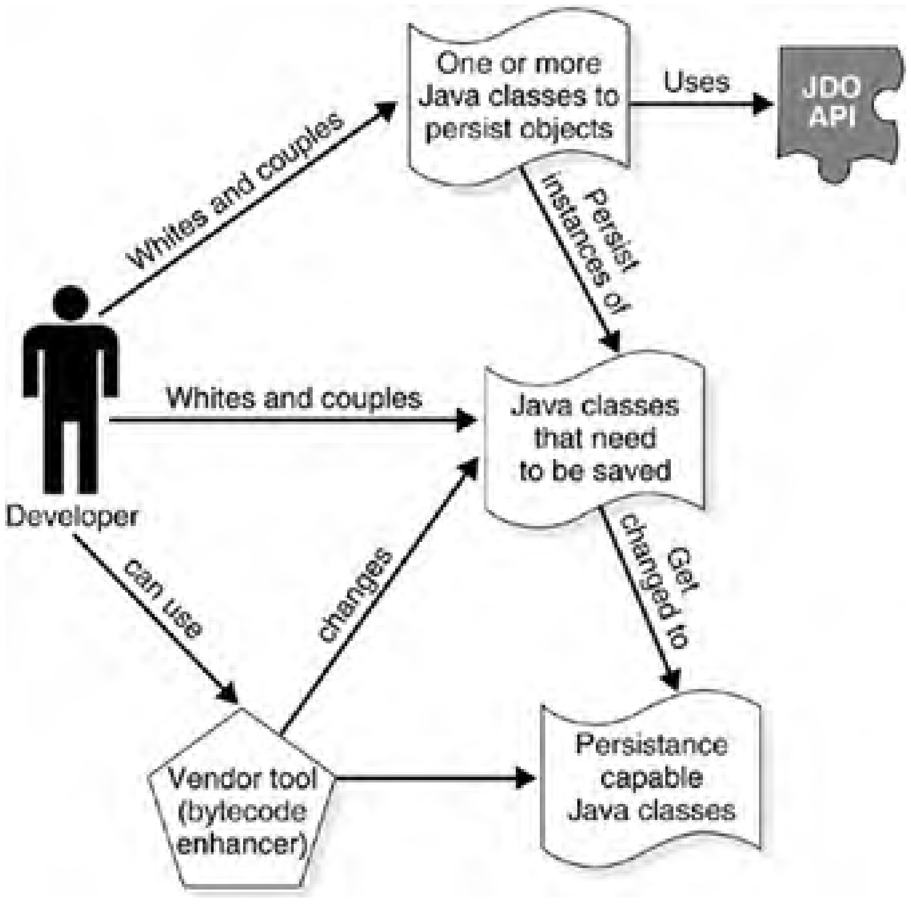
[5] Toward Class Evolution in Persistent Java. Ridgway & Wileden. The Third International Workshop on Persistence and Java-1998, available at http://research.sun.com/forest/com.sun.labs.pjw3.9_pdf.pdf.

Are persistence-capable instances persistent?

Just because a class is persistence-capable does not automatically mean that all instances are persistent. It means only that when JDO code is invoked to persist an instance of the class, it *will* be persisted to the datastore.

3. Using the JDO API as defined in the JDO specifications to persist instances of these classes, as illustrated in Figure 1-10.

Figure 1-10. Developer responsibilities.


One or more Java classes to persist objects
Uses
JDO API
Whites and couples
Persist instances of
Developer
Whites and couples
Java classes that need to be saved
can use
changes
Get changed to
Vendor tool (bytecode enhancer)
Persistance capable Java classes

1.5.3 The vendor's responsibilities

The JDO vendor plays an important role by providing the JDO implementation. The primary responsibility of the vendor is to implement the JDO-defined SPI and the JDO specifications. The JDO vendor must also pass the technology compatibility kit (TCK) that is provided by the JDO specifications.

Until now, we have used the term vendor as though it referred to a single company. In reality, several different vendors will be involved, including the following:

1. A vendor that provides the JDO implementation. A list of JDO vendors is contained Appendix E.
2. A vendor that provides the JCA resource adaptor or JDBC driver to connect to the underlying datastore.

3. A vendor that provides the underlying datastore. Because JDO vendors are free to use any underlying storage mechanism, the underlying store depends on what the JDO vendor supports and uses. It could be a relational database from another vendor, an object database from the JDO vendor itself, a file database, a hierarchical database, an EIS system, etc.
4. If a managed environment is used, a J2EE vendor must provide the J2EE application server. For example, JDO may be used for bean-managed persistence in EJBs, in which case the EJB must be deployed in an EJB container provided by a J2EE server. The relationship between JDO and EJBs is covered in Chapter 9.

These logical roles can be fulfilled by a single vendor. In reality, there are two categories of JDO vendors:

1. Vendors that provide a JDO implementation using some form of file-based persistence or relational database and third-party JDBC drivers for that database.
2. Data storage (usually relational or object) vendors that augment their product offering by providing a JDO implementation. Until recently, these implementations have been provided by existing object-database vendors (e.g., Versant) and vendors of object-relational mapping products (e.g., SignSoft). It is possible that, in the future, relational database vendors may also augment their offerings by providing JDO implementations along with their servers.

It is also typical for JDO vendors to provide implementations that come in two different flavors:

- A standalone implementation that can be used in a non-managed environment
- A more expensive version that integrates with J2EE servers from other vendors using JCA technology

1.6 Summary

In this chapter, we introduced the concepts around which JDO is centered: a standard specification for incorporating persistence facilities into Java applications. We looked at how JDO provides an API that gives developers a Java-centric view and insulates the application code from the complexities of the underlying datastores. We also looked at the how JDO has been designed to fit into the non-managed and managed environments and the logical roles and responsibilities of the different parties involved.

In the next chapter, we write our first JDO application and introduce you to development with JDO.

Chapter 2. Object Persistence Fundamentals

"We are made to persist. That's how we find out who we are."

—Tobias Wolff, In Pharaoh's Army

This chapter looks at the challenges that a persistence solution must address. A closer look is taken at object-relational mapping solutions in particular. The effort of a "from-scratch" implementation of such a framework is then outlined.

It is perfectly possible to use JDO without indulging in the underlying issues that a JDO implementation solves for you. However, it may still be helpful to at least be aware of them. Read the rest of this chapter to get an idea of the questions that you would generally trust the JDO implementation to have solved for you.

Alternatively, you can skip this chapter at this point and plunge straight into the usage of JDO in the following chapters. Be sure to come back and read this chapter, though; it's full of great information.

2.1 Persistence in Applications

The principal tenet of persistence is to save something to non-volatile storage, which can be a file, a disk, or flash memory, such that if a system is power cycled or an application is restarted, it is possible to retrieve what was previously stored.

A persistence mechanism can either be implemented by an application itself or can be made persistent *magically* by the operating system, for instance. Thanks to object-oriented programming, it is easy to implement the first case in a helper class that can read and write all basic types like int, short, byte, boolean, String, Date, and so on.

Such a helper class works as an adapter between the storage media and the object contents. If an interface is used for all classes that should become persistent, it is also easy to make a closure of objects persistent and to read the state back into memory. We drill more deeply into this approach later.

The other way to load and save the object state persistently can be implemented by the runtime system, the operating system, or in case of Java, by the virtual machine. The idea is that this environment already has all the information about the memory layout of objects, their references to other objects, and so on. It just needs to copy a chunk of data to or from a disk. This kind of *fully orthogonal persistence* can become difficult to realize because the underlying runtime or operating system cannot properly restore semantics without a notion of surrounding application environment. Examples of such persistence-incapable types are network connections (sockets), file handles, or window positions.

In the past, several vendor-neutral groups have tried to solve and propose standards for the *persistence* problem: Project Forest (Sun's Orthogonal Persistence research project), Java Blend, Object Data Management Group (ODMG), EJB CMP, and many others.

2.2 JDK Binary Serialization

Serialization is the basic persistence support that is built into the Java language. It is called serialization because the objects are written or read sequentially as a series of bytes. An object is **serializable** if its class implements the serializable interface and all its non-transient members are as well serializable. When an object is serialized, the default algorithm traverses the closure of the object graph and writes all reachable objects to a stream. If a member of a class is tagged transient, the default algorithm skips this field.

When the stream is read back, the equivalent object graph is rebuilt, keeping transient members initialized to their null values. By tagging a field as transient, its referenced object or content is not serialized by the default algorithm. On the contrary, fields that are not tagged transient are not checked by the Java compiler. For example, an ArrayList itself is serializable because it implements the Serializable interface, but because an instance of ArrayList might refer to any instance of java.lang.Object, the graph might still not be serializable at runtime.

2.2.1 The serialization API

The Serializable interface has no declared methods. It is merely a tagging interface, although two methods are frequently used in conjunction with Serializable: readObject and writeObject. The following code copies a SerializableBook into a byte array and back into another SerializableBook:

```
import java.io.*;
import java.util.*;

class SerializableBook implements Serializable
{
      String name;
      SerializableBook(String n)
      {
         name = n;
      }
}


public class SerialTest
{
  public static void main(String args[])
    throws Exception
    {
      SerializableBook object;
      object = new SerializableBook("foo");

      // create a buffer to write the object into:
      ByteArrayOutputStream bo;
      bo = new ByteArrayOutputStream();

      // create a stream for serial object data:
      ObjectOutputStream oo = new ObjectOutputStream(bo);

      // write the object (and all its members recursively)
      oo.writeObject(object);

      // don't forget to close the stream:
      oo.close();

      // now lets get the object back from the stream:
      // get a new stream that reads the buffer of the
      // previous stream:
      ByteArrayInputStream bi =
                new ByteArrayInputStream(bo.toByteArray());

      // get a stream that can read objects:
      ObjectInputStream oi = new ObjectInputStream(bi);

      // cast the read object to our type:
      SerializableBook back;
      back = SerializableBook)oi.readObject();

      // never ever forget to close a stream:
```

```
   oi.close();
   }
```

Three important things can be seen here:

- ObjectOutputStream and ObjectInputStream are the main working horses for serialization.
- A String seems to be serializable and is written to the stream by some magic. There is no code to read or write that member in the SerializableBook class.
- The returned object has to be cast to the destination type. Although the ObjectInputStream "knows" which object to create from the stream, there must be some generic method that returns simple objects.

In the example code, ByteArrayInputStream and ByteArrayOutputStream, respectively, may be replaced by any other Java stream implementation. This is what makes serialization a powerful component of the Java platform: Simply anything may be *streamed* through a pipe, a network connection, a file, or into a buffer. Other components make heavy use of serialization, such as Remote Method Invocation (RMI), for example. Needless to say, serialization works across platforms and is independent of the CPU byte order or other operating system dependencies.

2.2.2 Versioning and serialization

One of the obstacles with serialization is version handling. When anything changes in the class layout—the class name, package name, inheritance, fields, or access modifiers—the new version of the class must be recognized by the serialization runtime library. This is done by a so-called serial version unique identifier (serial version UID), which is a kind of hash code over all properties of a class. The default algorithm throws a StreamCorruptedException or an InvalidClassException if the serialized object data is incompatible with the class in memory. That is one reason why developers need to overload the readObject and writeObject methods with their own implementations to handle different versions of a class. Here is an example of how to handle the versioning problem:

```
class Author
   implements Serializable
{
   String name;

   Author(String n)
   {
      name = n;
   }

   private void writeObject(ObjectOutputStream out)
      throws IOException
   {
      int version = 1;
      out.writeInt(version);
      out.writeUTF(name);
   }

   private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException
   {
      int version = in.readInt();
      if (version == 1) {
         name = in.readUTF();
      } else {
         throw new IOException(
              "Invalid version: "+version);
      }
   }
 }
```

Now that a version is provided with every object in the stream, the readObject method can easily distinguish old and new objects. The second version of the Author class gets its name attribute split into first name and last name:

```
class Author implements Serializable
{
   String firstName;
   String lastName;

   Author(String first, String last)
   {
      firstName = first;
```

```
        lastName = last;
    }

    private void writeObject(ObjectOutputStream out)
        throws IOException
    {
        int version = 2;
        out.writeInt(version);
        out.writeUTF(firstName);
        out.writeUTF(lastName);
    }

    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        int version = in.readInt();
        if (version == 2) {
            firstName = in.readUTF();
            lastName = in.readUTF();
        } else if (version == 1) {
            // fallback:
            String name = in.readUTF8();
            firstName = guessFirstName(name);
            lastName = guessLastName(name);
        } else {
            throw new IOException(
               "Invalid version: "+version);
        }
    }
  }
```

2.2.3 When to use object serialization

An application should use serialization when simple data types have to be saved persistently. Configuration data, which is entered through dialog boxes or window positions in GUI applications, is a good candidate for serialization. When the application version changes, such settings may be simply ignored and complex version handling may not be required. Another use case is RMI: an RMI method may take any object as a parameter if the object and its referenced attributes are serializable. The same applies to method return values. Complex version handling is required only for applications that must support incompatible client and server application versions.

2.2.4 When not to use object serialization

Serialization does, however, have serious limitations that exclude it as a candidate technology for storing even simple domain objects (e.g.. clients, bills, issues, and so on) of real-world applications. Most notably, Java JDK serialization lacks the following:

- **Query facility:** After objects are serialized into a stream, there is no way to ask the stream to return all objects that have fields of certain values. The readObject() method simply returns the next object in the stream, with all objects reachable from it.
- **Partial read or update:** While partial reads and updates as well as caching features are "only" performance and memory usage enhancing features of other persistence solutions, the lack of such approaches in serialization makes it useless for big quantities of data.
- **Lifecycle management:** The deliberately simple serialization API (as seen above) has no notion of an object's "state," i.e., an object being "dirty," and so on. Objects exist in memory, and are explicitly written to and read from dumb streams, and it really doesn't go further. As mentioned, this is, of course, perfectly suitable for purposes such as RMI or persisting very simple configuration information, but not for domain data.
- **Concurrency and transactions:** Serialization completely lacks the notion of ACID[1] transactions and is not suitable for applications with concurrent data access. It is, for example, impossible to have two threads write to or read from one and the same stream at the same time.

[1] ACID is an acronym for **A**tomicity, **C**onsistency, **I**solation, **D**urability.

The next section examines how relational databases can be used with an object persistence framework as a data storage technology.

2.3 Object-Relational Mapping

Object-Relational (O/R) mapping is a technique to map between developed Java classes and relational databases. Java relations between objects, classes, and their fields have to be mapped somehow to database relations, tables, and columns in a relational database.

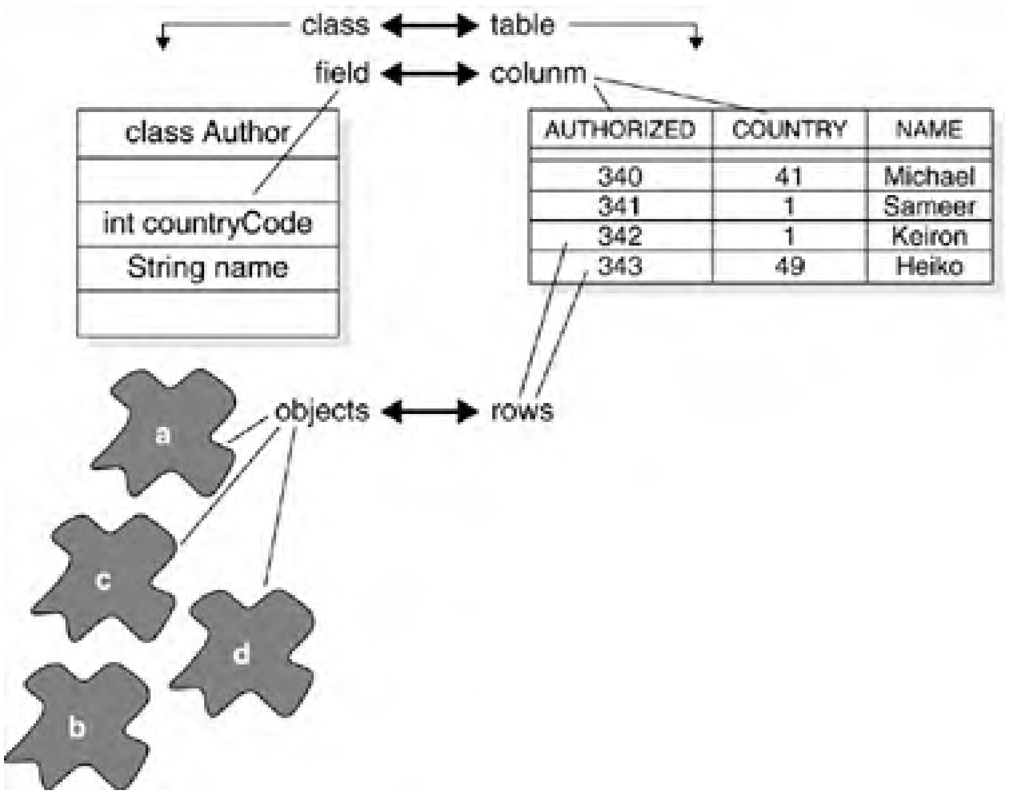
An object-oriented domain model of business entities is sometimes compromised for technical reasons like performance or space requirements when the application matures. In this case, object-relational mapping is also a technique to reengineer complex, existing database models. The object-oriented "is-a" and "has-a" relationships can only be poorly mapped to relational databases, where a record or related information often spans a number of tables and has to be assembled by join operations.

2.3.1 Classes versus tables

One talks about O/R mapping if tables and columns of a database are connected with classes and fields of an object-oriented language, so that an instance of a class can be used to access or modify data of one or more rows/columns of that database.

Figure 2-1 shows a Java class Author that has a countryCode and name field. This information can be put into an AUTHORS table that has corresponding COUNTRY and NAME columns:

Figure 2-1. The object versus the relational view of the world.



Instances of the Author class are plain Java objects that contain the countryCode and name data during runtime. The instance data corresponds to table rows, which are accessed by SQL (Structured Query Language) queries and INSERT, UPDATE, or DELETE operations. Starting with this basic, apparently simple mapping scheme, the details can get more complex rapidly:

1. In each of both worlds, there are different type systems.
2. Java supports inheritance; most relational database systems don't.
3. Deletion in Java is governed by a garbage collector, while relational databases have an explicit operation. Combining these two can lead to questions.
4. True objects are ultimately always "found" by following a reference from another object. Relational databases, however, have a different view: A table is a set of rows, on which query operations can be performed.

5. Naming conventions are different: Class names can be long, are case sensitive, and include a hierarchical package name; attribute names can also be long. SQL table and column labels, however, are often of (very) limited size. The class-to-table and attribute-to-field name mapping thus needs to be addressed. Automatic name mapping schemas or manual specification of table and column names are imaginable, and often both provided by JDO implementations and schema generators.

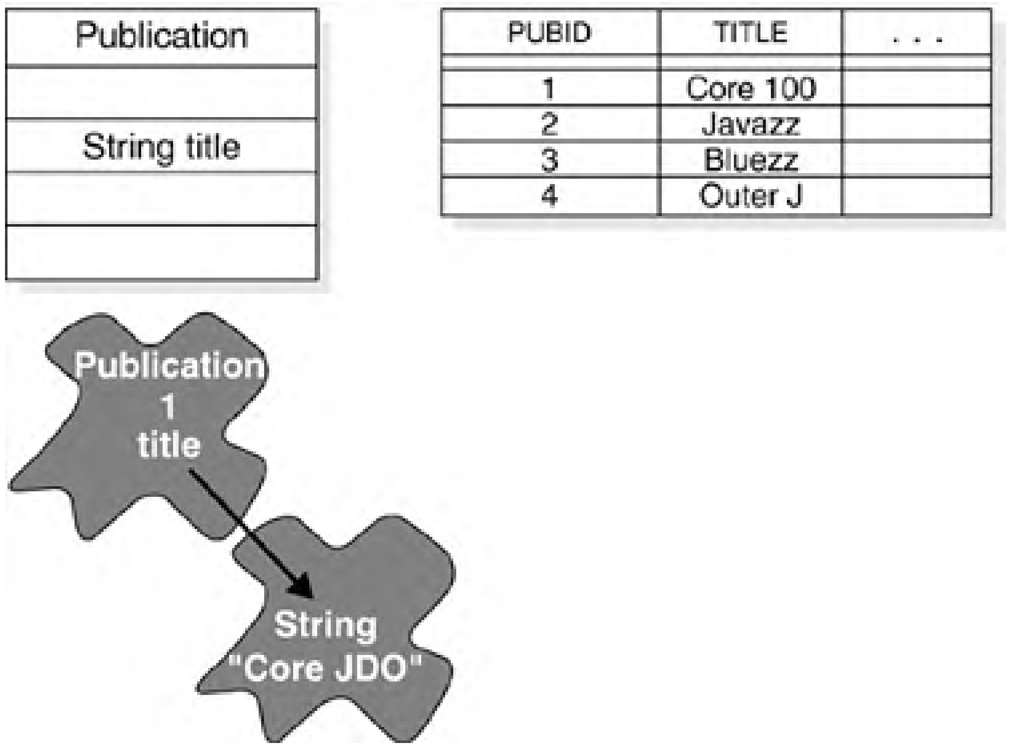
6. Ambiguities exist in mapping references to relations.

Such issues are examined in more detail over the next few pages.

2.3.2 String, date, and other type mappings

The Java type system is different from SQL's type system. Basic types like Java short, int, float, and double do not directly map to exact SQL types, and vice versa. Even simple types like Java String and Date have no exact equivalent in the relational database world because they are classes, and in principle, classes are mapped to tables. In Figure 2-2, the Publication class has a title field that refers a String object at runtime, but in a database, columns can be declared to store character data.

Figure 2-2. A Java string is really a reference to an independent object, but is generally mapped to a column of a table, not a row in a STRING table.



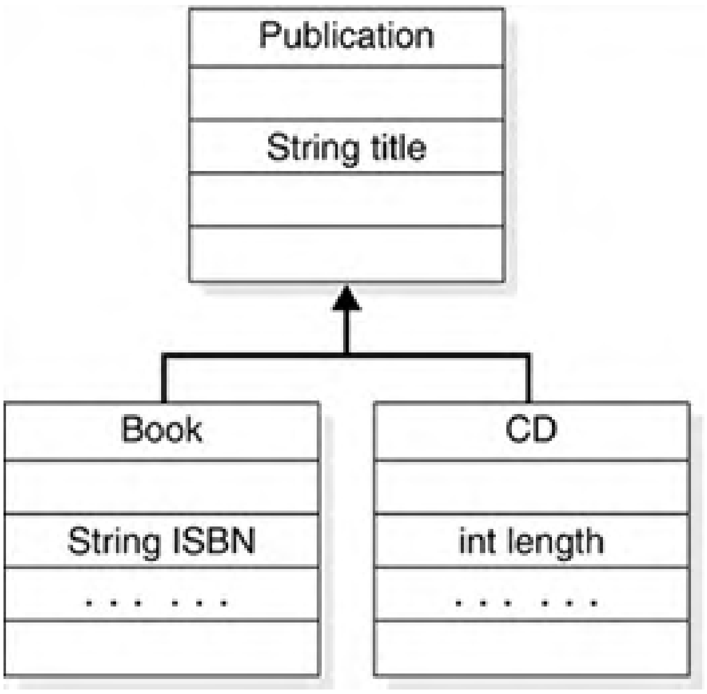
Using date types is even worse. In Java, the Date class counts milliseconds since January 1, 1970, and stores them internally in a 64-bit long value. In SQL, there are many different flavors of date and time or combinations of them available as column types.

Other mapping issues could arise if the out-of-the-box provided object-to-table mapping is not desired purely for efficiency reasons in the relational model; it is often the case to use object references to model a Java "enum" pattern that would generally be modeled as a simple number column. Even simpler, sometimes a String field of a Java class should actually be represented as a small 1-2 character type column with some sort of fixed back and forth mapping of some code, or currency symbol, and so on. (Such issues are more likely to arise if existing relational schemas should be used by, but sometimes even desired, with new schemas.)

2.3.3 Inheritance mapping

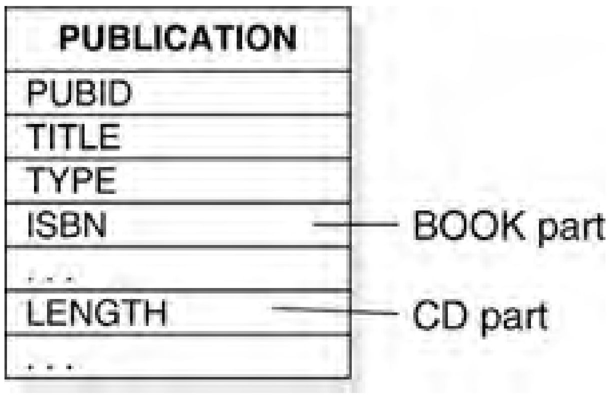
Object models using inheritance can be mapped to relational models by different mapping schemes. The class hierarchy shown in Figure 2-3 is used throughout the next paragraphs to explain these schemes.

Figure 2-3. Sample class inheritance.



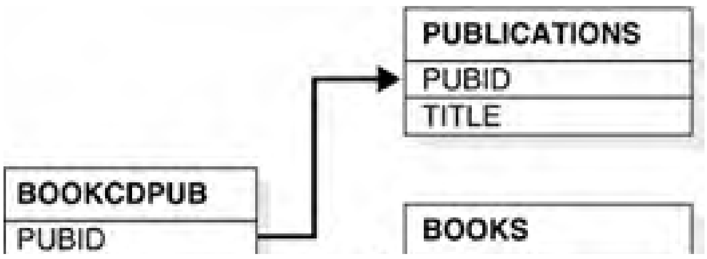
The first scheme maps every field of every class of the class hierarchy into a single, plain table. An additional TYPE column indicates whether a record is a Publication, Book, or CD object. Because an instance of Publication can be a Publication, a Book, or a CD, the table contains lots of empty fields. If the instance is a Book, the CD fields are not used and vice versa. Figure 2-4 shows this table structure.

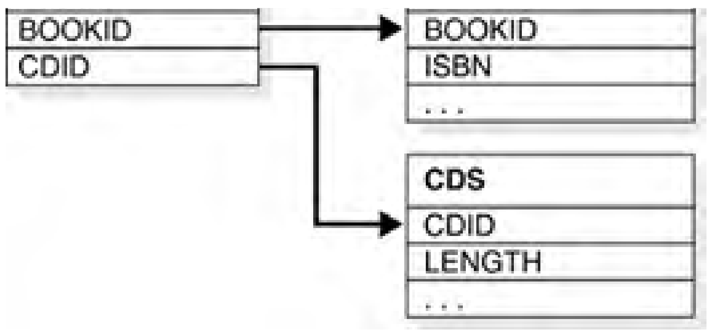
Figure 2-4. A "flat" inheritance mapping with an entire class hierarchy in one table.



A second scheme is on the other extreme—it puts the fields of each class in a separate table. A fourth table, BOOKCDPUB defines the relations of the instances. Figure 2-5 is an example of such a mapping scheme. This scheme should be used if Book and CD instances are rarely expected and a huge number of plain Publication objects exist in the database.

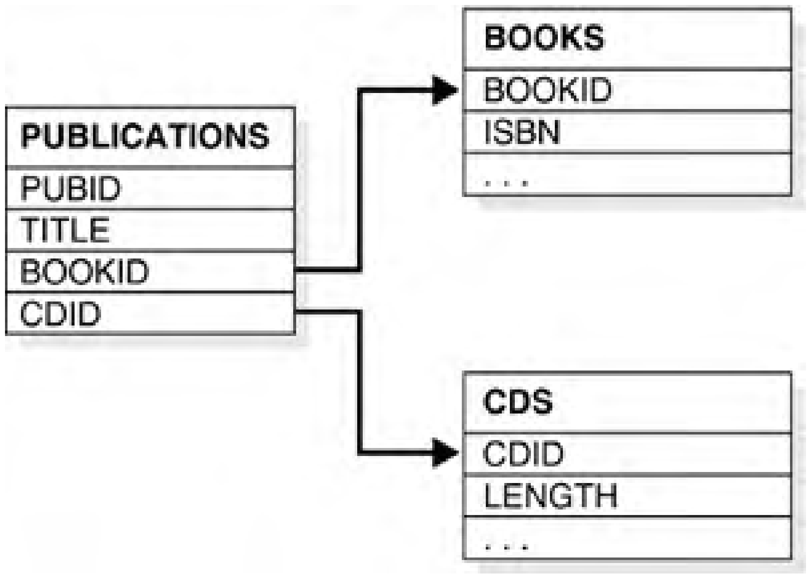
Figure 2-5. A "fat" inheritance mapping with one table for each class, plus an additional "forward join" table (rarely used by products in this form).





In a third scheme to save the extra table, if the Publication class is abstract or no pure Publication objects exist, the Book and CD relation fields BOOKID and CDID can be moved into the Publications table. This scheme, shown in Figure 2-6, is frequently used in applications.

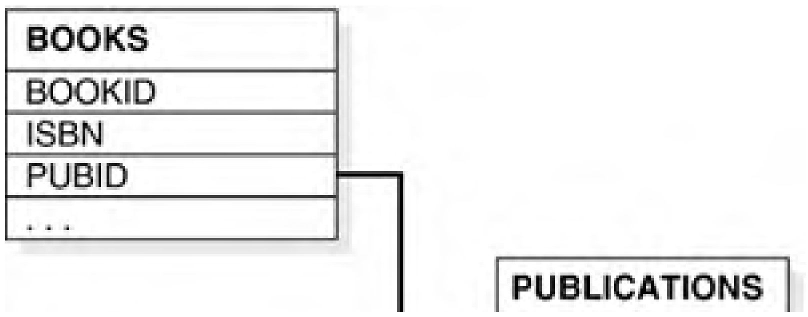
Figure 2-6. An inheritance mapping with one table for each concrete class, and "forward join foreign keys" from superclass to all possible subclasses.



It is a typical model with key/foreign-key relations that also illustrates the ambiguity of "is-a" and "has-a" references. One cannot tell from the table declarations whether Books and CDs are part of the Publication data or just referenced by it.

Figure 2-7 shows a fourth mapping scheme that eliminates the extra table by using backward references. Notice that the BOOKID and CDID columns are both primary keys of their respective tables, as well as foreign keys to PUBLICATIONS.PUBID. The insert, update, and delete operations for Book and CD objects is faster and less disk space is used for each object, although queries can become quite complex—for instance, if Book objects are searched by title. An often-seen sensible optimization is to provide some sort of TYPE (or CLASS) column in PUBLICATIONS that allows determining the exact type of a Publication instance, i.e., whether it is "just" a Publication, a Book or CD instance, without further queries. Notice in the previous mapping scheme that this was determined by the non-NULL presence of BOOKID or CDID foreign keys in the PUBLICATIONS table.

Figure 2-7. An inheritance mapping with one table for each concrete class, and "backward join foreign key" from each subclass to the superclass.



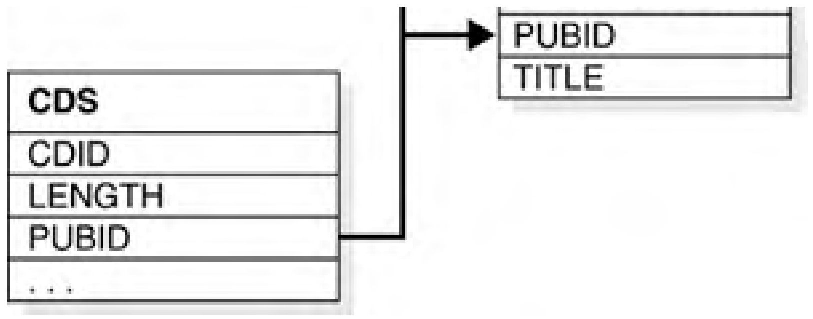
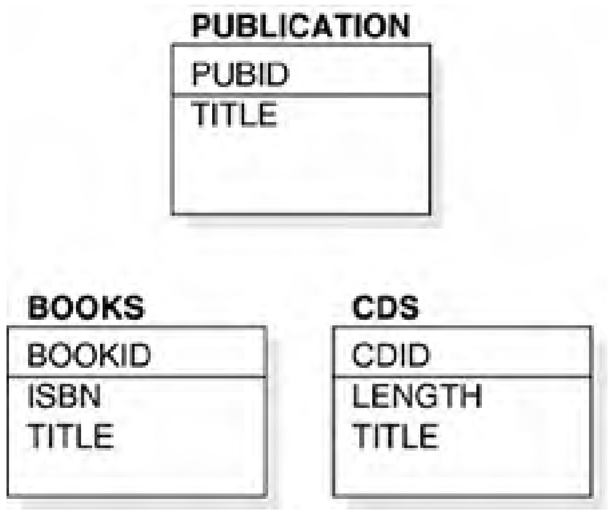


Figure 2-8 shows yet another possible mapping scheme in which one table per concrete class is used. The definitions of the attributes of the superclass are duplicated in each subclass, but no data is duplicated, because an object is always a Publication, Book, or CD. If the Publication class is not abstract, a PUBLICATIONS table, with only PUBID and TITLE columns, and storing only instances of real Publication objects (no subclass instances), can be used; if Publication is abstract, then that table is not necessary.

Figure 2-8. An inheritance mapping with separate tables for each concrete class.



This mapping has very good performance characteristics for most query operations, except for queries on the Publication superclass "with subclasses," which could not be expressed as a table join and would lead to two separate queries or a UNION. (It is imaginable, purely to optimize performance for this specific case, to duplicate some columns of records of the subclass tables [BOOKS and CDS] into the superclass PUBLICATIONS table to speed up respective queries; in this case, a persistence framework would have to ensure consistency.)

2.3.4 Security

When using O/R mapping, other problems result from different access restrictions to data. In Java, public, protected, private, and package access defines the rules for field read and write access, as well as method invocation.

Relational database systems have much more flexible access policies, based on user or group security control settings. Such mapping problems may occur if a column of a table must not be read by a user, but the class does contain a field for that column. Security issues may influence the way columns are mapped to fields and classes to get a NullPointerException for illegal field access.

2.3.5 Query language translation

Finding relational records is usually done by expressing a search in SQL. However, SQL is not well suited to express filtering expressions on object graphs.

Most O/R mapping tools thus have some form of non-SQL query facility. Queries are then translated into SQL before being sent to the database for execution. This facility generally comes either in the flavor of another string-based query language that mapping tools parse or directly as some sort of tree of Criteria, Operation, and other objects.

While again simple cases are easy to translate, the more advanced scenarios can rapidly get fairly interesting exercises. For example, if the objects query language allows to sort on attributes of referenced objects, such as JDOQL, then the query translation must find out which tables to join, when to use Outer Joins—if available, and so on.

2.3.6 Referential integrity, deletion, and so on

Pure Java objects exist as long as at least one other object references them. This familiar Java feature is known as **garbage collection.** The days of C and other low-level programming languages with the problem called "dangling pointers" (references to non-existing objects previously de-allocated memory) that led to screaming developers are long gone when using Java.

However, persistent objects can usually be deleted explicitly. Relational databases have long had their own mechanism (referential integrity constraints) to prevent "dangling records," i.e., foreign-key references to non-existent primary keys.

Again, marrying these two different views is an issue that a mapping layer needs to address, be it by explicitly supporting relational referential integrity constraints and performing correct statement ordering if required, maybe by supporting cascaded delete options, or by explicitly not supporting relational referential integrity.

2.3.7 Transparent persistence in O/R mapping

After the Java object model is mapped to columns and tables, an O/R mapping implementation must be able to read and write objects by some means. The basic operations on databases are query, insert, update, and delete. The fundamental idea of transparent persistence is to hide away direct database. A further objective is to minimize database access. This can be achieved by reading known data from a memory cache and updating only modified objects or even fields.

Tools are provided by O/R mapping solutions that create the required code behind field access methods to transparently execute appropriate SQL statements. Although it is possible to implement all the methods by hand, it can be erroneous. Especially when the model changes from one application version to another, or when the model becomes more complex, many project teams spend more time implementing JDBC and SQL mapping code than implementing application logic. A closer look at this problem is taken in the JDO and JDBC chapter.

A brief example of what happens is given below. Assuming that an instance of a Book already exists in memory, the following list shows how an O/R implementation retrieves an Author object. The Java code looks like this:

```
Book book = ... already in memory...;
Author author = book.getAuthor();
```

The O/R mapping implementation might perform something similar to these steps:

- Take the book's AUTHORID.
- Create an Author instance.
- Create a JDBC statement like

  ```
  SELECT * FROM AUTHORS WHERE AUTHORID = id.
  ```
- Copy necessary fields of the result record into the author instance. This step includes a more or less complex attribute to field mapping due to the different type systems.
- Return the author instance.

2.3.8 Identities

Each object in Java has an implicit object identity based on its memory location within the JVM. This identity is used to express relationships with other objects in memory. However, object identity cannot be guaranteed across space and time. It is thus not suitable for identifying relationships in a datastore.

A relational database's view of relationships is based on primary and foreign keys. It is essential that a one-to-one association exists between objects in memory and records in a database, or else it becomes impossible to identify the appropriate rows for query, update, and delete operations. Nobody would expect that a call to book.getAuthor() returns different objects in two adjacent calls. Therefore, the pseudo code from the above example must be extended to satisfy the constraint:

- Take the book's AUTHORID.

- Check if an instance corresponding to AUTHORID already exists in memory. If yes, return that instance; else continue.
- Create an Author instance.
- Create a JDBC statement like

  ```
  SELECT * FROM AUTHORS WHERE AUTHORID = id.
  ```

- Copy the necessary fields of the result record into the author instance.
- Return the author instance.

2.4 Rolling Your Own Persistence Mapping Layer

The preceding sections explored some object-relational specific mapping issues. The following section looks into some topics that are not strictly O/R-related, but apply to any persistence framework that maps a Java object from and to an external datastore, be it relational, an object database, XML repositories, or any other imaginable data store.

While again such issues are addressed by existing products and do not need to be a source of concern for application-level developers, an understanding, or at least awareness of it may be of interest.

Furthermore, such issues have to be addressed if you want to realize your own JDO-like persistence layer with good performance characteristics. This section concludes with a short "build versus buy" discussion.

2.4.1 Caching

Now that objects can be instantiated on demand and unique references are returned for unique data in the database, it is a small step to realize further caching. Why should a call to getAuthor in the above example do anything at all, if the author object was previously retrieved? A reference to the in-memory object can be returned again. The following pseudo code illustrates the idea:

- Check whether the book instance already contains valid data. If yes, return the Author reference.
- Take the book's BOOKID.
- Create a JDBC statement like

  ```
  SELECT * FROM BOOKS WHERE BOOKID = id.
  ```
- Copy the necessary fields of the result record into the book instance.
- Create an Author instance.
- Copy the AUTHORID from the result record into the Author instance.
- Mark the Author instance so that it is loaded when a get() method is called.
- Return the author instance.

In this case, the on-demand loading of objects is split into instantiation of objects that are just placeholders for corresponding database identities and the actual data retrieval from the database.

Whenever objects are modified, the actual update operation to the database can be delayed until all work related to the object graph is done. This would save unnecessary update operations on the same object or dependent objects. Diverse solutions to find out about changes to objects of a graph of objects exist and are realized in O/R mapping tools. One can copy objects and compare each original object with the application object through Java reflection. Set and get methods for fields may be created by a tool to track modifications and load data on demand.

What all these solutions have in common is that the application has the notion of some point in time when all modifications have to be copied out to the datastore. Obviously, the same is true for reading. At some point, the cached objects need to be thrown away (cache eviction) and data has to be reloaded. Various strategies exist, usually linked to some time or memory limit exhaustion, and often implemented in Java by using SoftReferences, WeakReferences, and the like.

2.4.2 Transactional database access and transactional objects

The above-mentioned cache consistency is directly connected to transactional database access. Developers trust a database system and rely on ACID properties. A transactional system fulfilling these properties guarantees that it can commit some pieces of work in a single step. Either everything or nothing is done, independent of other parts of the system or separated from them. All work is committed persistently so that the system can fail afterward without data loss.

A typical workflow with transactional access looks like the following:

- Start a new transaction.

- Look up a first instance as a root of the object graph by using a query or provided application identity.
- Navigate through the object graph.
- Modify or delete objects.
- Add new objects.
- Commit or abort the transaction.

Beyond the "simple" usage of underlying datastore transactions as just described, some persistence layers (and JDO certainly does, as we'll see later) add a real **transactional objects** mechanism. Pure Java objects clearly are not transactional. For example, imagine that a transaction is started, an object's attributes are changed, and the transaction is then aborted. The Java object would keep the old, now invalid values from within a transaction that was rolled back. In a persistence framework with true transactional object support, the values of the attributes of the object itself would "roll back" as well.

2.4.3 Locking

Another issue, somewhat related to correct transaction handling in persistence layers, is **locking.** What happens in the case of concurrent read and write access to the same object? Different strategies exist, as we examine in the transaction chapter, but whatever the details, a persistence layer again needs to make an extra effort for this purpose.

For example, to support optimistic locking on a relational database, an extra timestamp column or incremental numeric counter column on each table is often introduced, kept up to date, and checked. For pessimistic locking, the "... FOR UPDATE" SQL syntax (non-standard) may have to be used. Such underlying locking implementations must be implemented transparent to the application logic and translated to the respective API.

2.4.4 Arrays, sets, lists, and maps

The Java language and more so the standard java.util.Collection library have various ways of modeling associations between objects, from simple Arrays, to Sets, to Lists, and finally Maps.

Different datastores have different and sometimes limited direct support for such models. Relational datastores—for example, out-of-the-box (with the ubiquitous "join table in the middle")—really model only what a correct Java model would express as a Set, while in XML-to-object mapping, a collection of objects is always ordered, thus delivering what Java would model with a List type.

A persistence layer needs to provide mapping of such object collections in order to ensure that associations work exactly like their Java interface contract requests and perform well for large data.

2.4.5 Performance and efficiency

Questions of performance and efficiency arise with many aspects of the implementation of a persistence layer, including the following:

- How are attributes of the persistent class read and written? Access to fields of objects through reflection is slow. For example, JDO implementations generally do not actually use Java Reflection. Other persistence layers have different approaches.
- How are large query results (collection of objects) handled? Is everything loaded at once, or is some sort of Cursor or Iterator architecture used? Is pre-fetching a certain number of initial records possible (Like SQL's "OPTIMZE FOR x ROWS" or Oracle's "/* FIRST_ROWS */" syntax)?
- How are large associations between two objects handled? Are partial object reading with basic lazy loading of referenced objects and "non-default fetch group" attributes supported? Is a Set or List always fully loaded into memory, or are there optimized framework specific implementations of such interfaces, or are techniques such as Java Proxies implemented? Are such purely performance enhancing optimizations transparent to an application of a framework, or do some sort of ValueObject/DataTransferObject-type classes have to be used explicitly?
- Is there any optimization before interacting with the underlying database? In other words, are "redundant" operations within one transaction (e.g., multiple sets on same attributes) recognized and simplified? Are protocol-specific features utilized, e.g., for O/R mapping through JDBC, reusing of prepared statements, batch statements, and so on.

2.4.6 Building versus buying a persistence framework

The previous pages have outlined the issues that a persistence framework in general and an O/R mapping-based one in particular need to address. The issues presented are not exhaustive, and we have not discussed issues such as complex object lifecycle definition and implementation used to model some of the behavior outlined. In short, while such a framework certainly could be built in-house, it is worth a detailed "build versus buy" cost and effort analysis.

This book cannot attempt to provide estimate figures, but merely outlines what is involved in case of a "build" decision. People who have tried "build instead of buy" usually argue for a "buy instead build" afterward.

Alternatively, a much simpler and less generic framework may be envisioned for "build" in rare situations. As for any major project, clearly defining the scope of a framework to build is essential. Areas that possibly provide room for simplification include the following:

- Not real transparent orthogonal persistence, "no persistence by reachability," but simple explicit "save" or "update" methods (instead makePersistent) on each modified persistent object.
- Not real transactions and transactional objects. No locking. Not a great architecture, but many simple JDBC applications do work like that.
- Limited query capability, no generic query language, and thus no translation. Only special application-specific cases. Alternatively, usage of SQL as query language, with no support for queries via object navigation, and so on.
- No support for inheritance.
- No caching.

2.5 Conclusion

This chapter examined issues around persistence frameworks and how or when they should be used in applications.

O/R mapping should be used when existing database applications have to run together with newly developed applications—for instance, a mature procurement system that needs to be accessed via the Internet. Existing data analysis tools like OLAP-based data mining tools or reporting solutions can be a reason to use O/R mapping as well. To become independent of database vendors may be another advantage of O/R mapping tools. Although it is doubtful that an application runs out-of-the-box on a different database system with expected performance, a mapping layer helps to port such applications in a fraction of the time. The JDO and JDBC chapter in this book examines these issues in more detail.

If the application uses a deeply hierarchical object model, like XML document-oriented or CAD applications, a relational database system can be too slow. Navigational access may result in a number of queries, which can be difficult to tune. In such cases, object-oriented or XML-based databases are a better choice. In the embedded domain, even a database system may be too large because of restricted memory and code space. Nevertheless, there are small, file-based solutions that can be used via the JDO API.

Although a portion of this chapter focuses specifically on O/R-based transparent persistence, many of the issues discussed remain the same for non-relational datastores. For example, even if data is persisted in an object-database or native XML store, the essential questions of transactional objects, caches, and query translations from one language into another remain.

If the object model is simple and stable, using O/R mapping tools may be "overkill" compared to the effort of an application-specific persistence implementation. The decision should also depend on resource consumption, number of parallel accesses, data consistency requirements, and the need for query capabilities. A really bad idea is to design another mapping layer on top of JDO, to implement the application independent of various persistence frameworks like ODMG, JDO, and other, vendor-dependent solutions. The fundamental purpose of JDO, transparent persistence, will be lost in such a case.

Chapter 3 looks at how JDO addresses the issues discussed in this chapter.

Part 2: The Details

Chapter 3. Getting Started with JDO

"The expert at anything was once a beginner."

—Hayes

Using JDO to build an application that creates, reads, updates, and deletes persistent instances of Java classes is easy and requires only some basic knowledge about how JDO works and how to use it. Armed with this knowledge, you can develop your first JDO application and persist instances of Java classes transparently in a datastore. This chapter is a guide to getting started with JDO, providing an understanding of how JDO works and how to use the basic APIs, and exploring some of the more advanced concepts related to using JDO.

This chapter covers these topics:

- How JDO is able to transparently persist instances of Java classes.
- The basic JDO interfaces and how they are related.
- How to define a Java class that can be used with a JDO implementation.
- How to connect to a datastore.
- How to create, read, update, and delete persistent objects.
- The types of fields, system classes, collection classes, and inheritance supported by JDO.
- How to handle exceptions within an application.
- The concept of object identity.
- The different types of identity that can be used.
- How concurrency control is enforced between multiple applications.

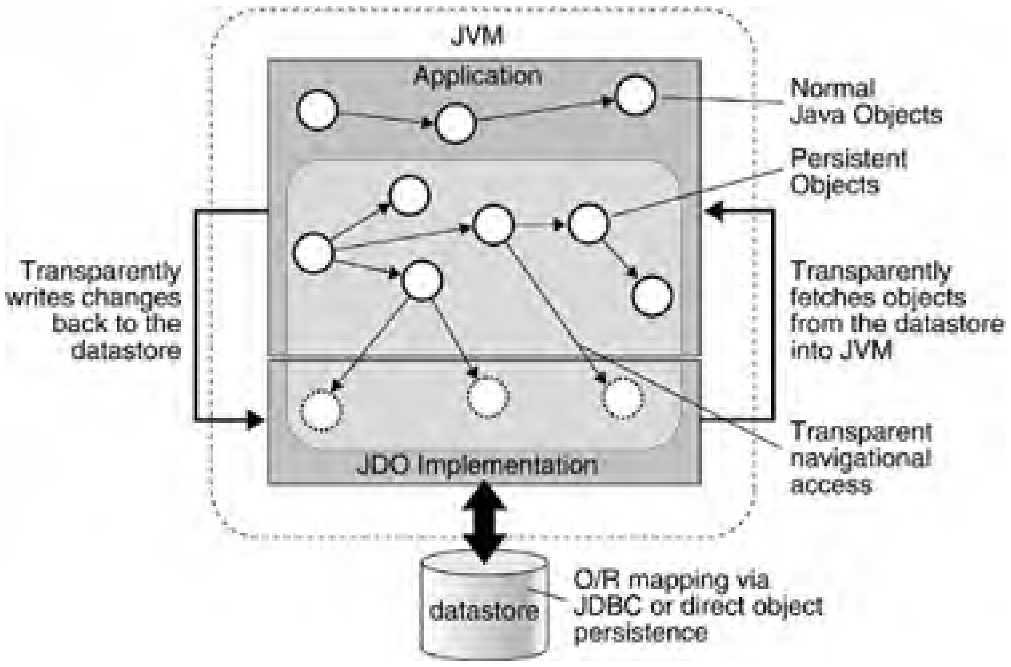
The examples for this chapter can be downloaded from the Internet at www.corejdo.com and are located in the com.corejdo.examples.chapter3 package. In many cases, the code snippets shown are simplified versions of the actual classes to allow the examples to focus only on the relevant concepts.

3.1 How Does JDO Work?

The goal of the JDO is to allow a Java application to transparently store instances of any user-defined Java class in a datastore and retrieve them again, with as few limitations as possible. This book refers to the instances that JDO stores and retrieves as *persistent objects.* From the application perspective, these persistent objects appear as regular, in-memory Java objects. However, the fields of these instances are actually stored in some underlying datastore persistently—all without any explicit action on behalf of the application.

JDO has nothing to do with where methods are executed; it does not provide a means of remote method invocation à la RMI and EJB, nor does it store and execute methods in some datastore. JDO simply specifies how the fields of a persistent object should be managed in-memory, being transparently stored to and retrieved from an underlying datastore. With JDO, methods are invoked on a persistent object by an application as per any regular in-memory Java object. Figure 3-1 provides a schematic of how JDO works.

Figure 3-1. JDO runtime environment.


JVM
Application
Normal Java Objects
Persistent Objects
Transparently writes changes back to the datastore
Transparently fetches objects from the datastore into JVM
Transparent navigational access
JDO Implementation
datastore
O/R mapping via JDBC or direct object persistence

The JDO implementation and the application run together in the same JVM. The application delegates to the JDO implementation to retrieve the fields of persistent objects as needed. The JDO implementation tracks modifications to the fields and writes these changes back to the datastore at the end of the transaction. The JDO implementation is responsible for mapping the fields of the persistent objects to and from memory and the underlying datastore.

JDO achieves transparency of access by defining a contract to which a class must adhere. Any class that implements this contract can then be used with any JDO implementation. JDO requires that a JDO implementation ensure that any class that adheres to the JDO persistence-capable contract can be used with any JDO implementation, without recompilation.

The ability to run a JDO application with any JDO implementation is akin to using JDBC, a JDBC application can be run "as is" using JDBC drivers from different vendors and even using different relational databases. In fact, JDO is somewhat better than this, because with JDBC an application is still prone to differences in SQL support across different databases. With JDO, SQL is not directly exposed. Although a JDO runtime may itself use JDBC to access a relational database as its datastore, it is the responsibility of the JDO implementation to resolve the differences in SQL support across databases.

Even better, unlike SQL, a JDO application can work "as is" across different types of databases, not just relational: object databases, flat-files, and so on. All that is required is a JDO implementation that supports the datastore.

The JDO specification defines the persistence-capable contract as a Java interface, called PersistenceCapable, and a programming style that the class implementation must follow. A class that adheres to this contract is referred to as being "persistence-capable."

A class is said to be persistence-capable if its instances can be stored in a datastore by a JDO implementation. However, just because a class is persistence-capable doesn't mean that all its instances have to be persistent; it just means the option is there. Whether a particular instance is persistent depends on the application. It's similar to Java serialization. Just because a class implements the Serializable interface doesn't mean that all its instances have to be serialized.

However, the intention of JDO is not to expect the developer to worry about making a class persistence-capable; it's a tedious job better left to tooling.

You can create a persistence-capable class in three main ways:

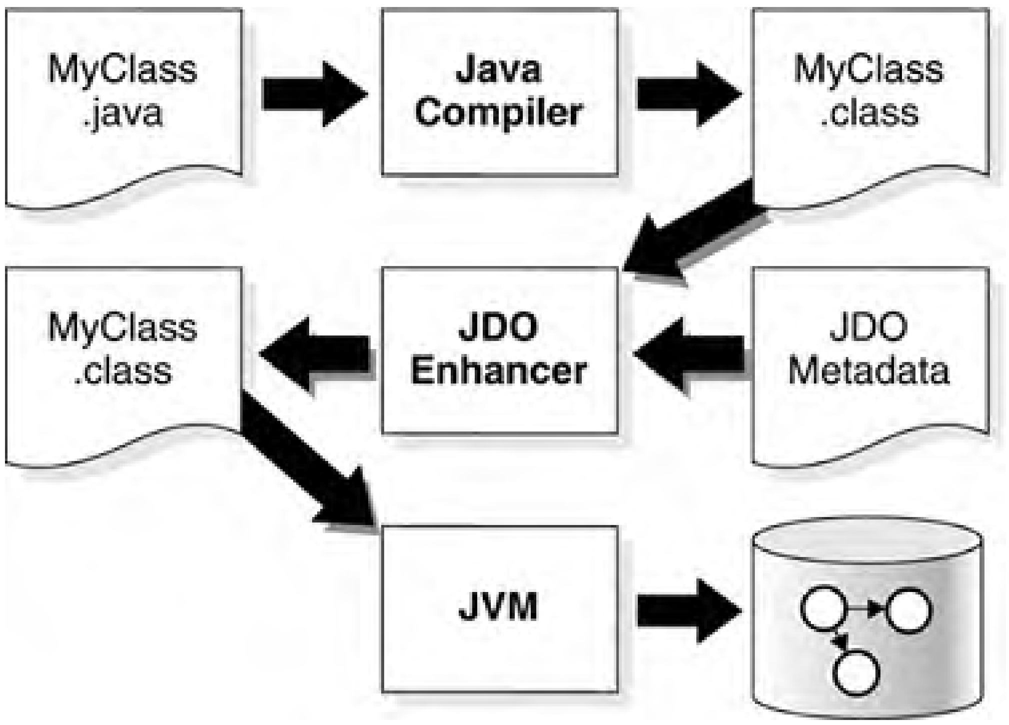
Source code generation: With this method, the source code for a class is generated from scratch. This approach works well if the object model is defined in a modeling tool and is being automatically generated, or the datastore schema already exists and the object model can be generated from it. Tools supplied by the JDO implementation would be used to generate source code adhering to the persistence-capable contract. The drawback of this approach is that it won't work for existing classes and won't appeal to those who like to write their own code.

Source code preprocessing: With this method, existing source code is preprocessed and updated. This approach works well if the source code for a class is available. Tools supplied by the JDO implementation would be used to read the original source code and update it to adhere to the persistence-capable contract. The drawback of this approach is that it won't work unless the original source code is available, but it does have the benefit that a developer can write his or her own source code. Typically, the preprocessing is a precompilation step in the build process, and the generated code may be kept to aid in debugging.

Byte code enhancement: With this method, the compiled Java byte code for a class is enhanced directly. This approach works well even if the source code is not available. Tools supplied by the JDO implementation would be used to read a class file and insert additional byte code directly to make the class adhere to the persistence-capable contract. This approach has the benefit of being completely transparent to the developer, and the enhancement is simply a post-compilation step in the build process. Although the JDO specification requires that an enhanced class still function correctly when debugged against the original source code, some developers may be distrustful if they can't see the actual code for what has been changed (although they could, of course, always decompile the enhanced class file afterward).

Byte code enhancement is the approach used by the JDO reference implementation available from SUN Microsystems, and the enhancement tool is available for any developer to use. Some JDO implementations may provide their own enhancement tools also. Figure 3-2 provides a schematic of how the byte code enhancement process works.

Figure 3-2. The byte code enhancement process.



The Java classes are compiled using a Java compiler to generate class files. The byte code enhancement tool reads the class files along with the JDO metadata for the classes (this metadata is explained in Section 3.3.1) and either updates the existing class files or creates new ones. The "enhanced" class files are then loaded by a JVM along with the JDO implementation and the application. The application can then use JDO to store instances of the persistence-capable classes in the datastore.

[Team LiB]

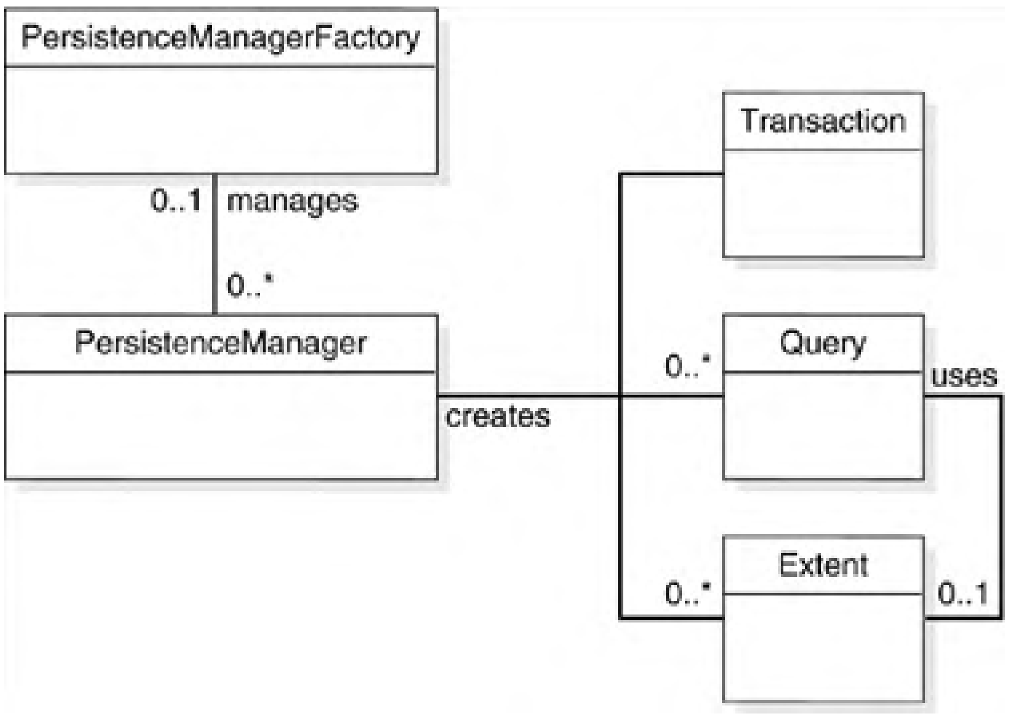
3.2 The JDO Basics

The JDO specification defines 20 or so classes and interfaces in total, but a developer needs to know only five main classes (which are actually Java interfaces):

- PersistenceManagerFactory
- PersistenceManager
- Extent
- Query
- Transaction

Figure 3-3 provides a simplified class diagram showing how these interfaces are related:

Figure 3-3. A simplified JDO class diagram.



A **PersistenceManagerFactory** is used to get a PersistenceManager instance. There is a one-to-many relationship between the PersistenceManagerFactory and the PersistenceManager. A PersistenceManagerFactory can create and manage many PersistenceManager instances and even implement pooling of PersistenceManager instances.

A **PersistenceManager** embodies a connection to a datastore and a cache of in-memory persistent objects. The PersistenceManager interface is the primary means by which the application interacts with the underlying datastore and in-memory persistent objects.

From a PersistenceManager, the application can get one or more **Query** instances. A Query is how the application can find a persistent object by its field values.

In addition to a Query, the application can get an **Extent** from a PersistenceManager. An Extent represents all the instances of a specified class (and optionally subclasses) stored in the datastore and can be used as input to a Query or iterated through in its own right.

A **Transaction** allows the application to control the transaction boundaries in the underlying datastore. There is a one-to-one correlation between a Transaction and a PersistenceManager; there can be only one ongoing transaction per PersistenceManager instance. Transactions must be explicitly started, and either committed or aborted.

3.3 Defining a Class

The first step in using JDO is to create a persistence-capable Java class. For the purposes of the examples in this book, it is assumed that byte code enhancement is being used, so all that is needed is a Java class. The following code snippet shows the code for a class called Author, which represents the author of a book:

```
package com.corejdo.examples.model;

public class Author {

  private String name;

  public Author (String name) {

      this.name = name;
  }

  protected Author () {}

  public String getName () {

    return name;
  }

  public void setName (String name) {

    this.name = name;
  }
}
```

This class is relatively simply with just one string field. JDO can, of course, support much more than this; see Section 3.9 for more details on what is supported.

Note that the fields of the class are declared as private. Unlike other approaches to object persistence, JDO does not require fields of classes to be declared public. In fact, JDO places few constraints on the definition of the Java classes themselves.

There is one requirement, however; JDO requires that a persistence-capable class have a no-arguments constructor. This constructor does not need to be declared public, it just needs to be accessible to the class itself and to any potential subclasses. If there are no subclasses, it can be declared as private. However, if there are subclasses, it should be declared as protected or packaged so that it is accessible from the subclasses.

The no-args constructor is used by the JDO runtime to create empty instances of the class prior to retrieving its fields from the datastore.

To make the Author class persistence-capable, it must be compiled and passed to the byte code enhancement tool. Before this can be done, however, an XML file must be created that contains the JDO metadata for the class.

3.3.1 JDO metadata

For each persistence-capable class, JDO requires additional metadata. This metadata is specified in an XML file. The metadata is primarily used when making a Java class persistence-capable (by byte code, source enhancement, or code generation). However, it is likely also to be used by a JDO implementation at runtime.

By convention, these files have a .jdo suffix. Each class can have a metadata file, in which case the file is named after the class itself, or there can be one metadata file per package, in which case the file is named package.jdo. The metadata files should be accessible at runtime as resources via the class loader that loaded the persistence-capable Java class.

For the Author class, a file called Author.jdo would be located in the same directory as the Author class file. This allows the byte code enhancement process and JDO implementation to easily locate it as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author"/>
  </package>
</jdo>
```

Alternately, the metadata for this class could be located in a file called package.jdo that would be located in the com/corejdo/examples/model directory. This file could contain metadata on all the classes in the "model" package (and subpackages), not just the Author class. Appendix B provides a detailed overview of the syntax for the JDO metadata. For this class, however, all that is needed is to specify the package name and class name.

When the Author source and metadata files are ready, they can be passed to the byte code enhancement tool. This takes care of implementing the persistence-capable contract. As an example, if using the JDO reference implementation (RI), the Author class would be compiled as normal using javac and then the RI byte code enhancement tool would be invoked from the command line as follows:

```
javac Author.java
java com.sun.jdori.enhancer.Main Author.class Author.jdo
```

By default, the RI enhancement tool updates the Author byte code in place, and the original Author.class file is replaced with an enhanced one.

The initial JDO 1.0 specification didn't actually mandate the location and name of the XML metadata files. To aid in portability, the 1.0.1 maintenance revision to the specification changed this to what has just been described.

Prior to this revision, the name of a metadata file that contained multiple classes was the package name itself with a .jdo suffix, and it was located in the directory containing the package directory. In the Author example above, it would have been called model.jdo and would have been located in the com/corejdo/examples/ directory. Some JDO implementations may still use this naming convention.

3.3.2 Mapping a class to the datastore

JDO does not define how to specify how the fields of a class should be mapped to the underlying datastore—for example, if using a relational database, what tables and columns should be used—nor does it define how the datastore schema should be created in the first place.

All this is datastore and JDO-implementation specific. A JDO implementation provides the necessary tools to allow a developer to define a mapping between the persistence-capable classes and the underlying datastore and to define the required datastore schema.

As an example, a JDO implementation on top of a relational database might use the "vendor-extensions" element in the JDO metadata for a class to specify table and column names or define how field values should be stored. An implementation using an object database or a flat file might require only the Java classes and nothing more.

3.4 Connecting to a Datastore

After a persistence-capable class has been defined, the next step is to implement an application that actually uses it. Any JDO application must get a connection to the underlying datastore before it can do anything. This is done by configuring an instance of PersistenceManagerFactory and then using it to get a PersistenceManager.

PersistenceManagerFactory is just a Java interface; JDO does not specify how an actual PersistenceManagerFactory instance should be created. However, it does provide a helper class called JDOHelper that, among other things, provides a common bootstrap mechanism to create a PersistenceManagerFactory instance based on a specified set of properties:

```
static public PersistenceManagerFactory
  getPersistenceManagerFactory(Properties props)
```

The getPersistenceManagerFactory() method takes a set of properties as input and constructs a concrete instance of a PersistenceManagerFactory. The one required property is javax.jdo.PersistenceManagerFactoryClass, which is used to identify the name of an implementation's PersistenceManagerFactory class. All other properties are simply passed to the implementation's class and are implementation specific. These include things like connection URL, name, and password.

At a minimum, every JDO implementation must support bootstrapping via JDOHelper. Additionally, an implementation might provide its own PersistenceManagerFactory constructors that can be used to directly construct a PersistenceManagerFactory instance. It may also provide PersistentManager constructors to avoid having to go through a PersistenceManagerFactory at all. From a portability standpoint, these approaches are all JDO implementation specific.

PersistenceManagerFactory also implements Serializable. This allows a previously configured PersistenceManagerFactory instance to be located via JNDI.

After a PersistenceManagerFactory instance has been created, the getPersitenceManager() method can be used to get a PersistenceManager instance:

```
public PersistenceManager getPersistenceManager()
```

When an application is finished with a PersistenceManager instance, it should close it by calling the close() method:

```
public void close()
```

Connection Pooling

JDO leaves it up to the JDO implementation as to whether its **PersistenceManagerFactory** implements connection pooling. If implemented, the **close()** method on **PersistenceManager** may not physically close anything, but may instead return the **PersistenceManager** instance to the **PersistenceManagerFactory** to be reused.

The following code snippet taken from MakeConnectionExample.java shows how to get a PersistenceManager instance from a PersistenceManagerFactory and then close it:

```
import java.util.Properties;
import javax.jdo.*;

public class MakeConnectionExample {

  public static void main(String[] args) {

    Properties properties = new Properties();

    properties.put(
      "javax.jdo.PersistenceManagerFactoryClass", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionURL", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionUserName", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionPassword", "XXX");
    PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(properties);

    PersistenceManager pm = pmf.getPersistenceManager();

    /* Do something interesting */

    pm.close();
  }
}
```

Of course, before this example can be used, the "XXX" strings would need to be replaced with values appropriate to the JDO implementation being used.

As an alternative to hard coding the properties as in the previous example, it is generally better to specify them as system properties or read them from a file. The code for this chapter contains examples of how these two alternative approaches would work. See the following for more information:

- MakeConnectionFromSystemPropertiesExample.java
- MakeConnectionFromFileExample.java

For a full list of the standard JDO property names that can be specified, see Chapter 5.

3.5 Creating an Object

After a PersistenceManager has been retrieved, it is possible to actually begin a transaction and do something interesting with the datastore. To begin a transaction, get the current Transaction instance from the PersistenceManager:

```
public Transaction currentTransaction()
```

And call the begin() method on Transaction to start a datastore transaction:

```
public void begin()
```

Typically, any interaction with a datastore is done within the context of a transaction. A transaction simply marks the beginning and end of a unit of work, whereby all changes made either happen or do not happen.

A transaction is also the mechanism that a datastore uses to enforce concurrency control between multiple applications accessing the datastore at the same time. Not all datastores support concurrent access, but those that do ensure that each ongoing transaction is isolated from the others. For more details on concurrency control and transactions, see Section 3.14.

Instances of persistence-capable classes can now be created and made persistent. They are constructed in the same manner as normal Java and, once constructed, need to be passed to the makePersistent() method on PersistenceManager to notify the JDO implementation that the instances need to be persisted:

```
public void makePersistent(Object pc)
```

The instances won't actually be stored in the datastore until the transaction is committed, which is done using the commit() method on Transaction:

```
public void commit()
```

Alternatively, a transaction can be rolled back—in this case, the instances won't become persistent and won't be stored in the datastore. This can be done using the rollback() method on Transaction, instead of commit():

```
public void rollback()
```

The following code snippet taken from CreateExample.java shows how to create an instance of the Author class and make it persistent in the datastore:

```
Transaction tx = pm.currentTransaction();

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();
```

When they are successfully committed, the fields of the new Author instance are stored in the datastore, and the in-memory instance becomes what is termed "hollow." This means that its fields are cleared so that when the instance is used in the next transaction, they will be retrieved again from the datastore.

Object Lifecycle

JDO defines a number of lifecycle states for persistent objects ("hollow" being one of them). These states are used by the JDO implementation to manage the in-memory persistent objects. While it is not necessary to really understand the JDO object lifecycle (it's of primary concern only to the JDO implementation), an appreciation of what happens under the covers is always a good idea. See Section 3.13 for more details.

Not every instance that needs to be persistent has to be passed to makePersistent(). JDO defines that any instance of a persistence-capable class that is reachable from a persistent object automatically becomes persistent also.

This means that only the root of a graph of objects actually needs to be passed to makePersistent(); all other objects in the graph implicitly become persistent also. This makes programming much easier.

PREVIOUS NEXT

3.6 Reading an Object

After an instance has been made persistent and its fields stored in the datastore, it can be retrieved again, either within the same application or by a different application.

There are three primary ways of finding an instance in the datastore with JDO—by navigation, via an Extent, or by a Query.

3.6.1 Reading by navigation

Retrieving an instance by navigation is simple. Extending the previous example, this code snippet taken from ReadByNavigationExample.java shows how the Author instance can be retrieved again in the next transaction simply by using it:

```
tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

Underneath the covers, the JDO implementation retrieves the fields for the Author instance from the datastore. The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

Using navigation to retrieve persistent objects from the datastore also applies when one persistent object is referenced from another. Essentially, anytime a persistent object is referenced, the JDO implementation retrieves the instance's fields from the datastore, if it has not already done so within the current transaction.

A common question often arises from the previous example: Why do the fields of the Author instance have to be retrieved from the datastore again after it was initially created?

The answer has to do with transactions. After each transaction ends (either by a commit or rollback), all in-memory persistent objects become hollow. This is because the representation of a persistent object in the datastore could be updated by another application after the transaction ends, making any in-memory values out of sync. For this reason, JDO's default behavior is to clear the fields of each persistent object and retrieve them again if used within the next transaction, thereby ensuring that the in-memory instance remains consistent with the datastore.

JDO has some advanced options that change this default behavior and allow instances to be retained over transaction boundaries or even accessed outside of a transaction altogether. See Chapter 5 for more details.

3.6.2 Reading using an extent

What happens if another application wants to access the new Author instance? Obviously, this application won't have a reference to which it can navigate. In this case, the application can use an Extent to find all Author instances.

Extents

An Extent represents a collection of all instances in the datastore of a particular persistence-capable class and can consist of just instances of the class or instances of all subclasses. An application can get an Extent for a class from a PersistenceManager and then use an Iterator to iterate through all instances, or it can use the Extent as input to a query.

By default, it is possible to get an Extent for any persistence-capable class. If an Extent is not required, then as a potential optimization, it is possible to specify explicitly that an extent is not required in the JDO metadata for the class. Depending on the underlying datastore, this may or may not make any difference.

An application can get an Extent using the getExtent() method on PersistenceManager:

```
public Extent getExtent(Class pc, boolean subclasses)
```

The subclasses flag determines whether the returned Extent represents instances of the specified class only or includes instances of subclasses as well. If true, then it includes instances of all subclasses as well.

The iterator() method on Extent can be used to get an Iterator that iterates through all the instances in the Extent:

```
public Iterator iterator()
```

The following code snippet taken from ReadByExtentExample.java shows how an Extent can be used to iterate through the instances of the Author class:

```
tx.begin();

Extent extent = pm.getExtent(Author.class, true);

Iterator authors = extent.iterator();

while (authors.hasNext()) {

  Author author = (Author) authors.next();

  String name = author.getName();

  System.out.println("Author's name is '" + name + "'.");
}

extent.close(authors);

tx.commit();
```

It is a good practice to use the close() or closeAll() method on Extent when an iterator has been used. The close() method closes an individual iterator, whereas closeAll() closes all iterators retrieved from the Extent:

```
public void close(Iterator iterator)
public void closeAll()
```

Depending on the underlying datastore, this may free resources that were allocated when the Iterator was created (a cursor, for example).

3.6.3 Reading using a query

An Extent is useful if an application wants to retrieve all instances of a class in no particular order. However, if an application is looking for a particular instance that has certain field values, an Extent is not very useful. In this case, a query is needed to find an instance based on the values of its fields.

JDO defines a query language called JDOQL (JDO Query Language). JDOQL is essentially a simplified version of the Java syntax for an "if" statement. The PersistenceManager is used to create a Query instance, and once created, a Query can be executed to return a collection of matching persistent objects. There are approximately nine methods on PersistenceManager to create a Query instance; the most straightforward requires only a Java class and a filter:

```
public Query newQuery(Class cln, String filter)
```

The class specifies the persistence-capable class that the query is for, and the filter, which is just a string, specifies which instances of the class should be returned.

A Query can also be created using an Extent. A query created using a Java class is equivalent to a Query created using an Extent whose subclasses property is true. This means that a Query created from a Java class may include instances of the specified class and subclass (which is generally the desired behavior).

If an application specifically wants to restrict the query to instances of a given class only, then it can create an Extent whose subclasses property is false and use the Extent to create the Query. When executed, the Query includes only instances of the class itself and ignores subclasses.

The basic syntax for a filter is as follows:

```
<field> <operator> <constant>
```

where <field> is the name of the field as declared in the Java class, <operator> is one of the standard Java comparison operators (==, !=, <, >, and so on), and <constant> is the value to be compared against. Much more sophisticated queries than this can be defined; see Chapter 6 for details.

The following code snippet taken from ReadByQueryExample.java shows how to use a query to find a previously created Author by name. For simplicity, it assumes that at least one author with the given name is in the datastore:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close(result);

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

The execute() method on Query is used to execute the query and return the result. A half dozen ways of executing a query exist. The simplest is this:

```
public Object execute()
```

The execute() methods all return java.lang.Object, which must be cast to a java.util.Collection type for JDOQL queries. This collection is the set of instances that matched the specified filter. It can be iterated through in the normal manner to retrieve the persistent objects themselves.

The execute() methods on Query return java.lang.Object rather than java.util.Collection directly to allow for implementation-specific extensions. JDO requires that JDOQL be supported; however, an implementation can optionally support alternative query languages (maybe SQL, OQL, or something proprietary). In this case, the result of executing a query may not be a java.util.Collection; instead, it needs to be cast to something specific to the query language being used.

It is a good practice to use the close() method on Query to release the result returned from a call to execute():

```
public Object close(Object result)
```

Depending on the underlying datastore, this may free resources that were allocated during the execution of the query (a database cursor, for example).

3.7 Updating an Object

It is possible to modify a persistent object in the same manner as normal Java with no additional coding required. Assuming that the class defines methods that can be used to set the fields, then the application simply needs to invoke these methods within a transaction. The JDO implementation automatically detects when a field of a persistent object has been modified and marks the field as "dirty." At commit time, the dirty fields of all the modified persistent objects are written back to the datastore as part of the transaction.

The following code snippet taken from UpdateExample.java shows how to find a previously created Author instance and change its name. To validate that the name has changed in the datastore, it is printed in a new transaction. For simplicity, it assumes that at least one author with the specified name is in the datastore:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close();

author.setName("Sameer Tyagi");

tx.commit();

tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Sameer Tyagi'.
```

If it is decided that the changes should not be written to the datastore and should be discarded instead, then the rollback() method on Transaction can be used.

The following code snippet taken from UpdateExampleWithRollback.java is the same as the previous one, except that a rollback is used instead of a commit to undo the name change:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close(result);

author.setName("Sameer Tyagi");

tx.rollback(); // rollback() rather than commit()

tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

Although the name of the Author instance is modified, because the transaction is rolled back, the modification is not persisted in the datastore. Therefore, when the instance is retrieved in the next transaction, the name is unchanged.

3.8 Deleting an Object

The last basic operation is deletion. Unlike Java, most datastores do not perform automatic garbage collection. Indeed, the semantics of what constitutes garbage when instances are made persistent changes. Just because a persistent object is no longer referenced by any other does not mean that it is garbage. Typically, it can always be retrieved at any time via Query or Extent.

JDO provides a mechanism to explicitly delete persistent objects. The deletePersistent() method on PersistenceManager can be used to permanently remove persistent objects from the datastore.

```
public void deletePersistent(Object pc)
```

Of course, the persistent object isn't actually deleted until the transaction is committed. Rather, the JDO implementation marks the persistent object as deleted and, upon commit, requests that the datastore remove all the deleted persistent objects. The following code snippet taken from DeleteExample.java shows how to delete an instance of Author. For simplicity, it assumes that at least one author instance with the specified name is in the datastore:

```
tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author = (Author) result.iterator().next();

query.close(result);

pm.deletePersistent(author);

tx.commit();
```

Trying to access the fields of a deleted instance within the same transaction results in JDOUserException being thrown.

After the transaction commits, the in-memory instance reverts to being a normal transient Java object, with its fields reset to their default values. If a rollback is called instead, then the deletion is undone and the in-memory instance reverts to being a "hollow" persistent object again. The following code snippet taken from DeleteWithRollbackExample.java uses a rollback rather than a commit and then validates that the persistent object still exists by printing its name:

```
pm.deletePersistent(author);

tx.rollback(); // rollback() rather than commit()

tx.begin();

String name = author.getName();

System.out.println("Author's name is '" + name + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
```

JDO is unlike a JVM, where garbage collection automatically determines whether an instance is still referenced and, if not, releases it. In JDO it is up to the application to ensure that unwanted persistent objects are deleted from the data-store and that a persistent object being deleted is no longer referenced by another.

Depending on the datastore, it may be possible to define constraints to guard against deleting instances that are still referenced, but this is implementation specific. If a persistent object does reference a previously deleted persistent object, then JDOUserException is thrown if the deleted persistent object is later accessed (because it no longer will be found in the datastore).

There is no equivalent of "persistence by reachability" when deleting. Deleting a persistent object deletes only the specified instance; it does not automatically delete any referenced instances.

It is possible for an application to implement a cascading delete behavior by using the JDO InstanceCallbacks interface and implementing the jdoPreDelete() method to explicitly delete referenced instances. See Chapter 5 for more details.

3.9 JDO Object Model

So far, the examples have used a simple Java class to demonstrate basic concepts. JDO can, of course, be used with much more sophisticated classes than this. In addition to supporting fields of primitive Java types and system classes like String, JDO can handle object references, Java collections, and inheritance of both classes and interfaces.

This section outlines what can and can't be used when developing a persistence-capable class, including the following:

- The basic field types that can be used within a persistence-capable class.
- Using references to persistence-capable classes, non-persistence-capable classes, and interfaces.
- Using standard Java collections.
- Using arrays.
- Support for inheritance.
- The class and field modifiers that can be used.
- What JDO doesn't support.

3.9.1 Basic types

A persistence-capable class can have fields whose types can be any primitive Java type, primitive wrapper type, or other supported system immutable and mutable classes. Table 3-1 provides a summary of these basic types.

If a persistence-capable class has fields of any of the above types, then by default these fields are persisted in the datastore (providing they are not declared as transient). If a field is a reference to a persistence-capable class, then by default these fields are persisted also. Fields of any other types are not persisted in the data-store by default and need explicit metadata to indicate that they should be persisted. The types of these fields might be a reference to the Java interface, a java.lang.Object, a non-supported system class/interface, or a non-persistence-capable class.

Table 3-1. Supported Basic Types

Primitive Types	Wrapper Classes	Supported System Interfaces	Supported System Classes
boolean	java.lang.Boolean[¥]	java.util.Collection[§]	java.lang.String[¥]
byte	java.lang.Byte[¥]	java.util.List[*][§]	java.math.BigDecimal[¥]
char	java.lang.Character[¥]	java.util.Map[*][§]	java.math.BigInteger[¥]
double	java.lang.Double[¥]	java.util.Set[§]	java.util.Date
int	java.lang.Integer[¥]	javax.jdo.PersistenceCapable	java.util.Locale[¥]
float	java.lang.Float[¥]		java.util.ArrayList[*][§]
long	java.lang.Long[¥]		java.util.HashMap[*][§]
short	java.lang.Short[¥]		java.util.HashSet[§]
			java.util.Hashtable[*][§]
			java.util.LinkedList[*][§]
			java.util.TreeMap[*][§]
			java.util.TreeSet[*][§]
			java.util.Vector[*][§]

[¥] See Section 2.9.3 for more details on using collection classes.

[§] Immutable class (Its value can't be changed.)

[*] Support is an optional JDO feature.

JDO defines both mandatory and optional features. A JDO implementation is said to be compliant if it supports all the mandatory features defined by the JDO specification. An implementation may support one or more optional features, in which case it must adhere to what is defined for the optional feature by the JDO specification if it is to be JDO compliant.

From a portability standpoint, an application should not rely on support of an optional feature. But should a particular feature be required, then at least the application is still portable across implementations that support the feature or set of features used. See Chapter 5 for more details on specific optional features.

3.9.2 References

A persistence-capable class can have fields that reference instances of other persistence-capable classes. The following code snippet shows a Book class that has an "author" field, which is a reference to an instance of Author:

```
public class Book {

  private String name;
  private Author author;

  public Book(String name, Author author) {

    this.name = name;
    this.author = author;
  }

  protected Book() {}

  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  public Author getAuthor() {

    return author;
  }

  public void setAuthor(Author author) {

    this.author = author;
  }
}
```

The JDO metadata for this class in Book.jdo is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Book"/>
  </package>
</jdo>
```

The following code snippet taken from CreateWithReachabilityExample.java shows persistence by reachability at work. Both an Author and a Book instance are created. The Book instance is added to the Author instance, and the Author instance is explicitly made persistent, so the Book instance is automatically made persistent because it is reachable from the Author instance:

```
tx.begin();

Author author = new Author("Keiron McCammon");

Book book =
  new Book("Core Java Data Objects", "0-13-140731-7");

author.addBook(book);

pm.makePersistent(author);

tx.commit();
```

As well as supporting references to persistence-capable classes, JDO also supports references to Java interfaces and even java.lang.Object. Because fields of these types are not persistent by default, additional metadata is required to denote that the field should be persistent. Any referenced instances should still be instances of a persistence-capable class. (Support for references to non-persistence-capable classes is a JDO optional feature.)

The following code snippet shows a revised Book class that has an "author" field, which now is of type java.lang.Object instead of Author:

```
public class Book {

  private String name;
  private Object author; // Uses Object rather than Author

  /* Rest of code not shown */
}
```

The major difference here compared to the previous example is the metadata in Book.jdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Book">
      <field
        name="author"
        persistence-modifier="persistent"/>
    </class>
  </package>
</jdo>
```

Because the field's type is java.lang.Object, it is not persistent by default. It must be explicitly denoted as persistent in the metadata for the class. The same would be true if it were a reference to an interface.

A JDO implementation may optionally support references to instances that do not belong to a persistence-capable class. If not supported, a Java class cast exception would be thrown at the time of assignment. (An application intended to run against multiple JDO implementations should not rely on support for persisting non-persistence-capable classes.)

If references to non-persistence-capable classes are supported, then these instances exist in the datastore only as part of the referencing persistent object. It is the responsibility of the application to inform the JDO implementation when an instance of a non-persistence-capable class is modified because the JDO implementation is unable to automatically detect when a field of a non-persistence-capable class is changed.

The JDO implementation can't detect when an instance of a non-persistence-capable class is modified because the class doesn't adhere to the persistence-capable programming style. A persistence-capable class automatically informs the JDO implementation before a field is changed (this is what the byte code enhancement process takes care of), but a non-persistence-capable class does not.

If the JDO implementation is not explicitly notified of changes to non-persistence-capable classes, then the modifications do not get written to the datastore on commit. The easiest way to notify a JDO implementation that a non-persistence-capable instance has been modified is to use the makeDirty() method on JDOHelper:

```
static void makeDirty(Object pc, String field)
```

The first argument is the persistent object that references the non-persistence-capable instance. The second argument is the field name of the reference.

JDOHelper

JDOHelper was previously introduced as a way to bootstrap a JDO application and get an instance of a

PersistenceManagerFactory. As well as this, JDOHelper has a number of additional methods that act as shortcuts to the various JDO APIs.

The following code snippet shows a non-persistence-capable class called Address that can be used to store the address of an Author:

```
import java.io.Serializable;

public class Address implements Serializable {

  private String street;
  private String zip;
  private String state;

  public Address(String street, String zip, String state) {

    this.street = street;
    this.zip    = zip;
    this.state  = state;
  }

  /* Additional getters and setters not shown */

  public void setZip ( String zip ) {

    this.zip = zip;
  }
}
```

Exactly how a JDO implementation manages the fields of non-persistence-capable classes is undefined by JDO. Some implementations may require that a class implement Serializable or follow the JavaBean pattern, or may require that its fields be declared public.

The following code snippet shows a revised Author class that contains a reference to an Address and shows how to set the zip code of the address. The method first calls makeDirty() to inform the JDO implementation that the address is being be modified:

```
import javax.jdo.*;

public class Author {

  private String  name;
  private Address address;

  public Author(String name, Address address) {

    this.name    = name;
    this.address = address;
  }

  protected Author () {}

  public void setZip(String zip) {

    JDOHelper.makeDirty(this, "address");

    address.setZip(zip);
  }
}
```

JDOHelper.makeDirty()

It is a best practice to call **makeDirty()** before making any modifications to a non-persistence-capable instance. Depending on the underlying datastore, the call to **makeDirty()** may result in a concurrency conflict indicating that changes aren't allowed at that time.

The revised JDO metadata for this class in Author.jdo is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author">
      <field
          name="address"
          persistence-modifier="persistent"/>
    </class>
  </package>
</jdo>
```

Because the address field is a reference to a non-persistence-capable class, it is necessary to specifically specify that the field should be persistent.

When using non-persistence-capable instances, it is best to modify them only through methods on the referencing persistent object. This way, the persistent object can call makeDirty() on itself before making any changes. If a non-persistence-capable object is passed by reference and modified elsewhere, it becomes hard for the application to ensure that the referencing persistent object is notified of the changes, in which case any changes are ignored.

3.9.3 Collection classes

A persistence-capable class can have fields that reference the standard Java collection classes. Instances of these collection classes exist in the datastore only as part of the referencing persistent object. At a minimum, JDO mandates support for java.util.HashSet; support for ArrayList, HashMap, Hashtable, LinkedList, TreeMap, TreeSet, and Vector is optional. However, most JDO implementations support all these collection classes.

Optional Features

To determine whether an implementation supports the optional collection classes, use the **supportedOptions()** method on PersistenceManagerFactory. See Chapter 5 for more details.

The following code snippet shows a revised Author class that uses a java.util.HashSet to hold the books that an author has written:

```
import java.util.*;

public class Author {

  private String name;
  private Set books = new HashSet();

  public Author (String name) {

    this.name = name;
  }

  protected Author () {}

  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  public void addBook(Book book) {

    books.add(book);
  }

  public Iterator getBooks() {

    return books.iterator();
  }
}
```

The revised JDO metadata for this class in AuthorWithBooks.jdo is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author">
      <field name="books">
        <collection
            element-type=
              "com.corejdo.examples.model.Book"/>
      </field>
    </class>
  </package>
</jdo>
```

Because this class uses a Java collection, additional metadata can be used to declare the class of the instances that are stored in the collection. Because Java collections are un-typed, it is impossible for a JDO implementation to determine this information from the Java class itself.

Additional Metadata

The additional metadata shown for the "books" field is optional; it does not need to be specified. If not specified, the JDO runtime assumes that the collection may contain a reference to any persistent object, as does normal Java.

Even though it is optional, it is best practice to define the element type of a collection because it potentially allows the JDO runtime to optimize how it handles the field both in-memory as well as in the datastore.

Unlike user-defined, non-persistence-capable classes, JDO mandates that the supported collections automatically detect changes and notify the referencing persistent object. A JDO implementation does this by replacing instances of system-defined collection classes with instances of its own equivalent classes that perform this detection. For a new persistent object, this replacement happens during the call to makePersistent(); for existing persistent objects, this occurs when they are retrieved from the datastore. All this is completely transparent to the application. It just means that an application should not reply on a collection being a particular concrete Java class (java.util.HashMap, for example).

3.9.4 Arrays

Support for a persistence-capable class with fields that use Java arrays is optional in JDO. If supported, arrays are similar in nature to user-defined, non-persistence-capable classes except that a JDO runtime may automatically detect changes and notify the referencing persistent object of the modification. For portability, an application should assume responsibility for notifying the referencing persistent object when an array is modified.

With non-persistence-capable classes, it is a best practice to modify arrays only through methods on the referencing persistent object. This way, the referencing persistent object can call makeDirty() on itself before making any change. If an array is passed by reference and modified elsewhere, it becomes hard for the application to ensure that the owning persistent object is notified of the changes, in which case any changes are ignored.

The following code snippet shows a revised Author class that uses an array to store the books that an author has written. As a simplification, the array holds only a single book; in real life, any sized array could have been used:

```
import java.util.*;

public class Author {

  private String name;
  private Book   books[] = new Book[1];
  public Author(String name) {

    this.name = name;
  }

  protected Author()

  public String getName() {

    return name;
  }

  public void setName(String name) {

    this.name = name;
  }

  public void addBook(Book book, int i) {

    JDOHelper.makeDirty(this, "books");

    books[i] = book;
  }

  public Book getBook(int i) {

    return books[i];
  }
}
```

The revised JDO metadata for this class in Author.jdo is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author"/>
  </package>
</jdo>
```

Because the field is an array of a persistence-capable class, no additional metadata is required and the array field is persistent by default.

3.9.5 Inheritance

JDO supports inheritance of both interfaces and classes. Because Java interfaces don't define any fields, a JDO implementation needs to do nothing, so Java interfaces can be used with no real consideration as far as JDO is concerned.

When using class inheritance (abstract or otherwise), JDO needs to be aware of the implementation hierarchy, because each class can define its own fields. In the usual case, where all classes in the inheritance hierarchy are themselves persistence-capable, it is straightforward. The only requirement is to denote the persistence-capable superclass of a class in the JDO metadata.

The following code snippet shows a class that extends the Book class:

```
public class RevisedBook extends Book {

  private Book original;

  public RevisedBook(String name, Book original) {

    super(name);

    this.original = original;
  }

  protected RevisedBook() {}

  public Book getOriginal() {

    return original;
  }
  }

  public void setOriginal(Book original) {

    this.original = original;
  }
}
```

The JDO metadata for this class in RevisedBook.jdo is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class
       name="RevisedBook"
           persistence-capable-superclass="Book"/>
  </package>
</jdo>
```

The additional metadata denotes the persistence-capable superclass of the class. The persistence-capable-superclass name uses the Java naming rules: If no package is included in the name, the package name is assumed to be the same package as the persistence-capable class.

It is possible for a persistence-capable class to extend a non-persistence-capable class. In this case, any fields defined by the non-persistence-capable class are ignored by JDO and are not persisted in the datastore. It is also possible to have a persistence-capable class extend a non-persistence-capable class, which itself extends a persistence-capable class. In all cases, any fields of a non-persistence-capable class are ignored by JDO; it is the responsibility of the application to manage these fields.

It is best to make all classes in an inheritance hierarchy persistence-capable. This avoids added complications. If this isn't possible and non-persistence-capable classes have to be used, then a persistence-capable subclass should implement the InstanceCallbacks Interface and use the jdoPostLoad(), jdoPreStore(), and jdoPreClear() methods to manage the fields of the non-persistence-capable classes explicitly.

3.9.6 Modifiers

JDO supports all Java class and field modifiers—private, public, protected, static, transient, abstract, final, synchronized, and volatile—with the following caveats:

- A field declared as transient is not persisted it in the datastore by default. This default behavior can be overridden by explicitly declaring the field as persistent in the class's metadata.
- A field declared as final cannot be persisted in the datastore; its value can be set only during construction.
- A field declared as static cannot be persisted in the datastore; it has meaning only at the class level within the JVM.

Apart from static and transient, use of field modifiers is essentially orthogonal to the use of JDO.

3.9.7 What isn't supported

JDO supports most of what can be defined in Java. The major exception is that JDO can't make Java system classes persistence-capable; rather, it mandates support for specific system classes (as already outlined) as fields of a persistence-capable class. Also, classes that depend on an inaccessible or remote state, such as those that use JNI or those that extend system-defined classes, cannot be made persistence-capable and cannot be used as fields of persistence-capable classes.

3.10 Exception Handling

All the examples so far have ignored the possibility of exceptions. JDO defines a set of exceptions that can be thrown by the JDO implementation. All JDO exceptions are declared as runtime exceptions because they can be thrown at anytime, not just when calling JDO methods. As an example, navigating from one instance to another may result in an exception if there is a problem with accessing the datastore when retrieving the fields of an instance.

JDO classifies exceptions broadly into fatal or non-fatal exceptions. Non-fatal exceptions indicate that an operation failed but can be retried, whereas fatal exceptions indicate that the only recourse is to start again. Beyond this, there are user exceptions (fatal or non-fatal), which indicate user error; datastore exceptions (fatal or non-fatal), which indicate a datastore error; an internal exception (fatal), which indicates a problem with the JDO implementation; and an unsupported feature exception (non-fatal), which indicates that the JDO implementation doesn't support a particular feature.

An application should ensure that it handles all exceptions correctly, not just JDO exceptions. In particular, if connection pooling is being used, an application should ensure that a PersistenceManager instance is closed properly even if an exception is thrown. The following code snippet taken from CreateWithExceptionsExample.java shows how to use a try/final block to catch any exception and ensure that the PersistenceManager instance is closed:

```
PersistenceManager pm = null;

try {

  pm = pmf.getPersistenceManager();

  Transaction tx = pm.currentTransaction();

  tx.begin();

  Author author = new Author("Keiron McCammon");

  pm.makePersistent(author);

  tx.commit();

  pm.close();
}

finally {

  if (pm != null && !pm.isClosed()) {

    if (pm.currentTransaction().isActive()) {

      pm.currentTransaction().rollback();
    }

    pm.close();
  }
}
```

In this simple example, it really doesn't matter whether the transaction is rolled back or the PersistenceManager is closed because it exits immediately thereafter anyway.

3.11 Object Identity

One of the key concepts defined in JDO is that of object identity, although in most cases a developer is not even aware that it exists. Every persistent object has a JDO object identity. This identity associates the in-memory Java object with its representation in the underlying datastore. JDO ensures that there is only one in-memory representation of a given persistent object for a given PersistenceManager. This is known as "uniquing." Uniquing ensures that no matter how many times a persistent object is found, it has only one in-memory representation. All references to the same persistent object within the scope of the same PersistenceManager instance reference the same in-memory object.

The following code snippet taken from UniquingExample.java shows uniquing at work. It creates a new Author instance, begins a new transaction, and finds the Author again using a query. The two references are then compared to validate that they both refer to the same in-memory object:

```
tx.begin();

Author author1 = new Author("Keiron McCammon");

pm.makePersistent(author1);

tx.commit();

tx.begin();

Query query =
  pm.newQuery(Author.class, "name == \"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author2 = (Author) result.iterator().next();

tx.commit();

if (author1 == author2)
  System.out.println("There is only one object in memory");
```

The output would be as follows:

```
There is only one object in memory
```

However, because it is possible to create multiple PersistenceManager instances within a JVM, it is possible that a persistent object may have multiple in-memory representations at any given time—at most, one per PersistenceManager. Each would have the same JDO object identity, but would be a different in-memory Java object. To determine whether two in-memory objects represent the same persistent object, their JDO object identities can be compared. The JDOHelper class provides a method to get the JDO object identity of an object:

```
static Object getObjectId(Object pc)
```

The returned object can be compared with another using the equals() method to determine whether two in-memory objects represent the same persistent object in the datastore.

The following code snippet taken from ObjectIdentityExample.java creates an Author using one PersistenceManager, and then using a different PersistenceManager, it finds the Author again. The two references are compared to validate that they refer to different in-memory objects. The JDO identities are then compared to validate that they do, however, represent the same persistent object:

```
tx1.begin();

Author author1 = new Author("Keiron McCammon");

pm1.makePersistent(author1);

tx1.commit();

PersistenceManager pm2 = pmf.getPersistenceManager();
```

```
Transaction tx2 = pm2.currentTransaction();

tx2.begin();

Query query =
  pm2.newQuery(Author.class, "name==\"Keiron McCammon\"");

Collection result = (Collection) query.execute();

Author author2 = (Author) result.iterator().next();

tx2.commit();

if (author1 != author2)
  System.out.println(
    "There are multiple objects in memory");

Object author1Id = JDOHelper.getObjectId(author1);
Object author2Id = JDOHelper.getObjectId(author2);

if (author1Id.equals(author2Id))
  System.out.println("But they represent the same Author");
```

The output would be as follows:

```
There are multiple objects in memory
But they represent the same Author
```

JDO actually defines three types of object identity for persistent objects: datastore identity, application identity, and non-durable identity.

3.12 Types of Object Identity

The JDO specification requires that either datastore or application identity be supported (support for both is optional) and support for non-durable identity is optional. Most JDO implementations support datastore identity as their default, and some also support application identity (notably, those that are designed to run on top of a relational database).

3.12.1 Datastore identity

Datastore identity is the simplest form of identity. The JDO object identity of a persistent object is assigned and managed by the JDO implementation, typically in conjunction with the underlying datastore. The identity is orthogonal to the values of any of the fields of a persistent object.

There is nothing an application itself needs to do or provide when using datastore identity other than the Java classes themselves.

Unless there is a specific need otherwise, datastore identity should always be used. It's the simplest and most portable form of JDO object identity, and it is widely supported by most JDO implementations.

3.12.2 Application identity

Application identity is more akin to the concept of a relational primary key. The JDO object identity of a persistent object is determined by the values of certain of its fields.

If using application identity, the application needs to specify which fields are part of the primary key in the JDO metadata for the class and provide a class that can be used by a JDO implementation to represent the persistence-capable class's object identity.

Application identity is particularly useful if using JDO on top of a predefined datastore schema where primary keys have already been defined.

The following code snippet shows how the Author class does not need to change from the original Author class. Regardless of the type of identity being used, the persistence-capable class remains the same:

```
public class Author {

  private String name;

  public Author (String name) {

     this.name = name;
  }

  protected Author () {}

  public String getName () {

    return name;
  }

  public void setName (String name) {

    this.name = name;
  }
}
```

However, the JDO metadata for the class does change. The identity-type attribute needs to be set to "application" and the objectid-class attribute should denote the class that the JDO implementation can use to represent the object identity of a persistent object. Also, the primary-key attribute should be set to "true" for each field of the persistence-capable class that is part of the primary key.

The revised JDO metadata for the Author class in Author.jdo is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Author"
        identity-type="application"
        objectid-class="AuthorUsingApplicationIdentityKey">
      <field
          name="name"
          primary-key="true"/>
    </class>
  </package>
</jdo>
```

If the package of the object identity class is not specified, it is assumed to be the same as the persistence-capable class itself.

The final requirement when using application identity is to provide the implementation of the object identity class. JDO mandates that this class adhere to the following rules:

- The class must be public.
- The class must implement Serializable.
- The class must have a public no-arg constructor.
- There must be a public field of the same name and type as each field that is part of the primary key for the persistence-capable class.
- The equals() and hashCode() methods must use the values of all the primary key fields.
- The toString() method must return a string representation of the object identity instance.
- The class must have a constructor that takes a string returned from the toString() method of an object identity instance and constructs an equivalent object identity instance.

The following code snippet taken from AuthorKey.java shows the implementation of the object identity class for the Author persistence-capable class:

```
import java.io.Serializable;

public class AuthorKey implements Serializable {

  /*
   * JDO requires that an identity class have a
   * public field of the same name and type for each of
   * the primary key fields of the persistence-capable
   * class.
   */
  public String name;

  /*
   * JDO requires a public no-arg constructor
   */
  public AuthorKey () {}

  /*
   * JDO requires a public constructor that takes a string.
   */
  public AuthorKey (String oid) {

    name = oid;
  }

  /*
   * JDO requires that the toString() method return
   * a string representation of the primary key fields that
   * can later be used to recreate an identity instance
   * using the string constructor.
   */
  public String toString() {

    return name;
  }

  /*
   * JDO requires the equals() method to compare based on
   * all the primary key fields.
   */
  public Boolean equals(Object obj) {

    return name.equals(
     ((AuthorUsingApplicationIdentityKey) obj).name);
  }

  /*
   * JDO requires the hashCode() method to return a
   * hashcode based on all the primary key fields.
   */
  public int hashCode() {

    return name.hashCode();
  }
}
```

Primary Key Fields

A persistence-capable class can have multiple primary-key fields. The **equals()** and **hashCode()** methods on the object identity class must take into account each of the primary-key fields. Also, the **toString()** method should concatenate all the primary-key fields into a single string, which the string constructor can later tokenize to recreate the identity instance again.

When using application identity, only fields declared in abstract superclasses and the least-derived concrete class in the inheritance hierarchy can be part of the primary key. For example, if a class inherited from the Author class, then it could not provide its own object identity class or set the "primary-key" attribute to "true" for any of its locally declared fields. Instead, it would use the same application identity as defined for the Author class.

If using application identity, a JDO implementation may optionally allow an application to change a persistent object's identity by modifying the values of the primary key fields. If not supported, a JDOUnsupportedOptionException is thrown.

3.12.3 Non-durable identity

Non-durable identity is essentially no identity. The JDO object identity of a persistent object is valid only within the context of a given transaction. If the same persistent object is accessed in a different transaction, it may have a different object identity.

Non-durable identity should be used only when object identity has no relevance, i.e., when there is no need to maintain references between objects. Essentially, it only makes sense to query for objects with non-durable identity.

There are few, if any, JDO implementations that actually support non-durable identity. Its use is limited to a small number of use cases, and as such, most developers can safely ignore non-durable identity.

[Team LiB]

3.13 Object Lifecycles

JDO defines a number of different states in which an in-memory persistent object can be. Principally, these states are used by the JDO implementation to determine whether the fields of an instance need to be retrieved from the datastore and, if modified, whether they need to be written back. Together, these states represent an in-memory persistent object's lifecycle with respect to JDO.

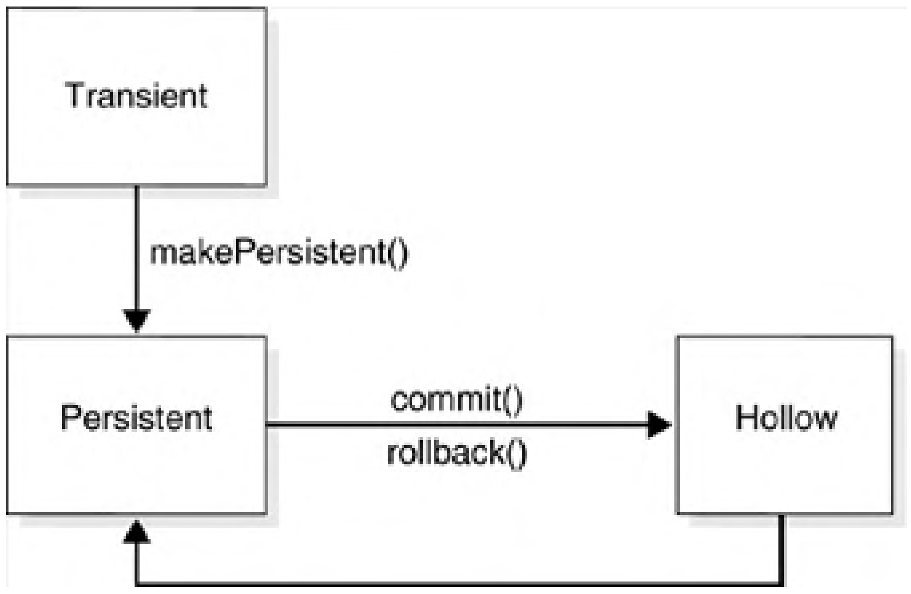
Chapter 4 describes the object lifecycle states in greater detail; however, in general, only the following basic states need to be understood:

- Transient
- Persistent
- Hollow

Figure 3-4 shows a simplified state diagram for these basic lifecycle states:

> **Transient** denotes a normal, non-persistent Java object. When an instance of a persistence-capable class is created, it starts life as being transient. By default, JDO does not do anything to manage transient instances.
>
> When made **persistent,** either explicitly via makePersistent() or because an instance is reachable from another persistent object, an instance transitions to being persistent (it actually transitions to a "persistent-new" state). Five different states are used to represent persistence. These allow the JDO implementation to differentiate between whether an instance is new ("persistent-new"), has been modified ("persistent-dirty") or deleted ("persistent-deleted"), and so on. All that really matters is that the instance is persistent, it has a JDO object identity, and its fields have been retrieved from the datastore within the context of the current transaction.
>
> A **hollow** instance is also a persistent object (it has a JDO object identity); however, its fields have not yet been retrieved from the datastore, hence the term "hollow." The first time that a hollow instance is accessed, its fields are retrieved from the datastore within the context of the current transaction and the instance transitions to being persistent. (It actually transitions to "persistent-clean" to indicate the instance hasn't yet been modified.) Likewise, by default, after a transaction ends, all instances that are persistent transition to being hollow.

Figure 3-4. The simplified lifecycle state diagram.



There are several optional features that provide support for transient instances being managed as transactional objects (changes to their fields being undone on rollback) and persistent objects being accessed outside of transactions. See Chapter 5 for more details.

3.14 Concurrency Control

All the examples so far have been single-user applications (only one transaction was accessing the datastore at a time). It is, of course, possible for multiple applications to be running concurrently. The JDO specification does not make any specific statements about concurrency control, because it is a feature of the underlying datastore rather than of JDO itself.

JDO does define the concept of a *datastore transaction* and says that access to the datastore should be done within the context of a transaction and that these transactions should exhibit ACID properties. This allows the JDO implementation to use the underlying datastore's own concurrency control mechanisms, which may be based on a pessimistic or optimistic locking scheme or some other approach. Because of this, JDO does not expose APIs to explicitly lock persistent objects. If locks are required by the underlying datastore to enforce concurrency control, then it is the responsibility of the JDO implementation to manage them implicitly.

3.14.1 ACID transactions

ACID stands for Atomic, Consistent, Isolated, and Durable, and it defines standard properties for a transaction:

Atomic: Upon commit, either all changes are in the datastore or none of them are in the datastore. This means that, should something fail, arbitrary changes aren't left behind in the datastore.

Consistent: Prior to and after commit, all data is consistent as far as the datastore is concerned. This means that all data accessed during a transaction is initially consistent and that, after commit, all changes leave the data consistent. Consistent means that the data does not violate any datastore constraints or invariants.

Isolated: Prior to commit, any changes made are not visible to other transactions and vice versa. This means that one transaction is not able to see changes made by another transaction; it can see only committed changes. Many datastores support varying levels of isolation that trade off strict transaction isolation against improved concurrency.

Durable: After commit, any changes are guaranteed to persist even in the event of failure. This means that when a transaction ends, any changes are in the datastore.

Different transactional datastores use different mechanisms to enforce ACID transactions. Therefore, when using JDO, when and how concurrency conflicts are detected and resolved is dependent upon which underlying datastore is being used. All that can be guaranteed is that a transaction is ACID in nature.

3.14.2 Optimistic transactions

JDO provides optional support for optimistic transactions. Optimistic transactions are useful for applications that have long running transactions where it is undesirable to hold datastore resources (such as locks) for an extended period of time.

With optimistic transactions, all concurrency control is deferred until the end of the transaction. See Chapter 5 for more details on optimistic transactions.

3.15 Summary

After reading this chapter, you should understand what transparent persistence is all about and what it means when a Java class is persistence-capable. You have seen how to create, read, update, and delete persistent objects, and you have explored how JDO supports the full Java object model: basic types, references, collection classes, and inheritance. You should also understand some of the more advanced JDO concepts like handling exceptions, object identity, object lifecycles, and concurrency control.

In the next chapter, you learn more about the different states in the JDO object lifecycle and how the JDO implementation manages persistent objects in-memory.

Chapter 4. Object Lifecycle

"'Forty-two,' said Deep Thought, with infinite majesty and calm."

—Douglas Adams, The Hitchhiker's Guide to the Galaxy, *1979*

This chapter discusses the lifecycle of objects and the reasons why state changes can be meaningful to the application developer.

Why are object lifecycles and state transitions important to know? Imagine what happens inside of a JVM whenever "new" is called by an application: A small portion of memory needs to be allocated and a reference is returned to the application, but the world is more complex behind the scenes. First, the JVM might load the byte code from some resource and then compile the byte code to native machine instructions. Then static initialization takes place, the base classes are loaded, and constructors are called. Even more complex is the destruction of memory instances due to garbage collection, weak references, queues, and finalize callbacks. There must be some well-defined lifecycle model inside the JVM that lets Java objects be used predictably and without "Deep Thought." As Java instances in memory have their lifecycle, persistent instances in JDO have a similar one: Its life starts as a plain, transient Java object, and it can become persistent, can be deleted, can be made transient again, and so on. Knowing about the object's persistent lifecycle is not needed in general, especially in every little detail, but whenever callbacks or other re-entrant code is used, it is strongly recommended that you think about the side effects of state transitions.

First, we present a brief introduction into mandatory and optional states of instances, the methods to gather information about states, and methods that lead to state transitions. Java source code is provided to illustrate state changes and to print state information about persistent instances. A section about instance callbacks explains their use and conditions. Last, optional states and operations are explained.

4.1 A Persistent Object's Lifecycle

JDO defines seven required and three optional states: transient, persistent-new, persistent-new-deleted, hollow, persistent-clean, persistent-dirty, and persistent-deleted are the mandatory lifecycle states. Transient-dirty, transient-clean, and persistent-non-transactional are optional states and are explained later in this chapter.

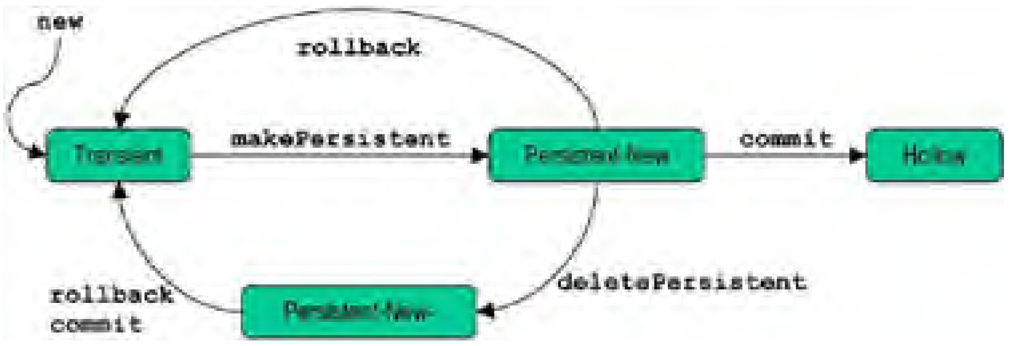
Some of the state transitions are directly triggered by the application itself—for instance, making an object persistent. Starting or terminating a transaction can also trigger an object's state change, which might be invoked directly by the application developer through a call to Transaction methods begin(), commit(), or rollback(). Note that in managed environments, state changes can happen due to an external transaction controller. A persistence-capable instance can enter the JDO object lifecycle by two operations: a call to makePersistent or an instantiation by the JDO vendor's implementation when an existing object is loaded from a datastore.

The chapter starts with two typical sequences making objects persistent and re-creating objects from the datastore. A short code sample is provided, and JDOHelper is used to gather information about the state changes in these sequences.

4.1.1 Making objects persistent

Operations to make an instance persistent follow the state diagram shown in Figure 4-1: Making transient instances persistent.

Figure 4-1. Making transient instances persistent.



4.1.1.1 Transient

Instances of persistence-capable classes behave like any other Java objects when they are created by the new operator. Field access is not mediated for these objects, resulting in almost the same access performance as for unenhanced classes. Transient instances are not influenced by transactional methods like begin, commit, or rollback except when they are reachable by other persistent objects.

4.1.1.2 Persistent-new

A call to PersistenceManager.makePersistent() assigns a new object ID to the instance, and the state changes from transient to persistent-new. A persistent-new object behaves like a transient object, but is flushed to the datastore at commit or whenever the object state needs to be synchronized with internal caches. Its state is controlled by the JDO implementation. Other objects can have a persistent-new state if they are reachable through persistent fields. They are made persistent, and so on, until all reachable objects are either of a transient type or persistent-new. Between makePersistent() and a final commit, references may change and reachable instances may become unreachable. The state of these *provisionally*-persistent instances is also persistent-new. These objects behave like persistent-new objects, but their state may change during commit, if the *provisionally*-persistent object isn't reachable anymore.

4.1.1.3 Hollow

After successful completion of the transaction, the persistent-new object's state changes to hollow. A hollow object is a reference that represents some data in the datastore, which is not yet in memory or outdated. Usually, other clients or applications are free to access or change data in the datastore referenced by hollow objects—for instance, in a different transaction, process, or JVM. Later, the application might access hollow objects, and the JDO implementation has to lazily reload data. This scenario is explained in the following section. Additionally, the contents of such hollow objects are cleared after commit, to free memory. If references were kept between hollow objects, the garbage collector may not free a graph of objects. After some transactions, a bunch of hollow objects might be accumulated.

The hollow state cannot be interrogated by the application. There is no JDOHelper method, and access to fields may result in a state change so that it is impossible to tell the previous state.

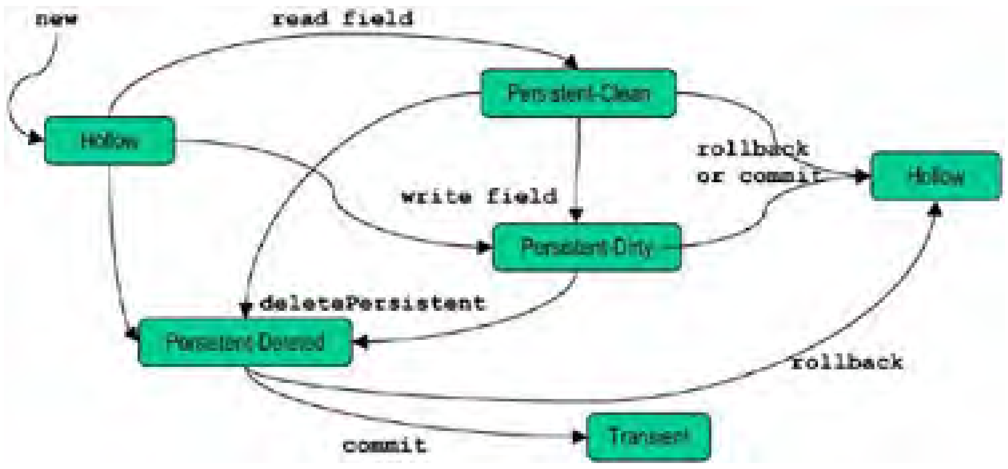
4.1.1.4 Persistent-new-deleted

Whenever persistent-new objects are deleted by a call to PersistenceManager.deletePersistent(), their state changes to persistent-new-deleted. There is no way to undelete an object, which means that this state is final until transaction termination. Accessing a field of a deleted object results in a JDOUserException, except when reading primary-key fields. After transaction termination (commit or rollback), the deleted instance's state changes back to transient.

4.1.2 Creating objects from the datastore

The next steps explain typical state changes for instances that are initially loaded from the datastore, which is shown in Figure 4-2: Creating instances from the datastore.

Figure 4-2. Creating instances from the datastore.



4.1.2.1 Hollow

As mentioned above, hollow is also the initial state for objects that are first created to represent a reference to data in a datastore, but the objects contain no valid data yet. When fields are accessed, data is retrieved on demand. Objects having the hollow state are instantiated by calls such as Extent.iterator().next(), PersistenceManager.getObjectById(), or a query execution. Usually, persistent fields of hollow objects are initialized to the field's default values (zero or null), but because persistence-capable instances are created through their no-args constructors, they may be filled with other values as well. This might be the only visible difference between objects made hollow after they have been inserted in a previous transaction as mentioned in the preceding section and hollow objects that were created by the JDO implementation.

4.1.2.2 Persistent-clean

Objects transition from hollow to persistent-clean after they have been filled with valid data from the datastore, but only when the data has not yet been modified in the current transaction. The state is called clean because the object's data is consistent with the corresponding data in the datastore. Athough an object is filled with valid data by the JDO implementation, field references to other objects are resolved by the PersistenceManager. Either those references may point to objects already in memory, or the PersistenceManager creates new hollow objects. Persistent-clean objects might be collected by the JVM's garbage collector if the application no longer refers to the objects. The PersistenceManager is required to cache the references to objects and instantiate new hollow objects in case the objects were already garbage-collected.

4.1.2.3 Persistent-dirty

Objects transition from either persistent-clean or hollow to persistent-dirty when one of their persistent fields is changed. This state is called dirty because the in-memory data of the object does not reflect the persistent data corresponding to the object in the datastore. If the value assigned to a managed field is equal to the previous value, the instance's state nevertheless changes to persistent-dirty. Furthermore, an instance does not change back to persistent-clean or hollow when previous field values or modifications are restored. Whenever transient objects are assigned to persistent-clean or persistent-dirty objects, they do not transition to persistent-new at that moment.

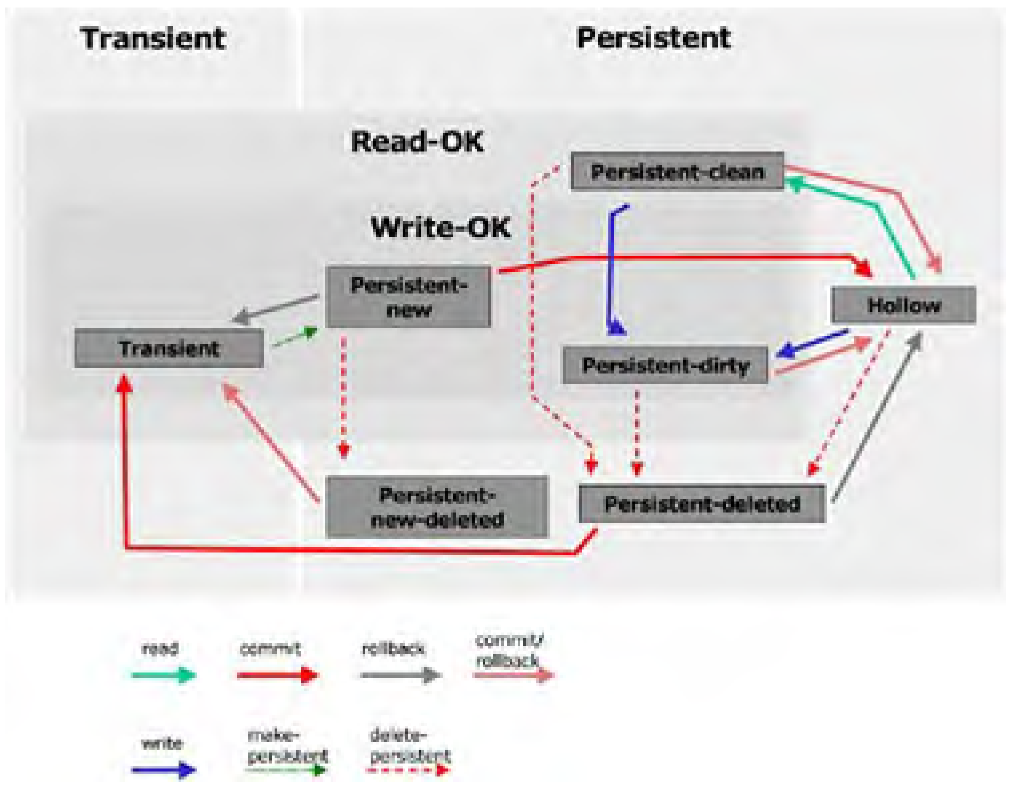
4.1.2.4 Persistent-deleted

In a call to PersistentManager.deletePersistent(), the state of the persistent instance is changed to persistent-deleted. It is impossible to leave this state except by terminating the transaction, which means that there is no method to undo a deletePersistent call except by aborting the transaction. A successful transaction termination makes a deleted instance transient. An object cannot be made persistent again with the same identity as long as it is deleted.

4.1.3 Simplified lifecycle

The state diagram in Figure 4-3: Lifecycle: Mandatory states shows a simplified lifecycle. The marked area Read-OK encloses all states in which instances can be read (or accessed) without forcing a state change. Write-OK marks the same area for instance modification.

Figure 4-3. Lifecycle: Mandatory states.



[Team LiB]

4.2 Finding Out about an Object's State

The previously mentioned seven states transient, persistent-new, persistent-new-deleted, hollow, persistent-clean, persistent-dirty, and persistent-deleted are mandatory for all JDO implementations, although there is no API to get the internal state of an instance used within the JDO implementation. The exact state of a JDO instance is known only by the JDO implementation, and some states might not even be implemented by some vendors. The JDO states are defined in a behavioral manner. Nevertheless, some methods let us interrogate the persistence attributes coupled with the object and find information about the object's state.

These static methods are in the JDOHelper class:

```
boolean isDeleted(Object o)
boolean isDirty(Object o)
boolean isNew(Object o)
boolean isPersistent(Object o)
boolean isTransactional(Object o)
```

There is no method to find out whether an object is hollow. It is a vendor's choice how to handle the lazy on-demand reading of fields and objects to alleviate the responsibility from the programmer. Later in this chapter, another way is shown to get information about an object's hollow state. The following class prints the persistence flags of an instance by using the JDOHelper class:

```
public class Example01
{
  public static void printInfo(String msg, Object obj)
  {
    StringBuffer buf = new StringBuffer(msg);
    if (JDOHelper.isPersistent(obj))
        buf.append(" persistent");
    if (JDOHelper.isNew(obj))
        buf.append(" new");
    if (JDOHelper.isDirty(obj))
        buf.append(" dirty");
    if (JDOHelper.isDeleted(obj))
        buf.append(" deleted");
    if (JDOHelper.isTransactional(obj))
        buf.append(" transactional");
    System.out.println(buf.toString());
  }
```

The following lines of code illustrate states and methods that we have used up to now:

```
  public static void main(String args[])
  {
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();

    Book book = new Book();

    printInfo("a new book ",book);

    pm.makePersistent(book);
    printInfo("persistent book ",book);

    pm.deletePersistent(book);
    printInfo("deleted book ",book);

    pm.currentTransaction().rollback();
    printInfo("rolled back ",book);

    pm.currentTransaction().begin();
    pm.makePersistent(book);
    pm.currentTransaction().commit();
    printInfo("new commit ",book);

    pm.currentTransaction().begin();
    String title = book.title;
    printInfo("read field ",book);

    String title = "new "+book.title;
    printInfo("modified field ",book);

    pm.deletePersistent(book);
    printInfo("deleted book ",book);

    pm.currentTransaction().commit();
    printInfo("last commit ",book);
  }
}
```

When the example is run, it prints the following:

```
a new Book: dirty
persistent book: persistent new dirty
deleted book: persistent new dirty deleted
rolled back: dirty
new commit: persistent
read field: persistent
modified field: persistent dirty
deleted book: persistent dirty deleted
last commit: dirty
```

4.3 Operations That Change State

In the previous section, we introduced basic states, and obvious methods—like reading a field or deleting a persistent instance from the datastore—were taken for granted. This section drills deeper into operations and explains behaviors in more detail. Some methods alter the object's state by a direct call; others indirectly change the state of multiple objects.

4.3.1 PersistentManager.makePersistent

The PersistenceManager method makePersistent causes transient objects to become persistent-new. In a call to makePersistent, the object graph is traversed to find other reachable, still transient objects. Those objects may indirectly become persistent-new as well.

4.3.2 PersistenceManager.deletePersistent

A call to deletePersistent makes clean, dirty, or hollow objects persistent-deleted, whereas persistent-new objects become persistent-new-deleted. This operation does not involve other reachable objects.

4.3.3 PersistenceManager.makeTransient

An object can be made transient if the previous state is persistent-clean or hollow. The persistent object is detached from the PersistenceManager and loses its identity. This PersistenceManager method does not involve other reachable objects; only the passed instance is made transient.

4.3.4 Transaction.commit

Instances that are part of a PersistenceManager's current transaction are processed by the JDO implementation during commit. Hollow instances are not changed. New, clean, and dirty objects become hollow; all other objects become transient. After commit, either hollow objects or regular, transient Java objects are associated with a PersistenceManager.

4.3.5 Transaction.rollback

Clean, dirty, and deleted objects become hollow at rollback. Other objects become transient. As a rule, hollow and transient objects are set back to their previous state after rollback.

4.3.6 PersistenceManager.refresh

A call to the PersistenceManager method refresh reloads the object's data and any modifications are lost. Persistent-dirty objects become persistent-clean after refresh.

4.3.7 PersistenceManager.evict

Evict helps the PersistenceManager caching implementation to save memory. Persistent-clean objects that are evicted become hollow. The fields of the objects are cleared to allow garbage collection of unreferenced subobjects.

4.3.8 Reading fields within a transaction

After reading the data from the datastore, hollow objects become persistent-clean. If pessimistic concurrency control is used, the object may be locked against concurrent modifications.

4.3.9 Writing fields within a transaction

When a field of an object that is attached to a PersistenceManager is modified inside of a transaction, the persistent object must be remembered by the JDO implementation to later update the object's data in the datastore. A persistent-clean or hollow object becomes persistent-dirty in this case. If pessimistic locking is used, a JDO implementation might also lock the object against access by other clients of the same datastore. Field modification of transient objects or objects that are persistent-dirty or persistent-new has no effect on the object's state. Whenever another object is assigned to a field, the modified object is marked persistent-dirty, but the referenced object is kept unchanged.

4.3.10 PersistenceManager.retrieve

The PersistenceManager.retrieve() method performs essentially the same operation as reading of a field within an active transaction.

[Team LiB]

4.4 Callbacks

By implementing callback methods, an application can intercept some of the above lifecycle events. There are three major cases in which an application developer might wish to implement some special behavior of persistent classes. These cases are as follows:

- Read or write data of fields that is not persistence-capable or should not be stored by the default JDO algorithm
- Delete contained objects
- Clear unneeded references

The callback methods are placed in an extra interface, which is handled like the java.io.Serializable tag interface. If a persistence-capable class implements the InstanceCallbacks interface, the callback methods are called by the JDO implementation; otherwise, they are not called.

4.4.1 Uses of jdoPostLoad

The jdoPostLoad method is called whenever a persistent object is loaded with data from the datastore. This happens when the instance's state transitions from hollow to persistent-clean. It is precisely called after the object has consistent data in its default fetch group.

InstanceCallbacks

If a class implements the InstanceCallbacks interface, the methods jdoPostLoad, jdoPreStore, jdoPreClear, and jdoPreDelete are automatically called by the JDO implementation when instances are read, written, cleared, or deleted, respectively. Unlike the Serializable interface, a class is responsible to call its superclass InstanceCallbacks methods. The InstanceCallback methods are declared public, so other clients should never call them.

By implementing the jdoPostLoad method, the code to check for initial assignment of transient values can be saved, because JDO defines that this method is called only once, when an object is resolved. A class might implement the jdoPostLoad method to assign a value to a transient field, which depends on the value of a persistent field. This is especially useful for extensive calculations that should be done only once.

Another use case can be to register the persistent object with other objects of the application after it has been loaded from the datastore. As an example, if deferring the initialization of a collection, memory is conserved and performance is gained until the object is actually used:

```
public class Book
   implements javax.jdo.InstanceCallbacks
{
   transient Map myMap; // not initialized here!

   public Book(String title) { // used by application
      ...
   }

   private Book() { // used by JDO runtime ...
   }

   public void jdoPostLoad() {
      myMap = new HashMap(); // initialized when loaded
   }

   // other callbacks:
   public void jdoPreStore() {
   }
   public void jdoPreDelete() {
   }
   public void jdoPreClear() {
   }
}
```

4.4.2 Uses of jdoPreStore

The prestore callback method is called before fields are written to the datastore. One application of this callback is to update persistent fields with values of non-persistent fields before the transaction completes. Another idea can be to check values of persistent fields for consistent values or data constraints. In the event that a constraint has not been followed, an exception can be thrown so that current values cannot be stored. On the other hand, business objects should expose validation methods to check for business constraints. These methods should be invoked by the application before data is written to the datastore. It is not a good practice to couple this kind of validation with the persistency framework. It is better to use the jdoPreStore callback only to test for programming mistakes in this case.

Here is an example of how to update a byte array field with a serialized stream of some non-persistent object:

```
public class Book
   implements InstanceCallbacks
{
   transient Color myColor; // not persistence-capable
   private byte[] array; // supported by most JDO
                         implementations

     public void jdoPostLoad() {
        ObjectInputStream in = new ObjectInputStream(
            new ByteArrayInputStream(array));
        myColor = (Color)in.readObject();
        in.close();
   }

   public void jdoPreStore() {
        ByteArrayOutputStream bo =
                         new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(bo);
        out.writeObject(myColor);
        out.close();
        array = bo.getBytes();
   }
   // other callbacks:
   public void jdoPreDelete() {
   }
   public void jdoPreClear() {
   }
}
```

4.4.3 Uses of jdoPreDelete

This callback is invoked before an object is deleted from the datastore—for instance, when the application calls PersistenceManager.deletePersistent. The jdoPreDelete method is not called for objects that are deleted by other clients of the same datastore, nor for instances in other, parallel transactions. Instead, the jdoPreDelete callback should be used for in-memory instances only. A typical application for the callback is to delete dependent subobjects:

```
public class BookShelf
   implements InstanceCallbacks
{
   // a BookShelf can contain either
   Collection shelves;
   Collection books;
   public void jdoPreDelete() {
      // get our PM:
        PersistenceManager pm =
            JDOHelper.getPersistenceManager(this);
        // delete shelves but not the books!
        pm.deleteAll(shelves);
     }
     // other callbacks:
     public void jdoPostLoad() {
   }
   public void jdoPreStore() {
   }
   public void jdoPreClear() {
   }
}
```

Long discussions came up in the JDO community about whether the handling of dependent objects should be coded in the Java source or declared somewhere outside in the JDO XML file. In another use case, the jdoPreDelete callback is responsible for de-registering the object in the referencing object, as was mentioned before. The coding might become quite complex if the application's object model gets larger. Here is a small example:

```
public class Aisle
   implements InstanceCallbacks
{
   // an Aisle contains BookShelves:
   Collection shelves;
}

public class BookShelf
   implements InstanceCallbacks
{
   final Aisle aisle; // location of this shelf

   // can be created together:
   public BookShelf(Aisle ai) {
              this.aisle = ai;
     aisle.shelves.add(this);
   }

   private BookShelf() { /* JDO required */ }

   // on deletion, remove ourselves from
   // the enclosing object:
   public void jdoPreDelete() {
      aisle.shelves.remove(this);
     }

     // other callbacks:
     public void jdoPostLoad() {
   }
   public void jdoPreStore() {
   }
   public void jdoPreClear() {
   }
}
```

In JDO, it is always an option to model the relations between objects either as a forward reference (the Aisle has BookShelves) or as a backward reference (the BookShelf belongs to an Aisle). A query is necessary to navigate from an enclosing object to its child (get a BookShelf that references a specific Aisle). It is application-specific and depends on the use cases and which option is taken, but when callbacks are used to de-register an object from its referencing object, a back reference (Bookshelf.aisle) is necessary anyway.

4.4.4 Uses of jdoPreClear

The jdoPreClear callback is called when an object becomes hollow or transient from persistent-deleted, persistent-new-deleted, persistent-clean, or persistent-dirty during transaction termination. It should be used to clear non-persistent references to other objects to help the garbage collector, or to de-register it from other objects. It can also be used to clear transient fields, or to set them to illegal values, to prevent accidental access of cleared objects.

4.5 Optional States

JDO defines optional states and behavior that may be implemented by a JDO vendor. These states help the developer to do things like use objects, although they may not contain transactional consistent data outside of active transactions. The states can also be used for transient objects that should act like persistent objects in an active transaction. Before any such optional features are used by an application, the JDO implementation's PersistenceManagerFactory.supportedOptions() method should be checked.

4.5.1 Transient-transactional instances

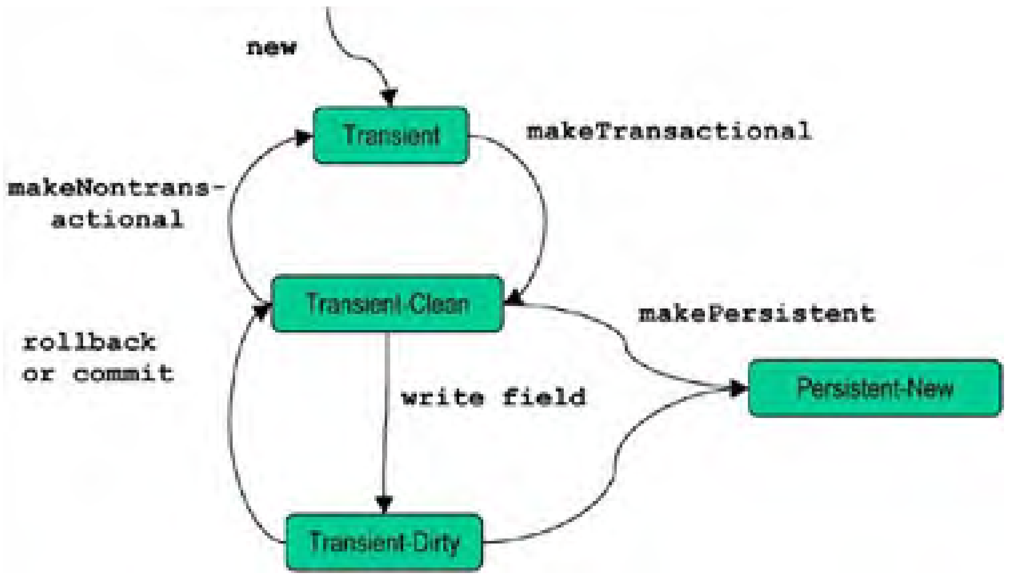
Transient instances of persistence-capable classes can become transactional, but not persistent, in the datastore by a call to PersistenceManager.makeTransactional. Those objects are called transient-transactional instances. Their data is somehow saved when the application calls makeTransactional and is restored when the transaction rolls back, but no data is stored in the backend. This feature is helpful when a complex operation changes many objects and the operation may be rolled back. The transient in-memory objects are also transactionally connected with other persistent objects of the same PersistenceManager instance.

Transient-transactional is an optional JDO feature; nevertheless, most of the vendors have implemented it at the time of this writing. To support lazy copy-on-write, a fresh transient-transactional instance enters the state transient-clean by a call to makeTransactional. Although its data is new or changed, this is only a marker for the PersistenceManager that any modification must be tracked. When the transient instance is modified, it becomes transient-dirty and a copy of its persistent fields is saved for later restore. There are two cases when makeTransactional can be invoked: before an active transaction is begun and afterward. In the first case, the data of the object it had at the time the transaction started will be saved. In the other case, the field data at the time makeTransactional was called will be saved.

Transient-dirty instances become transient-clean at either commit or rollback. During rollback, the previously saved values of managed fields are restored. Transient-clean instances can also be reset to non-transactional by invoking makeNontransactional. Both transient-clean and transient-dirty instances can be made persistent-new by a call to makePersistent.

The diagram in Figure 4-4 shows the state transitions for transient-transactional objects.

Figure 4-4. Lifecycle: Transient transactional instances.



4.5.2 Usages for transient-transactional instances

The basic idea for transient-transactional instances is transparency for the application. It is irrelevant whether or not an object is persistent; the state or data can be restored at rollback. An example is a GUI application that uses dialog boxes to display and edit configuration data. A dialog box may contain both persistent and local, transient configuration information. By using the transient-transactional feature, the implementation of the dialog box can simply roll back all data when "cancel" is clicked by the end-user.

4.5.3 Non-transactional instances

The idea behind non-transactional instances is to support slow changing, inconsistent data and optimistic transactions. In some scenarios, an application wishes to read data explicitly into objects, but doesn't care for modifications by other clients of the same datastore. This is the lowest isolation level with dirty read semantics. Non-transactional instances can also be the result of terminated transactions.

JDO defines four different properties that change the behavior of the JDO object lifecycle:

- NontransactionalRead
- NontransactionalWrite
- RetainValues
- Optimistic

The application has to set one or some of these properties to enable the specific options before the first use of a PersistenceManager. The reason for the options is safety. An application might modify an object by accident in a non-transactional context, but the modification gets lost or wrong data was read. Here is an example in which modification can get lost:

Keiron gets Book A.

Michael gets Book A.

Keiron updates Book A.

Michael updates Book A.

Michael commits transaction.

Keiron commits transaction (overwriting Michael's updates).

A similar problem can happen if data is updated between two consecutive reads:

Keiron counts all borrowed books.

Michael borrows a book.

Michael commits transaction.

Keiron counts all books in the library (with one book missing).

4.5.3.1 Non-transactional read

This option allows an application to read fields of hollow or non-transactional instances outside of transactions. It also enables navigation and queries outside of active transactions. The state to support non-transactional, inconsistent field data outside of active transactions is called persistent-non-transactional. An application can force a persistent-clean object to become persistent-non-transactional by invoking PersistenceManager.makeNontransactional. Consider the side effects and potential problems with non-transactional reads.

4.5.3.2 Non-transactional write

With this option set to true, modification of non-transactional instances is allowed outside a transaction. But changes to such objects are never written back to a datastore. JDO does not define how or when instances transition to persistent-non-transactional if they are modified outside a transaction with nontransactionalWrite set to true.

4.5.3.3 Usages for non-transactional instances

From an architectural standpoint, it is not really recommended to use non-transactional instances because of consistency problems. The design guidelines for transaction handling are discussed in Chapter 11. On the other hand, there are special scenarios in which non-transactional data has advantages:

- An application needs to display statistical data from time to time, and it is irrelevant whether the data is accurate at a specific point in time. Another design goal can be that the displaying part must not disturb the rest of the data processing. Under some circumstances, it might even be necessary to cache the displayed data and reread only changes.

- A GUI application uses JDO for configuration data, which is loaded at a defined point in time by another part of the system. While editing the configuration data, transactional behavior is not needed. Only the change from one configuration set to another (new) one must be atomic, consistent, isolated, and durable.

4.5.3.4 RetainValues

JDO defines that all persistent instances are cleared when they become hollow to remove unwanted references and clean up memory. With the RetainValues option set to true, persistent-clean, persistent-new, or persistent-dirty objects become persistent-non-transactional instead of hollow at commit. The field data is kept for later use, and it is allowed to read fields of such objects after the transaction terminated. Compared to non-transactional read, it is impossible to navigate to hollow objects and retrieve their data after transaction completion.

4.5.3.5 RestoreValues

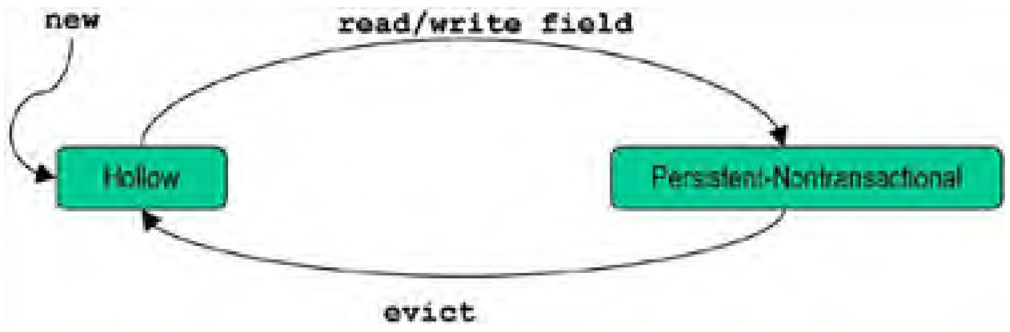
Without the RestoreValues option set to true, the above case is valid only if the transaction commits. At rollback, instances still become hollow and they are unusable for the application. By setting the RestoreValues option to true, the JDO runtime must reread the old state of persistent-dirty objects at rollback, or by other means restore the values of modified fields. All persistent-clean and persistent-dirty objects become persistent-non-transactional at rollback, making them readable by the application.

4.5.4 Optimistic transactions

JDO also has the ability to do optimistic transaction management in the same way that a source code version control system allows multiple people to work on code concurrently rather than locking users out of a particular source document. It is interesting to note that optimistic transaction management changes the state transitions of persistent instances. When a transaction starts optimistically, the application believes that locking objects ahead of their access delays operations or locks other users out of the system. Instead, conflicting operations are dealt with when transactions are being committed. JDO accomplishes this technique by allowing parts of the program to modify persistent objects that contain transactional data. When the application finally instructs JDO to commit the persistent object, it checks the datastore to see if the data has been modified since the data was loaded (probably by using a version identification mechanism internal to JDO), and it rolls back, letting you know that there has been a versioning problem. This allows more than a single client to use a persistent object while ensuring that no data is lost. The drawback is that you have to recommit your data (with the new data merged). For this reason, the state management has the option of letting the application refresh data of some hollow or non-transactional instance before it is used inside the transaction.

When the application selects to use optimistic transactions, hollow instances transition to persistent-non-transactional, instead of persistent-clean, when a field is read (Figure 4-5). Upon updates of persistent fields, the instance transitions back to persistent-dirty like in pessimistic transactions. But the application has to explicitly invoke a refresh before any update is made to retrieve fresh values from the datastore. The pros and cons of optimistic transaction are explained in Chapter 11.

Figure 4-5. Lifecycle: Access outside transactions.



4.6 Putting It All Together

The following code excerpts show how to use the state interrogation methods and instance callbacks to debug common problems. A class is developed that serves as a helper to find out about the actual state of objects, number of objects in memory, objects per state, and so on. Because most JDO vendors don't provide methods to get information about a PersistenceManager's in-memory cache, it is quite handy to have a vendor-independent debug facility.

All the following methods can be found in the DebugStates.java source file.

The first method is a helper to get the memory ID of an object. A simple way to get an informal ID of an object is to call identityHashCode. Although this is not a unique ID, it works on common virtual machines and can help to debug objects in memory:

```
private static String getMemoryId(Object obj)
{
  long id = System.identityHashCode(obj);
  return Long.toString(id,16); // Hex value
}
```

The next method returns a string that contains both the persistent object ID and the in-memory hash code of an object. If the object is not yet persistent, only the memory ID is returned:

```
private static String getId(Object obj)
{
  if (obj instanceof PersistenceCapable) {
    PersistenceManager pm =
       JDOHelper.getPersistenceManager(obj);
    if (pm != null) {
      return "@"+getMemoryId(obj)+
                  " OID:"+pm.getObjectId(obj);
    }
  }
  return "@"+getMemoryId(obj)+" no OID";
}
```

To get even more information about an object's lifecycle, a table of counters is used to keep track of persistent instances. These counters are incremented in the InstanceCallbacks interface methods and the no-args constructor, which is used by the JDO implementation to create new hollow objects. A table entry object keeps a weak reference to each persistent instance, so that it is possible to get the number of objects that were garbage-collected.

Here is a sample class that uses the DebugStates class to count its callback invocations:

```
public class Author
   implements InstanceCallbacks
{

    protected String name;

    public Author( String name ) {
        this.name = name;
    }

    public String toString()
    {
        return "Author: " + name;
    }

  /**
   * This is the constructor called for hollow objects.
   * It registers with the debug class.
   */
  private Author()
  {
    DebugStates.constructor(this);
  }

  // JDO callbacks, delegate to DebugStates:
  public void jdoPostLoad() {
    DebugStates.postLoad(this);
  }

  public void jdoPreStore() {
    DebugStates.preStore(this);
  }

  public void jdoPreClear() {
    DebugStates.preClear(this);
  }

  public void jdoPreDelete() {
    DebugStates.preDelete(this);
  }
}
```

The above DebugStates methods delegate to a function that gets a table entry object from a static table, based on the memory identity. If the entry does not exist, it is added to the table:

```
private static ObjectTable objectTable = new ObjectTable();

private static ObjectTableEntry tableEntry(Object obj)
{
  return objectTable.register(obj);
}

/**
 * Should be called in the no-args constructor of a
 * persistence-capable
 * class.
 */
public static void constructor(Object obj)
{
  tableEntry(obj).constructor++;
}

... equivalent methods for InstanceCallbacks ...


public ObjectTableEntry register(Object obj)
{
  Long id = new Long(System.identityHashCode(obj));
  ObjectTableEntry entry = (ObjectTableEntry)table.get(id);
  if (entry == null) {
    entry = new ObjectTableEntry(id,obj);
    table.put(id,entry);
  }
  return entry;
}
```

Now that every object is put into the table at some point, it is easy to count used objects or find objects of a specific kind. The following method can help to get some figures of the memory usage:

```
public String getStatistics()
{
  int constructors = 0;
  int postLoad = 0;
  int preStore = 0;
  int preClear = 0;
  int preDelete = 0;
  int objects = 0;
  int gced = 0;
  for (Iter i = new Iter();i.hasNext();) {
    ObjectTableEntry e = i.next();
    objects++;
    if (e.get() == null) gced++;
    constructors += e.constructor;
    postLoad += e.postLoad;
    preStore += e.preStore;
    preClear += e.preClear;
    preDelete += e.preDelete;
  }
  String NL = System.getProperty("line.separator");
  String result = "   Number of Objects:   "+objects+NL+
    "   Garbage collected:   "+gced+NL+
    "   Constructors called: "+constructors+NL+
    "   postLoad called:     "+postLoad+NL+
    "   preStore called:     "+preStore+NL+
    "   preClear called:     "+preClear+NL+
    "   preDelete called:    "+preDelete+NL;
  return result;
}
```

The result is interesting because even the first simple allocation of two objects like the following code:

```
pm.currentTransaction().begin();
Author a = new Author("Heiko Bobzin");
Book b = new Book("Core JDO", a);
```

prints the following result:

```
Number of Objects:   2
Garbage collected:   0
Constructors called: 2
postLoad called:     0
preStore called:     0
preClear called:     0
preDelete called:    0
```

Although the no-args constructor wasn't called in the sample code, it was traced that two other objects have been created. These two objects are held as factory objects by the JDO implementation to create new Author and Book instances. The code to call the no-args constructor has been added by the enhancer for security reasons, which are explained later in Chapter 10.

When these two objects are made persistent (Author by reachability):

```
pm.makePersistent(b);
pm.currentTransaction().commit();
```

the preStore and preClear instance callbacks are called for each object:

```
Number of Objects:   2
Garbage collected:   0
Constructors called: 2
postLoad called:     0
preStore called:     2
preClear called:     2
preDelete called:    0
```

In another test, all objects of the Book Extent were printed:

```
pm.currentTransaction().begin();
Extent e = pm.getExtent(Book.class,true);
Iterator iter = e.iterator();
while (iter.hasNext()) {
  Book a = (Book)iter.next();
  String title = a.title; // creates a hollow Author !
}
```

The database contained 20 Book instances and 20 Author instances, so that 42 objects were created in total (two for the factory objects). postLoad and preClear have been called for the Book instances only, whereas the other objects were hollow instances:

```
Number of Objects:   42
Garbage collected:   10
Constructors called: 42
postLoad called:     20
preStore called:     0
preClear called:     20
preDelete called:    0
```

4.7 Summary

The JDO instance lifecycle has been explained in this chapter to give an in-depth understanding of the dynamic aspects of persistent and transactional instances. This chapter has provided information about how and when to use state interrogation functions as well as callback methods. JDO optional states like persistent non-transactional and transient transactional were introduced, and a commonly usable class to explore and trace JDO states has been developed.

With the understanding of the JDO object lifecycle, it should be possible to drill into more complex problems and design JDO applications. For a complete table of state transitions, refer to Appendix A.

Chapter 5. Developing with JDO

"If debugging is the art of removing bugs, then programming must be the art of inserting them."

—Unknown

Becoming proficient with using JDO to build applications requires a thorough understanding of the JDO concepts as well as all the classes and interfaces defined in the javax.jdo and javax.jdo.spi packages. In this chapter, we explore these concepts in greater detail and look at all the classes and APIs defined by the JDO specification.

This chapter covers the following topics:

- JDO concepts.
- The names, roles, and relationships for all the JDO interfaces and classes.
- Explanation for each method defined by the PersistenceManagerFactory, PersistenceManager, Transaction, Extent, Query, and InstanceCallbacks interfaces.
- An overview of the JDO exception classes.
- Explanation for each method defined by the JDOHelper and I18NHelper classes.
- An overview of the classes and interfaces in the service provider interface package.

The examples for this chapter can be downloaded from the Internet at www.corejdo.com and are located in the com.corejdo.examples chapter 5 package; you can also use the persistence-capable classes contained in the com/corejdo/examples/model directory.

5.1 JDO Concepts

Being proficient at using JDO to build applications requires a good understanding of the concepts underlying JDO. Some of these were introduced in previous chapters to varying degrees, and others are dealt with in greater detail in the chapters that follow.

5.1.1 Persistence-capable

The concept of a persistence-capable class was explored in detail in Chapter 3. It's the basic mechanism that enables JDO to transparently persist instances of a Java class in a datastore. Any Java class that implements the JDO PersistenceCapable interface and adheres to the JDO programming style is said to be persistence-capable and can be used with any JDO implementation without recompilation.

Typically, a developer would use vendor-supplied tooling to either generate a persistence-capable class from scratch or take an existing Java class and make it persistence-capable (through source code or byte code manipulation).

5.1.2 JDO Metadata

As introduced in Chapter 3, JDO requires additional metadata to be defined for each persistence-capable class. This metadata denotes that a given Java class is actually persistence-capable, whether it has a persistence-capable superclass, and how the class and fields of the class should be treated by the JDO implementation.

The XML file containing the metadata file should contain the following:

- A header that specifies the location of the JDO Metadata XML DTD. The location can be the URL as published by SUN Microsystems, or it can be some DTD file stored elsewhere:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE jdo SYSTEM "jdo.dtd">
  ```

- A <jdo> element, within which the metadata for the persistence-capable classes is contained:

  ```
  <jdo>
  </jdo>
  ```

- The <jdo> element should contain one or more <package> elements, each having a name attribute that specifies the name of the package. A <package> element contains the metadata for the persistence-capable classes contained in that package:

  ```
  <package name="com.corejdo.examples.model">
  <\package>
  ```

- A <package> element should contain one or more <class> elements, each having a name attribute that specifies the name of the Java class. A <class> element can have a number of additional attributes: identity-type; objectid-class; requires-extent, and persistence-capable-superclass that define how the persistence-capable class should be treated by the JDO implementation:

  ```
  <class name="Author">
  </class>
  ```

- A <class> element can contain one or more <field> elements, each having a name attribute that specifies the name of the field. A <field> element can have a number of additional attributes: persistence-modifier; primary-key; default-fetch-group, embedded, and null-value that define how the field should be treated by the JDO runtime:

  ```
  <field name="books">
  </field>
  ```

- A <field> element can contain <collection>, <map>, or <array> elements, depending on the type of the field.

These elements specify additional metadata about the field (such as the class of the instances in a collection):

```
<collection element-type="com.corejdo.examples.model.Book">
</collection>
```

JDO defines various defaults for the optional attributes that avoid the need to specify them explicitly. This keeps the average metadata file short and simple. In addition to the elements outlined above, the DTD also allows vendors to specify extensions using the <extension> element, which has vendor-name, key, and value attributes.

The metadata for a persistence-capable class should be located in a metadata file specific to the class or in a file that contains the metadata for all persistence-capable classes in a given package hierarchy. A metadata file specific to a single class should be named after the class itself, but with a .jdo suffix. For example, the metadata for the Author class could be located in a file called Author.jdo. Alternatively, the metadata could be located in a file called package.jdo. This file would then contain the metadata for the Author class as well as any other persistence-capable class in the same package hierarchy. In both situations, the metadata file should be available at runtime as a resource that can be loaded by the same class loader that loaded the class itself (i.e., it should be in the classpath). For example, the metadata for the class com.corejdo.examples.model.Author should be contained in one of the following files:

- package.jdo
- com/package.jdo
- com/corejdo/package.jdo
- com/corejdo/examples/package.jdo
- /corejdo/examples/model/package.jdo
- com/corejdo/examples/model/Author.jdo

The metadata is searched for in the above sequence, and the first definition that is found is used.

As explained in Chapter 3, the initial JDO 1.0 specification did not mandate the location and name of the XML metadata files. To aid in portability, the 1.0.1 maintenance release of the specification changed this to what has just been described.

Prior to this release, the name of a metadata file that contained multiple classes was the package name itself with a .jdo suffix, and it was located in the directory that contained the package directory. In the Author example, it would have been called model.jdo and would have been located in the com/corejdo/examples/ directory. Some JDO implementations may still use this naming convention.

Appendix B provides greater details on the exact syntax of the JDO metadata XML file and the purpose of each element and attribute.

5.1.3 Default fetch group

When a persistent object is retrieved from the datastore, the JDO implementation initially retrieves the values of those fields that are part of what is referred to as the *default fetch group* for the class. If a persistent object has been retrieved from the datastore, then as a minimum the values of all the fields that are part of the default fetch group are in memory. This concept allows the JDO implementation to optimize access to the fields in the default fetch group because it doesn't have to mediate access to each field individually once a persistent object has been retrieved. For fields not in the default fetch group, the JDO implementation mediates each access, ensuring that the field has first been retrieved from the datastore.

By default, fields of primitive types, java.util.Date and supported system classes from java.lang and java.math packages, automatically are part of the default fetch group. A field can be explicitly added to the default fetch group by setting the default-fetch-group attribute to true in the metadata for the field. If a field that references another persistent object is made part of the default fetch group, then effectively the JDO implementation retrieves the referenced persistent object also.

5.1.4 Persistence by reachability

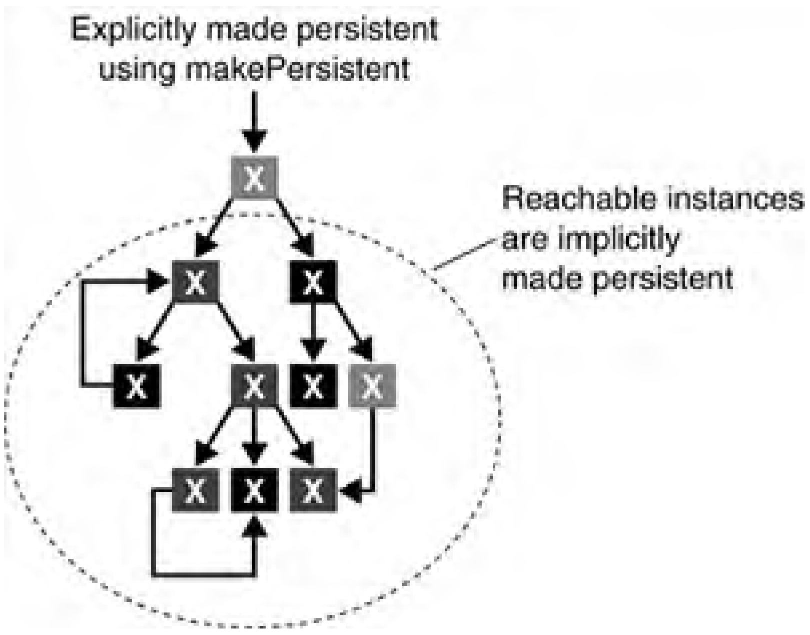
Chapter 3 introduced the concept of persistence by reachability. This simply means that instances of persistence-capable classes or supported-system classes that are referenced from an instance that is itself made persistent is implicitly made persistent also, and so on for the entire graph of reachable instances. Likewise, any transient instance that is later added to an already persistent instance is made persistent, too.

JDO defines that the persistence by reachability calculation is initially done during PersistenceManager.makePersistent() and then redone during Transaction.commit(). Therefore, a transient instance that is added to a graph after makePersistent()

does not itself become persistent until commit() – or as a result of a later call to makePersistent().

Figure 5-1 shows a graph of instances and identifies those that would be implicitly made persistent.

Figure 5-1. Persistence by reachability.


Explicitly made persistent using makePersistent
Reachable instances are implicitly made persistent
X

Even if the graph of reachable instances is cyclic in nature, or a particular instance is referenced from multiple others or is already persistent, the JDO implementation ensures that each is made persistent only once.

Persistence by reachability negates the need for the application to explicitly track all instances that it needs to make persistent as it creates them; instead, it just needs to make the root instance of a graph persistent, and the JDO implementation implicitly takes care of the rest.

5.1.5 First-class and second-class objects

JDO uses the concept of first-class and second-class objects to differentiate how a JDO implementation manages instances of user-defined, persistence-capable classes (persistent objects) versus instances of supported-system defined classes (like Java collection classes) and arrays.

A first-class object is an instance of a persistence-capable class; it has a JDO object identity and can be stored in the datastore by itself. A second-class object does not have a JDO object identity; it can be stored only in the datastore as belonging to a first-class object.

As an example, consider a persistence-capable class that contains a java.util.HashSet. Instances of the persistence-capable class are first-class objects; instances of java.util.HashSet are second-class. Essentially, instances of supported-system classes or arrays are not themselves persisted in the datastore; rather a representation is persisted as part of the owning first-class object.

Because second-class objects can be persisted only as belonging to a first-class object, it is not possible for multiple first-class objects to reference the same second-class object in the datastore. As an example, at some point within a transaction, multiple in-memory persistent objects may reference the same java.util.HashSet instance. However, when the transaction commits and the persistent objects are persisted in the datastore, each creates its own representation of the contents of the java.util.HashSet. When retrieved again into memory, each persistent object gets its own java.util.HashSet instance.

In addition, it is not necessary to explicitly delete second-class objects. When a first-class object is deleted using the deletePersistent() method on PersistenceManager, the JDO implementation automatically ensures that the representation of any second-class objects is also deleted from the datastore. Of course, if a java.util.HashSet contains references to other persistent objects, only the representation of the java.util.HashSet is deleted; the referenced persistent objects are not deleted because they are first-class objects in their own right.

5.1.6 Object identity

Chapter 3 introduced the concept of JDO object identity and the different types of identity that can be used with JDO.

Essentially, every persistent object has a unique JDO object identity. An in-memory instance is associated with its representation in the underlying datastore via this identity. Object identity is central to how a JDO implementation manages and persists instances of persistence-capable classes.

JDO defines three types of object identity:

- **Datastore identity:** A persistent object's identity is automatically generated.
- **Application identity:** A persistent object's identity is determined by the values of certain fields (those whose primary-key attribute is true in the metadata for the class).
- **Non-durable identity:** A persistent object's identity is not persistent at all and is only valid while in-memory.

By default, a persistence-capable class is assumed to have a datastore identity. The identity type of a persistence-capable class can be explicitly specified using the identity-type attribute in the class's metadata. The attribute can be data-store (the default), application, or nondurable.

If using the application identity, the objectid-class attribute is also required to identify the user-defined class used to represent the persistence-capable class's identity.

5.1.7 Object lifecycle

JDO defines a number of states that an in-memory instance of a persistence-capable class can be in. These states are used by the JDO implementation to keep track of whether the fields of a given instance have been retrieved from the underlying datastore within the current transaction, whether any fields have been modified by the application or if the application has deleted it. The transition from state to state is referred to as an object's lifecycle and the JDO specification denotes which transitions are valid and which aren't.

As explained in Chapter 3, these are the basic states:

- **Transient:** A non-persistent, normal Java object.
- **Persistent:** A persistent object whose fields have been retrieved from the datastore in the current transaction.
- **Hollow:** A persistent object whose fields have not yet been retrieved from the datastore.

The object lifecycle is of real concern only to a JDO implementation; a JDO application itself needs to do nothing. However, using the InstanceCallbacks interface or via a JDOHelper class, an application can interact with the JDO implementation as certain state transitions occur or interrogate the state of a particular instance of a persistence-capable class.

See Chapter 4 for a comprehensive description of all the states and state transitions that make up a persistent object's lifecycle.

5.1.8 Transactions

Typically, persistent objects must be created, retrieved, modified, or deleted within the context of a transaction. The underlying datastore uses transactions to enforce concurrency control between multiple applications and ensure that, upon commit, all changes within a transaction get written to the datastore or none of them do (in the case of failure).

JDO supports both non-managed and managed transaction control. For non-managed transactions, the application itself must explicitly start and end a transaction. For managed transactions, transactions are implicitly started and ended by an external coordinator (a là the Java Transaction Service). Managed transactions are typical if using JDO within the J2EE environment and are covered in more detail in Chapter 9.

Because different datastores use different approaches to managing concurrency control, JDO does not explicitly define the semantics of how concurrency control should be enforced or when concurrency conflicts should be detected. It instead defines the concept of a *datastore transaction* and simply says that a datastore transaction should exhibit ACID properties. See Chapter 3 for more details on ACID transactions.

5.1.8.1 Non-transactional read/write

Basic database principles dictate that a transaction should be short-lived to avoid concurrency conflicts between different applications running concurrently. If an application begins a transaction and retrieves persistent objects, it can potentially block other applications from accessing or updating those same persistent objects.

For applications that need to access persistent objects over an extended period of time, JDO specifies a number of optional features that allow access to persistent objects outside of a normal transaction and therefore avoid potential concurrency issues.

The non-transactional read and write features allow an application to retrieve persistent objects and to update them without first beginning a transaction. The following PersistenceManagerFactory options denote whether these features are supported by a JDO implementation:

- javax.jdo.option.NontransactionalRead
- javax.jdo.option.NontransactionalWrite

Persistent objects accessed in this way are referred to as being non-transactional.

Because the persistent objects are accessed without using a transaction, the underlying datastore is not going to enforce any concurrency control. After it is retrieved into memory, a persistent object is potentially no longer in sync with the underlying datastore; another application might have modified or even deleted the persistent object after it was retrieved.

A persistent object can also be modified outside of a transaction, but if the instance is later accessed within a database transaction, any changes are lost (the persistent object is effectively re-retrieved from the datastore within the current transaction). If using optimistic transactions, then it is possible to preserve changes.

Non-transactional support is useful for applications that want to keep persistent objects that are rarely modified in-memory for reference purposes to avoid the overhead of having to keep retrieving them from the datastore within each transaction. However, it is the application's responsibility to ensure that the in-memory instances are kept up-to-date with respect to changes in the underlying datastore.

The methods setNontransactionalRead() and setNontransactionalWrite() on Transaction and PersistenceManagerFactory can be used to explicitly enable or disable these features independently.

5.1.8.2 Optimistic transactions

Similar to non-transactional read and write, this feature allows an application to retrieve and modify persistent objects over a prolonged period of time without requiring an actual datastore transaction for the entire duration. But unlike non-transactional read and write, it does ensure transactional consistency. The following PersistenceManagerFactory option denotes whether this feature is supported by a JDO implementation:

- javax.jdo.option.Optimistic

Using an optimistic transaction is similar to using non-transactional read in that persistent objects are retrieved from the datastore and managed in memory as non-transactional instances outside of an actual datastore transaction. This means that any other application can modify the persistent objects after they have been retrieved, in which case the in-memory instances are no longer in sync with the actual persistent objects in the datastore. With optimistic transactions, however, the JDO implementation ensures that any conflicting updates are detected at commit time.

When a persistent object is first retrieved within an optimistic transaction, it becomes a non-transactional instance. If modified, the in-memory instance is marked as being modified and transitions to being transactional. The same is true for new or deleted persistent objects. An application can also choose to explicitly make non-transactional instances transactional using the makeTransactional() method on PersistenceManager.

At commit time, all transactional instances are checked against the datastore to ensure that no other application has modified them because they were first retrieved. If there has been a conflicting update, a JDOOptimisticVerificationException is thrown and the transaction is automatically rolled back, in which case an application must retry in a new transaction.

JDOOptimisticVerificationException

This exception was introduced in the JDO 1.0.1 maintenance release. Prior to this, if a conflict was detected upon commit, a **JDOUserException** was thrown. However, this leaves the transaction in an indeterminate state as far as the application is concerned, because it is not possible to determine whether the commit indeed failed because of a conflict or because of user error. If user error caused the failure, some corrective action might be taken and the transaction retried. Also, assuming that a conflict is involved, there is no way to determine which persistent objects caused it.

The JDO 1.0.1 maintenance release addresses these issues by introducing the **JDOOptimisticVerificationException**. This differentiates the failure of an optimistic transaction as a result of a conflict from a failure as a result of user error. The exception contains nested **JDOOptimisticVerificationException** instances, one for each persistent object that caused the conflict. Also the exception is a fatal rather than non-fatal exception, therefore, the transaction is automatically rolled back.

The method setOptimistic() on Transaction and PersistenceManagerFactory can be used to explicitly enable or disable this feature.

5.1.8.3 Retaining values

By default, at the end of a transaction, all in-memory persistent objects transition to being "hollow" and if needed are re-retrieved from the datastore within the next transaction. This default behavior can be changed.

The retain values feature allows an application to retain the fields of persistent objects in memory after a transaction ends. The following PersistenceManagerFactory option denotes whether this feature is supported:

- javax.jdo.option.RetainValues

If using retain values, at the end of a transaction, all in-memory persistent objects transition to being non-transactional rather than hollow. Assuming that non-transactional read has been enabled, an application can continue to access the fields of the persistent objects, although they are no longer in sync with the underlying datastore.

The method setRetainValues() on Transaction and PersistenceManagerFactory can be used to explicitly enable or disable this feature.

5.1.8.4 Restoring values

The restore values feature is a mandatory JDO feature and can used to revert the fields of in-memory persistent objects on rollback to their values as of the beginning of the transaction or the point in time they were made persistent within the transaction.

If used by itself, then on rollback the fields of any persistent objects that where made persistent within the transaction will be reverted to their original values and the persistent objects will transition to being transient instances again.

If used in conjunction with the RetainValues option, then on rollback the fields of all in-memory persistent objects will be reverted and they will transition to being non-transactional rather than hollow, as per the behavior for RetainValues on commit.

If restore values is not used then the fields of any modified persistent objects retain the modified values, even though these have not been committed to the datastore.

The method setRestoreValues() on Transaction and PersistenceManagerFactory can be used to explicitly enable or disable these options.

5.1.8.5 Transient transactional

In addition to managing persistent objects transactionally, JDO has an optional feature that allows an application to make transient (i.e., non-persistent) instances of persistence-capable classes transactional.

The transient transactional feature allows an application to specify that changes to a transient instance of a persistence-capable class should be managed transactionally by the JDO implementation. An application can make a transient instance transactional so that changes made to the instance during a transaction are undone if the transaction rolls back. The following PersistenceManagerFactory option denotes whether this feature is supported:

- javax.jdo.option.TransientTransactional

A transient instance is made transactional using the make transactional methods defined on PersistenceManager. Any modification of a transient transactional instance during a transaction is subject to rollback. Any modification made outside of a transaction is not. A transient transactional instance remains transactional until explicitly made non-transactional using the make non-transactional methods defined on PersistenceManager.

On rollback, a transient transactional transient instance's fields are reverted to the values they had at the beginning of the current transaction (or the values they had when they were made transactional, if this occurred within the current transaction).

Transient transactional can be useful for applications that need to keep changes made to transient instances transactionally consistent with changes made to persistent objects.

5.2 JDO Interfaces and Classes

Chapter 3 provides a simplified overview of the key JDO classes and interfaces. Figures 5-2 and 5-3 show all the JDO classes and interfaces defined in the javax.jdo and javax.jdo.spi packages.

Figure 5-2. JDO class diagram.

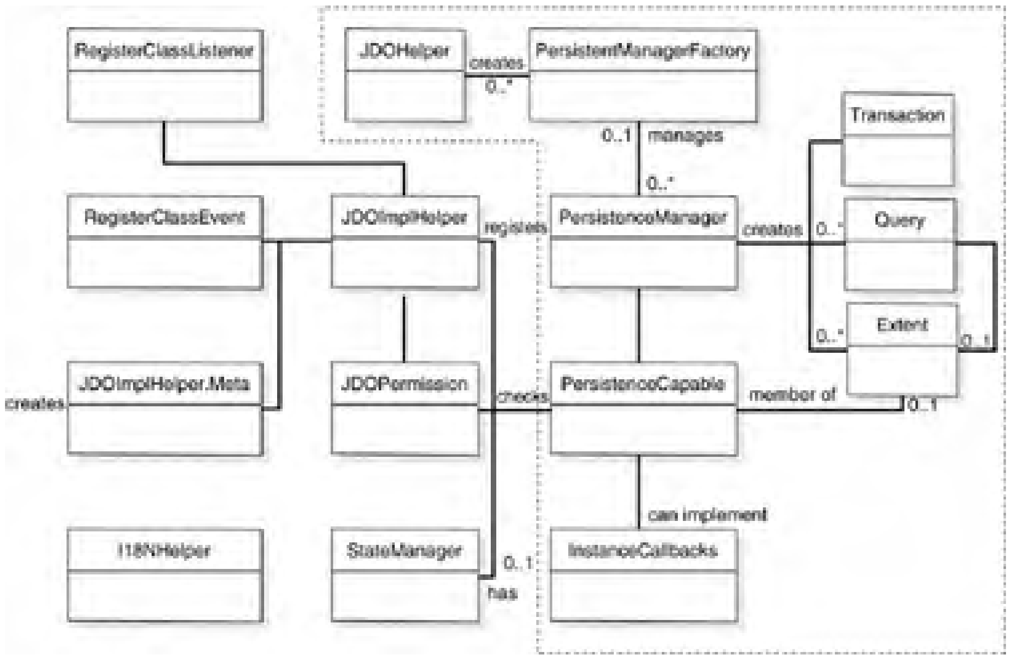
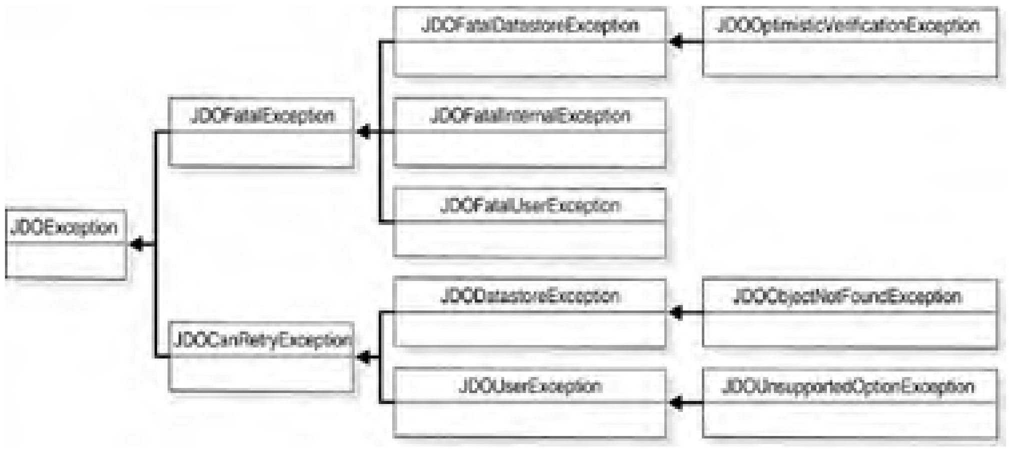


Figure 5-3. JDO exceptions class diagram.


JDOFatalDatastoreException
JDOOptimisticVerificationException
JDOFatalException
JDOFatalInternalException
JDOFatalUserException
JDOException
JDODatastoreException
JDOObjectNotFoundException
JDOCanRetryException
JDOUserException
JDOUnsupportedOptionException

The javax.jdo package contains the classes and interfaces that a developer would use to build an application using JDO. The javax.jdo.spi package contains the classes and interfaces that would be used to actually implement a JDO compliant interface to a datastore.

A *PersistenceManagerFactory* is used to get a PersistenceManager instance. There is a one-to-many relationship between PersistenceManagerFactory and PersistenceManager. A PersistenceManagerFactory can create and manage many PersistenceManager instances and even implement pooling of PersistenceManager instances.

A *PersistenceManager* embodies a connection to a datastore and a cache of in-memory persistent objects that implement the PersistenceCapable interface. The PersistenceManager interface is the primary means by which an application interacts with the underlying datastore and persistent objects.

From a PersistenceManager, an application can get one or more Query instances. A Query is the how an application can find persistent objects by their field values.

In addition to a Query, an application can get an Extent from a PersistenceManager. An Extent represents all persistent objects of a specified class (and optionally subclasses) in the datastore and can be used as input to a Query or iterated through in its own right.

A *Transaction* allows an application to control the transaction boundaries in the underlying datastore. There is a one-to-one correlation between a Transaction and a PersistenceManager (there can be only one ongoing Transaction per PersistenceManager instance). Transactions must be begun and either committed or aborted.

The *InstanceCallbacks* interface can be implemented by a PersistenceCapable class and, if implemented the PersistenceManager, invokes the relevant callback methods as a persistent object's lifecycle state changes.

JDOHelper can be used to create a PersistenceManagerFactory from a set of properties and provides convenience methods to interrogate the state of instances of classes that implement PersistenceCapable.

Every persistent object has an associated instance of StateManager. This instance manages the persistent fields and lifecycle state of the in-memory persistent object.

JDOImplHelper is used by the JDO implementations to get metadata on persistence-capable classes; it is not supposed to be used by an application. JDOImplHelper.Meta contains the metadata for a given class. To receive notification when a persistence-capable class is loaded, a JDO implementation registers a RegisterClassListener with the JDOImplHelper. For every persistence-capable class that is loaded, the registered listeners are invoked using a RegisterClassEvent.

JDOPermission contains the access privileges that restrict what can call certain methods on JDOImplHelper and persistence-capable classes that deal with metatdata and modification of the state of a persistent object.

I18Nhelper is used by the JDO classes to get localized messages from a resource bundle. A default resource bundle (Bundle.properties) is provided in the JDO JAR.

JDOException is the superclass for all JDO exceptions and extends java.lang.RuntimeException. JDO classifies exceptions broadly into fatal or non-fatal exceptions. JDOFatalException is the superclass for all fatal exceptions, and JDOCanRetryException is the superclass for all non-fatal exceptions.

Non-fatal exceptions indicate that an operation failed but can be retried. JDODataStoreException indicates a problem with the datastore. JDOUserException indicates invalid or incorrect application behavior, and JDOUnsupportedOptionException indicates that an application has tried to use a JDO optional feature that isn't supported by the JDO implementation being used. JDOObjectNotFoundException indicates that a persistent object could not be found in the datastore; this exception was added in the JDO 1.0.1 maintenance revision of the specification.

Fatal exceptions indicate that the only recourse is to start again. JDOFatalDatastoreException indicates a serious problem with the datastore. JDOFatalInternalException indicates a serious problem within the JDO runtime, and JDOFatalUserException indicates a serious problem caused by incorrect application behavior. JDOOptimisticVerificationException indicates that the completion of an optimistic transaction failed due to a conflicting update; this exception was added in the JDO 1.0.1 maintenance revision of the specification.

The rest of this chapter provides a detailed overview of the JDO classes and interfaces. The classes and interfaces are grouped as follows:

Basic APIs

PersistenceManagerFactory
PersistenceManager
Transaction
Query
Extent
InstanceCallbacks

Exception Classes

JDOException
JDOCanRetryException
JDODataStoreException
JDOUserException
JDOUnsupportedOptionException
JDOFatalException
JDOFatalDatastoreException
JDOFatalInternalException
JDOOptimisticVerificationException
JDOFatalUserException

Additional APIs

JDOHelper
I18NHelper

Service Provider APIs

PersistenceCapable
StateManager
JDOImplHelper
JDOPermission

The following classes are used internally by the JDO classes themselves and are not covered here:

JDOImplHelper.Meta
RegisterClassListener
RegisterClassEvent

PREVIOUS NEXT

5.3 Basic APIs

The following six interfaces provide the basic set of JDO APIs that most applications need to use:

- PersistenceManagerFactory
- PersistenceManager
- Transaction
- Query
- Extent
- InstanceCallbacks

For each interface described in the following sections, you find a description of its purpose, followed by an overview of each method, and then examples that show how to use those methods in practice.

5.3.1 javax.jdo.PersistenceManagerFactory

The PersistenceManagerFactory interface is typically the starting point for any JDO application; from it, an application gets instances of PersistenceManager. The PersistenceManagerFactory has a number of standard properties that can be configured; in addition, vendors can add their own properties to their PersistenceManagerFactory implementations, although all properties should follow the JavaBean pattern of setters and getters.

PersistenceManagerFactory is responsible for managing the physical connections to the underlying datastore and may or may not implement connection pooling. In addition, a PersistenceManagerFactory can be used by both managed and non-managed applications. For non-managed applications, connection-related properties are used to indicate how to make a connection to the datastore. For managed applications, connection management is delegated to an external connection factory.

Chapter 7 provides more details on managed and non-managed applications. Simply put, a managed application typically runs within a J2EE container; a non-managed application is a standalone Java application.

A non-managed application creates its own connections to the datastore and explicitly manages its own transactions. A managed application gets connections from a connection factory and may delegate transaction management to the container within which it runs.

5.3.1.1 Creating a PersistenceManagerFactory

There are three main ways to create a PersistenceManagerFactory:

JNDI Because PersistenceManagerFactory extends Serializable, it can be stored and looked up via JNDI. This is the typical approach for a managed application. The properties of a PersistenceManagerFactory created in this way cannot be modified, and the PersistenceManagerFactory instance cannot be serialized again.

JDOHelper The getPersistenceManagerFactory() method creates a PersistenceManagerFactory from a set of properties, one of which specifies the actual class name of the vendor's PersistenceManagerFactory. This is the expected and portable approach for non-managed applications to get a PersistenceManagerFactory instance. JDO requires each vendor to implement a static method on their PersistenceManagerFactory called getPersistenceManagerFactory(); JDOHelper uses this to create a specific PersistenceManagerFactory instance. The properties of a PersistenceManagerFactory created in this way cannot be modified, and the PersistenceManagerFactory instance cannot be serialized.

Constructors Each vendor can provide their own constructors that allow an application to directly construct a PersistenceManagerFactory instance. This is suitable for non-managed applications, but has the disadvantage that it is non-portable and the code would need to be changed per JDO implementation. The properties of a PersistenceManagerFactory created in this way can be modified up to the first call to getPersistenceManager() and the PersistenceManagerFactory instance can be serialized.

A PersistenceManagerFactory instance created by either a JNDI lookup or by JDOHelper is not serializable in order to avoid a security hole. Because a PersistenceManagerFactory can potentially contain a password, if it were serializable, it could expose this password.

5.3.1.2 PersistenceManagerFactory Properties

The standard properties defined by JDO for PersistenceManagerFactory are split between those that control the creation of connections to the underlying datastore (such as ConnectionURL and ConnectionUserName) and those that configure the default behavior of PersistenceManager instances (such as IgnoreCache and Multithreaded).

Depending on whether the application is managed or non-managed, different connection-based properties apply. Table 5-1 lists the non-managed connection properties.

Table 5-1. Non-Managed Connection Properties

Property	Method Summary
ConnectionDriverName	String getConnectionDriverName() void setConnectionDriverName(String name)
This property specifies the class name of the driver used when connecting to the datastore. A JDO implementation that uses JDBC might use this property to allow the application to specify the name of the JDBC driver to be used.	
ConnectionPassword[*]	void setConnectionPassword(String password)
This property specifies the password of the user used to connect to the datastore.	
ConnectionURL	String getConnectionURL() void setConnectionURL(String url)
This property specifies the URL used to make the connection and is datastore-specific. A JDO implementation that uses JDBC might expect this to be of the form jdbc:subprotocol:subname. Some implementations may allow the username and password to be specified in the connection URL.	
ConnectionUserName[*]	String getConnectionUserName() void setConnectionUserName(String name)
This property specifies the user name used to connect to the datastore.	

[*] The user name and password can also be specified separately in the call to getPersistenceManager().

The values of these properties are dependent on the underlying datastore being used, but not all may be required or even relevant. The documentation for a given JDO implementation should identify which of these properties are used and what values they should be given.

For managed applications, the PersistenceManagerFactory uses an external connection factory to create datastore connections. If any of the connection factory properties are set (non-null), then the un-managed properties above are effectively ignored and the external connection factory is used instead.

Typically, a managed application would not explicitly create or configure a PersistenceManagerFactory; it would look one up via JNDI instead. A PersistenceManagerFactory instance would be initialized and registered with JNDI beforehand (although exactly how this is done depends on the managed environment being used and the JDO implementation).

Table 5-2 lists the managed connection properties.

Table 5-2. Managed Connection Properties

Property	Method Summary
ConnectionFactory[*]	Object getConnectionFactory() void setConnectionFactory(Object factory)
This property specifies the external connection factory used to create connections. The factory interface itself is not defined by JDO and is implementation dependent. If set, the PersistenceManagerFactory does not use its own connection pooling. A JDO implementation that uses JDBC might accept an instance of javax.sql.DataSource as a connection factory for JDBC connections.	
ConnectionFactoryName[*]	String getConnectionFactoryName() void setConnectionFactoryName(String name)
This property specifies the JNDI name of the connection factory; if the ConnectionFactory property has not been set, then the PersistenceManagerFactory looks up the connection factory via this name using JNDI.	
ConnectionFactory2[*]	Object getConnectionFactory2() void setConnectionFactory2(Object factory)

This property specifies an alternative external connection factory used to create connections for optimistic transactions. Typically, in managed environments, transaction control is delegated to an external coordinator; however, when using optimistic transactions, implicit transaction control is not appropriate.

ConnectionFactory2Name[*]	String getConnectionFactory2Name() void setConnectionFactory2Name(String name)

This property specifies the JNDI name for an alternative connection factory; if the ConnectionFactory2 property has not been set, then the PersistenceManagerFactory looks up the connection factory via this name using JNDI.

[*] A PersistenceManagerFactory that implements its own connection pooling may utilize these properties also, even for a non-managed application.

Table 5-3 lists the configuration properties that provide default values for each PersistenceManager instance created by a given PersistenceManagerFactory.

Table 5-3. Configuration Properties

Property	Method Summary
IgnoreCache	boolean getIgnoreCache() void setIgnoreCache(boolean flag)
This property specifies whether the PersistenceManager can ignore in-memory changes when creating Extent and Query instances. See the section on PersistenceManager for more details.	
Multithreaded	boolean getMultithreaded() void setMultithreaded(boolean flag)
This property specifies whether the PersistenceManager should synchronize itself for mutli-threaded use. See the section on PersistenceManager for more details.	
NontransactionRead	boolean getNontransactionalRead() void setNontransactionalRead(boolean flag)
This property specifies whether the PersistenceManager should allow non-transaction reads by default. See the section on PersistenceManager for more details.	
NontransactionWrite	boolean getNontransactionalWrite() void setNontransactionalWrite(boolean flag)
This property specifies whether the PersistenceManager should allow non-transactional writes by default. See the section on PersistenceManager for more details.	
Optimisitic	boolean getOptimistic() void setOptimistic(boolean flag)
This property specifies whether the PersistenceManager should use optimistic transactions by default. See the section on Transaction for more details.	
RestoreValues	boolean getRestoreValues() void setRestoreValues(boolean flag)
This property specifies whether the PersistenceManager should restore the fields of persistent and transactional instances after rollback by default. See the section on PersistenceManager for more details.	
RetainValues	boolean getRetainValues() void setRetainValues(boolean flag)
This property specifies whether the PersistenceManager should retain the fields of persistent instances after commit by default. See the section on PersistenceManager for more details.	

5.3.1.3 Method summary

In addition to the methods to get and set properties, the PersistenceManagerFactory has five other methods:

void close()

Closes the PersistenceManagerFactory instance and all associated PersistenceManager instances.

If any associated PersistenceManager instances have currently active transactions, they are not closed and JDOUserException is thrown. This exception contains a nested JDOUserException for each PersistenceManager instance that could not be closed.

This method checks that the caller has the JDOPermission closePersistenceManagerFactory. If not, the method simply returns with no action.

This method was added in the JDO 1.0.1 maintenance release.

PersistenceManager getPersistenceManager()

Returns a PersistenceManager instance.

The PersistenceManager is configured based on the connection and configuration properties of the PersistenceManagerFactory.

After the first call to this method, the properties of the PersistenceManagerFactory can no longer be modified.

PersistenceManager getPersistenceManager (String name, String password)

Returns a PersistenceManager instance using the specified user name and password.

The PersistenceManager is configured based on the connection and configuration properties of the PersistenceManagerFactory, except that the specified user name and password are used. If the PersistenceManagerFactory implements connection pooling, it ensures that it returns a connection for the specified user.

After the first call to this method, the properties of the PersistenceManagerFactory can no longer modified.

Properties getProperties()

Returns the non-configurable, vendor-specific properties of the JDO implementation

JDO defines two standard vendor properties, VendorName and VersionNumber. All others are implementation-specific.

Collection supportedOptions()

Returns a collection of strings that represent the optional features and/or query languages that the JDO implementation supports.

Table 5-4 lists the strings used to denote support for the standard JDO optional features and query language:

Table 5-4. Optional Features

Optional Feature	Optional Feature
javax.jdo.option.TransientTransactional	javax.jdo.option.LinkedList
javax.jdo.option.NontransactionalRead	javax.jdo.option.TreeMap
javax.jdo.option.NontransactionalWrite	javax.jdo.option.TreeSet
javax.jdo.option.RetainValues	javax.jdo.option.Vector
javax.jdo.option.Optimistic	javax.jdo.option.Map
javax.jdo.option.ApplicationIdentity	javax.jdo.option.List
javax.jdo.option.DatastoreIdentity	javax.jdo.option.Array
javax.jdo.option.NonDurableIdentity	javax.jdo.option.NullCollection
javax.jdo.option.ArrayList	javax.jdo.option.ChangeApplicationIdentity
javax.jdo.option.Hashtable	javax.jdo.query.JDOQL
javax.jdo.option.HashMap	

5.3.1.4 Usage guidelines

A non-managed application can create a PersistenceManagerFactory via JNDI, JDOHelper, or explicitly using a vendor-supplied constructor.

If using JNDI, a PersistenceManagerFactory needs to be initially created via an implementation-specific constructor and bound to its JNDI name. When the application looks up the PersistenceManagerFactory via JNDI, it gets a de-serialized instance. Typically, a PersistenceManagerFactory instance is shared within an application, in which case a singleton pattern should be used to manage the lookup and sharing of the PersistenceManagerFactory.

The following code snippet from PMFSingletonFromJNDIExample.java shows how a PersistenceManagerFactory can be created and registered with JNDI, and then looked and managed as a singleton instance:

```
import javax.naming.*;
import javax.jdo.*;

public class PMFSingletonFromJNDIExample {

  // JNDI name for the PersistenceManagerFactory instance
  private static String name =
    PMFSingletonFromJNDIExample.class.getName();

  // PersistenceManagerFactory Singleton
  private static PersistenceManagerFactory pmf;

  /*
   * This method returns the singleton
   * PersistenceManagerFactory. If required, it first
   * initializes the singleton by looking it up via JNDI.
   * If it can't lookup the PersistenceManagerFactory via
   * JNDI, it throws JDOFatalUserException
   */
  public static PersistenceManagerFactory
  getPersistenceManagerFactory() {

    if (pmf == null) {

      try {

        InitialContext context = new InitialContext();

        pmf = (PersistenceManagerFactory)
          context.lookup(name);
      }

      catch (NamingException e) {

        throw new JDOFatalUserException(
          "Can't lookup PersistenceManagerFactory: " +
            name, e);
      }
    }

    return pmf;
  }

  /*
   * This class can be run to create a
   * PersistenceManagerFactory instance
   * and register it with JNDI.
   * The PersistenceManagerFactoryClass property
   * would need to be changed from "XXX".
   */
  public static void main(String[] args) {

    if (args.length != 3) {

      System.out.println(
        "Invalid arguments, use: <url> <name> <password>");

      System.exit(-1);
    }

    Properties properties = new Properties();

    properties.put(
      "javax.jdo.PersistenceManagerFactoryClass", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionURL", args[0]);
    properties.put(
      "javax.jdo.option.ConnectionUserName", args[1]);
    properties.put(
      "javax.jdo.option.ConnectionPassword", args[2]);
    PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(properties);

    try {

      InitialContext context = new InitialContext();

      context.bind(name, pmf);
    }

    catch (NamingException e) {

      System.out.println("Can't bind PersistenceManager");

      e.printStackTrace();
    }
  }
}
```

If not using JNDI, the singleton needs to be created and initialized directly. JDOHelper can be used to do this based on a set of properties. The following code snippet taken from PMFSingletonExample.java shows how this could work. It uses an initialize() method that creates the PersistenceManagerFactory singleton from the specified properties:

```
import java.util.Properties;
import javax.jdo.*;

public class PMFSingletonExample {

  // PersistenceManagerFactory singleton
  private static PersistenceManagerFactory pmf;

  /*
   * This method returns the singleton
   * PersistenceManagerFactory.
   * If it the singleton hasn't been initialized, it throws
   * JDOFatalUserException.
   */
  public static PersistenceManagerFactory
  getPersistenceManagerFactory() {

    if (pmf == null) {

     throw new JDOUserException(
       "PersistenceManagerFactory not initialized.");
    }

    return pmf;
  }

  /*
   * Creates a PersistenceManagerFactory based on the
   * specified properties and initializes the singleton.
   * The actual PersistenceManagerFactoryClass needs to
   * be specified rather than "XXX".
   */
  public static void intialize(
    String url, String name, String password) {

    Properties properties = new Properties();

    properties.put(
      "javax.jdo.PersistenceManagerFactoryClass", "XXX");
    properties.put(
      "javax.jdo.option.ConnectionURL", url);
    properties.put(
      "javax.jdo.option.ConnectionUserName", name);
    properties.put(
      "javax.jdo.option.ConnectionPassword", password);

    pmf =
      JDOHelper.getPersistenceManagerFactory(properties);
  }
}
```

Of course, before this example can be used, the "XXX" strings would need to be replaced with values appropriate to the JDO implementation being used.

The previous example has the disadvantage that the initialize() method needs to be explicitly called with the appropriate connection properties. An alternative to this is shown in the following code snippet taken from PMFSingletonFromFileExample.java. It reads the properties from a file whose name is specified using a system property:

```
import java.util.Properties;
import java.io.*;
import javax.jdo.*;

public class PMFSingletonFromFileExample {

  // PersistenceManagerFactory singleton
  private static PersistenceManagerFactory pmf;

  /*
   * This method returns the singleton
   * PersistenceManagerFactory. If the singleton
   * hasn't been initialized, it creates a
   * PersistenceManagerFactory from a properties
   * file denoted by the system property "jdo.properties"
   * and initializes the singleton.
   */
  public static PersistenceManagerFactory
  getPersistenceManagerFactory() {

    if (pmf == null) {

      String filename =
        System.getProperty("jdo.properties");

      if (filename == null) {
        throw new JDOFatalUserException(
          "System property 'jdo.properties' not defined");
      }

      else {

        Properties properties = new Properties();

        try {

          properties.load(new FileInputStream(filename));

          pmf =
            JDOHelper.
              getPersistenceManagerFactory(properties);
        }

        catch (java.io.IOException e) {

          throw new JDOFatalUserException(
            "Error reading '" + filename + "'", e);
        }
      }
    }

    return pmf;
  }
}
```

The following code snippet taken from SupportedOptionsExample.java shows how to print out the options that a PersistenceManagerFactory instance supports:

```
Iterator iter = pmf.supportedOptions().iterator();

System.out.println(
  "PersistenceManagerFactory class '" +
  pmf.getClass().getName() + "' supports:");
```

```
while (iter.hasNext()) {
  String option = (String) iter.next();
  System.out.println("  " + option);
}
```

Using the getPersistenceManager() methods is explained in the section on PersistenceManager.

5.3.2 PersistenceManager

PersistenceManager is the primary interface that a JDO application would use. It has a number of standard properties that govern how it manages persistent objects in memory and provides methods for an application to interact with those persistent objects and the underlying datastore. It also acts as a factory for Query, Extent, and Transaction instances.

5.3.2.1 Creating a PersistenceManager

There are two main ways to create a PersistenceManager:

PersistenceManagerFactory The getPersistenceManager() methods on PersistenceManagerFactory return a PersistenceManager instance. For a managed application, this is the only way to get a PersistenceManager. For a non-managed application, this is the expected and portable approach to getting a PersistenceManager. A PersistenceManager got in this way inherits its properties from its PersistenceManagerFactory and remains associated with the PersistenceManagerFactory during its lifetime.

Constructors A JDO implementation may provide constructors that allow an application to directly create a PersistenceManager. This is suitable for non-managed applications, but has the disadvantage that it is non-portable and the code would need to be changed per JDO implementation.

5.3.2.2 PersistenceManager properties

Table 5-5 lists the three properties defined by PersistenceManager.

Table 5-5. PersistenceManager Properties

Property	Method Summary
IgnoreCache	boolean getIgnoreCache() void setIgnoreCache(boolean flag)
This property is a hint to the PersistenceManager as to whether it can ignore in-memory changes when executing a Query or iterating through an Extent.	
Multithreaded	boolean getMultithreaded() void setMultithreaded(boolean flag)
This property controls whether the PersistenceManager synchronizes internally to ensure that internal data structures do not get corrupted when used by multiple threads concurrently.	
UserObject	object getUserObject() void setUserObject(Object obj)
This property allows the application to associate an arbitrary object with a PersistenceManager instance.	

IgnoreCache Normally, the result from a Query or Extent reflects any in-memory changes made within the current transaction. If a new persistent object was created or a persistent object was modified or deleted, any result would reflect these changes. Generally, this is the desired behavior. However, it may come at a cost or be undesirable in certain situations. Depending on the underlying datastore, it might be necessary to synchronize any in-memory changes with the datastore prior to executing the Query or iterating through the Extent.

As an optimization, an application can choose not to incur this overhead, in which case the IgnoreCache property should be set to true. A result from a Query or iterating through an Extent may then not reflect in-memory changes made within the current transaction. If a new instance is created that matches a given query, it may not be part of the returned result. The advantage is that executing the query no longer requires in-memory changes to be synchronized with the datastore beforehand.

This property is just a hint; even if set, it may be ignored by a JDO implementation. An application can't rely on a Query to ignore in-memory changes just because this property is true. Unless there is a specific performance requirement, an application typically shouldn't set this property to true.

Multithreaded It is safe for an application to use multiple threads with JDO as long as there is only one thread using a given PersistenceManager or instances created by a PersistenceManager (Query, Extent, PersistenceCapable instances, and so on) at a given time. Simply put, if each thread has its own PersistenceManager, there is no problem. However, if the application requires multiple threads to be able to use the same PersistenceManager instance at the same time, this property should be set to true.

If this property is true, the PersistenceManager synchronizes internally to ensure that internal data structures do not get corrupted when multiple threads use the PersistenceManager instance concurrently. Synchronization may incur additional overhead and adversely affect performance; this property should be set only if multi-threaded access is really required. See Chapter 7 for more details on developing multithreaded applications.

5.3.2.3 Method summary

PersistenceManager has methods that allow an application to interact with persistent objects and the underlying datastore, and to get Query, Extent, and Transaction instances. There are many methods, but these are the key ones that a JDO application is likely to use:

Transaction currentTransaction()

The returned Transaction instance can be used to begin and end transactions.

void makePersistent (Object obj)

This makes a new instance of a persistence-capable class persistent.

void deletePersistent (Object obj)

This deletes a persistent object in the datastore.

Query newQuery (Class cls, String filter)

The returned Query instance can be executed to find persistent objects based on the specified filter.

void close()

Indicates that the PersistenceManager instance is no longer needed.

The PersistenceManager methods can be grouped into the following categories:

- Connection management methods
- Cache management methods
- Instance management methods
- Factory methods

Connection Management Methods These methods allow the application to indicate that it no longer needs the PersistenceManager instance or allows it to get the associated PersistenceManagerFactory.

void close()

Indicates that the PersistenceManager instance is no longer needed. Depending on whether the PersistenceManagerFactory implements pooling, this may just return the PersistenceManager instance back to the PersistenceManagerFactory for reuse.

An attempt to close a PersistenceManager instance, in a non-managed environment where there is an active transaction, results in JDOUserException being thrown.

An attempt to use a PersistenceManager instance after it has closed (apart from the isClose() method) results in JDOFatalUserException being thrown.

Boolean isClosed()

Returns true if the PersistenceManager has been closed.

PersistenceManagerFactory getPersistenceManagerFactory()

Returns the PersistenceManagerFactory that created the PersistenceManager instance.

This may return null if the PersistenceManager was explicitly created by the application via a constructor.

Cache Management Methods Typically, an application does not need to explicitly manage persistent objects in-memory; the JDO runtime implicitly takes care of them. In some situations, the application may need to explicitly release, retrieve, or refresh a persistent object or group of persistent objects. A common reason for this is as a performance optimization: An application can explicitly retrieve a collection of persistent objects from the datastore into memory in one go, rather than retrieving them implicitly one by one.

The methods below all follow the same set of conventions. There is a base method that takes a single Object argument. Then there are additional methods whose names are post-fixed with "All" to indicate that they take multiple objects (either an Object[] or Collection argument). There may also be an "All" method that takes no argument, in which case it operates on all relevant in-memory instances.

If null is used as an argument to a method that takes Object, then the null is ignored and the method is effectively a no-op. If null is passed to a method that takes an Object[] or Collection, then NullPointerException is thrown. If the Object[] or Collection argument itself contains null elements, then these are ignored.

void evict(Object obj)
void evictAll(Object[] objs)
void evictAll(Collection objs)
void evictAll()

By default, persistence objects are automatically evicted by the PersistenceManager at the end of a transaction. However, the evict methods allow an application to provide a hint to the PersistenceManager that the specified instances are no longer required in-memory and they can be made available for garbage collection during a transaction (although they are garbage collected only if actually no longer referenced by the application).

These methods evict instances even if the RetainValues property is true for the transaction and evicted instances transition to being hollow. The evictAll() method with no argument evicts all unmodified in-memory (persistent-clean) instances.

This is useful for applications that need to retrieve many persistent objects, more than would normally fit into memory, during a transaction. The application can call the evict methods to release instances after it has finished with them so that they can potentially be garbage collected during the transaction and make room for other persistent objects.

Typically, it makes sense only to evict unmodified instances, although evicting a modified instance is guaranteed to not cause any lose of in-memory changes. An implementation may choose to ensure this by ignoring the eviction of modified instances. Some implementations may support eviction of modified instances, perhaps by storing the changes in the underlying datastore.

These methods are generally useful within a transaction; however, if the RetainValues property is true, then these methods can be used to evict specific instances after a transaction ends also.

Garbage Collection

During a transaction, a **PersistenceManager** needs to maintain a reference to all in-memory persistent objects, even if the application no longer references them the JDO runtime does (although some implementations may choose to use weak references). This means that during a transaction, in-memory persistent objects typically won't be available for garbage collection.

Eviction allows the garbage collector to do its job and collect instances that are no longer referenced by the application.

void refresh (Object obj)
void refreshAll (Object[] objs)
void refreshAll (Collection objs)
void refreshAll()

The refresh methods re-retrieve the values of the fields from the data-store for the specified instances.

These methods re-retrieve field values, even for instances that have been modified (persistent-dirty), in which case any in-memory changes are lost. Refreshed instances transition to being in-memory and unmodified (persistent-clean).

Typically, these methods make sense only for optimistic transactions where the in-memory state of a persistent object is not necessarily up to date with the datastore. These methods allow an application to refresh an instance, perhaps before making a modification, to reduce the likelihood of a conflicting update being detected when the transaction ends.

These methods can typically be used only within a transaction, unless a JDO implementation supports non-transactional read, in which case they can be used outside of a transaction also. If used outside of a transaction, refreshAll() is a no-op.

[View full width]

```
void retrieve (Object obj)
void retrieveAll (Object[] objs)
void retrieveAll (Object[] objs, boolean flag)[*]
void retrieveAll (Collection objs)
void retrieveAll (Collection objs, boolean flag)[*]
```

[*] These methods are defined in the 1.0.1 maintenance revision to the JDO specification and may not be supported by all JDO implementations.

The retrieve methods retrieve the values of the fields from the datastore for the specified instances. By default, these methods retrieve values for all fields, not just those in the default fetch group. A flag can optionally be used to control this behavior: If true, all field values are retrieved; if false, only values for fields in the default fetch group are retrieved.

Unlike the refresh methods, retrieve does not overwrite field values that have already been retrieved; therefore, modified fields are not overwritten.

These methods have two primary uses. One is to ensure that all the field values for an instance are in memory. This is useful before trying to make a persistent object transient. If a persistent object is made transient before retrieving all its field values, then some of the fields will have only their default values.

The second is as a performance optimization. This is useful when iterating through the result of a query: Rather than retrieving each object individually, an application can call a retrieve method and retrieve them all at once, avoiding multiple calls to the underlying datastore.

These methods can typically be used only within a transaction, unless a JDO implementation supports non-transactional read, in which case they can be used outside of a transaction also.

Instance Management Methods These methods allow an application to interact with the PersistenceManager and PersistenceCapable instances that it manages:

```
Object getObjectId (Object obj)
Object getTransactionalObjectId (Object obj)
```

Every persistent object has an identity associated with it. An application can get the identity of a persistent object by calling the getObjectId() method.

If a persistence-capable class uses application identity, then it may be possible to modify a persistent object's identity during a transaction (by modifying the values of the primary key fields). The PersistenceManagerFactory option javax.jdo.option.ChangeApplicationIdentity denotes whether this is supported. If a persistent object's identity is changed during a transaction, then getObjectId()returns the original identity, as it was at the start of the transaction. If an application needs the changed identity, then it can call getTransactionalObjectId(). If changing application identity is not supported or the application identity hasn't been changed or application identity isn't being used, then calling getTransactionalObjectId() is the same as calling getObjectId().

These methods are useful to get the identity of a persistent object and use this to retrieve the persistent object at a later time or with a different PersistenceManager by calling getObjectById().

These methods can be used both in and outside of a transaction. If used outside of a transaction, getTransactionalObjectId() is equivalent to getObjectId().

Object Identity

The object identity returned from **getObjectId()** is essentially an opaque type. The application need know nothing about it. If using application identity, the implementer of the persistence-capable class is responsible for providing the object identity class; otherwise, the class is provided by the JDO implementation itself. Regardless, any object identity class has a number of characteristics on which the application can rely.

One of these is that it must implement **Serializable**—this allows a persistent object's identity to be sent to another application or stored somewhere for later use.

Another one is that the **equals()** and **hashCode()** methods for the class have to work based on the underlying object identity being represented. This allows an application to compare two persistence objects from different **PersistenceManager** instances based on their identities.

Yet another is that it must have a constructor that takes a single string argument that re-creates the object identity based on what was returned by the identity class's **toString()** method. The constructor is used by the **newObjectIdInstance()** method on **PersistenceManager** to recreate an object identity from a string. This allows an application to convert an object identity to a string that can be later used to re-create it, avoiding the need to serialize the object identity instance itself.

Class getObjectIdClass (Class pcClass)

If a persistence-capable class uses application identity, then this method can be used to retrieve the actual class used to represent the object identity for the class. If the specified class does not use application identity or is null, then this method returns null.

This method is useful only if the application needs to know for some reason the class that implements the object identity for a persistence-capable class.

This method can be used both in and outside of a transaction.

Object newObjectIdInstance (Class pcClass, String oid)
Object getObjectById (Object oid, boolean validate)

These methods can be used to return a persistent object given its identity. If the persistence-capable class uses application identity, then these methods can be used between PersistenceManager instances from different JDO implementations; otherwise, they only work across PersistenceManager instances from the same JDO implementation.

The newObjectIdInstance() method is used to create an object identity instance from a string previously returned by the toString() method. The persistence-capable class of the persistent object whose identity is represented by the specified string needs to be provided.

The getObjectById() method returns the persistent object represented by the specified object identity instance. If the persistent object is already in memory, a reference to it is simply returned. If it is not, an in-memory instance is created with the specified identity. Whether the persistent object is actually retrieved from the datastore is dependent on the JDO implementation and the validate flag.

If validate is true, then the PersistenceManager checks that a persistent object with the specified identity actually exists and, if called within a transaction, actually retrieves it from the datastore. If there is no persistent object with the specified identity in the datastore, then JDOObjectNotFoundException[*] is thrown.

[*] This exception was introduced in the 10.1 maintenance release. Prior to this **JDODatastoreException** would have been thrown.

If validate is false, it is up to the JDO implementation as to whether it validates, whether there is actually a persistent object in the datastore with the specified identity, and whether it retrieves it. If this method doesn't validate that the persistent object exists, then JDOObjectNotFoundException[*] is thrown when the in-memory instance is used instead.

These methods can be used both in and outside of a transaction:

void makePersistent (Object obj)
void makePersistentAll (Object[] objs)
void makePersistentAll (Collection objs)

These methods can be used to explicitly make transient instances persistent so that they are stored in the datastore after the transaction successfully completes. A transient instance transitions to being a new persistent object (persistent-new) as a result of this method. If a specified instance is already persistent, it is left unchanged, although if it belongs to another PersistenceManager instance, JDOUserException is thrown.

After the transaction is committed, all persistent objects transition to being hollow (unless RetainValues is being used). If the transaction rolls back instead, then any new persistent objects (persistent-new) transition back to being transient again.

These methods can be used only within a transaction; if called outside of a transaction, JDOUserException is thrown:

```
void deletePersistent (Object obj)
void deletePersistentAll (Object[] objs)
void deletePersistentAll (Collection objs)
```

These methods can be used to explicitly delete persistent objects, so that they are no longer stored in the datastore after the transaction successfully commits. A persistent object transitions to being a deleted persistent object (persistent-deleted or persistent-new-deleted) as a result of this method. If a specified instance has already been deleted within the transaction, it is ignored. If an instance belongs to a different PersistenceManager instance or is transient, JDOUserException is thrown.

After the transaction is committed, deleted persistent objects transition to being transient and lose their object identity. If the transaction rolls back instead, then deleted persistent objects transition to being a hollow again (unless RestoreValues is being used).

These methods can be used only within a transaction; if called outside of a transaction, JDOUserException is thrown.

Cascade Delete

Unlike making objects persistent, deleting objects deletes only the specified instances. There is no reachability algorithm; referenced persistent objects are not deleted (although some JDO implementations may provide vendor extensions that do this).

If an application needs to ensure that referenced persistent objects are also deleted, then a persistence-capable class can implement the **jdoPreDelete()** method defined by the **InstanceCallbacks** interface and explicitly delete the referenced instances. See section on **InstanceCallbacks** for more details.

```
void makeTransient (Object obj)
void makeTransientAll (Object[] objs)
void makeTransientAll (Collection objs)
```

These methods can be used to detach a persistent object from its PersistenceManager and make it transient. This allows the instance to be used outside of the context of the PersistenceManager instance. These methods do not affect persistent objects in the underlying datastore; it's just a change to the in-memory instance itself. The instance retains any field values as-is, but no longer has any object identity.

The effect of this method is immediate and is not subject to rollback. If a modified persistent (persistent-dirty) object is made transient, JDOUserException is thrown. If an instance is already transient, it is ignored.

These methods are useful if an application needs to move a persistent object to a different PersistenceManager or pass it to non-JDO application. If required later, the application needs to maintain the object identity of the persistent object because, when it is made transient, it loses its identity. Because the field values of the persistent object are left as-is when it is made transient, the application should ensure that all the field values have been retrieved from the datastore prior to making it transient. This can be done using retrieve methods on PersistenceManager.

```
void makeTransactional (Object obj)
void makeTransactionalAll (Object[] objs)
void makeTransactionalAll (Collection objs)
```

These methods can be used to make a transient or persistent but non-transactional instance transactional.

If the specified instances are transient and the JDO implementation supports the transient transactional optional feature, then the instances transition to being transient transactional (transient-clean). If not supported, JDOUnsupportedOptionException is thrown. When used with transient instances, these methods can be used both in and outside of a transaction.

If the specified instances are non-transactional persistent objects, then they transition to being persistent again (persistent-clean). The field values of the persistent objects are re-retrieved from the datastore. When used with non-transactional persistent objects, these methods can be called only within a transaction; otherwise, JDOUserException is thrown.

The affect of these methods is immediate and is not subject to rollback (i.e., once made transient transactional, an instance remains so, even if the transaction is rolled back).

```
void makeNontransactional (Object obj)
void makeNontransactionalAll (Object[] objs)
void makeNontransactionalAll (Collection objs)
```

These methods can be used to make transient transactional or persistent instances non-transactional.

If the specified instances are transient transactional, then they transition to being transient. If an instance has been modified (transient-dirty) within the current transaction, then JDOUserException is thrown.

If the specified instances are persistent and the JDO implementation supports the non-transactional optional feature, then they transition to being persistent, non-transactional (persistent-nontransactional). If not supported, JDOUnsupportedOptionException is thrown. If an instance has been modified (persistent-dirty) within the current transaction, then JDOUserException is thrown.

The affect of these methods is immediate and is not subject to rollback (i.e., once made non-transactional an instance remains so, even if the transaction is rolled back).

These methods can be used both in and outside of a transaction.

Factory Methods PersistenceMananger acts as a factory for Query, Extent, and Transaction instances. The following factory methods can be used to construct these instances:

```
Query newQuery()
Query newQuery (Object query)
Query newQuery (String language, Object query)
Query newQuery (Class cls)
Query newQuery (Extent extent)
Query newQuery (Class cls, Collection cln)
Query newQuery (Class cls, String filter)
Query newQuery (Class cls, Collection cln, String filter)
Query newQuery (Extent extent, String filter)
```

These methods return a Query instance that can be used to find persistent objects based on a filter. See the section on Query for more details.

These methods can be used both in and outside of a transaction.

```
Extent getExtent (Class cls, boolean subclasses)
```

This method returns an Extent instance that represents all the persistent objects of the specified class in the datastore. See section on Extent for more details.

If subclasses is true, then the returned Extent also represents all persistent objects of the specified class and all subclasses. If false, the Extent represents only persistent objects of the specified class itself.

This method can be used both in and outside of a transaction.

If the requires-extent attribute is false in the JDO metadata for the persistence-capable class, then this method throws JDOUserException.

```
Transaction currentTransaction()
```

This method returns the current Transaction instance associated with the PersistenceManager. This allows the application to begin and end transactions. See the section on Transaction for more details.

5.3.2.4 Usage guidelines

The following examples demonstrate how to use the various APIs defined on PersistenceManager:

Connection Management Methods Getting a PersistenceManager instance is straightforward; however, for a server-side application, care needs to be taken to ensure that the PersistenceManager instance is always closed. The easiest way to do this is to wrap the use of a PersistenceManager instance within a try block and use a finally block to ensure that the PersistenceManager instance gets closed, even if an exception is thrown. The finally block can also ensure that any active transaction is first rolled back before the PersistenceManager instance is closed.

The following code snippet taken from PersistenceManagerExample.java shows how to use try and finally to ensure that a PersistenceManager instance is closed:

```
PersistenceManager pm = null;

try {

  pm = pmf.getPersistenceManager();

  // Do something interesting

  pm.close();
}

finally {

  if (pm != null && !pm.isClosed()) {

    if (pm.currentTransaction().isActive()) {

      pm.currentTransaction().rollback();
    }

    pm.close();
  }
}
```

Transactions The currentTransaction() method returns the Transaction instance for the PersistenceManager instance. For a non-mananged application, the Transaction instance can be used to begin and end transactions. The following code snippet taken from TransactionExample.java shows how to begin and commit a transaction:

```
Transaction tx = pm.currentTransaction();

tx.begin();

// Do something interesting

tx.commit();
```

Extents The getExtent() method can be used to get a collection of all the instances of a given persistence-capable class (and optionally subclasses) in the datastore. The following code snippet taken from ExtentExample.java shows how to iterate through all instances of the Book class (and subclasses):

```
Extent extent = pm.getExtent(Book.class, true);

Iterator books = extent.iterator();

System.out.println("Listing of all books:");

while (books.hasNext()) {

  Book book = (Book) books.next();

  System.out.println("  " + book.getTitle());
}
extent.close(books);
```

Evict Methods The previous example retrieved all instances of the Book class; however, if a large number of books are in the datastore, then they may not all fit into memory at the same time. The evict methods can be useful when an application needs to access a large number of persistent objects within a single transaction, but not all instances can possibly fit in memory. The following code snippet taken from EvictExample.java shows how to use the evict() method to evict each Book instance after it has been accessed:

```
Extent extent = pm.getExtent(Book.class, true);

Iterator books = extent.iterator();

System.out.println("Listing of all books:");
while (books.hasNext()) {

  Book book = (Book) books.next();

  System.out.println("  " + book.getTitle());

  pm.evict(book);
}

extent.close(books);
```

Retrieve Methods The previous example retrieved each persistent object one at a time from the datastore. As an optimization, the retrieveAll() method can be used to potentially retrieve multiple persistent objects from the datastore in one go. This can improve the performance of the application by reducing the number of times the underlying datastore needs to be accessed. The following code snippet taken from RetrieveExample.java shows how the retrieveAll() method can be used to retrieve a group of instances:

```
Extent extent = pm.getExtent(Book.class, true);

Iterator books = extent.iterator();

System.out.println("Listing of all books:");

List list = new ArrayList(1024);

while (books.hasNext()) {

  list.add(iter.next());

  if (list.size() == 1024 || !books.hasNext()) {

    pm.retrieveAll(list);

    for (int i = 0; i < list.size(); ++i) {

      Book book = (Book) list.get(i);

      System.out.println("  " + book.getTitle());
    }

    pm.evictAll(list);

    list.clear();
  }
}

extent.close(books);
```

If supported, a further optimization would be to use a retrieveAll() method that takes a Boolean flag to indicate that only the fields in the default fetch group are required. These methods were introduced in the 1.0.1 maintenance revision, so they may not be supported by all JDO implementations initially.

Timing the two examples EvictExample.java and RetrieveExample.java using the same dataset shows how retrieving objects can dramatically improve the performance of an application.

Refresh Methods These methods are primarily useful when using an optimistic transaction or using non-transactional instances to re-retrieve the fields of an instance from the datastore. However, another use is as a way to undo an in-memory modification to a persistent object within a transaction without rolling back the entire transaction. If an application decides that a change should not have been made to a particular instance, it is possible to use refresh() to re-retrieve the instance's fields from the datastore and thus overwrite any modified fields. The following code snippet taken from RefreshExample.java shows how retrieve() can be used to undo a change made to a book's title:

```
tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

tx.begin();

System.out.println(
  "Author's name is '" + author.getName() + "'.");

author.setName("Sameer Tyagi");

System.out.println(
  "Author's name changed to '" + author.getName() + "'.");

pm.refresh(author);

System.out.println(
  "Author's name after refresh is '"
  + author.getName() + "'.");

tx.commit();
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
Author's name changed to 'Sameer Tyagi'.
Author's name after refresh is 'Keiron McCammon'.
```

Object Identity Methods A persistence object's identity is represented by an instance of a normal Java class. An object identity instance is a useful way to pass a reference to a persistent object between PersistenceManager instances and even between PersistenceManager instances in different JVMs (because it is serializable).

The following code snippet taken from ObjectIdentityExample.java shows how to get the identity of a persistent object and use it to create a reference to the persistent object in a different PersistenceManager instance:

```
PersistenceManager pm = pmf.getPersistenceManager();

Transaction tx = pm.currentTransaction();

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

Object oid = pm.getObjectId(author);

System.out.println("Author's object identity is: " + oid);

pm.close();

PersistenceManager pm2 = pmf.getPersistenceManager();

Transaction tx2 = pm2.currentTransaction();

tx2.begin();

Author author2 = (Author) pm.getObjectById(oid, true);

System.out.println("Author is: " + author2.getName());

tx2.commit();

pm2.close();
```

Although the above example shows how to use a persistent object's identity to re-create a reference to it in a different PersistenceManager instance within the same JVM, it could easily have been passed via RMI to a different JVM altogether. In this case, the object identity instance just gets serialized and passed over the wire.

In some situations, it is desirable to pass the identity of a persistent object to a non-JDO application. This application would then use the identity as a parameter in additional requests (a Web browser, for example). In this situation, the object identity instance can be converted to a string representation. The following code snippet taken from StringObjectIdentityExample.java shows how to convert a persistent object's identity to a string and re-create a reference to the persistent object again from the string:

```
Transaction tx = pm.currentTransaction();

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

String oid = pm.getObjectId(author).toString();

System.out.println("Author's object identity is: " + oid);

pm.close();

PersistenceManager pm2 = pmf.getPersistenceManager();

Transaction tx2 = pm2.currentTransaction();

tx2.begin();

Author author2 =
  (Author) pm.getObjectById(
    pm.newObjectIdInstance(Author.class, oid),
    true);

System.out.println("Author is: " + author2.getName());

tx2.commit();
```

The toString() method is used to convert the persistent object's identity to a string. In a different PersistenceManager instance, newObjectIdInstance() is used to re-create an object identity instance from the string; this is then used to get a reference to the original persistent object.

Make and Delete Persistent Methods The makePersistent() and deletePersistent() methods create and delete persistent objects in the underlying datastore.

The makePersistent() methods mark in-memory transient instances of persistence-capable classes as persistent. The methods can be called at any point within a transaction, and because of persistence by reachability, only the root of a graph of interconnected instances needs to be made persistent. The following code snippet taken from MakePersistentExample.java creates instances of Author, Address, Book, and Publisher and associates them together. Because an Author references an Address and a Book, and a Book references a Publisher, only the Author instance needs to be made persistent. However, it wouldn't matter if the application also made the Address, Book and Publisher instances explicitly persistent (which might be the case if there is doubt about what is reachable from where):

```
tx.begin();

Author author =
  new Author(
    "Keiron McCammon",
    new Address(
      "6539 Dumbarton Circle",
      "Fremont", "CA", "94555"));

Publisher publisher =
  new Publisher("Sun Microsystems Press");

Book book =
  new Book("Core Java Data Objects", "0-13-140731-7");

author.addBook(book);

publisher.addBook(book);

pm.makePersistent(author);

tx.commit();
```

The deletePersistent() methods mark in-memory persistent objects as deleted. The methods can be called at any point during a transaction, but unlike makePersistent(), they affect only the specific persistent object being deleted. It is the responsibility of the application to delete any referenced persistent objects that are no longer required. The following code snippet taken from DeletePersistentExample.java gets the first Author instance from the class Extent, deletes the associated Address instance, and removes each Book from the Author:

```
Iterator authors = extent.iterator();

if (authors.hasNext()) {

  Author author = (Author) authors.next();

  pm.deletePersistent(author.getAddress());

  Iterator books = author.getAllBooks();

  while (books.hasNext()) {

    Book book = (Book) books.next();

    author.removeBook(book);
  }

  pm.deletePersistent(author);
}
```

Rather than relying on the application to remember to tidy up every time it deletes a persistent object, a persistence-capable class can implement the InstanceCallbacks interface and use the jdoPreDelete() method to tidy up automatically upon deletion. See the section on InstanceCallbacks for more details.

Make Transient Methods An in-memory persistent object can be made transient using the makeTransient() methods. After it is made transient, the in-memory instance has no object identity and is no longer associated with a PersistenceManager. Because the fields of the instance are left as-is, the application should first use one of the retrieve() methods to ensure that all the fields of the instance have been retrieved from the datastore. The following code snippet taken from MakeTransientExample.java first creates an Author instance in one transaction, and then in another retrieves all its fields and makes it transient. Even after the PersistenceManager instance is closed, the transient Author instance can still be used:

```
PersistenceManager pm = pmf.getPersistenceManager();

Transaction tx = pm.currentTransaction();

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

tx.begin();

pm.retrieve(author);

pm.makeTransient(author);

tx.commit();

pm.close();

System.out.println(
  "Author's name is: " + author.getName());
```

The makeTransient() methods affect only the specific instance being made transient. If referenced persistent objects should also be made transient, it is the application's responsibility to make each instance explicitly transient.

Usage Warning

Care must be taken when making instances transient. If an in-memory persistent object that has been modified references an instance that was made transient, then at commit, due to persistence by reachability, the transient instance is treated as a new persistent object.

It is the application's responsibility to ensure that there are no references to the transient instance. For this reason, great care must be taken when making persistent objects transient.

Transactional and Non-Transactional Methods The behavior of these methods depends on whether a transient or persistent instance is used.

The following code snippet taken from TransientTransactionalExample.java shows how a transient instance of a persistence-capable class can be made transactional so that, on rollback, the instance's fields revert to their values at the beginning of the transaction and how, when made non-transactional again, the fields are left as-is on rollback:

```
Author author = new Author("Keiron McCammon");

System.out.println("Making instance transactional.");

pm.makeTransactional(author);

Transaction tx = pm.currentTransaction();

tx.begin();

System.out.println(
  "Author's name is '" + author.getName() + "'");

author.setName("Sameer Tyagi");

System.out.println(
  "Modified name is '" + author.getName() + "'");

tx.rollback();

System.out.println(
  "Name after rollback is '" + author.getName() + "'");

System.out.println("Making instance non-transactional.");

pm.makeNontransactional(author);

tx.begin();

System.out.println("Name is '" + author.getName() + "'");

author.setName("Sameer Tyagi");

System.out.println(
  "Modified name is '" + author.getName() + "'");

tx.rollback();

System.out.println(
  "Name after rollback is '" + author.getName() + "'");
```

The output would be as follows:

```
Making instance transactional.
Author's name is 'Keiron McCammon'.
Modified name is 'Sameer Tyagi'.
Name after rollback is 'Keiron McCammon'.
Making instance non-transactional.
Name is 'Keiron McCammon'.
Modified name is 'Sameer Tyagi'.
Name after rollback is 'Sameer Tyagi'.
```

The following code snippet taken from PersistentTransactionalExample.java shows how a persistent object can be made non-transactional and accessed outside of the transaction and even modified. It then shows how, when the instance is made transactional again, any modified fields are simply discarded:

```
tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

System.out.println("Making instance non-transactional.");

pm.makeNontransactional(author);

System.out.println(
  "Author's name is '" + author.getName() + "'");

author.setName("Sameer Tyagi");

System.out.println(
  "Modified name is '" + author.getName() + "'");

tx.begin();

System.out.println("Making instance transactional.");

pm.makeTransactional(author);

System.out.println(
  "Transactional name is '" + author.getName() + "'");

tx.commit();
```

The output would be as follows:

```
Making instance non-transactional.
Author's name is 'Keiron McCammon'.
Modified name is 'Sameer Tyagi'.
Making instance transactional.
Transactional name is 'Keiron McCammon'.
```

5.3.3 Extent

An Extent instance essentially represents a collection of all the persistent objects of a persistence-capable class in the datastore. An Extent can be used to iterate through all the persistent objects of the class in no particular order or used to construct a Query.

By default, the JDO runtime in conjunction with the underlying datastore maintains an Extent for every persistence-capable class. Some datastores don't implicitly maintain a collection of all persistent objects being stored per class, in which case a collection of instances must be explicitly maintained instead. This can add overhead. If the application has no need to get an Extent or Query the instances of a class, then it can avoid any unnecessary overhead by setting the requires-extent attribute to false in the metadata for the class. For many datastores, Extent management is implicit (for example, a relational database always knows all the rows in a table). In this case, setting the requires-extent attribute to false makes no difference.

The Extent instance itself doesn't actually reference persistent objects; rather, it just maintains a reference to a persistence-capable class, a subclasses flag, and a collection of open Iterators. It's the Iterators created by the Extent or Query instances created using the Extent that return the persistent objects.

Because the instances of persistence-capable classes can vary over time as different applications create and delete persistent objects, Iterators created by an Extent at different points may return different sets of persistent objects. An Iterator returns the persistent objects in the datastore at the time the Iterator is created. Depending on the IgnoreCache property of the PersistenceManager, the Iterator may or may include instances made persistent or deleted within the current transaction.

5.3.3.1 Creating an Extent

An Extent is created using the getExtent() method on PersistenceManager:

```
public Extent getExtent(Class cls, Boolean subclasses)
```

The boolean flag controls whether the Extent contains persistent objects of just the specified class or includes persistent objects of all subclasses also.

5.3.3.2 Extent properties

Extent defines three read-only properties that are set when the Extent is created, as shown in Table 5-6.

Table 5-6. Extent Properties

Property	Method Summary
CandidateClass	Class getCandidateClass()
This property denotes the persistence-capable class specified when the **Extent** instance was created.	
PersistenceManager	PersistenceManager getPersistenceManager()
This property denotes the PersistenceManager that created the Extent instance.	
Subclasses	boolean hasSubclasses()
This property determines whether the Extent contains persistent objects of the specified class and its subclasses. If true, then the Extent includes all persistent objects of the specified class and all subclasses. If false, it includes persistent objects of the specified class only.	

5.3.3.3 Method summary

Extent provides a method to get an Iterator (that can then be used to iterate through all the persistent objects represented by the Extent) and methods to close iterators after they have been used.

void close (Iterator iter)

This method closes the specified Iterator. Once closed, the iterator's hasNext() method returns false, and next()throws NoSuchElementException.

void closeAll()

This method closes all iterators that have been created from the Extent. Once closed, any iterator's hasNext() method returns false, and next() throws NoSuchElementException.

This method closes only open iterators; the Extent itself is still valid and can be used to create new iterators or Query instances.

Iterator iterator()

This method returns an Iterator. The iterator can be used to iterate through all the persistent objects represented by the Extent.

If the IgnoreCache property is false, then the Iterator returns instances that have been made persistent within the current transaction and doesn't return instances that have been deleted within the current transaction. As an optimization, if the PersistenceManager instances IgnoreCache property is true, then the returned Iterator may or may not return instances that have been made persistent within the current transaction and may or may not return instances that have been deleted within the current transaction.

When an application has finished with an Iterator, it is closed using the close() method. Alternatively, all iterators can be closed in one go using the closeAll() method.

The returned Iterator does not implement the remove() method; if called, UnsupportedOperationException is thrown.

Closing Iterators

It is a best practice to ensure that any Iterator is closed when finished with. Depending on the JDO implementation, this may release datastore resources used by the Iterator (a database cursor, for example).

5.3.3.4 Usage guidelines

The following code snippet taken from ExtentExample.java shows how to iterate through all instances of the Book class, closing the Iterator when finished:

```
Extent extent = pm.getExtent(Book.class, true);

Iterator books = extent.iterator();

System.out.println("Listing of all books:");

while (books.hasNext()) {

  Book book = (Book) books.next();

  System.out.println("  " + book.getTitle());
}

extent.close(books);
```

5.3.4 Query

JDO defines both a query API and a query language. The API allows an application to construct and execute a query programmatically, and the query language allows an application to specify the persistent objects that it wants. The query language is called JDOQL (JDO Query Language), and its syntax is a Java Boolean expression.

Essentially, a JDO application uses a JDOQL string (referred to as a filter) to create a Query instance; this instance is then executed and returns a collection of persistent objects that satisfy the specified filter.

As well as its own query language, JDO allows implementations to offer additional query languages. An application would use the same query APIs defined by JDO; however, the syntax of the query string would instead be SQL, OQL, or some other proprietary language. JDO mandates support for JDOQL, but leaves support for additional languages as optional. An implementation that supports additional query languages would include them in the list of supported options returned from the supportedOptions() method on PersistenceManager.

A Query consists of a set of candidate instances, which can be specified using a Java class, an Extent, or a Java collection, and a filter string. In addition, it is possible to also declare import statements, parameters, and variables, as well as an ordering for the set of results. When executed, a Query takes the set of candidate instances and returns a Java collection of references to the instances that satisfy the query filter.

The goal of JDOQL is to provide a simple query grammar that is familiar to Java programmers and that can be executed by the JDO implementation, possibly by converting it to a different representation and passing it to the underlying data-store. JDO makes no assumptions about where the query is actually executed. Depending on the JDO implementation and the underlying datastore being used, a query may be executed be retrieving all the persistent objects into memory, or it may simply be passed to the underlying datastore.

See Chapter 6 for more details on the JDOQL syntax.

5.3.4.1 Creating a Query

A Query is created using the newQuery() methods on PersistenceManager. There are essentially five ways to create a Query:

- Create an empty one, and specify the candidate instances and filter later.
- Create a copy of another one.
- Create one in which the candidate instances are denoted by a Java class.
- Create one in which the candidate instances are denoted by an Extent.
- Create one in which the candidate instances are denoted by a Java collection.

The set of candidate instances for a Query should be specified one way, using a Java class, an Extent, or a Java collection.

Query newQuery()

Creates an empty Query.

The candidate instances and filter have to be specified before the query can be executed.

Query newQuery (Object query)

Creates a Query as a copy of the specified Query.

This is useful when creating a Query based on an existing Query owned by a different PersistenceManager. It can also be used to create a Query based on a Query that was previously de-serialized.

Query newQuery (String language, Object query)

Creates a Query for the specified query language.

The string used to denote the language being used and the query parameter are both implementation-specific.

Query newQuery (Class cls)
Query newQuery (Class cls, String filter)

Creates a Query using the specified Class to denote the set of candidate instances.

The candidate instances for a Query created using a class include all persistent objects in the datastore of the specified persistence-capable class and all subclasses.

Query newQuery (Extent extent)
Query newQuery (Extent extent, String filter)

Creates a Query using the specified Extent to denote the set of candidate instances.

If the subclasses flag for the Extent is false, then the set of candidate instances includes persistent objects in the datastore of persistence-capable class used to create the Extent only. Otherwise, the candidate instances include persistent objects of all subclasses also.

Query newQuery (Class cls, Collection cln)
Query newQuery (Class cls, Collection cln, String filter)

Creates a Query using the specified Collection to denote the set of candidate instances. The specified class denotes the persistence-capable class of the instances contained in the collection.

A Query created using a Collection queries on only the instances contained in the collection.

5.3.4.2 Query properties

Query defines two properties, as shown in Table 5-7.

Table 5-7. Query Properties

Property	Method Summary
IgnoreCache	boolean getIgnoreCache() void setIgnoreCache(boolean flag)
This property is a hint as to whether the Query can ignore in-memory changes when executed. The initial value is the same as the setting of the PersistenceManager instance's IgnoreCache property at the time the Query was constructed.	
PersistenceManager	PersistenceManager getPersistenceManager()
This property denotes the PersistenceManager used to create the Query.	

5.3.4.3 Method summary

The Query methods can be grouped into the following categories:

- Query Definition Methods
- Query Execution Methods

Query Definition Methods The following methods are used to define a Query. Before a Query can be executed, a set of candidate instances and a filter have to be set as a minimum.

void declareImports (String imports)

It is possible to provide import statements that are used to resolve type ambiguities within a filter expression.

The imports string is a list of Java import statements separated by semicolons (as per normal Java import statements).

void declareParameters (String params)

It is possible to declare parameters that can be used within a filter expression. The parameters are substituted for actual values when the Query is executed.

The params string is a list of Java parameters (type and identifier) separated by commas (as per normal Java parameter declarations for a method).

void declareVariables (String vars)

It is possible to declare variables that can be used within a filter expression.

The vars string is a list of Java local variables separated by semicolons (as per normal Java local variable declarations).

void setClass (Class cls)

This method sets the candidate class for the Query as the specified persistence-capable class.

If a setCandidates() method is not used, then the set of candidate instances is all those persistent objects in the datastore belonging to the specified class and its subclasses.

void setCandidates (Extent extent)

This method specifies the set of candidate instances as those belonging to the specified Extent.

If the subclasses flag for the Extent is false, then the set of candidate instances includes persistent objects in the datastore of the persistence-capable class used to create the Extent only. Otherwise, the candidate instances include persistent objects of all subclasses also.

There is no need to call setClass() because an Extent implicitly specifies the candidate class for the Query as the persistence-capable class used to create the Extent.

void setCandidates (Collection collection)

This method restricts the set of candidate instances for a class to those contained in the collection.

The setClass() method must be used to specify the persistence-capable class of the instances in the collection.

void setFilter (String filter)

This method sets the filter for the Query.

See the chapter on JDOQL for more details on the format of the filter string.

void setOrdering (String ordering)

It is possible to specify that persistent objects should be returned sorted in an ascending or descending order.

The ordering string is a list of fields, followed by the ascending or descending keyword, each separated by a comma.

Query Execution Methods The following methods are used to execute a Query and get a collection of persistent objects that satisfy the filter.

void close (Object result)

This method closes the result returned from an execute() method. Any Iterators retrieved from a result after it has been closed return false to hasNext(), and next() throws NoSuchElementException.

void closeAll()

This method closes all the results returned from an execute() method.

The Query itself can still be used; only the results returned by the execute() methods are closed.

void compile()

This method validates that the Query is valid; if not, JDOUserException is thrown.

It also acts as a hint to the JDO implementation that the Query should be prepared ready for execution.

Object execute()
Object execute (Object param)
Object execute (Object param1, Object param2)
Object execute (Object param1, Object param2, Object param3)
Object executeWithArray (Object[] params)
Object executeWithMap (Map params)

After a Query has been defined, it can be executed. An execute() method returns an Object. If the query language being used is JDOQL, then returned Object is an un-modifiable java.util.Collection that contains the persistent objects that satisfy the filter. If the query language is not JDOQL, then the class of the returned Object is implementation-specific.

The returned java.util.Collection can be used to iterate through the set of persistent objects that satisfy the specified filter. UnsupportedOperationException is thrown if the collection is modified in any way. The size() method may return Integer.MAX_VALUE if the number of instances is unknown. Depending on the JDO implementation, the execute() method may use a cursor of some kind, so the actual number of instances in the result set is not known until they have all been retrieved.

If the Query does not have any parameters, then the execute() method with no parameters can be used.

If the Query defines up to three parameters, then one of the execute() methods that takes one, two, or three Object parameters can be used. The Object parameters are bound in order to the parameters defined by the declareParameters() method (the first parameter defined is bound to the first value specified in the execute() method and so on). The types of the parameters specified in the execute() method should match those specified by the declareParamaters() method; otherwise, JDOUserException is thrown. For primitive types, an instance of the equivalent wrapper class should be used.

If more than three parameters are defined, then the executeWithArray() or executeWithMap() method should be used.

The executeWithArray() method takes an array of parameters and binds each one in order to the parameters defined by the declareParameters() method.

The executeWithMap() method takes a map of parameters. Each key should be a string that matches the name of a parameter defined by the declareParameters() method. The associated value is bound to the matching parameter in the Query.

If a value for a parameter is not specified when a Query is executed, JDOUserException is thrown.

Closing Results

It is a best practice to ensure that any result returned from an **execute()** method is closed when finished with. Depending on the JDO implementation, this may release datastore resources used during the execution of the Query (a cursor, for example).

5.3.4.4 Usage guidelines

Chapter 6 provides greater details on the JDOQL syntax and how to use queries within a JDO application. The following examples serve to provide an introduction to the basics.

Finding a persistent object by a field value requires only a simple query. The following code snippet taken from SimpleQueryExample.java shows how to create and execute a simple query to find all instances of Book with a given title. The Book's title is hard coded into the filter string used to create the Query:

```
Query query =
  pm.newQuery(
    Book.class,
    "title == \"Core Java Data Objects\"");

Collection result = (Collection) query.execute();

Iterator iter = result.iterator();

while (iter.hasNext()) {

  Book book = (Book) iter.next();

  System.out.println(
    book.getTitle() + " - " + book.getISBN());
}

query.close(result);
```

Rather than hard coding a literal in the filter string, a parameter can be used instead. The following code snippet taken from SimpleQueryWithParameterExample.java is the same as the previous example, except that it defines a parameter to specify the Book's title rather than hard coding it. The this keyword is used to resolve the ambiguity between the class's field name and the parameter's name (both of which are called title):

```
Query query =
  pm.newQuery(Book.class, "this.title == title");

query.declareParameters("String title");

Collection result = (Collection)
  query.execute("Core Java Data Objects");

Iterator iter = result.iterator();

while (iter.hasNext()) {

  Book book = (Book) iter.next();

  System.out.println(
    book.getTitle() + " - " + book.getISBN());
}
query.close(result);
```

The results from a query are not returned in any particular order by default. To specify an order, use the setOrdering() method. The following code snippet taken from SimpleQueryWithOrderingExample.java extends the previous example, but specifies that the returned Book instances should be ordered by their ISBN numbers:

```
Query query =
  pm.newQuery(Book.class, "this.title == title");

query.declareParameters("String title");

query.setOrdering("isbn ascending");

Collection result = (Collection)
  query.execute("Core Java Data Objects");

Iterator iter = result.iterator();

while (iter.hasNext()) {

  Book book = (Book) iter.next();

  System.out.println(
    book.getTitle() + " - " + book.getISBN());
}

query.close(result);
```

A Query can also use navigation to specify fields of referenced instances. The following code snippet taken from SimpleQueryWithNavigationExample.java shows how to use the "." notation to navigate to fields of a referenced instance. The query returns all books published by a given publisher, using the "." notation to check the name field of the Publisher instance referenced by the publisher field for each book:

```
Query query =
  pm.newQuery(Book.class, "publisher.name == name");

query.declareParameters("String name");

Collection result = (Collection)
  query.execute("Sun Microsystems Press");

Iterator iter = result.iterator();

while (iter.hasNext()) {

  Book book = (Book) iter.next();

  System.out.println(
    book.getTitle() + " - " + book.getISBN());
}

query.close(result);
```

A Query can also navigate to multiple persistent objects contained in a collection and check the fields of those contained persistent objects. If there is at least one persistent object in the collection that matches, then the referencing persistent object is returned. The following code snippet taken from SimpleQueryUsingContainsExample.java shows how to use contains() and a variable to navigate to fields of persistent objects contained in a collection. The query returns all Books by a given Author. Essentially, for each Book, the Authors contained in the authors collection are checked to see whether the name matches that given. If there is at least one Author contained in the collection with the given name, then the Book is returned:

```
Query query =
  pm.newQuery(
    Book.class,
    "authors.contains(author) && author.name == name");

query.declareParameters("String name");
query.declareVariables("Author author");

Collection result = (Collection)
  query.execute("Keiron McCammon");

Iterator iter = result.iterator();

while (iter.hasNext()) {

  Book book = (Book) iter.next();
  System.out.println(
    book.getTitle() + " - " + book.getISBN());
}

query.close(result);
```

See Chapter 6 for more details on using JDOQL.

5.3.5 Transaction

The Transaction interface provides the methods that allow an application to control the underlying datastore transactions. Each PersistenceManager has an associated Transaction instance. A Transaction is initially inactive, becomes active after begin(), and remains active until either commit() or rollback() (or JDOFatalDataStoreException is thrown), at which point it becomes inactive again.

5.3.5.1 Creating a Transaction

The currentTransaction() method defined on PersistenceManager can be used to get the Transaction instance associated with a PersistenceManager; each PersistenceManager has one Transaction instance:

```
public Transaction currentTransaction()
```

5.3.5.2 Transaction properties

Transaction defines a number of properties that control various semantics of a transaction. Once set, the properties of a Transaction remain so until changed again (they are not affected by transaction completion).

Support for a number of the Transaction properties is optional; if an implementation does not support a particular optional feature, then JDOUnsupportedOptionException is thrown if the property is set to true, as shown in Table 5-8.

Table 5-8. Transaction Properties

Property	Method Summary
NontransactionalRead[*]	boolean getNontransactionalRead() void setNontransactionalRead(boolean flag)
This property determines whether non-transactional reads are allowed. If true, they are allowed; if false, an attempt to access a persistent object outside of a transaction results in JDOUserException being thrown. If non-transactional read is not supported by a JDO implementation, JDOUnsupportedOption is thrown if an attempt is made to set this property to true.	
NontransactionalWrite[*]	boolean getNontransactionalWrite() void setNontransactionalWrite(boolean flg)
This property determines whether non-transactional writes are allowed. If true, they are allowed; if false, an attempt to modify a persistent object outside of a transaction results in JDOUserException being thrown. If non-transactional write is not supported by a JDO implementation, JDOUnsupportedOption is thrown if an attempt is made to set this property to true.	
Optimistic[*]	boolean getOptimistic() void setOptimistic(boolean flag)
This property determines whether the transaction is optimistic or not. True indicates that the Transaction is optimistic. This property cannot be changed if a transaction is currently active; JDOUserException is thrown. If optimistic transactions are not supported by a JDO implementation, JDOUnsupportedOption is thrown if an attempt is made to set this property to true.	
PersistenceManager	PersistenceManager getPersistenceManager()
This property denotes the PersistenceManager associated with the Transaction instance.	
RestoreValues[*]	boolean getRestoreValues() void setRestoreValues(boolean flag)
This property determines whether the fields of persistent objects are restored to their values as of the beginning of a transaction on rollback. If true, they are restored and the persistent objects transition to being persistent, non-transactional at the end of the transaction.	

This property cannot be changed if a transaction is currently active; JDOUserException is thrown. If restoring values is not supported by a JDO implementation, JDOUnsupportedOption is thrown if an attempt is made to set this property to true.

RetainValues[*]	boolean getRetainValues() void setRetainValues(boolean flag)

This property determines whether the field values of persistent objects should be retained in memory after a transaction ends. If true, they are retained and in-memory persistent objects transition to being persistent, non-transactional at the end of the transaction.

If retaining values is not supported by a JDO implementation, JDOUnsupportedOption is thrown if an attempt is made to set this property to true.

Synchronization	Object getSynchronization() void setSynchronization(Synchronization s)

This property allows an application to register an instance of the Synchronization interface with the Transaction. If not null, then the beforeCompletion() and afterCompletion() methods of the specified Synchronization instance are invoked before and after commit(), and afterCompletion() is invoked after rollback().

If no Synchronization callback is required, then null can be used.

[*] Support is optional.

5.3.5.3 Method summary

The Transaction methods allow an application to explicitly control transactions. If used within a managed environment where transactions are controlled by an external coordinator, any attempt to explicitly begin or end a transaction results in JDOUserException being thrown. Only isActive() can be called when using an external coordinator.

void begin()

This method can be used to begin a transaction.

If called when an external coordinator is being used, JDOUserException is thrown.

void commit()

This method can be used to end a transaction and commit all new, modified, and deleted instances to the datastore.

If called when an external coordinator is being used, JDOUserException is thrown.

boolean isActive()

This is method returns true if the Transaction is active (i.e., begin() has been called, but no commit() or rollback() has yet occurred), false otherwise.

void rollback()

This method can be used to roll back a transaction and discard all changes (new, modified, or deleted instances).

If called when an external coordinator is being used, JDOUserException is thrown.

5.3.5.4 Usage guidelines

Using the Transaction interface is simple. The following code snippet taken from TransactionExample.java shows how to begin and commit a transaction:

```
PersistenceManager pm = pmf.getPersistenceManager();

Transaction tx = pm.currentTransaction();

tx.begin();

// Do something interesting

tx.commit();
```

The following code snippet taken from NonTransactionalReadExample.java shows how a persistent object can be accessed outside of a transaction using a non-transactional read. If setNontransactionalRead(true) was not supported, then accessing a persistent object outside of a transaction would result in JDOUserException being thrown:

```
Transaction tx = pm.currentTransaction();

tx.setNontransactionalRead(true);

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

// Author instance transitions to hollow after commit

System.out.println(
  "Author's name is '" + author.getName() + "'.");

// Author instance is retrieved from the datastore and
// transitions to being non-transactional

if (!JDOHelper.isTransactional(author)) {

  System.out.println("Author is non-transactional.");
}
```

The output would be as follows:

```
Author's name is 'Keiron McCammon'.
Author is non-transactional.
```

The following code snippet taken from RetainValuesExample.java differs from the previous example only in that it calls setRetainValues() instead of setNontransactionalRead(). In the previous example, the Author instance transitions to being hollow after the commit, and when accessed outside of the transaction, its fields are re-retrieved from the datastore and the instance transitions to being non-transactional. With retainValues(), the Author instance transitions to being non-transactional after the commit and its field values are kept in memory. This has the advantage that the fields of the instance don't have to be re-retrieved again:

```
Transaction tx = pm.currentTransaction();

tx.setRetainValues(true);

tx.begin();

Author author = new Author("Keiron McCammon");

pm.makePersistent(author);

tx.commit();

// Author instance transitions to being non-transactional
// after commit and its fields remain in-memory

if (!JDOHelper.isTransactional(author)) {

  System.out.println("Author is non-transactional.");
}

System.out.println(
  "Author's name is '" + author.getName() + "'.");
```

The output would be as follows:

```
Author is non-transactional.
Author's name is 'Keiron McCammon'.
```

RetainValues can be used to retain the fields of persistent objects in memory after a transaction commits. However, if a rollback is used instead of a commit, then the in-memory instances retain the values of any fields that were modified during the transaction, even though the modifications have been rolled back in the datastore. If this is undesirable, then the restore values option can be used to revert the modified fields to the values they had at the beginning of the transaction. The following code snippet taken from RestoreValuesExample.java shows that if the name of an Author instance is modified but the transaction is rolled back, then the modified name remains in memory unless RestoreValues is used:

```
Transaction tx = pm.currentTransaction();

tx.setRetainValues(true);

tx.begin();

Author author = new Author("Keiron McCammon");
pm.makePersistent(author);

tx.commit();

// Author instance is retained after commit and transitions
// to being non-transactional

if (!JDOHelper.isTransactional(author)) {

  System.out.println("Author is non-transactional.");
}

System.out.println(
  "Author's name is '" + author.getName() + "'.");

tx.begin();

author.setName("Sameer Tyagi");

tx.rollback();

// Transaction was rolled back rather than committed
// therefore the name change is not in the datastore.
// However, the changed name is retained in-memory.

System.out.println(
  "Author's name is '" + author.getName() +
  "' after rollback.");

tx.setRestoreValues(true);

tx.begin();

author.setName("Sameer Tyagi");

tx.rollback();

// This time because RestoreValues is true the author's
// name is reverted back to its original name on rollback

System.out.println(
  "Author's name is '" + author.getName() +
  "' after rollback with RestoreValues.");
```

The output would be as follows:

```
Author is non-transactional.
Author's name is 'Keiron McCammon'.
Author's name is 'Sameer Tyagi' after rollback.
Author's name is 'Keiron McCammon' after rollback with RestoreValues.
```

The following code snippet taken from OptimisticExample.java changes the zip code of an Author's address in an optimistic transaction and makes the Author instance transactional. In another transaction, the name of the Author is modified and committed before the original optimistic transaction is committed. When the optimistic transaction is committed, an exception is thrown because the Author instance was modified by the other transaction:

```
Transaction tx = pm.currentTransaction();

tx.begin();

Author author =
  new Author(
    "Keiron McCammon",
    new Address(
      "6539 Dumbarton Circle", "Fremont", "CA", "94555"));

pm.makePersistent(author);

tx.commit();

tx.setOptimistic(true);

tx.begin();

Address address = author.getAddress();

address.setZipcode("12345");

pm.makeTransactional(author);

/*
 * Another PersistenceManager instance is used to modify
 * the author's name in a different transaction.
 */
PersistenceManager pm2 = pmf.getPersistenceManager();

Transaction tx2 = pm2.currentTransaction();

tx2.begin();

/*
 * The object identity of the Author is used to retrieve
 * the Author instance within the context of the new
 * PersistenceManager instance.
 */
Author author2 = (Author)
  pm.getObjectById(JDOHelper.getObjectId(author), false);

author.setName("Sameer Tyagi");

tx2.commit();

pm2.close();

// Because the Author's name was updated in a different
// transaction, the optimistic transaction will throw
// an exception on commit.

try {

  tx.commit();
}

catch (JDOOptimisticVerificationException e) {

  System.out.println("Transaction failed.");
}
```

If the Author instance hadn't been explicitly made transactional, the commit would have been successful, because in the first transaction the Address instance was modified, and in the second transaction the Author instance was modified, which would not have resulted in a conflicting update.

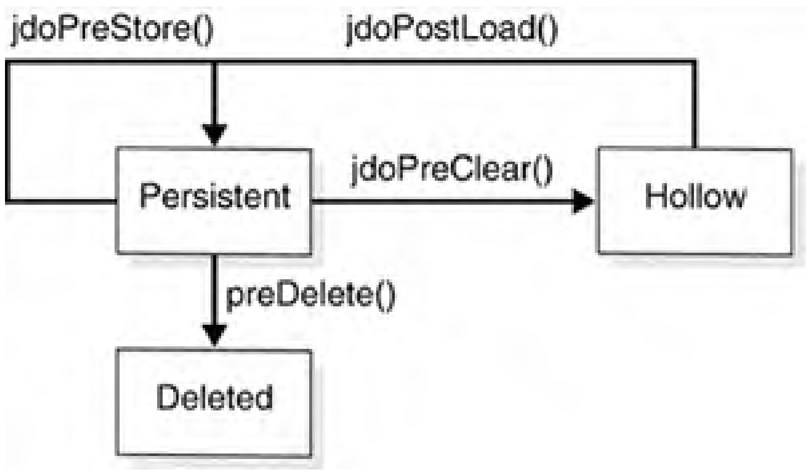
5.3.6 InstanceCallbacks

The InstanceCallbacks interface provides a means whereby instances of persistence-capable classes can take action on certain state changes in a persistent object's lifecycle. The JDO implementation invokes these methods as persistent objects that are stored, retrieved, and deleted.

5.3.6.1 Method summary

Figure 5-4 provides a simplified view of when the various InstanceCallbacks methods are invoked.

Figure 5-4. Invocation of InstanceCallbacks.



When a persistent object is retrieved from the datastore, jdoPostLoad() is invoked. Before a persistent object is sent to the datastore (typically during commit), jdoPreStore()is invoked. If a persistent object is deleted, then jdoPreDelete() is invoked, and before a persistent object transitions to being hollow, jdoPreClear() is invoked.

void jdoPostLoad()

This method is invoked after the fields in the default fetch group have been retrieved from the datastore.

Access to fields in the default fetch group is permitted, but other fields are uninitialized and should not be accessed. Access to other persistent objects from within this method is also not allowed. Any exceptions thrown during this method are ignored.

This method is useful to initialize the values of any non-persistent fields or to register the in-memory persistent object with other instances in the JVM.

void jdoPreClear()

This method is invoked before the fields of an instance are cleared. The fields are cleared when a persistent object transitions to being hollow, typically at the end of a transaction or the result of an eviction.

Access to fields in the default fetch group is permitted but other fields may not have been initialized and should not be accessed. Any exceptions thrown during this method are ignored.

This method is useful to clear the values of any non-persistent, non-transactional fields or to un-register the in-memory persistent object with other instances in the JVM.

void jdoPreDelete()

This method is invoked before an instance is deleted. An instance is deleted using the deletePersistent() method on PersistenceManager.

Access to all fields is permitted within this method. Any exceptions thrown during this method are propagated to the application immediately, and the persistent object won't be deleted.

This method is useful to CascadeDelete other referenced instances.

void jdoPreStore()

This method is invoked before the fields of an instance are sent to the datastore. This typically happens during commit.

Access to all fields is permitted within this method. Any exceptions thrown during this method are propagated to the application immediately, and the persistent object won't be stored in the datastore.

This method is useful to update persistent fields based on the values of non-persistent fields or implement simple constraint checks on field values.

5.3.6.2 Usage guidelines

The following code snippet taken from Author.java shows how jdoPreDelete() can be used to delete an Author's Address and remove the Author from each of its referenced books:

```
public class Author implements InstanceCallbacks {

 private String name;
 private Address address;
 private Set books;

 // Additional methods not shown

 public void jdoPostLoad() {
 }

 public void jdoPreClear() {
 }

 public void jdoPreDelete() {

   JDOHelper.getPersistenceManager(this).
     deletePersistent(address);
   if (books != null) {

     Iterator iter = books.iterator();

     while (iter.hasNext()) {

       Book book = (Book) iter.next();

       book.removeAuthor(this);
     }
   }
 }

 public void jdoPreStore() {
 }
}
```

5.4 Exception Classes

JDO defines a set of exception classes that a JDO runtime can throw. The exceptions are all runtime exceptions because they can occur whenever an in-memory persistent object is accessed, not just when a JDO method is invoked. The classes are divided into fatal and non-fatal exceptions. Refer to Figure 5-3 for the JDO exception class hierarchy.

A JDO exception behaves the same as a normal Java runtime exception; however, in addition to an error string, a JDO exception may also have a reference to the persistent object to which the exception relates. For operations that process more than one persistent object (a commit, for example), a JDO exception contains an array of nested exceptions, one for each persistent object that encountered a problem.

5.4.1 JDOException

This is the base class for all JDO exceptions and defines the methods common to all exception classes. In addition, the toString() and printStackTrace() methods have also been overridden to include details on any nested exceptions.

5.4.1.1 Method summary

An application can get a reference to the persistent object that the exception relates to and an array of nested exceptions where multiple problems were encountered.

Object getFailedObject()

This method returns the persistent object that the exception relates to or returns null if there is no specific persistent object.

Throwable[] getNestedExceptions()

This method returns an array of nested exceptions where multiple problems were encountered or returns null if there are no nested exceptions.

These methods were added in the JDO 1.0.1 maintenance release.

5.4.2 JDOFatalException

This is the base class for all fatal exceptions. An operation that resulted in a fatal exception cannot be retried. Typically, the only recourse is to restart a transaction or even get a new PersistenceManager instance.

5.4.3 JDOFatalUserException

This exception indicates a fatal problem with the application. It typically means that the application has called a method at an inappropriate time. This exception might be thrown in these cases:

- A method, other than isClosed(), is called on PersistenceManager after it has been closed or instances associated with a closed PersistenceManager are accessed (Query, Extent, Transaction, and so on).
- The getPersistenceManagerFactory() method on JDOHelper cannot find the specified PersistenceManagerFactory class or unsupported options have been specified.

5.4.4 JDOFatalInternalException

This exception indicates a fatal internal problem with the JDO implementation. There is little recourse other than to report the problem to the vendor for rectification.

5.4.5 JDOFatalDataStoreException

This exception indicates a fatal problem with the datastore. Any active transaction is rolled back. This exception might be thrown in these cases:

- The current transaction has been automatically rolled back by the datastore, perhaps due to a timeout.
- The connection to the datastore has been lost, perhaps due to a network problem.

5.4.6 JDOOptimisticVerificationException

This exception indicates the completion of an optimistic transaction failed due to a conflicting update in the datastore. Any active transaction is rolled back. The only case where this exception is thrown is as a result of calling the commit() method on a Transaction instance whose Optimistic property is true.

5.4.7 JDOCanRetryException

This is the base class for non-fatal exceptions. An operation that resulted in a non-fatal exception can be retried.

5.4.8 JDOUnsupportedOptionException

This exception indicates a non-fatal application problem. The application has tried to use an optional feature that the JDO implementation does not support. This exception might be thrown in these cases:

- Invoking setOptimistic() on Transaction when optimistic transactions aren't supported by the JDO implementation.
- Trying to use collection classes not supported by the JDO implementation.

5.4.9 JDOUserException

This exception indicates a non-fatal application problem. The application can continue and potentially rectify the problem and retry the operation. This exception might be thrown in these cases:

- An instance of a non-persistence-capable class is passed to a method that expects an instance of a persistence-capable class.
- The getExtent() method is used on PersistenceManager when the class does not have a managed Extent.

5.4.10 JDODataStoreException

This exception indicates a non-fatal problem with the datastore. The application can continue and potentially rectify the problem and retry the operation.

5.4.11 JDOObjectNotFoundException

This exception indicates a non-fatal application problem. Essentially, a persistent object could not be found in the datastore. This exception might be thrown in these cases:

- Dereferencing a reference to a persistent object that has previously been deleted, perhaps by a different application.
- The getObjectById() method on PersistenceManager is unable to find a persistent object with the specified object identity.

The exception contains the persistent object that could not be found, which is returned by the getFailedObject() method. The only valid operation on the in-memory instance is to the get its JDO object identity, because anything else simply results in another exception.

5.5 Additional APIs

JDO defines two additional classes that are usable by an application:

- JDOHelper
- I18NHelper

5.5.1 JDOHelper Class

This class defines a number of static methods that a persistence-capable class or JDO application can use. Each method simply delegates to other JDO classes, like PersistenceManager and PersistenceManagerFactory, as well as classes defined in the javax.jdo.spi package like PersistenceCapable.

5.5.1.1 Method summary

This class defines methods that can be used as a bootstrap mechanism for creating a PersistenceManagerFactory instance, as well as convenience methods for interacting with persistent objects.

static PersistenceManager getPersistenceManager (Object pc)

This method returns the PersistenceManager instance associated with the specified object. It returns null if the specified instance is transient or is not an instance of a persistence-capable class.

static PersistenceManagerFactory
getPersistenceManagerFactory (Properties props)
static PersistenceManagerFactory
getPersistenceManagerFactory (Properties props, ClassLoader cl)

These methods create a PersistenceManagerFactory instance based on the specified properties. See the section on PersistenceManagerFactory for more details on each property.

If no class loader is specified, then the calling thread's context ClassLoader is used to find the PersistenceManagerFactory class.

Table 5-9 provides a list of the standard properties that can be used.

The only required property is javax.jdo.PersistenceManagerFactoryClass. All others are optional.

In addition to these standard properties, a JDO implementation may also support its own additional properties. Any properties not recognized by a JDO implementation are ignored.

Table 5-9. JDOHelper Properties

Property Name	Value
javax.jdo.PersistenceManagerFactoryClass	Class name of JDO implementation's PersistenceManagerFactory class
javax.jdo.option.Optimistic	true or false
javax.jdo.option.RetainValues	true or false
javax.jdo.option.RestoreValues	true or false
javax.jdo.option.IgnoreCache	true or false
javax.jdo.option.NontransactionalRead	true or false
javax.jdo.option.NontransactionalWrite	true or false
javax.jdo.option.Multithreaded	true or false
javax.jdo.option.ConnectionDriverName	Undefined[*]
javax.jdo.option.ConnectionUserName	Undefined[*]
javax.jdo.option.ConnectionPassword	Undefined[*]

javax.jdo.option.ConnectionURL	Undefined[*]
javax.jdo.option.ConnectionFactoryName	Undefined[*]
javax.jdo.option.ConnectionFactory2Name	Undefined[*]

[*] The value of this property is specific to the JDO implementation being used.

- static Object getObjectId (Object pc)
  static Object getTransactionalObjectId (Object pc)

  These methods provide shortcuts for the getObjectId() and getTransactionalObjectId() methods defined on PersistenceManager.

- static boolean isDirty (Object pc)
  static boolean isTransactional (Object pc)
  static boolean isPersistent (Object pc)
  static boolean isNew (Object pc)
  static boolean isDeleted (Object pc)

  These methods allow an application to interrogate the lifecycle state of a persistent object and determine if it is dirty, transactional, persistent, new, or deleted.

  See Chapter 4 for more details on the object lifecycle.

- static void makeDirty (Object pc, String fieldName)

  This method allows an application to explicitly mark the specified field of the given persistent object as dirty. The fieldName is the name of the field, optionally fully qualified with the package and class name. If the specified object is null, transient, or not an instance of a persistence-capable class or the specified field name is not a managed field, then this method does nothing.

5.5.2 I18NHelper class

This class provides localized strings for the classes in the javax.jdo and javax.jdo.spi packages (primarily the exception classes). It finds a Java ResourceBundle for the current locale using java.jdo.Bundle as the baseclass. If no ResourceBundle is found for the current locale, then the Bundle.properties file located in the javax.jdo package is used. If an application needed to provide localized strings, it could provide its own ResourceBundle.

Although this class is in the javax.jdo package, it is not supposed to be used by an application. As of the JDO 1.0.1 maintenance release, this class has been deprecated and in future releases will be moved to the javax.jdo.spi package.

5.6 Service Provider Interface APIs

In addition to the application visible classes contained in the javax.jdo package, a number of classes are in the javax.jdo.spi package. These classes are used by the JDO implementations themselves, and an application should not use them directly. A brief summary of the following classes is provided as background information:

- PersistenceCapable
- JDOPermission
- JDOImplHelper
- StateManager

5.6.1 PersistenceCapable

This is the interface that all persistence-capable classes implement. It defines the methods that a JDO implementation uses to manage the instances of persistence-capable classes in memory. It primarily has methods to get and set the fields of an instance and to interrogate lifecycle state. The implementation of the methods defined by PersistenceCapable delegate to a StateManager instance to actually manage the fields.

An application should use the methods defined on PersistenceManager and JDOHelper to interact with persistent objects in memory.

5.6.2 JDOPermission

JDO defines three privileged operations that a JDO implementation or application requires:

- **SetStateManager:** Allows an implementation to set the StateManager instance associated with an instance of a persistence-capable class and therefore accesses the fields of that instance.
- **getMetaData:** Allows an implementation to access the metadata for a persistence-capable class via JDOImplHelper.
- **closePersistenceManagerFactory:** Allows an application to call the close() method on PersistenceManagerFactory to close a PersistenceManagerFactory instance and any associated PersistenceManager instances. This permission was added in the JDO 1.0.1 maintenance release.

The JDOPermission class is used to determine what can perform a privileged operation. If a JDO application is run using a security manager (perhaps as a applet), then the classes that constitute the JDO implementation need to be granted the setStateManager and getMetaData privileges in a policy file.

5.6.3 JDOImplHelper

This class is a singleton that maintains a registry of metadata for each persistence-capable class. When a persistence-capable class is loaded, it registers itself with JDOImplHelper, and the JDO implementation can then use the metadata directly and avoid using reflection.

5.6.4 StateManager

The StateManager interface defines how a JDO implementation actually interacts with instances of persistence-capable classes and manages the fields. The PersistenceCapable and StateManager interfaces are closely related.

5.7 Summary

After reading this chapter, you should have a thorough understanding of the concepts underlying JDO, all the JDO classes and interfaces, and how to use them.

Chapter 6 will now provide a more detailed understanding of performing queries using JDOQL.

Chapter 6. Finding Your Data

"If you can find a path with no obstacles, it probably doesn't lead anywhere."

—Frank A. Clark

The capability to transparently persist object instances directly is a powerful feature; however, it would be incomplete without a mechanism to query and retrieve the information stored in those persistent instances. The JDO specification include a query facility, pieces of which have already been introduced and used in previous chapters. The query facility consists of two parts—the API that can be used in application code to programmatically locate the information and a query language called JDO Query Language (JDOQL) that defines the grammar associated with structuring the queries. This concept is analogous to how the traditional relational database persistence works in Java. There is an API called JDBC that specifies the programmatic contract and the query language (SQL) that defines the grammar. In this chapter, we look at the query API that the JDO provides as a part of the javax.jdo package, and we also look at JDOQL, its syntax, and how it can be used for specifying meaningful queries.

6.1 Finding an Object by Identity

In Chapter 1, we introduced the concept of persistence by reachability and how an entire object graph of dependent objects is persisted when an object is saved. The simplest way to find an object is through its identifier. In Chapters 3 and 5, we covered the concepts surrounding object identity and how the getObjectID() can be used. You may recall that the JDOHelper class exposes a getObjectID(Object pc) method that returns the JDO identity associated with the parameter.

The PersistenceManager interface exposes a corresponding getObjectById() method that can be used in the application code to retrieve the instance through its JDO identity.

This example code taken from the example ObjectIdentityExample.java in Chapter 5 demonstrates how the identity of a persistent object can be used to retrieve it:

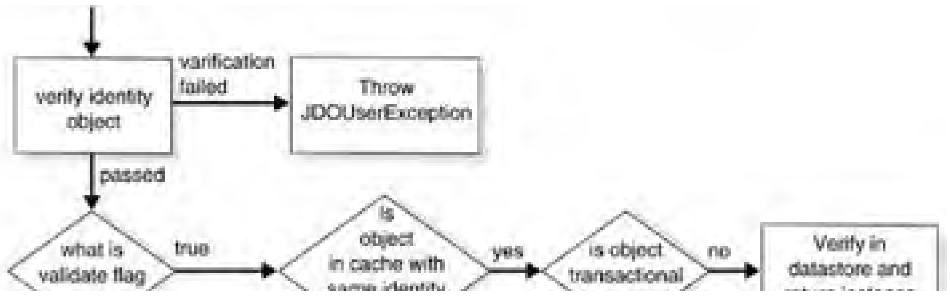
```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
tx.begin();

Author author =
  new Author(
    "Sameer Tyagi",
    new Address(
      "1 Main Street", "Boston", "MA", "01822"));
// make the Author and all its fields persistent
pm.makePersistent(author);
tx.commit();
// obtain the object id
Object oid = pm.getObjectId(author);
pm.close();
// conceptually the oid is now available and can be reused

PersistenceManager pm2 = pmf.getPersistenceManager();
Transaction tx2 = pm2.currentTransaction();
tx2.begin();
Author author2 = (Author) pm.getObjectById(oid, true);
// Should print out the name "Sameer Tyagi" below
System.out.println("Author is: " + author2.getName());
tx2.commit();
pm2.close();
```

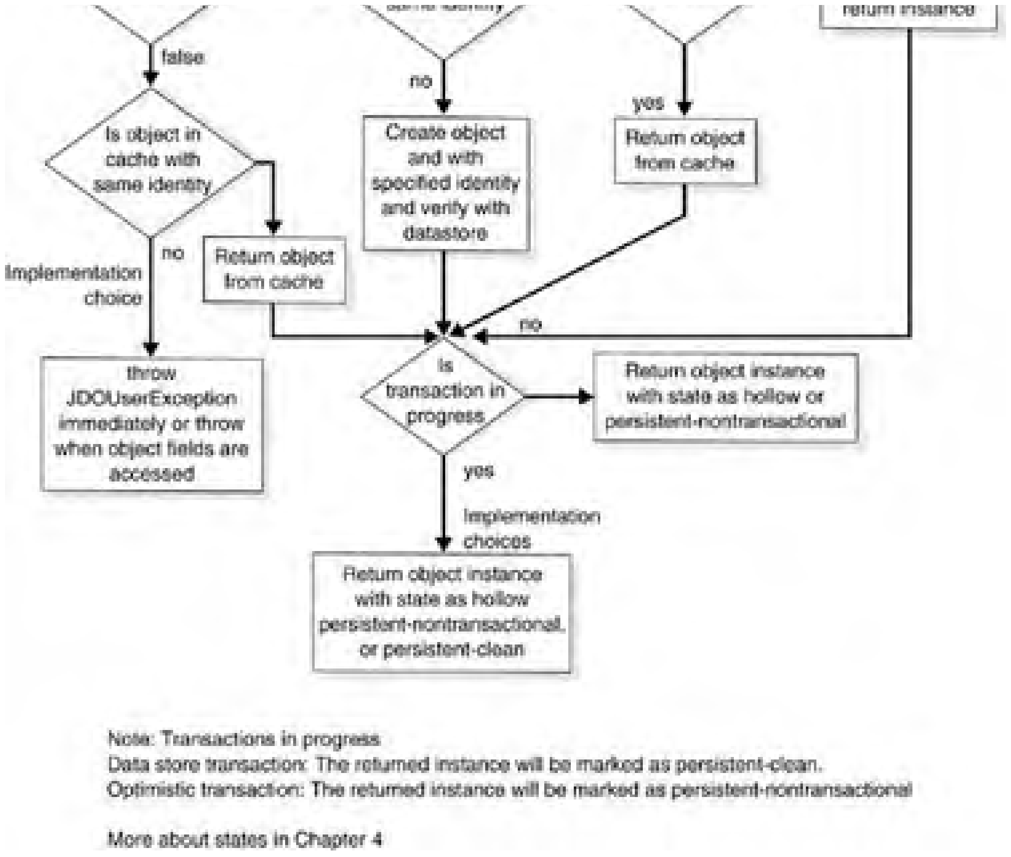
Object ID must be serializable

The object specifying the identity and returned by getObjectId() is required to be serializable so that applications can store it.

Figure 6-1 shows the logical flow that the JDO runtime goes through when the getObjectById() method is invoked. The JDO implementation tries to locate the object in the PersistenceManager's cache. If it is found, the object is verified with the underlying datastore and then returned. If the object cannot be found in the cache with the corresponding identity, the JDO implementation creates an instance, assigns it the specified identity, and returns it. Note that the JDO implementation verifies state with the underlying datastore only if the object instance in cache is non-transactional. Transactional instances do not necessarily need their state to be verified because the state is synchronized with the datastore.

Figure 6-1. Flowchart for getObjectById().




false
Is object in cache with same identity
no
Implementation choice
Return object from cache
throw JDOUserException immediately or throw when object fields are accessed
no
Create object and with specified identity and verify with datastore
yes
Return object from cache
return instance
no
Is transaction in progress
Return object instance with state as hollow or persistent-nontransactional
yes
Implementation choices
Return object instance with state as hollow persistent-nontransactional, or persistent-clean
Note: Transactions in progress
Data store transaction: The returned instance will be marked as persistent-clean.
Optimistic transaction: The returned instance will be marked as persistent-nontransactional
More about states in Chapter 4

The semantics associated with retrieving an object instance in JDO by using the JDO identity and the getObjectById method are in many ways analogous to the getHomeHandle() method in the EJB specifications. EJB can serialize and store the home handle and use it to re-obtain a reference to the remote home object at a later time.

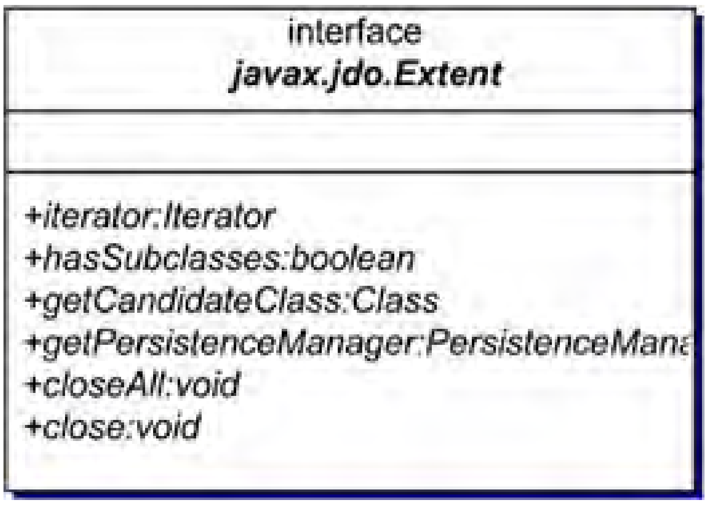
Object ID State

getObjectById returns a hollow instance, which can be deleted afterward by some other client of the datastore. When the application tries to use the object, i.e., when it is resolved, a JDOUserException is thrown.

6.2 Finding a Set of Objects Using an Extent

Earlier in Chapters 3 and 5, we discussed the javax.jdo.Extent interface. The Extent interface offers a convenient programmatic way to retrieve a collection of similar objects or an object hierarchy from the underlying datastore. Recall that an Extent like the one shown in Figure 6-2 represents a collection of all instances in the datastore of a particular persistence-capable class. An Extent can consist of just instances of the single class or can also include instances of subclasses. By default, it is possible to get an Extent for any persistence-capable class.

Figure 6-2. The javax.jdo.Extent interface.



An application obtains an Extent for a class from a PersistenceManager and then uses the iterator() method to go through all instances using a java.util.Iterator.

The following code snippet shown again from ReadByExtentExample.java in Chapter 3 demonstrates the use of an Extent in finding instances of a particular class:

```
tx.begin();
Extent extent = pm.getExtent(Author.class, true);
Iterator results = extent.iterator();
while (results.hasNext()) {
Author author = (Author) results.next();
String name = author.getName();
System.out.println("Author's name is '" + name + "'.");
}
extent.closeAll();
tx.commit();
```

Note that when used alone in the manner shown above, the Iterator can potentially include lots of objects because it returns all the instances of Author and the subclasses.

The Extent interface offers a convenient mechanism to locate persistent instances of a particular persistence-capable class in the datastore; however, most applications would need a more specific mechanism to locate instances that match specific criteria.

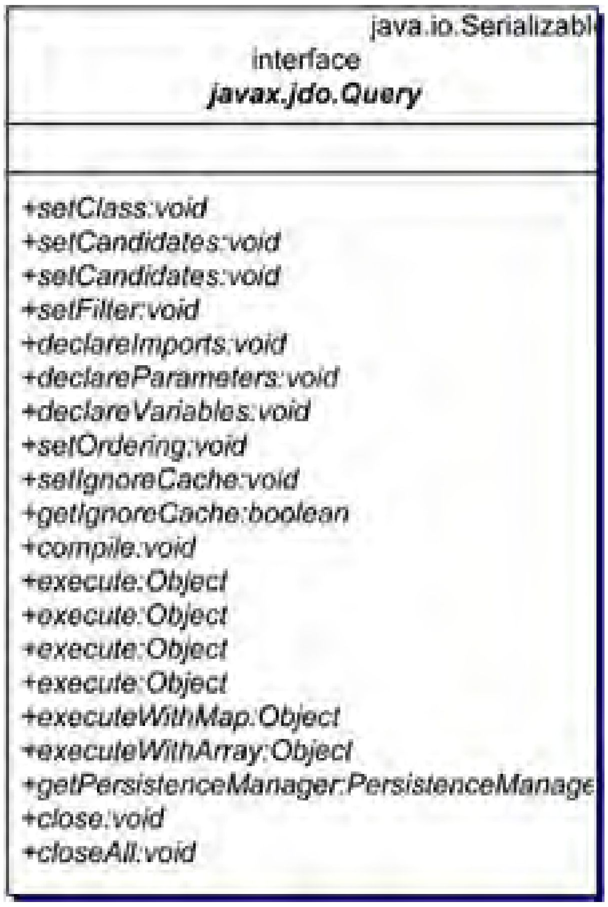
6.3 Finding Objects with the Query Facility

Obtaining specific objects by using their JDO Identity or obtaining a collection of objects through the use of an Extent is simple and convenient. However, both these techniques have drawbacks. It is not always possible or desirable to do the following:

- Store the JDO identity of an object.
- Iterate through potentially large collections of object graphs to locate a few objects that might be useful to the application. For example, if the previous snippet were applied to Stock objects that had a symbol attribute, the application would need to iterate through all the Stocks and compare the symbol until it found the one it was interested in!

The API portion of the query facility in JDO consists primarily of the Query interface shown in Figure 6-3. This was also introduced in Chapters 3 and 5.

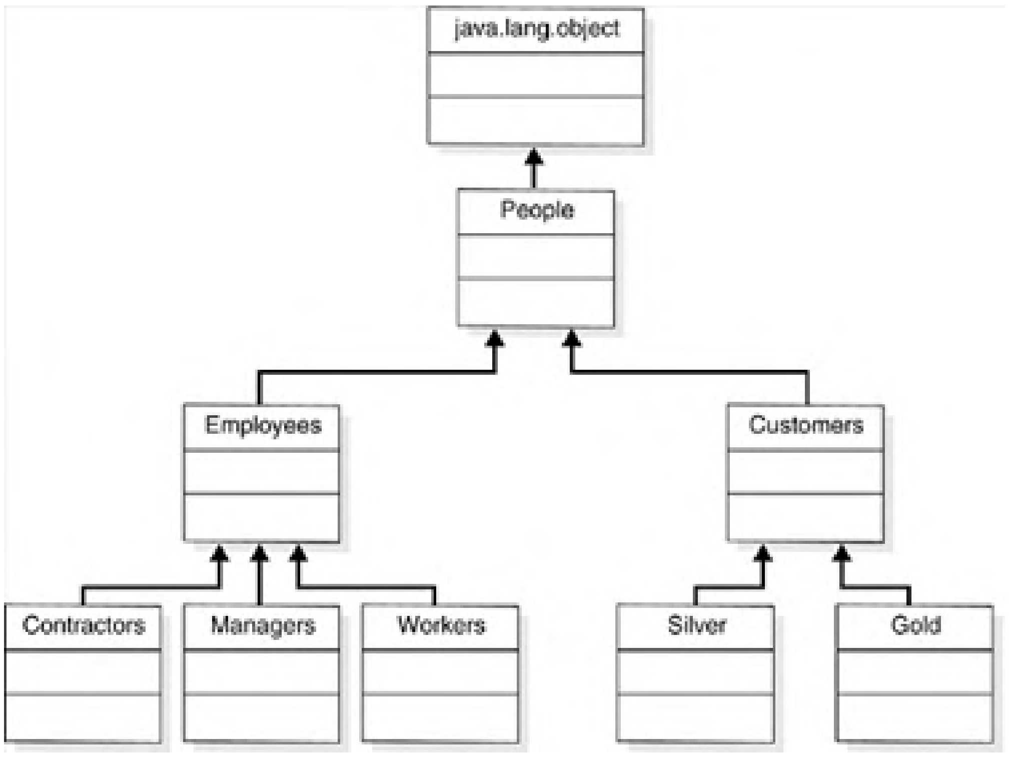
Figure 6-3. The javax.jdo.Query interface.



The Query interface abstracts the concept of interrogating the underlying data-store and retrieving *objects* that might match specific *criteria.* Three terms are important in this regard:

1. **The candidate class:** This corresponds to the highest level in the class inheritance hierarchy to which the results must belong. This can be conceptually thought of as the branch where the JDO implementation will *prune* the class inheritance tree. For example, in Figure 6-4, if the application were only interested in finding specific Employees, then the candidate class would be Employee rather than People. If the application were interested in finding gold customers, then the class would be Gold, rather than Customers or People.

Figure 6-4. Deciding on the candidate class.



2. **The candidate collection:** The candidate collection is a Collection or Extent of objects from a corresponding candidate class that are present in the data-store. An application can pass a candidate collection that it constructs to the query facility and have that collection narrowed or filtered as a result of the query execution.

3. **The query filter:** This is an expression that evaluates to some Boolean condition and is used to narrow the candidate collection returned as a result of the query execution.

Let's rework the previous example to incorporate the Query facility. Note that the results produced by the code shown below are identical to the earlier example. It returns all persistent Author objects because no Extent or *filter* has been specified in the query.

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
tx.begin();
    Query query = pm.newQuery(Author.class);
    Collection results = (Collection) query.execute();
    if (results.isEmpty())
        System.out.println("No results found");
    else {
      Iterator iter = results.iterator();
      while (iter.hasNext()) {
        Author author = (Author) iter.next();
        String name = author.getName();
        System.out.println("Author's name is " + name);
        }
      }
    query.closeAll();
    tx.commit();
```

When the Query is executed via the execute() method, the results are contained in an unmodifiable or immutable Collection. The actual method signature is declared to return an Object to allow for future or vendor extensions, but the application code must explicitly cast the result of the execute() method to a Collection. Elements of the collection can be accessed through the Iterator, but any attempt to change the collection causes an UnsupportedOperationException to be thrown.

The Collection object containing as the query results is required only to support the iterator() method because the implementation of the class is up to the vendor. For example, a vendor using a relational database may not be able to support the size() method because the database rows may be available only the first time they are accessed. For this reason, the size() method may return Integer.MAX_VALUE.

The results of a query may hold onto resources linked to the underlying datastore (such as connections). Therefore, all the query instances must be closed explicitly when the results are no longer needed. The individual result can be closed using the close(Object queryResult) method, or all open results associated with the Query can be closed at once with the closeAll() method. The idea of closing Query results is conceptually analogous to the way the close() method works for the java.sql.ResultSet object in JDBC.

6.3.1 Queries with an Extent

Although the Extent by itself may not be very meaningful, when used alongside the Query, it can be used to refine the results. The Query can be refined to return only instances of a class and its subclasses by specifying an Extent when the Query is constructed using the factory method. Code to accomplish this is shown below:

```
tx.begin();
Extent extent = pm.getExtent(Author.class, true);
Query query= pm.newQuery(extent);
Collection results=(Collection)query.execute();
Iterator iter = results.iterator();
while (iter.hasNext()) {
Author author = (Author) iter.next();
String name = author.getName();
System.out.println("Author's name is '" + name + "'.");
}
query.closeAll();
tx.commit();
```

Again, the results produced by the above code are identical to those produced in Section 6.2 because the Extent specifies the Author class.

The preceding discussion was meant to introduce you to the API itself. Before we look at queries and the API in greater detail, let's look at the second part of the query facility—JDOQL, the query language. We return then to the API and look at examples in greater detail by incorporating the query language in them.

6.4 JDOQL

The filter is essentially a string passed to the query facility that is used by the query facility to refine the results returned. The syntax for the filter is specified using JDOQL or the JDO query language. This begs the question: Why use a new syntax? Why not use Structured Query Language (SQL) that is standard to all relational databases? One of the goals of JDO is to isolate the developer from the specifics of the underlying datastore. This means that the query language must be vendor- and implementation-agnostic. The vendor is free to choose SQL, Object Query Language (OQL), or whatever other query language the underlying data-store supports. To quote the specifications, the primary design goals for the query language were:

- **Query language neutrality:** The underlying query language might be a relational query language such as SQL, an object database query language such as OQL, or a specialized API to a hierarchical database or mainframe EIS system.
- **Optimization to specific query language:** The Query interface is capable of optimizations; therefore, the interface has enough user-specified information to allow for the JDO implementation to exploit data source specific query features.
- **Accommodation of multi-tier architectures:** Queries may be executed entirely in memory, or may be delegated to a back end query engine. The JDO Query interface provides for both types of query execution strategies.
- **Large result set support:** Queries might return massive numbers of JDO instances that match the query. The JDO Query architecture provides for processing the results within the resource constraints of the execution environment.
- **Compiled query support:** Parsing queries may be resource-intensive, and in many applications can be done during application development or deployment, prior to execution time. The query interface allows for compiling queries and binding run-time parameters to the bound queries for execution.

The other goal was to keep the query facility simple and developer-friendly. JDOQL achieves this by using the standard Java operators and syntax with which developers are already familiar. This feature has been acclaimed and criticized equally. On the one hand, JDOQL is easy to construct and program, but on the other hand, most developers are already familiar with SQL, and database operators are capable of producing complex, highly optimized SQL queries. For many architects, embedding queries in a middle-tier transparent persistence API such as JDO is an unwelcome step. However, this has more to do with the overall use of JDO in an enterprise than the query facility. As per its design goal, JDO needs to isolate the application from the underlying datastore. Using SQL would mean tying the implementation to some sort of relational implementation. However, it should be noted that JDOQL queries can return only JDO instances. There are no projections or simple type/value queries, or even COUNT queries, as in SQL.

Portability

Using a query language other than JDOQL may make the application non-portable across vendor implementations.

JDOQL is one of many languages that an implementation can support and all implementations are required to support the specification-defined behavior of JDOQL. This keeps the application code portable. In reality, vendor implementations that use relational databases as the underlying store allow the use of SQL directly, as we see in Section 6.6.4.

Query language string

A query can be constructed for other vendor-supported languages using the newQuery (String language, Object query) method in the PersistenceManager. The value of the string to specify JDOQL is javax.jdo.query.JDOQL. For other languages, it is a vendor-specific string.

"How is JDOQL specified?" you may ask. The syntax for specifying syntax, i.e., the meta-syntax is done using a standard notion like Backus Normal Form (BNF). BNF is a widely used and widely accepted notation for specifying the syntax of programming languages such as Java and C++ and query languages such as SQL, JDOQL, and so on. Fortunately, the only people who need to know BNF are the specification authors! Developers need only know how JDQL works. Interested readers can refer to Appendix C for the BNF notation of JDOQL.

6.4.1 Specifying filters

The filter string defined in JDQL must evaluate to a Boolean condition. The string itself can be composed of multiple expressions that are separated using the standard Java syntax for AND, OR and NOT operators. The logical operators are explained further in Table 6-1. Each expression can use the standard Java operators used for comparison. These are explained further in Table 6-2.

The Boolean expression specified through the filter can be used to:

- Specify the criteria for the query.
- Determine whether an instance in the candidate collection is a member of the result set that is specified by the query.

For example, the following code segment from SimpleQueryWithFilter.java shows how a query can be constructed to find all Author objects where the author's name is "Snoop Smith":

```
tx.begin();
  // find all author objects where the name is Snoop Smith
String filter = "name==\"Snoop Smith\" ";
Query query = pm.newQuery(Author.class, filter);
Collection results = (Collection) query.execute();
if (results.isEmpty())
     System.out.println("No results found");
else {
    Iterator iter = results.iterator();
    while (iter.hasNext()) {
    Author author = (Author) iter.next();
    String name = author.getName();
    System.out.println("Author's name is " + name);
    System.out.println("Author's address is " +
                         author.getAddress());
    System.out.println("Author's books are " );
    Iterator books=author.getBooks().iterator();
    while(books.hasNext())
        System.out.println("Book:" +books.next()) ;
         }
       }
query.closeAll();
tx.commit();
```

The filter and candidate collection can be specified when the query is constructed through one of the newQuery() methods or dynamically using the setFilter() and setCandidates() methods respectively.

The setCandidates() method binds the candidate collection to the query instance. If the collection is modified after this invocation, it is up to the vendor implementation to allow the modified elements to participate in the query or throw a NoSuchElementException during execution of the Query for those elements.

The elements in the candidate Collection must be persistent instances associated with the same PersistenceManager as the Query instance. Vendors can optionally choose to support transient instances in the collection, but relying on that feature would make the code non-portable to other vendor implementations. If there are multiple PersistenceManagers and any element in the collection is associated with a persistence manager other than the one under which the query is constructed, a JDOUserException is thrown when the query is executed.

The code segment below from SimpleQueryWithFilterAndCandidates.java reworks the previous example to use a candidate collection and filter:

```
  tx.begin();
      Author a1= new Author("Author Smith");
      Author a2= new Author("Author Jones");
      Author a3= new Author("Author Knowles");
      Collection candidates = new ArrayList();
      candidates.add(a1);
      candidates.add(a2);
      candidates.add(a3);
      pm.makePersistentAll(candidates);

// find all author objects where the name is Author Smith
      String filter = "name==\"Author Smith\" ";
      Query query = pm.newQuery(Author.class, filter);
      query.setCandidates(candidates);
```

```
     Collection results = (Collection) query.execute();
```

The preceding example makes the candidate collection persistent. This is important for the code above to work because the candidate collection must represent objects in the datastore. If the vendor doesn't support transient instances in the candidate collection, then the code segment below throws a JDOUserException:

```
     Author a1= new Author("Author Smith");
     Author a2= new Author("Author Jones");
     Author a3= new Author("Author Knowles");
     Collection candidates = new ArrayList();
     candidates.add(a1);
     candidates.add(a2);
     candidates.add(a3);
// find all author objects where the name is Author Smith
     String filter = "name==\"Author Smith\" ";
     Query query = pm.newQuery(Author.class, filter);
     query.setCandidates(candidates);
```

Transient instance support

Support for transient instances in the collection is optional, and vendors may or may not implement it.

Table 6-1. Logical Operators

Operator	Description	Supported data types	Example	SQL Equivalent
!	Logical compliment	Boolean, boolean	String filter = "!(employed==true)"	NOT
&&	Conditional AND	Boolean, boolean	String filter = "(name==\"John\") && (age==30)"	AND
||	Conditional OR	Boolean, boolean	String filter = "(name==\"John\")|| (name==\"Johnathan\")"	OR

The "&" and "|" operators

JDOQL filters cannot change any fields. They are non-mutating. Therefore, the Java operators "&" and "|" are should not be used.

Table 6-2. Comparison Operators

Operator	Description	Supported data types	Example	SQL Equivalent
==	Equals	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Boolean, boolean, Date, any class instance or array	Name equals John:= String filter = "(name==\"John\")"	==
!=	Not equal	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger Boolean, boolean, Date, any class instance or array	Name does not equal John:= String filter =" (name!=\"John\")"	<>
>	Greater than	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String	Age is greater than 30:= String filter = "age > 30"	>
<	Less than	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String	Age is less than 30:= String filter = "age<30"	<
>=	Greater than or equal	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger, Date, String	Age is greater than or equal to 30:= String filter = "age >= 30"	>=
<=	Less than or equal	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal,	Age is less than or equal to 30:= String filter =	<=

		BigInteger, Date, String	"age<=30"	
+	Binary addition	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger	The persons age is 2 yrs below the minimum age String filter ="age+2>=minage"	+
-	Binary subtraction	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger	The persons age is 2 years more than the minimum age String filter ="age-2>minage"	-
~	Negation (operator is followed by a primitive)	byte, short, int, long, char, Byte, Short, Integer, Long, Character	Assume field x is negative, change to positive value String filter="age==(~ x)"	(-expr) - 1
*	Multiplication	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger	Profile is 2% of price String filter=".02 * price"	*
/	Division	byte, short, int, long, char, Byte, Short, Integer, Long, Character float, double, Float, Double, BigDecimal, BigInteger	Profile is 2% of price String filter="(2/100) * price"	/

Table 6-3. Special Operators

Operator	Description	Example
()	Used to cast object types.	Author names start with 'S": String filter = "((String)name).startsWith(\"S\") "; Note that the cast in the above filter is not explicitly required but is only meant to show usage.
. (period)	Used to reference fields as in the Java language	Author names start with 'S" : String filter="name.startsWith(\"S\")" Author living in zip code 02929: String filter = "address.zipcode==\"02929\"";
this	Used to access the hidden fields. this refers to an object of the candidate class.	String filer= "this.name == name"; More details on this in Section 6.5.5

6.4.1.1 Differences between Java and JDOQL operators

Although the JDOQL operators follow the syntax of the Java operators, there are some subtle and explicit differences between the operators as used in the query filters and as used in the Java language.

6.4.1.1.1 Equality and ordering

In Java, the equality operator "==" cannot be used between a primitive and its corresponding wrapper. JDOQL allows this operator to be used between primitives and their corresponding wrapper classes. For example, (10==objectten) is a valid use, whereas objectten is an instance of Integer.

The same concept extends to the ordering operators (>, <, >=, <=). In Java, you cannot use these operators between a primitive and a wrapper. For example, the syntax (10>objectten) would be illegal in Java if the variable objectten referred to an instance of Integer and not a primitive. Like the equality operator, JDOQL allows for this type of usage between primitives and wrappers.

The use of the equality and ordering operators extends to the String and Date objects. For example, if d1 and d2 represent two Date instances, (d1>d2) would be illegal in Java, but valid as a part of the filter in JDOQL.

6.4.1.1.2 Assignment operators

The query filter is not allowed to change the value of a persistent field. This means that the assignment operators =, /=, +=, and so on, and pre/post increment/decrement are not allowed. For example, the following string, although legal in Java, is illegal as a filter in JDO: String filter="author.name=\"Snoop Smith\"";

6.4.1.1.3 Methods

For performance reasons, a JDO implementation is not required to instantiate an object in order to evaluate the query filter. Additionally, the query may execute on a server where the application is not available. This means that the method invocation as a part of the filter is not supported and is considered illegal. For example, the following filter is illegal: String filter="author.getName()=="Snoop Smith"; the only exceptions when it comes to methods are the Collection interface and the String class. Two methods that can be invoked on a Collection instance in a filter:

- **isEmpty():** This returns true if the collection contains no elements or if the reference is null.
- **contains(Object obj):** This returns true if the collection contains the object passed as the parameter. Note that the reference passed must be a persistence-capable object. The JDO implementation uses the JDO identity, as opposed to the equals() method, to check whether the object exists in the collection.

Two methods can be invoked on a String instance in a filter:

- **startsWith(String str):** If the persistence field (a String instance) starts with the specified parameter, then the method returns true. For example, the name is a persietent field in the Author class. A valid filter could be String filter="name.startsWith(\"S\")".
- **endsWith(String str):** This is similar to the startsWith() method, except that it checks whether the field ends with the specified parameter.

6.4.1.1.4 Navigation

The term *navigation* in JDOQL is used to describe the ability to explicitly traverse and resolve references to fields in a persistence-capable class. As shown in Table 6-3, this is done using the period (".") operator.

JDOQL requires that navigation through a field with a null value must result in the subexpression being evaluated to false. In the above example, if the zipcode field in the candidate class were null, it would result in the filter evaluating to false. For example, the filter to determine whether an author resides in the zip 02929 would look like this:

```
String filter = "address.zipcode==\"02929\"".
```

In this case, the filter *navigates* to the field zipcode via the address reference.

JDOQL also supports navigation through a Collection types using variables and the contains() method. This is covered further in Section 6.5.3.

6.5 Queries, Filters, and Optional parameters

Let us now look at the slightly more advanced features of the query facility. These features allow the use of parameters in a query string, the use of declarative imports and variables, respectively.

6.5.1 Declaring parameters

Rather than hard coding the values for certain objects in the filter string, the filter can contain placeholders that can be replaced with different values. This allows a single query to be executed multiple times with new values for the placeholder. For repeated queries, vendors can optimize their underlying implementation specific to the datastore for optimizing performance. Conceptually, the usage of parameters in JDOQL is similar to the way parameters are passed in JDBC prepared statements. For example, in JDBC, the hard coded SQL would look like this:

[View full width]

```
Statement stmt= con.createStatement("UPDATE AUTHORS SET BOOK= 'Core JDO" WHERE
    NAME=='Snoop Smith'");
```

This can be replaced with a PreparedStatment as follows:

```
PreparedStatement stmt = con.prepareStatement(
    "UPDATE AUTHORS SET BOOK =? WHERE NAME ==?");
```

The prepared statement can be executed repeatedly with new values replacing the "?" through the stmt.setXXX() methods. The database precompiles the SQL and executes faster.

JDOQL works in a similar manner. Rather than containing "?" in the filter string, actual variable names may be used. The values for the variables are passed using the declareParameters() method in the Query interface, which takes a String as an argument. The syntax of the string is the same as that used in standard Java method signatures; i.e., it is a type declaration followed by the variable name. Let's look at an example of how this works. The code segment below from FilterExamples.java shows how to find all books with a given publisher name and a publication date earlier than the current date. The same query is then re-executed with new values for the publisher's name.

```
  // find all books with a certain publisher name and date
// First create the filter
  String filter = "publisher.name== pubname &&
                    PublicationDate < today";
  Query query = pm.newQuery(Book.class, filter);
  Date today= new Date();
  String pubname="Pet publishin Co";
// Declare any import statements (see next paragraph for
// explaination)
  query.declareImports("import java.util.Date;")  ;

// Declare the parameters using the standard Java method
// signature syntax
  query.declareParameters("Publisher pubname,Date today");
// execute the query with the given parameters
  results = (Collection) query.execute(pubname,today);
  if (results.isEmpty())
      System.out.println("No results found");
  else {
     Iterator iter = results.iterator();
      while (iter.hasNext()) {
           Book book = (Book) iter.next();
            System.out.println("Book found "+book);
            }
       }

  // repeat query execution with new value  for publisher
   pubname="Books123.com";
```

```
results = (Collection) query.execute(pubname,today);
if (results.isEmpty())
     System.out.println("No results found");
else {
    Iterator iter = results.iterator();
    while (iter.hasNext()) {
        Book book = (Book) iter.next();
        System.out.println("Book found "+book);
         }
  }
```

The Query interface has four convenient methods for execution that can use parameters. They take one, two or three objects as parameters. Usually, one of these overloaded methods will suffice. However, if the query takes a number of parameters, then the more generic form, which takes an array of objects, can be used. These methods are described in Table 6-4.

Table 6-4. Query Execution Methods

Query execute method	Description
execute(Object p1)	Execute the query and return the filtered Collection.
execute(Object p1, Object p2)	Execute the query and return the filtered Collection.
execute(Object p1,Object p2, Object p3)	Execute the query and return the filtered Collection.
executeWithArray(Object[] parameters)	Execute the query and return the filtered Collection

6.5.2 Declaring imports

Sometimes, the class or interface definitions of parameters passed to the query may reside in a package other than the candidate class. These definitions can be imported on demand by the JDO implementation using the declareImports() method. The method takes a semicolon-separated list of import statements in the standard Java syntax. In the preceding example, the publicationDate was used as a parameter, but the Date class resides in the java.util package. The declareImports() method was used to pass the import declaration to tell the JDO runtime where to locate the parameter definition. The standard java.lang.* classes, the candidate class, and classes in the same package as the candidate class do not need to be explicitly imported. All imported classes must, however, be available in the class path.

6.5.3 Declaring variables

JDOQL uses the concept of variables to test whether some items in a multi-valued collection match specific criteria. If the filter involves iteration through a Collection, a variable can be used along with the Collection.contains() method to check whether a specific element in the collection matches the filter criteria. For example, the Author class contains a field named books that is of type java.util.Set. To check whether the Set (a subinterface of java.util.Collection) contains any Book object with the title "Learn Java Today", the code would look like this:

```
String filter = "books.contains(myfavouritebook) && " +
         "(myfavoritebook.title==\"Learn Java Today\")";
Query query = pm.newQuery(Author.class, filter);
query.declareVariables("Book myfavoritebook");
Collection results = (Collection) query.execute();
```

The contains() method is passed a variable named myfavoritebook. This variable is then declared using the declareVariables() method, following the standard Java syntax for variable declarations (i.e., variableType variableName).

In the preceding example, the Collection is iterated and each element is set to the variable myfavouritebook. The condition (myfavouritebook.title== \"Learn Java Today\") is then evaluated for each value and the result formed. Of course, this serial access is only conceptual, and the JDO implementation may use a strategy specific to the indexing features of the underlying data-store. For example, in the preceeding code segment, a vendor implementation may generate SQL for a relational datastore that looks like this:

[View full width]

```
Select Distnct T0.Jdoid T0.Address, T0.Name From Author T0, Author_Books T1, Book T2 Where
    (T2.Title = ? And T0.Jdoid = T1.Jdoid And T1.Books = T2.Jdoid)
```

Tip for Declaring Variables

When declaring variables, the names used should be unique and cannot conflict with other parameter names. Also, variables must be specified in a single call. For example, if multiple variables need to be passed in a filter, then all of them should be declared in a single declareVariables() call.

6.5.4 Ordering results

Sometimes, it is desirable to have the results of a query ordered in a particular manner. For example, Author objects are logically ordered by name in ascending alphabetical order. The ordering criteria for a query can be specified as a comma-separated list of field names followed by the keywords "ascending" or "descending". The results returned are ordered using the left-to-right evaluation of the ordering criteria. The declarations are passed to the Query interface using the setOrdering() method. For example, the code segment below from FilterExamples.java shows how a query can be executed to return all Books sorted in ascending order by title. If two books have the same title, then they are sorted by the next criteria, descending order of the publicationDate:

```
query = pm.newQuery(Book.class);
query.setOrdering("title ascending,
                  publicationDate decending");
results = (Collection) query.execute();
```

6.5.5 Namespaces in a query

A namespace refers to a logical separation of type names, fields, and variables. A Query has two namespaces:

- The namespace of types and classes
- The namespace of fields, variables, and parameters

The setClass() method adds the name of the candidate class, and the declareImports() method adds the names of imported classes or interfaces to the type namespace.

When the setClass() method is used to set the candidate class explicitly, the names of fields of the candidate class are also added to the namespace of the fields and variables for the Query. The same happens when the declare-Parameters() or declareVariables() methods are invoked. In other words, the names of the parameters and variables are added to the same namespace. A parameter name or variable name *overrules and hides* the field name set by the candidate class, if they are the same. The hidden field may be accessed using the operator shown in Table 6-3. The keyword in a filter always refers to an object of the candidate class. For example, consider the code snippet from FilterExamples.java. The filter looks for an Author with a name field that equals "Steven Jones" and declares a variable of type Book also called name:

```
String filter = "books.contains(name) && " +
          "(this.name==\"Steven Jones\";
Query query = pm.newQuery(Author.class, filter);
query.declareVariables("Book name");
Collection results = (Collection) query.execute();
```

In this case, the persistent field name of the candidate class (Author) will be *hidden* by the declared variable name and the this keyword must be used to refer to it explicitly.

Invoking setClass()

The setClass() method needs to be invoked only if the candidate class was not passed to the newQuery() factory method.

6.6 More on the Query Interface

Let us now look at the Query interface a bit more in detail. In this section we look at the different factory methods available for query construction, how queries are cached, how JDO for compiled queries, using templates for query construction and the use of other query languages.

6.6.1 Query construction

There are a number of factory methods in the PersistenceManager interface that can be used to construct a Query instance. We have seen some of these methods in preceding code snippets. Table 6-5 lists all the factory methods. The methods dealing with query compilation and creating queries from other queries are covered further in Sections 6.6.4 and 6.6.5, respectively.

Table 6-5. Factory Methods for Creating Queries

Factory method	Description
Query newQuery()	Creates a new Query with no elements.
Query newQuery(Class class)	Creates a new Query specifying the Class of the candidate instances.
Query newQuery(Class cls, Collection candaidates)	Creates a new Query with the candidate Extent; the class is taken from the Extent.
Query newQuery(Class class, Collection candaidates, String filter)	Creates a new Query with the Class of the candidate instances, candidate Collection and filter.
Query newQuery(Class class, String filter)	Creates a new Query with the Class of the candidate instances and filter.
Query newQuery(Extent extent)	Creates a new Query with the Class of the candidate instances and candidate Extent.
Query newQuery(Extent extent, String filter)	Creates a new Query with the candidate Extent and filter; the class is taken from the Extent.
Query newQuery(Object compiled)	Creates a new Query using elements from another Query.
Query newQuery(String language, Object query)	Creates a new Query using the specified language.

6.6.2 Queries and the cache

In Chapter 5, the concept of the cache maintained by the PersistenceManager was introduced. The Query interface exposes two methods that specify the run-time behavior and query execution characteristics: the setIgnoreCache(boolean ignoreCache) method and the getIgnoreCache() set and get the javax.jdo.option.IgnoreCache property. When this property is set to true, the query execution ignores the instances in the PersistenceManager's cache. These cached instances may have changed from their persistent representations in the datastore (see Chapter 4 and Appendix A for state changes). Although synchronizing state for the query execution may result in more accurate results (this may also depend on the isolation settings in the datastore), it also negatively affects the performance characteristics of the system. When the cache is ignored, the query executes entirely in the datastore. This feature, when used in conjunction with optimistic transactions or read only type transactions (see Chapter 10), can dramatically improve performance.

6.6.3 Compiled queries

JDO queries can be compiled with the compile() method shown in Figure 6-3. This does not mean that the query is compiled into some native format; it is merely a hint to the implementation to optimize an execution plan for the query. It is up to the vendor to support this optional feature and choose the compilation strategy depending on the datastore. If the vendor chooses a compile-time optimization technique, a potential disadvantage could be that the execution strategy for the query may actually become sub-optimal at runtime due to updates to the datastore at runtime.

Compiled Queries

The notion of compiled queries is analogous to the concept of compiled queries in object-oriented databases that use OQL. OQL is an object-oriented SQL-like query language and is a part of the ODMG-93 standard. It can be used in two different ways: either as an embedded function in a programming language or as an ad-hoc query language. Some vendors compile OQL declarations into native constructs such as C++ to perform query optimization at preprocessing time rather than at runtime. Therefore, all type errors are caught at compile time. (Note that precompiled queries are also available in relational databases.)

6.6.4 Template queries

The vendor-provided implementation class for the Query interface is required to be serializable. This means that the application code can serialize and save the Query instances indefinitely and recreate them subsequently. Because a Query may hold onto active resources in the underlying datastore, the deserialized Query object cannot actually be re-executed. However, the details associated with the Query (e.g., the candidate class, filter string, imports, parameters, variables, and ordering) are retained. This deserialized query can be used as a sort of template to recreate another query instance from the PersistenceManager with the original details that were saved. For example, this code snippet from TemplateExample.java shows how the Query can be serialized and reused:

```
// create and serialize the query
if(args[0].equalsIgnoreCase("serialize")) {
  String filter = "books.contains(myfavoritebook) && " +
         "(myfavouritebook.title==\"Learn Java Today\")";
  Query query = pm.newQuery(Author.class, filter);
  query.declareVariables("Book myfavoritebook");
 // close resources just to be safe
   query.closeAll();
  serializeQuery(query);
      }
// other code here

/**Method to serialize and save the query object*/
public static void serializeQuery(Query query){
 try{
   ObjectOutputStream os = new ObjectOutputStream (new
                         FileOutputStream("query.ser"));
   os.writeObject(query);
   os.close();
   } catch(Exception e){
     System.out.println("An exception occurred during query
                       serialization :"+e);
      }
    }
```

The serialized query can then be read later and used as a template and executed. It is required that the Query instance being reused must come from the same JDO vendor; in other words, you should not expect to serialize and reuse Query instances across vendor implementations. The corresponding snippet below from the same example shows how the serialized query can be read and used as a template:

```
if (args[0].equalsIgnoreCase("deserialize")) {
// deserialize and execute
Transaction tx = pm.currentTransaction();
tx.begin();
Object deserQuery = deserializeQuery();
Query recreatedQuery = pm.newQuery(deserQuery);
Collection results = (Collection) recreatedQuery.execute();
if (results.isEmpty())
    System.out.println("No results found");
else {
    Iterator iter = results.iterator();
    while (iter.hasNext()) {
    Author author = (Author) iter.next();
    String name = author.getName();
    System.out.println("Author's name is " + name);
    System.out.println("Author's address is " +
                         author.getAddress());
    System.out.println("Author's books are ");
    Iterator books = author.getBooks().iterator();
    while (books.hasNext())
        System.out.println("Book:" + books.next());
        }
      }
  recreatedQuery.closeAll();
  tx.commit();
        }
// other code here

/** Method to Deserialize the query */
public static Query deserializeQuery() {
 try {
    ObjectInputStream os = new ObjectInputStream(new
                        FileInputStream("query.ser"));
    Query q = (Query) os.readObject();
    os.close();
    return q;
     } catch (Exception e) {
       System.out.println("An exception occurred during
                           query deserialization :" + e);
        return null;
        }
```

6.6.5 Choosing a different query language

The JDOQL discussed in this chapter is required to be implemented by all vendors. However, vendors can choose to support other query languages natively in their implementations. For example, a vendor using a relational database as the underlying datastore may choose to allow SQL as the query language; another vendor using an object database may support OQL directly. A query can be constructed in a format other than JDOQL using the overloaded newQuery() method in the persistence manager.

```
public Query newQuery(String language, Object query);
```

If the vendor supports this feature, the application may potentially leverage features that are specific to the datastore; however, using anything other than JDOQL may render the application code non-portable to other vendor implementations. The code example below shows how an SQL query can be used in a particular vendor's implementation:

```
String SQL= "Select * from Authors";
Query query = pm.newQuery(VendorInterface.SQL, SQL);
Collection results = (Collection)query.execute();
```

Note that the value of the String parameter language is not specified in the JDO specifications for any language; it depends on the vendor.

6.7 Summary

In this chapter, we looked at how persistent instances can be queried and retrieved from the underlying datastore using the query facilities specified in JDO. We looked at two parts of the facility, the API and JDOQL, which is a Java-like syntax for specifying the query filters. We looked at detailed examples that showed different features of the API and JDOQL that are available to developers.

In Chapter 7, we look at architecture scenarios and how JDO fits into different environments.

Chapter 7. Architecture Scenarios

"I call architecture frozen music."

—Johann Wolfgang von Goethe

This chapter outlines the different types of application architectures that can be used with JDO, beyond the standalone, single-threaded Java applications that have been the focus of the book up to this point. Using JDO with these different architectures is not very different from using JDO to build a standalone application. The concepts and APIs are all the same. The only difference is how PersistenceManagerFactory, PersistenceManager and Transaction instances are used.

This chapter covers the following topics:

- When to use JDO, and when to use JDBC
- The different types of datastores that can be used with JDO
- Using JDO with multi-tier applications
- Differences when using JDO in managed and non-managed environments
- Multi-threaded programming with JDO

7.1 JDO versus JDBC

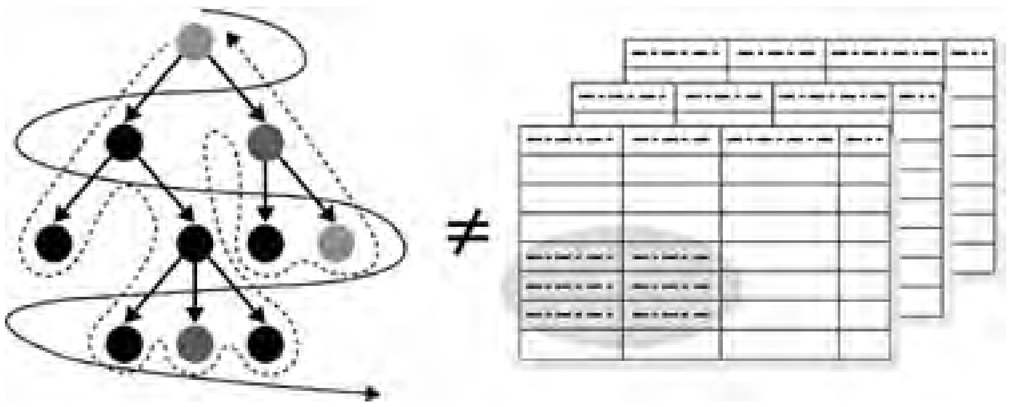
Put in its simplest form, JDO can be used to build any Java application that needs to access data stored in a datastore, whatever the architecture of the Java application. To this extent, JDO and JDBC can be viewed as competing ways of accessing persistent data from Java. However, the difference between JDO and JDBC lies in JDO's object perspective.

For data-centric applications, whose main purpose is to store and retrieve record-oriented data, JDBC is likely the best tool for the job. Record-oriented data can easily be mapped directly to a relational model, and an application can leverage the power of SQL for ad hoc queries and data access. The focus is on the data.

For object-centric applications, whose main purpose is to process a complex graph or hierarchy of interrelated data, JDO is likely the best tool for the job. An object model can be used to represent the complex graph or hierarchy of interrelated data that the application can navigate through to complete its processing. The focus is on the object model.

Simply put, JDBC is for applications that need to think of data as relations (tables, rows, and columns), and JDO is for applications that need to think of data as interrelated objects. Figure 7-1 shows the difference in structure between object-centric and data-centric applications.

Figure 7-1. An object-oriented application versus a relational-oriented application.



7.2 RDBMS, ODBMS, and Flatfiles

Unlike JDBC, which is specific to databases that support SQL, JDO is datastore-neutral. The same JDO application can be used with a relational database, an object database, a flatfile, or even an in-memory database. A JDO implementation for the underlying datastore being used is all that's required.

Although the application itself remains the same across databases, how a database is set up and used with a JDO implementation varies. The JDO specification does not define how a persistence-capable class is actually mapped to its representation in the underlying datastore; this is left to the JDO implementation. Typically, before a JDO application can be run, the following steps must be performed:

- Specify the mapping between the persistence-capable classes and the data-store schema. JDO implementations typically use the extension element in the JDO metadata to specify the mapping between the persistence-capable class and its representation in the datastore, and provide tools that can be used to generate this mapping.
- Create a database, and define the schema. Again, JDO implementations typically provide tools that can be used to define the schema in the database.

How each step is performed varies between JDO implementations; an implementation's documentation should be consulted for further details.

7.2.1 Using JDO with a relational database

There are three approaches to using JDO with a relational database:

- Define the persistence-capable classes, and generate the database schema.
- Define the database schema, and generate the persistence-capable classes.
- Define the persistence-capable classes and database schema independently, and define a mapping between the two.

The first approach is suitable for applications in which there is no existing database and the application is free to define its own schema. The database schema can be generated from the persistence-capable classes. This has the advantage that the application can leverage the power of object-orientation when defining its persistence-capable classes. A JDO implementation's tools are used to automatically define the schema in the database for the persistence-capable classes and to create the mapping specification between the persistence-capable classes and the schema.

The second approach is suitable for applications in which a database already exists and the application must use the existing schema. The persistence-capable classes can be generated from the schema. This has the advantage that a JDO application can easily access existing data in a database; however, the generated classes won't be particularly object-oriented in nature. A JDO implementation's tools are used to generate the Java classes from the database schema for the application and to create the mapping specification between the persistence-capable classes and the schema.

The third approach is suitable for applications in which both the persistence-capable classes and the database schema already exist. A mapping is specified between the persistence-capable classes and the existing database schema. This has the advantage that a JDO application can leverage the power of object-orientation when defining its persistence-capable classes but can still access existing data in a database. A JDO implementation's tools are used to define the mapping between the persistence-capable classes and the database schema. This approach is the most complex of the three because defining a mapping between two arbitrary models is a difficult task. Typically, for this approach to work, the persistence-capable classes and database schema must have a similar structure.

A JDO implementation that supports relational databases may not support all the approaches outlined. Developers must choose an implementation that supports the approach required by their applications.

7.2.2 Using JDO with an object database

Object databases differ from relational databases in that they do not use the relational model (tables, columns, and rows) to store data; instead, they use the object model (classes, fields, and instances). Direct support for the object model means that a JDO persistence-capable class can be mapped directly to its object representation in the database.

The approach to using an object database with JDO is similar to the first approach outlined for using a relational database. The persistence-capable classes are defined, and the database schema is generated. A JDO implementation's tools are used to define the schema in the database for the persistence-capable classes. There is typically no mapping to specify because the persistence-capable classes map one-for-one to the classes defined in the database.

7.2.3 Object versus relational databases

As with most things, no single technology is best for all needs. Although a heated debate has taken place over the years about the pros and cons of using object versus relational databases, much of it has failed to recognize that the different technologies satisfy different needs. This doesn't necessarily make one technology better than the other; it simply means that a developer must understand which is most appropriate for the job at hand.

The primary strength of an object database is its ability to manage arbitrarily complex object models, including the following:

- Inheritance
- Multi-valued fields (arrays of values)
- One-to-one, one-to-many, and bi-directional relationships
- Relationships that include semantics—sets (uniqueness), lists (ordering), and maps (associative lookup)—where the relationships may be complex objects in themselves, perhaps containing hashed values for efficient in-memory lookup and retrieval

Complex object models are typically hard to map to an efficient representation in a relational database, and because an object database is able to directly represent a persistence-capable class without any mapping, a JDO application gets the following benefits:

- **Development:** Additional metadata is not required to specify how a persistence-capable class should be represented in the database. This may appear only a minor benefit, but as the number of persistence-capable classes increases along with the complexity of those classes (use of inheritance and relationships), the additional metadata can add up.
- **Performance:** Because the persistence-capable classes are directly represented in the database, there is less overhead to retrieve a persistent object (if using a relational database, it is likely that the representation of a persistence-capable class would require several tables). As the complexity of the classes increases, typically the performance delta between relational and object databases increases.

Although a discussion of the relative merits of object and relational databases is beyond the scope of this book, a simple perspective is that an object database can be very effective as an application-specific database, whereas a relational database is a much better choice as an enterprise-wide database. Essentially, if an application persists data for its own needs only, an object database can be a good candidate. If an application needs to persist data to be accessed enterprise-wide, then a relational database is a good choice.

7.3 J2EE, RMI, and CORBA

As well as standalone Java applications, JDO can be used to build J2EE, RMI, or CORBA applications also. In particular, JDO has been specifically designed to work in conjunction with J2EE. Essentially, wherever a Java application can directly use JDBC today, JDO likewise could be used.

In many ways, JDO offers a programming model and approach to object persistence similar to Container Manager Persistence (CMP) from the Enterprise JavaBeans (EJB) specification. However, unlike CMP, JDO keeps persistence orthogonal to the component architecture being used. This means that the power of object persistence can be taken advantage of directly within a Java application or a Servlet without incurring the overhead of the J2EE component architecture. Or it can be used by any other non-J2EE distributed platforms like RMI and CORBA.

That is not to say that JDO has no role to play within EJB. In fact, JDO combined with SessionBeans makes for a powerful architecture. SessionBeans provide the remote, component-level interface, leveraging the container services like resource pooling, connection, and transaction management, while JDO provides portable, lightweight object persistence.

Although the EJB 2.0 specification provides an enhanced specification for CMP that aims to address some of the performance issues of the previous version, from a development standpoint, CMP is still an order of magnitude more complex than JDO to use.

With CMP, a developer needs to do the following:

- Provide a local and/or remote Home interface that defines the methods used to locate instances of the EntityBean class.
- Provide a local and/or remote EntityBean interface that defines the public methods of the EntityBean class.
- Provide a local and/or remote abstract class that implements the methods as declared in the associated EntityBean's interface. The class doesn't actually declare the fields of the EntityBean; instead, it declares JavaBeans-style getters and setters for each field. Fields and relationships to other EntityBeans must be declared in the deployment descriptor for the EntityBean.
- Provide a deployment descriptor that declares all of the following: the EntityBean's local and/or remote Home and EntityBean interfaces, the abstract class that implements the EntityBean's methods, and the fields and relationships of the EntityBean.
- Deploy the EntityBean, at which point a concrete implementation of the EntityBean's abstract class is generated by the container along with the necessary code to store and retrieve the fields or the EntityBean to/from the underlying datastore.

When these things have been done, the EntityBean can be run and tested; however, any issue that requires a code change requires a redeployment of the Entity Bean. Also, the EntityBean itself can be run only within a J2EE container, although a local interface at least allows an application within the same JVM to bypass the overhead of remote invocation.

With JDO, a developer needs to do the following:

- Provide a normal Java class. Fields are declared as normal; relationships are implemented using Java references and standard collection classes. Inheritance can be used.
- Provide an XML file containing the metadata for the persistence-capable class. Because most of the metadata is derived, often all that is needed is the class name.
- Compile the class using a standard Java compiler, and use a JDO-compliant byte-code enhancement tool.

At this point, the class can be used by any Java application, a Servlet or EJB SessionBean, RMI, or CORBA application.

7.4 Managed and Non-Managed Environments

The terms **managed** and **non-managed** have been used at various points in the book to differentiate between traditional two-tier Java applications and multi-tier Java applications.

A two-tier Java application explicitly connects to the underlying datastore (which constitutes the second tier) and begins and ends its own local transactions. A two-tier application typically starts, does something at the bequest of a user and ends.

A multi-tier Java application requests connections from a connection manager and may delegate transaction control to an external coordinator. This coordinator begins and ends transactions implicitly, potentially coordinating multiple different data sources into a single distributed transaction. A multi-tier application typically runs continuously, processing requests that its receives from other applications. Servlets, EJBs, and applications that use RMI or CORBA are all examples of multi-tier applications.

7.4.1 Connection management

To interact with the underlying datastore, a JDO application must get a connection. A two-tier application typically requires a connection for the duration of the application, during which time it begins and ends many transactions. A multi-tier application typically requires a connection for the duration of a single transaction only. For this reason, multi-tier applications rely on resource pooling to avoid connections being continuously opened and closed. Instead, once connections are opened, they are managed in a resource pool. Each time a multi-tier application requires a connection to process a request, one is allocated from the pool; when finished, the connection is given back to the pool. Although typically used by multi-tier applications, some two-tier applications can also benefit from resource pooling.

7.4.1.1 Two-tier applications

A two-tier Java application constructs a PersistenceManagerFactory, gets one or more PersistenceManager instances, and closes them when finished. The act of getting a PersistenceManager instance and closing it may or may not correlate with physically opening and closing connections to the underlying datastore.

Depending on the JDO implementation being used and how it is configured, the PersistenceManagerFactory may pool the underlying datastore connections or PersistenceManager instances. This means that when the application closes a PersistenceManager instance, it doesn't physically close anything, but rather gives back the resources to the PersistenceManagerFactory to be reused. All this is transparent to the application, which simply gets a PersistenceManager instance when it needs one and closes it when it is finished.

A JDO implementation may also allow use of external connection factories. If configured to use an external connection factory, the PersistenceManagerFactory relies on the external factory to pool datastore connections. It requests a connection when needed and returns it to the factory when finished. See the ConnectionFactory and ConnectionFactoryName properties on PersistenceManagerFactory for more details on using external connection factories.

Connection pooling is more typically useful for multi-tier applications, but some multi-threaded, two-tier applications may benefit from pooling also, particularly those that can have many threads where a thread doesn't necessarily need a PersistenceManager instance all the time. By pooling, an application can reduce the resources it needs at any given time.

7.4.1.2 Multi-tier applications

A multi-tier application looks up a PersistenceManagerFactory (typically using JNDI), from which it gets one or more PersistenceManager instances.

If used in a J2EE runtime environment, a PersistenceManagerFactory instance is no longer responsible for connection management; instead, it relies on the J2EE container to pool connections and manage the quality of service associated with use of those connections.

Most JDO implementations use the J2EE Connector Architecture (JCA) as the means by which the PersistenceManagerFactory instance interacts with the J2EE container. The JCA resource adapter provided with a JDO implementation is configured and deployed into a J2EE runtime environment. The PersistenceManagerFactory instance retrieved via JNDI then delegates connection management to the J2EE container.

The JDO specification does not mandate that a JDO implementation use JCA; an implementation may choose a different way of integrating with a J2EE container. Although how the JDO implementation is configured and deployed may differ, the application itself remains the same. It looks up a PersistenceManagerFactory instance from which it gets PersistenceManager instances.

Non-J2EE multi-tier applications must ensure that the JDO implementation being used implements its own connection pooling.

7.4.2 Transaction management

Transactions provide the context within which persistent objects are stored in the datastore and retrieved again. A two-tier application typically begins and ends transactions explicitly, while a multi-tier application may delegate transaction control to an external coordinator, in which case the coordinator begins and ends transactions implicitly.

7.4.2.1 Two-tier applications

A two-tier application gets a Transaction instance from a PersistenceManager, and uses it to explicitly begin and end transactions.

7.4.2.2 Multi-tier applications

A multi-tier application can delegate transaction control to an external coordinator, typically the Java Transaction Service (JTS). When using an external coordinator, transactions are automatically controlled.

If used in a J2EE runtime environment, a PersistenceManagerFactory instance ensures that a PersistenceManager instance is associated with the externally coordinated transaction (if there is one) when it is retrieved. This is relevant primarily for EJBs. If a method has been declared as transactional, then when the Bean method gets its PersistenceManager instance from the PersistenceManagerFactory, the PersistenceManagerFactory returns a PersistenceManager instance associated with the external transaction. The external coordinator then implicitly begins and ends the transaction.

A multi-tier application doesn't have to delegate transaction management to an external coordinator. In this case, the application explicitly begins and ends transactions via the Transaction interface. This is relevant for Servlets or EJBs with non-transactional methods.

For transaction management, most JDO implementations use JCA as the means by which the PersistenceManagerFactory interacts with the J2EE container.

However, this is not mandated, and an implementation may use a different mechanism.

Non-J2EE applications that require external coordination of transactions must ensure that the JDO implementation being used supports the Java Transaction APIs (JTA) so it can be integrated with JTS.

7.5 Multi-Threaded Applications

Many of the examples in the book so far have been of the simplest type of JDO application. The application runs, gets a PersistenceManager instance from a PersistenceManagerFactory, begins a transaction, does something, and ends the transaction. This is a single-threaded, single PersistenceManager application.

JDO supports much more sophisticated applications than this. An application can essentially get as many PersistenceManager instances at a time as available resources allow. Each PersistenceManager instance represents a connection to the datastore, a transaction context, and a cache of persistent objects. From the application perspective, each PersistenceManager instance is independent of the other, and from the datastore perspective, each PersistenceManager instance is essentially a separate user. An application can be multi-threaded and concurrently access the datastore using different PersistenceManager instances.

7.5.1 Multi-threaded programming

When using multiple threads, a developer must give thought to how threads interact with a PersistenceManager instance. A PersistenceManager instance is essentially a shared resource; therefore, care needs to be taken when multiple threads concurrently access it.

In fact, not only is the PersistenceManager instance itself a shared resource, but because the PersistenceManager acts as a factory for Query, Extent, and Transaction instances, any of those instances are likewise shared resources. Also, any persistent objects associated with the PersistenceManager instance are shared resources.

If multiple threads are going to concurrently access a PersistenceManager instance or any associated instances (Query, Extent, and so on), then the Multitheaded property of the PersistenceManager instance needs to be true. This ensures that the JDO implementation synchronizes internally and that concurrent access doesn't cause internal data structures to be corrupted. However, this synchronization may add additional overhead.

Beyond this, the application must ensure that multiple threads interact with the PersistenceManager instance correctly. Although multiple threads can access persistent objects at the same time, it is still the application's responsibility to synchronize this access. This is no different from normal multi-threaded programming in which multiple threads are concurrently accessing the same Java objects. Also, care must be taken when using various methods on PersistenceManager that affect the lifecycle states of persistent objects. For example, an application must ensure that persistent objects being evicted or refreshed by one thread are not being used by another.

When completing a transaction, the lifecycle states of all persistent objects associated with the PersistenceManager instance change (they all transition to being "hollow"). An application must ensure that only one thread at a time calls commit or rollback and that other threads are not accessing at this time any persistent objects associated with the PersistenceManager instance.

7.5.2 Simplified multi-threaded programming

Although an application may be multi-threaded, if there is only a single thread accessing a PersistenceManager instance or its associated instances at a given point, then as far as the JDO runtime is concerned, this is equivalent to the application being single-threaded. Because each PersistenceManager instance is independent of any other, multiple threads can concurrently access different PersistenceManager instances without conflict, and the Multithreaded PersistenceManager property does not need to be true.

7.6 Summary

This chapter provides an overview of how JDO can be used with different application architectures beyond a standalone Java application. The next chapters focus specifically on J2EE and how JDO can be used to build J2EE applications.

Part 3: J2EE

Chapter 8. JDO and the J2EE Connector Architecture

"Creativity is the power to connect the seemingly unconnected."

—William Plomer

This chapter explains how JDO can be used in conjunction the Java 2 Platform, Enterprise Edition (J2EE) via the J2EE Connector Architecture. This allows a JDO implementation to collaborate with a J2EE-compliant Application Server on security, transaction, and connection management. This chapter assumes familiarity with the J2EE concepts and development.

This chapter covers the following topics:

- A brief introduction to the J2EE Connector Architecture.
- How the J2EE Connector Architecture and JDO fit together.
- Using JDO and the J2EE Connector Architecture to build J2EE applications.
- Using JDO without the J2EE Connector Architecture to build J2EE applications.

Chapter 9 provides greater detail on actually using JDO to build J2EE applications. This chapter aims to provide an introduction by explaining how JDO and J2EE can interoperate. In this context, a J2EE application refers to either a Servlet or an Enterprise JavaBean.

8.1 J2EE Connector Architecture Overview

The J2EE Connector Architecture specification, part of the overall J2EE specification, is intended as a standard mechanism for plugging heterogeneous Enterprise Information Systems into a J2EE-compliant Application Server. It allows J2EE applications to securely interact with non-J2EE applications in a transactional manner. Provided that there is a J2EE Connector Architecture resource adapter for the Enterprise Information System (EIS), the EIS and Application Server can collaborate on security, transaction, and connection management. A resource adapter is a standard, system-level software driver for the EIS.

The J2EE Connector Architecture specification (version 1.0) defines three system-level contracts:

Connection Management Contract: This allows the J2EE platform to pool connections to the EIS and allows applications to use those connections to interact with the EIS when needed.

Transaction Management Contract: This allows the J2EE platform to control interaction with the EIS transactionally.

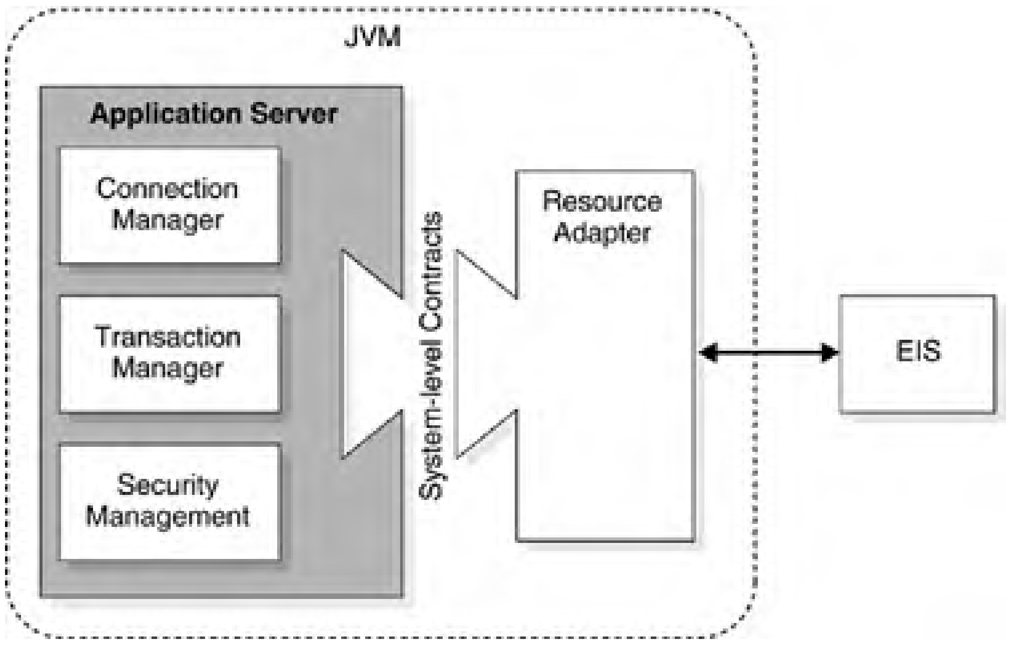
Security Contract: This allows the J2EE platform to secure access to the EIS.

These contracts allow the Application Server to manage interaction with the EIS in a standard way. However, they are system-level contracts and as such are of concern only to those developing J2EE Connector Architecture resource adapters.

The latest version of the J2EE Connector Architecture specification (version 1.5) has added additional system-level contracts; however, these are of little relevance when using JDO within the J2EE environment at this time.

Figure 8-1 depicts how a resource adapter plugs into an Application Server and, through the J2EE Connector Architecture system-level contracts, manages the interaction between the Application Server and the EIS. As can be seen, the J2EE Connector Architecture resource adapter runs within the same JVM as the Application Server.

Figure 8-1. The J2EE Connector Architecture.



The J2EE Connector Architecture specification also defines a Common Client Interface (CCI). The CCI defines standard APIs for J2EE applications to connect and interact with an EIS in a standardized, record-oriented manner.

8.2 JDO and the J2EE Connector Architecture

Although the J2EE Connector Architecture was principally designed as a standard mechanism to plug an EIS into an Application Server, it also provides a convenient mechanism for a JDO implementation to do likewise. As such, any JDO implementation designed to work within the J2EE environment likely provides a J2EE Connector Architecture resource adapter. This resource adapter means that J2EE applications can use JDO, because the JDO implementation collaborates with the Application Server on security, connection, and transaction management as defined by the J2EE Connector Architecture specification.

In the same manner that a J2EE application can use JDBC today and benefit from container-managed connections and transactions, via the J2EE Connector Architecture, an application using JDO gets the same benefits. So even though JDO is not officially part of the J2EE specification, it can still leverage the enterprise services offered by the J2EE environment.

Support for the J2EE Connector Architecture

The JDO specification does not actually require a JDO implementation to provide a J2EE Connector Architecture resource adapter. Therefore, a developer wishing to use JDO within the J2EE environment should ensure that the chosen JDO implementation does indeed support the J2EE Connector Architecture.

A JDO implementation is primarily interested in implementing the J2EE Connector Architecture system-level contracts because these allow it to collaborate with an Application Server. The J2EE Connector Architecture CCI is not of particular relevance to JDO, because JDO is itself the "client" API. The only CCI APIs of relevance to JDO are those involved with connections. The CCI defines two interfaces related to connection management: ConnectionFactory and Connection (which are covered later in this chapter).

8.3 Using JDO and the J2EE Connector Architecture

This section goes through how to use JDO to build a J2EE application. It covers these topics:

- Setup and Configuration— setting up and configuring a J2EE Connector Architecture resource adapter.
- Connection management— using the PersistenceManagerFactory, PersistenceManager interfaces, and the role of the J2EE Connector Architecture ConnectionFactory and Connection interfaces.
- Transaction management— bean-managed versus container-managed transactions and distributed transactions.
- Security management.

8.3.1 Setup and configuration

The first step in using a JDO implementation with an Application Server is to set up and configure an instance of the J2EE Connector Architecture resource adapter provided by the JDO implementation.

The resource adapter itself is contained in a resource adapter archive or RAR file. The RAR file contains, among other things, the classes used to implement the J2EE Connector Architecture system-level contracts and a deployment descriptor (called ra.xml). The deployment descriptor defines the classes that implement the J2EE Connector Architecture system-level contracts along with the configuration properties that are specified at deployment time. These configuration properties are akin to the properties used when creating a PersistenceManagerFactory instance (ConnectionURL; UserName; Password).

Different Application Servers provide different mechanisms for deploying an instance of a resource adapter. Some simply require the RAR file to be copied into a particular directory along with an XML configuration file containing, among other things, the configuration properties for the resource adapter instance. The deployment process results in a PersistenceManagerFactory instance being bound to a specified JNDI name. This name can then be used by a J2EE application to retrieve the PersistenceManagerFactory instance.

The PersistenceManagerFactory instance created during the deployment of the resource adapter automatically delegates connection pooling to the Application Server. It also ensures that the PersistenceManager instances are automatically enlisted in any managed transaction, as appropriate. In this case, the PersistenceManager instances delegate transaction management to the Application Server, and any attempt to explicitly start or end a transaction using JDO's Transaction interface results in a JDOUserException being thrown.

Container-Managed and Bean-Managed Transactions

A container-managed transaction is implicitly started and ended by an EJB container based on the deployment properties for a given bean. A bean-managed transaction is explicitly started and ended programmatically by the bean itself. This is done either using the **javax.transaction.UserTransaction** interface or via resource adapter specific APIs.

If there is no managed transaction, then JDO's Transaction interface can be used to control the transaction instead.

8.3.2 Connection management

The J2EE Connector Architecture connection management contract enables a JDO implementation to delegate connection management to an Application Server. This allows a JDO implementation to take advantage of the pooling features and quality of service guarantees offered by the Application Server. In addition, it means that administrators can use the interfaces provided by the Application Server to configure and manage all connection pools, JDO or otherwise.

8.3.2.1 Getting a PersistenceManagerFactory

After an instance of the resource adapter has been deployed, a J2EE application can use JNDI to look up the PersistenceManagerFactory instance.

This lookup would typically be done during initialization. For a Servlet, this would be done in its init() method; for an EJB, it would be in its setSessionContext() method. In both cases, an instance field defined by the class would be used to maintain a reference to the PersistenceManagerFactory instance so that the Servlet or EJB methods can use it to get PersistenceManager instances when needed.

The following code snippet shows how an EJB SessionBean would get a PersistenceManagerFactory instance. It assumes that an instance of a resource adapter was deployed using the JNDI name jdo/factory:

```
private PersistenceManagerFactory pmf;
private SessionContext ejbContext;

public void setSessionContext(SessionContext context)
  throws EJBException, RemoteException {

  ejbContext = context;

  try {

    InitialContext ctx = new InitialContext();

    pmf = (PersistenceManagerFactory)
      ctx.lookup("java:comp/env/jdo/factory");
  }

  catch (NamingException e) {

    throw new EJBException(
      "Can't find PersistenceManagerFactory");
  }
}
```

The following code snippet shows how a Servlet would do likewise:

```
private PersistenceManagerFactory pmf;

public void init() throws ServletException {

  try {

    InitialContext ctx = new InitialContext();

    pmf = (PersistenceManagerFactory)
      ctx.lookup("java:comp/env/jdo/factory");
  }
  catch (NamingException e) {

    throw new ServletException(
      "Can't find PersistenceManagerFactory", e);
  }
}
```

JDO can also be used with standalone Servlet engines that do not support the J2EE Connector Architecture. In this situation, the init() method simply creates a PersistenceManagerFactory instance as per any unmanaged JDO application.

8.3.2.2 Getting a PersistenceManager

After a PersistenceManagerFactory instance has been created, a Servlet or EJB can use it to get and close PersistenceManager instances as per normal. For an EJB, each bean method would need to get a PersistenceManager instance before it does anything; for a Servlet, the doGet() and doPost() methods would do likewise.

8.3.2.3 ConnectionFactory and Connection interfaces

Because the JDO specification doesn't require a JDO implementation to support the J2EE Connector Architecture or define how the J2EE Connector Architecture should be supported, there is the possibility of variance as to how a JDO implementation supports it.

The J2EE Connector Architecture CCI defines two interfaces related to connection management: ConnectionFactory and Connection. These are equivalent in purpose to JDO's PersistenceManagerFactory and PersistenceManager interfaces. A JDO implementation may choose to use the CCI interfaces as the means of getting PersistenceManager instances. In this case, rather than looking up a PersistenceManagerFactory instance via JNDI, a ConnectionFactory instance is looked up instead. Using the getConnection() method defined on ConnectionFactory, the J2EE application would get a Connection instance. Depending on the JDO implementation, it might be possible to cast the Connection instance to a PersistenceManager directly or else use an implementation-specific API to retrieve the PersistenceManager instance from the Connection instance.

Some JDO implementations may support the use of the CCI interfaces completely interchangeably with the JDO PersistenceManagerFactory and PersistenceManager interfaces. In this case, the implementation's PersistenceManagerFactory and PersistenceManager classes also implement the CCI ConnectionFactory and Connection interfaces. Depending on the application's perspective, it can decide to be either the J2EE Connector Architecture or JDO compliant in the way it gets its connections.

8.3.3 Transaction management

The J2EE Connector Architecture transaction management contract enables a JDO implementation to delegate transaction control to an Application Server.

With Servlets, only explicit transaction control is possible. However, when using JDO to develop an EJB, it is possible to either explicitly control transactions (bean-managed transactions) or delegate transaction control to the EJB container (container-managed transactions).

8.3.3.1 Servlets

When using JDO with Servlets, the doGet() and doPost() methods would get a PersistenceManager instance, begin a transaction, do something, end the transaction, and close the PersistenceManager instance.

The following code snippet shows how the doGet() method of a Servlet works:

```
public void doGet(
  HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {

  PersistenceManager pm = null;

  try {

    pm = pmf.getPersistenceManager();

    Transaction tx = pm.currentTransaction();

    tx.begin();

    // Do something...

    tx.commit();
  }

  catch (JDOException e) {

    throw new ServletException("JDO exception", e);
  }

  finally {

    if (pm != null && !pm.isClosed()) {

      if (pm.currentTransaction().isActive()) {

        pm.currentTransaction().rollback();
      }

      pm.close();
    }
  }
}
```

A try/finally block ensures that even in the event of an unhandled exception, the PersistenceManager instance is closed and any active transaction is rolled back.

8.3.3.2 Bean-managed transactions

When using EJB with bean-managed transactions, a bean method can either use JDO's Transaction interface or the javax.transaction.UserTransaction interface to demarcate transactions.

If using JDO's Transaction interface, the bean method would get a PersistenceManager instance, begin a transaction, do something, end the transaction, and close the PersistenceManager instance. The following code snippet shows how a bean method works in this case:

```
public void localTransactionExample() {

  PersistenceManager pm = null;

  try {

    pm = pmf.getPersistenceManager();

    javax.jdo.Transaction tx = pm.currentTransaction();

    tx.begin();

    // Do something...

    tx.commit();
  }

  catch (JDOException e) {

    throw new EJBException("JDO exception");
  }

  finally {

    if (pm != null && !pm.isClosed()) {
      if (pm.currentTransaction().isActive()) {

        pm.currentTransaction().rollback();
      }

      pm.close();
    }
  }
}
```

A try/finally block ensures that even in the event of an unhandled exception, the PersistenceManager instance is closed and any active transaction is rolled back.

If using the javax.transaction.UserTransaction interface, the bean method would begin a transaction using the begin() method on javax.transaction.UserTransaction, get a PersistenceManager instance, do something, and end the transaction using the commit() method on javax.transaction.UserTransaction. The following code snippet shows how a bean method works in this case:

```
public void userTransactionExample() {

  PersistenceManager pm = null;

  UserTransaction tx = ejbContext.getUserTransaction();

  try {

    tx.begin();

    pm = pmf.getPersistenceManager();

    // Do something...

    tx.commit();
  }

  catch (Exception e) {

    throw new EJBException("Exception encountered");
  }

  finally {

    if (pm != null && !pm.isClosed()) {
      pm.close();
    }
  }
}
```

A try/finally block ensures that even in the event of an unhandled exception, the PersistenceManager instance is closed.

When using JDO's Transaction interface to demarcate transactions, each time getPersistenceManager() is called, it returns a different PersistenceManager instance (because there is no managed transaction). If one bean method calls another and each calls getPersistenceManager(), each method gets its own PersistenceManager instance. This means each bean method operates within its own transaction.

8.3.3.3 Container-managed transactions

When using EJB with container-managed transactions, the EJB container implicitly begins and ends transactions. In this case, each bean method simply gets a PersistenceManager instance, does something interesting, and closes it. The following code snippet shows how a bean method works in this case:

```
public void containerTransactionExample() {

  PersistenceManager pm = null;

  try {

    pm = pmf.getPersistenceManager();

    // Do something...
  }

  finally {

    if (pm != null && !pm.isClosed()) {

      pm.close();
    }
  }
}
```

A try/finally block ensures that even in the event of an unhandled exception, the PersistenceManager instance is closed.

With container-managed transactions, multiple bean method invocations can be made within the same transaction (based on the transaction attribute declared for the bean methods). In this case, each time getPersistenceManager() is called, it returns the PersistenceManager instance associated with the managed transaction. This way, multiple bean methods share the same underlying PersistenceManager instance.

Because EJBs have remote interfaces, it is possible that one bean can call a method on another in a different JVM. In this situation, the bean methods cannot possibly share the same PersistenceManager instance. Instead, a PersistenceManager instance is associated with the transaction within each JVM, and the Java Transaction Service (JTS) is responsible for coordinating them all as a single, distributed transaction.

8.3.3.4 Distributed transactions

Usually, a transactional resource, like an RDBMS, manages its own transactions. When an application commits, the RDBMS itself determines whether the transaction is successful or not.

However, if an application needs to interact with several different systems but still requires transactional guarantees across all those systems, then it needs to use an external transaction manager. The external transaction manager is responsible for coordinating access to each system as a single transaction—either all systems are updated or none are updated.

Distributed transactions are of primary use when there is a requirement to coordinate access across different systems. For example, an application may need to read a message from a queue and update a database as a single operation (to ensure that the message gets written into the database). In this situation, a distributed transaction can be used to coordinate the message queue and the database.

The J2EE Connector Architecture specification defines three classifications for resource adapters, based on their level of transaction support:

NoTransaction: The resource adapter does not support transactions. Each interaction with the EIS is an atomic operation in itself.

LocalTransaction: The resource adapter supports local transactions only. Multiple interactions to the same EIS can be coordinated transactionally, but it cannot be used as part of a distributed transaction.

XATransaction: The resource adapter supports local and distributed transactions.

Some JDO implementations provide LocalTransaction resource adapters only. This means that a bean method using JDO cannot be coordinated transactionally with another bean method, unless both methods are invoked within the same JVM and both use the same PersistenceManagerFactory instance (and therefore each method gets the same PersistenceManager instance).

If an application needs to coordinate the following transactionally:

- Bean methods across multiple JVMs
- Bean methods that use different PersistenceManagerFactory instances
- Bean methods accessing other transactional systems (EIS, RDBMS)

Then the chosen JDO implementation needs to provide a XATransaction-capable resource adapter.

8.3.4 Security

Because JDO does not define its own security model and instead relies on the security provided by the underlying datastore, the J2EE Connector Architecture security contract is of less relevance to JDO than the connection and transaction contracts.

The JDO implementation's resource adapter provides an appropriate implementation of the security contract, given the underlying datastore being used. In most cases, this relies on a traditional username and password being specified when the resource adapter is deployed.

8.5 Using JDO without the J2EE Connector Architecture

As has already been outlined, JDO can be used with Servlets without needing to use a J2EE Connector Architecture resource adapter. In this case, the Servlet is the same as any other unmanaged JDO application. Likewise, if an EJB does not require container-managed transactions, then it is not necessary to use a J2EE Connector Architecture resource adapter.

Depending on needs, the Servlet or EJB can rely on the connection pooling provided by JDO implementation's PersistenceManagerFactory or instead configure an external connection factory using the ConnectionFactory or ConnectionFactoryName PersistenceManagerFactory properties. See Chapter 5 for more details on creating a PersistenceManagerFactory instance in this way.

8.6 Summary

This chapter provided an introduction to using JDO to build J2EE applications by examining how JDO-compliant and J2EE-compliant Application Servers can interoperate via the J2EE Connector Architecture.

The next chapter goes into greater detail on actually developing a J2EE application using JDO.

Chapter 9. JDO and Enterprise JavaBeans

"You can tell a lot about a fellow's character by his way of eating jelly beans."

—Ronald Reagan

This chapter attempts to give you a better understanding of Enterprise JavaBeans (EJB) in the context of JDO. The goal is not to simply show *how* to code EJB and JDO together, but to also talk about the *why* and *when,* which should allow you to ultimately decide *whether* the combination of these two powerful Java APIs makes sense.

More or less complete code examples are shown to clarify the points discussed. Brief summaries of EJB concepts are presented; however, a certain familiarity by the reader with the basic EJB architecture is assumed. This chapter is not intended to be a comprehensive tutorial on EJBs; several good ones already exist, such as the J2EE tutorial from Sun, which is available at http://java.sun.com/j2ee/tutorial.

The example code does not show home and remote interfaces or deployment descriptors. Instead, it focuses on relevant code snippets from within the bean implementation classes themselves.

9.1 Introduction

The Enterprise JavaBeans specification[1] defines EJB as "a component architecture for the development and deployment of component-based distributed business applications. Enterprise beans are components of distributed transaction-oriented enterprise applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification."

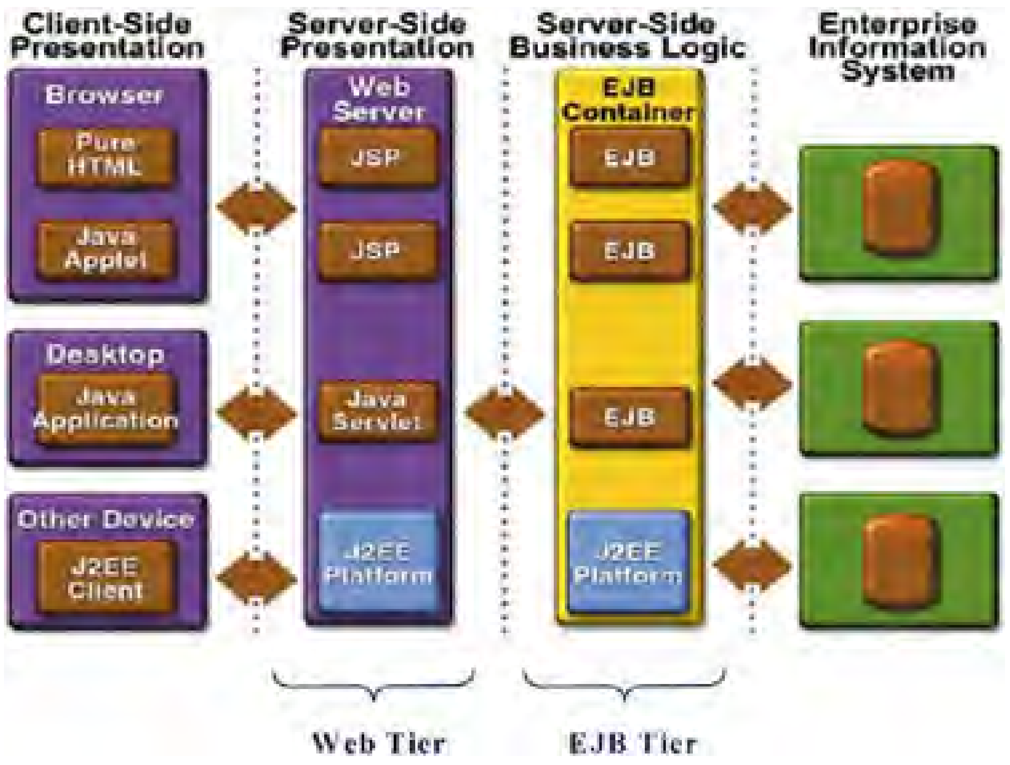
[1] http://java.sun.com/products/ejb

In practice, this translates into a number of concrete features:

- Transaction management (declarative or explicit via code, possibly distributed)
- Contracts (components, remote and local clients) and Views (including Web services)
- Simple security management (declarative or explicit)
- Instance and resource pooling
- Exception handling (standard conventions)
- Deployment (packaging, descriptors)
- Management (runtime monitoring)

Enterprise Java Beans are but one type of component in the bigger J2EE architecture picture of things. Figure 9-1 illustrates the relationship to other components, such as in the Web tier.

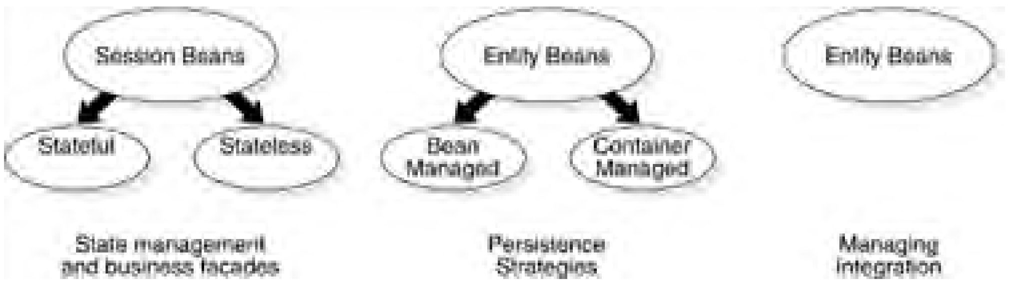
Figure 9-1. EJBs as server-side business logic in the classical J2EE Architecture.



There are currently three types of enterprise beans that model typical but fairly different and technically relatively unrelated requirements of modern enterprise software, see also Figure 9-2:

- **Session beans:** These are components that represent stateless services that are invoked synchronously from remote clients. They are possibly accessible as Web-service endpoint views. A stateful variation models components that represent a conversational session with a particular client. Session beans are accessible with remote invocation or via local interfaces.
- **Message-driven beans:** These are components that represent a stateless service and whose invocation is asynchronous, usually driven by the arrival of Java Messaging Service (JMS) messages.
- **Entity beans:** These are components that represent persistent objects. Entity beans are also accessible with remote invocation or via local interfaces. Note that entity beans are technically unrelated to JDO and predate it; more on this later.

Figure 9-2. Types of EJB components.



Of these three types, one is related to persistence and thus overlaps with JDO. The other two types of EJBs represent service-oriented interfaces that, in order to do their tasks, often rely on and thus interface with a persistence API, such as JDBC, entity beans, JDO, or others. The point to remember is that these three enterprise bean types do not depend on each other in any way, but offer different features. We look at the role of JDO with each throughout this chapter.

9.2 Session Beans and JDO

A typical session bean has the following characteristics, according to the EJB specification:

- Executes on behalf of a single client.
- Can be transaction-aware.
- Often updates shared data in an underlying database.
- Does not represent directly shared data in the database, although it may access and update such data.
- Is relatively short-lived.
- A typical EJB container provides a scalable runtime environment to execute a large number of session objects concurrently.

The EJB specification defines stateless session beans, which do no keep state between business method invocations, as well as stateful session beans, which can keep conversational state between business method invocations. In short, a stateful session bean holds a client-specific state in the business tier.

Additionally, such beans can rely on two types of transaction demarcation mechanisms:

Container-managed transactions (CMT) in which the EJB container coordinates transactions across business method calls. A bean using CMT declares in its XML deployment descriptor its transactional properties.

Bean-management transactions (BMT) in which a bean's implementation itself controls transactions via explicit code, i.e., begins and commits rollbacks transactions.

The choice of stateless versus stateful and the transaction management schema are technically unrelated and orthogonal to each other, which leads to four possible scenarios:

- Stateless session bean with container-managed transactions.
- Stateful session bean with container-managed transactions.
- Stateless session bean with bean-managed transactions.
- Stateful session bean with bean-managed transactions.

The first blend (stateless session bean with container-managed transactions) is generally the most useful and widely used, because it combines managed transactions with the stateless architecture. The stateless architecture best allows EJB containers to efficiently implement remote request load-balancing and fail-over features by reusing and sharing beans. Together, these are two of the major advantages of using EJBs over "plain vanilla" Java classes to implement business logic.

Subsequently, we first examine more details regarding transaction management, and then look at some concrete code specifically for the case of stateless session bean with container-managed transactions and stateful session bean with container-managed transactions.

9.2.1 Transaction management

Other chapters of this book (specifically Chapters 8 and 11) provide the background on the issue of managed versus unmanaged JDO environments with its different transaction mechanisms.

As a brief reminder, the transaction demarcation mechanism in a non-managed JDO environment uses the `javax.jdo.Transaction` interface returned by the `PersistenceManager.currentTransaction()` method. In a managed environment, however, a JDO application relies on the standard J2EE `javax.transaction.UserTransaction` from the Java Transaction API (JTA).

9.2.1.1 Bean-managed transactions (BMT) with JDO transaction

When building session beans using bean-managed transactions, both the JDO and the JTA transaction demarcation API can be used.

Should the choice be to use the javax.jdo.Transaction, then the code within a session bean's business method would not really differ much from standalone simple JDO code seen elsewhere in this book. So in a business method in a session bean with BMT, you would simply write:

```
PersistenceManager pm = pmf.getPersistenceManager();

// No J2EE JTA transaction can be active at this point!
pm.currentTransaction().begin();

// Do something using JDO now...

pm.currentTransaction().commit();
pm.close();
```

This architecture could make sense for simple BMT session beans that do not require the transaction context to be propagated when other methods are called from it.

If an enterprise component does require transaction propagation, however, it should either use the JTA API for explicit demarcation, or use CMT for implicit declarative demarcation. Both approaches are illustrated in the following paragraphs.

9.2.1.2 Bean-managed transactions (BMT) with JTA transaction

As is shown in Chapters 8 and 11, a JDO implementation that claims JCA-compliance (sometimes referred to as "n-tier ready" or "enterprise integration enabled" or listed as supporting "JDO managed-environment" or "application server environment" in documentation) is able to perfectly blend into the J2EE and thus the EJB environment. In this approach, a BMT-based business method would use a javax.transaction.UserTransaction, obtained via the EJB SessionContext.getUserTransaction() method, for transaction demarcation, as would any non-JDO related session bean. Only one condition must be true for this to happen: a request to the PersistenceManager from the PersistenceManagerFactory after having initiated a JTA transaction (BMT-scenario), or alternatively ensuring that code executes within an elsewhere initiated currently active JTA transaction (CMT scenario; see below). Respective code from a BMT session bean business method would look something like this:

```
sessionContext.getUserTransaction().begin();
PersistenceManager pm = pmf.getPersistenceManager();

// Do something using JDO now...

pm.close();
sessionContext.getUserTransaction().commit();
```

This allows a JDO implementation to implement PersistenceManagerFactory.getPersistenceManager() such that it returns a PersistenceManager associated with the javax.transaction.Transaction of the executing thread. If datastore transactions are used by JDO, the underlying data-store connection is also enlisted in the transaction.[2]

[2] If optimistic JDO transactions and a transactional in-memory cache are used, as opposed to datastore transactions, then there may not actually be an underlying datastore transaction associated. Note that it is still possible to use JDO optimistic transactions even when using JTA user transactions, by specifying setOptimistic(true) on the PersistenceManagerFactory instead of the javax.jdo.Transaction class, which would not be used in this scenario.

This returned PM is a new one just created, set up with the appropriate synchronizations and enlisted in the respective connection, as in the above example in which a user transaction was just initiated by the bean. If, however, a user transaction was already active and the BMT bean business method did not explicitly demarcate a transaction beginning, then a JDO implementation would ensure that the PM is an existing one currently correctly enlisted in the transaction. This is of particular importance for CMT as well, as explained below.

Each EJB business method should obtain its persistence manager from the persistence manager factory and close the persistence manager before returning, as shown in the above code snippet. This is not strictly necessary for stateful session beans under BMT, but the implied holding of a reference to a persistence manager and user transaction in an instance variable is a dubious practice. In fact, as we have just said, a client is guaranteed to obtain a PM enlisted in the same user transaction, if it is still active. Obtaining a persistence manager and closing it before returning in each EJB business method is thus perfectly safe and preferred practice, and it causes no issues for transactions that span multiple bean method invocations.

9.2.1.3 Container-managed transactions (CMT) via JTA

CMT enterprise beans can use neither the JDO transaction demarcation javax.jdo.Transaction nor the JTA user transaction management methods begin(), commit() or rollback() in their code. On the contrary, their business methods are guaranteed to be called within an active JTA transaction, if so declared. CMT relies on the container, other enterprise beans, or external plain Java classes for transaction initiation via the javax.transaction.UserTransaction interface.

As discussed above, this guarantees that the JDO PersistenceManager obtained in a business method of a CMT enterprise bean is always correctly associated and enlisted in the respective J2EE transaction.

When using CMT with declarative transaction management via the EJB deployment descriptor, the respective attributes in the XML descriptors must be understood and set correctly, as per the EJB specification. The intended configuration usually is one of the following:

- **Required:** Automatically begins a new transaction if none, and in this case, commits before returning to the caller.
- **Mandatory:** If existing, then as in Required, else throw Exception. Never automatically begin a transaction.
- **Never:** Always to be called without a transactional context, if existing then throw Exception. Could potentially be used together with JDO NontransactionalRead in respective methods.

These configurations would generally be more exotic and less used:

- **NotSupported:** "Unspecified transaction context." If existing, then suspend.
- **Supports:** Like Required if existing; like NotSupported if none.
- **RequiresNew:** If none, then begin new transaction and commit before return. If existing, then suspend.

In summary, it can be said that transaction management with enterprise session beans that rely on JDO for persistence matters really does not have to be any different from how EJB transaction demarcation has always been.

The remainder of this chapter focuses on examples of how to use JDO from CMT session beans in order to keep transaction management out of sample code.

9.2.2 Stateless session bean using CMT example

Without further ado, let's dive into practice and look at how a full stateless session bean with container-managed transaction would be coded. The class would be declared like this:

```
public class ExampleCMTBeanWithJDO implements SessionBean {
  private javax.ejb.SessionContext ejbCtx;
  private PersistenceManagerFactory jdoPMF;
  private InitialContext jndiInitCtx;
  // No other instance variables in a stateless bean!

  private final static String jndiName =
                       "java:comp/env/jdo/bookstorePMF";
```

In setSessionContext(), the JDO PersistenceManagerFactory is acquired from JDNI and assigned to an instance variable of the session bean:

```
public void setSessionContext(SessionContext sessionCtx)
                              throws EJBException {
    ejbCtx = sessionCtx;
    try {
      jndiInitialContext = new InitialContext();
      Object o = jndiInitCtx.lookup(jndiName);

      jdoPMF = (PersistenceManagerFactory)
            PortableRemoteObject.narrow (o,
              PersistenceManagerFactory.class);

    } catch (NamingException ex) {
      throw new EJBException(ex);
    }
  }
```

The other EJB service methods—ejbCreate(), ejbRemove(), ejbActivate(), and ejbPassivate()—usually stay empty unless code is required for other (non-JDO related) purposes in business methods later:

```
public void ejbCreate() throws javax.ejb.CreateException {
}

public void ejbRemove() throws EJBException {
}

public void ejbActivate() throws EJBException {
}

public void ejbPassivate() throws EJBException {
}
```

Now in the business methods of the EJB, JDO can be easily used according to this generic code template (ignore return and argument types for now; more on that later):

```
public void doSomething(int arg) {
        PersistenceManager pm;
        try {
            pm = jdoPMF.getPersistenceManager();

            //
            // Do something using JDO now...
            //

        } catch (Exception ex) {
            ejbCtx.setRollbackOnly();    // sic!
            throw new EJBException(ex);
        } finally {
            try {
                if (pm != null && !pm.isClosed())
                    pm.close();
            } catch (Exception ex) {
                // Log it
            }
        }
        // Maybe, return something;
    }
```

The core of the business logic, what it actually does, goes in the middle of the above method, after the getPersistenceManager(). We look closer at the details of such code, and any related JDO specifics, just below.

How Did the JDO PMF Get into JNDI?

In the code above, the **PersistenceManagerFactory** is obtained via a lookup from JNDI, instead of using the **JDOHelper** as in the simpler examples elsewhere in this book. How, when and by whom did that PMF get bound into JNDI in the first place?

The idea in line with global J2EE architecture is that the JDO PMF is bound into JDNI by a JCA adaptor; more details on this can be found in Chapters 11 and 12.

Alternatively, a simple plain vanilla Java "start-up class" could achieve a similar goal, by manually creating a **PersistenceManagerFactory**, presumably using the **JDOHelper** class, and binding it into JNDI. However, J2EE "start-up" classes are a non-standard feature that depend on respective proprietary Application Server APIs. Furthermore, and more importantly, such a manual approach would defeat the purpose of the JCA architecture that JDO can leverage in a managed scenario, and would make it difficult to use container-managed (possibly distributed) transactions.

9.2.3 Stateful session bean with CMT example

As mentioned in the introduction, the J2EE specification provides for another type of session bean, the stateful session bean, which automatically maintains its conversational state across multiple client-invoked methods.

Many business processes have simple conversations that can be modeled as completely stateless services via stateless session beans as already seen. Alternatively, if the service appears stateful but is simple enough, passing state from the client to the session bean as argument to each business method is a possibility.

However, some use-cases describe business processes that are inherently conversational and require multiple and possibly varying sequences of method calls to complete. Sometimes, it makes more sense to model this as a stateful session bean. This can be more efficient than reconstructing state inside the bean from arguments each time, or retrieved from any persistent datastore. This book is not the place to outline such a choice in more detail. However, we examine how to build such a stateful session bean with access to JDO, should this be a preference over a completely stateless architecture for a given scenario.

As above, for the stateless session bean example, we focus on container-managed transactions only in this code. BMT would be equally possible and look as outlined earlier.

The declarations at the beginning of the bean implementation class are largely similar to the stateless source code example above. The one important difference is that, because this is a stateful session bean, we are now permitted to keep instance variables across method calls—for example:

```
private String bookISBN;
```

This instance variable could remember the customer ID across business method invocations. Remember that instance variables of stateful session beans need to be serializable, so although an instance variable of type String or other simple type is just fine, holding onto an arbitrary persistent object would be asking for trouble and deserves a closer look:

```
private Category currentCategory;  // bad!
```

Instead, if a stateful session bean indeed needs to hold onto the same persistent object between business method calls, it is generally preferable to keep the JDO object identity instead of the persistent object itself in an instance variable, like this:

```
private Object catID;  // ok.
// ...

public void doOneThing(String name) {
        PersistenceManager pm;
        try {
            pm = jdoPMF.getPersistenceManager();
            // ...
            Category catPC = ... // e.g. from Query result
            catID = pm.getObjectId(catPC);
        }
        // ... Omitting error handling etc.—it's as above
        finally {
            pm.close();
        }
}

public void doAnotherThing() {
        PersistenceManager pm;
        try {
            pm = jdoPMF.getPersistenceManager();
            Category CatPC = pm.getObjectById(CatID, true);
            // ...
        }
        // ... Omitting error handling etc.—it's as above
        finally {
            pm.close();
        }
}
```

Just because the enterprise bean is stateful does not imply that it is a good practice to keep a reference to a JDO PersistenceManager in an instance variable. As in the example above for a stateless session bean, the business methods of stateful session beans should still obtain a PM at the beginning of every method and close it at the end. Transactional integrity can still be guaranteed and is unrelated to holding onto the same PM from different stateful entity bean method calls.

9.2.4 Service-oriented architecture (SOA)

In the above examples, we looked only at the structure of session enterprise beans, and abstracted the specifics of a service's method contract and inner workings. We examine those more closely here.

Let's first briefly think about what we mean by a *service.* A service in this context is something phenomenally simple: It has a signature, is called with arguments, and returns something. More importantly, however, a service in this context is usually a remote service, often referred to as a remote session "façade" because it fronts local code. Arguments and return values are thus passed by value, in a serialized form, not by reference.

Of course, stateless session beans are not simply dumb remote procedure call (in Java, i.e., RMI) endpoints; they include transaction context propagation, possible clustering, some security authentication mechanism, and so on. Still, the fundamental architecture is one of a remote invocation.

A service thus exposes coarse-grained use-cases to the public. Sometimes, such "services" offered by session beans have relatively straightforward incoming arguments and return simple outgoing results—for example, a createXYZ(String name) method with simple arguments such as Strings for a name, and similarly, methods of type double getXYZ(String code) or something along those lines. Business methods with such signatures are easy to implement and should not require further discussion after what was presented so far.

However, some session beans, more often than not, want to return (or receive as arguments in an updateXYZ()-style scenario) more complex types—for example, a Book, an Author or other objects "inspired" by the persistence-capable objects from the actual underlying domain model. It is this second case that needs to be examined more closely in the context of JDO.

The point here is that in a service-oriented architecture (SOA), a client cannot simply "navigate" to other objects of a domain model that were not explicitly returned by value. If a service does not return the data that a service invoker requires, then the service definition, the service implementation, and the service-invoking client need to be changed.

In such a pure SOA architecture, clients of a session bean service (be it the Web tier or standalone EJB clients) also cannot use the JDO API directly, because this would be a violation of the service-oriented architecture. Clients would invoke only well-defined services. The objects returned from a service would thus not have a JDO environment on the client. This immediately leads to the question: What instances of which class should a service return and receive as arguments? We look at this in more detail in the following two sections.

9.2.4.1 Data-transfer objects (aka value objects) and JDO PC serialization

Let's first examine the case of returning objects from an enterprise session bean service business method. We look at the reverse case, passing objects as arguments to a service, in the next section.

For the sake of an illustrative example, imagine a library application: Suppose that the session bean outlined above had a method to return a collection of books; it could be objects that meet a certain query condition, such as "has copies that the currently authenticated user has borrowed," or it could be a given number of objects, starting from a given index, or anything else that requires the method to return a collection of persistent objects. For the initial discussion, let's assume that the persistent Book class does not have any Java references to other persistent objects and looks something like this:

```
public class Book {
   //
   // JDO persistent-capable class that will be enhanced!
   //
   private String author;
   private String isbn;

   public String getAuthor() { return author; }
   public void setAuthor(String _a) { author = _a; }

   public String getISBN() { return isbn; }
   public void setISBN(String _isbn) { isnb = _isbn; }
}
```

This is often overly simplistic for most real-world applications, but bear with it as a first example for good reasons. We look into the matter of graphs of objects later in this chapter.

A developer may be tempted to write at first, without further thought, something like this (this code is intentionally wrong, so be sure to see the discussion below):

```
public Collection getBooks(/* e.g. int start, int count */) {
   PersistenceManager pm = null;

   // try/catch/finally error handling omitted, as above
   pm = jdoPMF.getPersistenceManager();
   Query q = pm.newQuery(Book.class);
   // q.setFilter("...");
   Collection results = (Collection)q.execute();

   pm.close();
   return results;  //  BAD.  WRONG.  See below.
}
```

This will not work, for several reasons:

- The persistence-capable class is not declared serializable.
- Persistent objects are accessed by the container for serialization before returning from method, but after transaction has been committed (both CMT and BMT).
- Most importantly, the collection returned by Query.execute() is not serializable. Worse yet, it could be holding onto datastore resources such as cursors, and so on.

The last point can be easily addressed, so let's get this out of the way first. The following code addresses that issue by simply copying JDO query results into a JDK standard Collection, here an ArrayList. It is assumed that a filter expression has limited the size of the result collection to a "sensible" size:

```
public Collection getBooks(/* int start, int count* /) {
   PersistenceManager pm = null;
   List results = null;

   // try/catch/finally error handling omitted, as above
   pm = jdoPMF.getPersistenceManager();

   Query q = pm.newQuery(Book.class);
   // q.setFilter("...");
   Collection jdoResult = (Collection)q.execute();
   results = new ArrayList(jdoResult);
   q.close(jdoResult);

   pm.close();
   return results;
}
```

This still won't work, though. As mentioned, the persistent objects are not serializable at this point. Because the Book instances contained in the ArrayList will "travel over the wire" to the remote client after the EJB business method returns, we have to stick something serializable into the collection. There are two ways to achieve this:

- You can use handcrafted (or code-generated) non-persistence-capable but serializable helper classes specifically for transporting the data between tiers. Readers familiar with other literature will recognize this as a familiar architecture choice, with Sun's J2EE patterns literature referring to this concept as value objects (VO), while other sources prefer the term of data transfer object (DTO) for the same idea. Instead of returning a collection of persistence capable Book instances, we would create instances of new transfer objects in the service. We restrain from the initial temptation of making this BookVO class extend the Book class. Similarly, we do not use a constructor that takes the persistent object. Instead of using a Constructor taking individual arguments, a dedicated Assembler class could have been used as well. This ensures decoupling and avoids dependency, so that only the Serializable class is required on the classpath of the EJB client, and the persistent Book class remains restricted to the server with access to the domain model.

  ```
  public class BookVO implements Serializable {
    private String author;
    private String isbn;

     public BookVO(String bookAuthor, bookISBN) {
        author = bookAuthor;
        isbn = bookISBN;
     }

     public String getAuthor() { return author; }
     public String getISBN() { return isbn; }

     // Note: Delibarately no setters yet!  See below.
  }

  public Collection getBooks() {
     PersistenceManager pm = null;
     List results = null;
     try {
         pm = jdoPMF.getPersistenceManager();
         Query q = pm.newQuery(Book.class);
         // q.setFilter("...");
         Collection jdoResult = (Collection)q.execute();

         results = new ArrayList(jdoResult.size());
         Iterator it = results.iterator();
         while ( it.hasNext() ) {
  ```

```
            Book pcBook = (Book)it.next();
            BookVO dtoBook = new BookVO(
                pcBook.getAuthor(), pcBook.getISBN());
            results.add(dtoBook);
        }

        q.close(jdoResult);
    } catch (Exception ex) {
        ejbCtx.setRollbackOnly();    // sic!
        throw new EJBException(ex);
    } finally {
        try {
            if (pm != null && !pm.isClosed())
                pm.close();
        } catch (Exception ex) {
            // Log it
        }
    }
    return results;
}
```

- Alternatively, it is perfectly possible to actually tag the Book persistence capable class as serializable in its declaration. An interesting and useful feature of JDO in this respect is that persistence-capable classes are guaranteed to implement Java JDK Serialization in a manner such that enhanced classes can be serialized to and from non-enhanced classes; i.e., a correct serialVersionUID will be calculated by an enhancer, and so on. It is thus perfectly possible to use an enhanced and persistence-capable version of a class in the service implementation on a server, and the non-enhanced code of the same class on a client. Alternatively, the same enhanced class could also be used on the client; it will deserialize into an environment without JDO runtime, and be in transient state. Transient objects are guaranteed to behave like non-enhanced ones. An additional issue in this option as opposed to the one above is that the commit that the EJB container performs before the return (in CMT, or the demarcation the bean would do in BMT) has possibly flushed the persistent objects. So if the container tries to access objects for serialization after the commit, it gets an exception because there is no active transaction. This can be addressed by a makeTransient() call. Related to this, if there was any chance that not all attributes are read into memory yet, maybe due to a pre-fetch limitation, then invoking retrieve() before makeTransient() ensures that all persistent fields are loaded into the persistent object by the respective PersistenceManager. Here is an example code illustrating this:

```
public class Book implements Serializable {
   ...

public Collection getBooks() {
   PersistenceManager pm = null;
   List results = null;
   try {
       pm = jdoPMF.getPersistenceManager();
       Query q = pm.newQuery(Book.class);
       // q.setFilter("...");
       Collection jdoResult = (Collection)q.execute();

       pm.retrieveAll(jdoResult);
       results = new ArrayList( jdoResult );
       pm.makeTransientAll(results);

       // Or use loop to iterated and individually
       // retrieve/makeTransient instead of xyzAll():
       //
       // results = new ArrayList( jdoResult.size() );
       // Iterator it = results.iterator();
       // while ( it.hasNext() ) {
       //    Book pcBook = (Book)it.next();
       //    pm.retrieve(pcBook);
       //    pm.makeTransient(pcBook);  // !
       //    results.add(pcBook);
       // }

       q.close(jdoResult);
   } catch (Exception ex) {
       ejbCtx.setRollbackOnly();    // sic!
       throw new EJBException(ex);
   } finally {
       try {
           if (pm != null && !pm.isClosed())
               pm.close();
       } catch (Exception ex) {
           // Log it
       }
```

```
        }
        return results;
    }
```

Possible issue: makeTransient() on dirty object

A specific corner case of the above described general pattern can cause an issue in some applications. A closer look at the State Transition table in the JDO specification reveals that invoking makeTransient() on a persistence-capable instance in persistent-dirty, persistent-new, persistent-deleted or persistent-new-deleted state is considered an error, and throws an exception.

In cleartext, if a persistent object is modified in the EJB business method, and the transaction is not committed at the time makeTransient() is called, then makeTransient() will fail.

We are calling this somewhat of a corner case because many real-world business methods (including the example above) may never modify persistent objects before they need to be returned to the client, thus made transient. Also, if the other approach is used, using data transfer objects (DTO), then this issue never occurs anyway because as we have seen above, the respective fields of the persistent object are copied into the DTO and the persistent object is never made transient. Furthermore, if bean-managed transactions (BMT) are used, then the correct ordering of operation (commit first to flush dirty objects, thus making dirty instances persistent-clean again, then makeTransient) can help to work-around this.

Still, future JDO specifications may explicitly address this case. If you encounter this issue in your project, contact your JDO vendor about this, they may be able to offer a proprietary solution until the standard catches up with this.

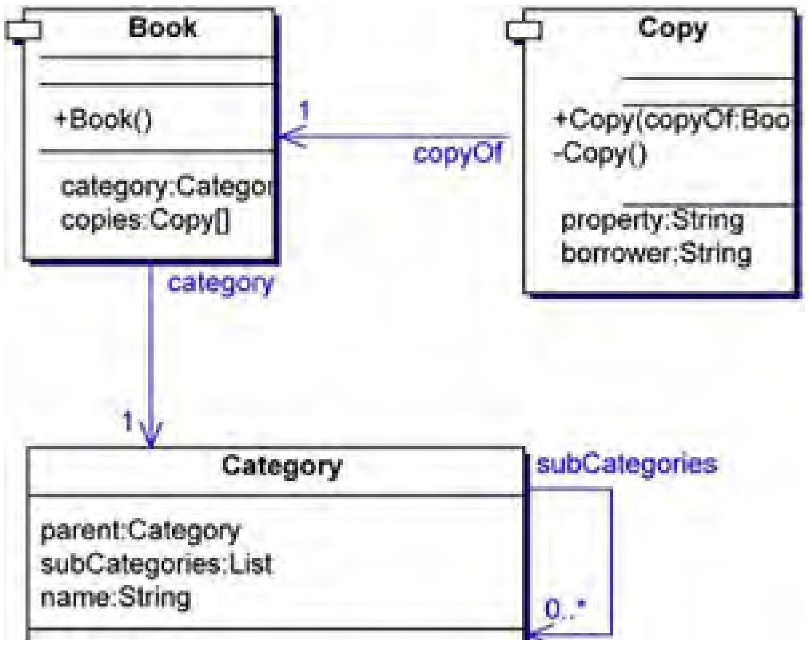
These are arguments for choosing one or the other of above strategies:

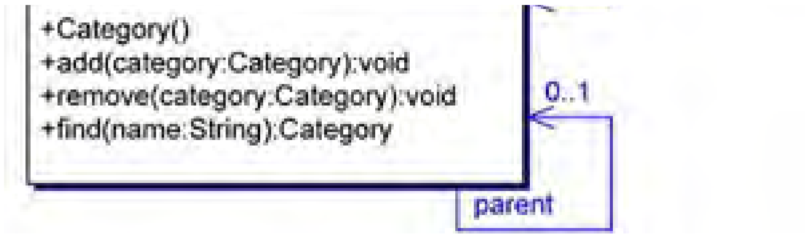
- Although coding additional classes for the first option may seem like a disadvantage of that approach at first, the implied reduction of coupling between the domain model used in the session bean implementation based on JDO, and the client, usually a presentation tier, can be an advantage in the longer term.
- If a service returns a condensed summary view, an analysis, results of a calculation, or any other information not directly represented by an existing class of the domain model, a dedicated non-persistence-capable DTO class likely makes sense anyway.

If a business method does not return a collection of objects, but merely one specific persistent object, the same issues would arise and the above points would equally apply.

Let's now extend the domain model of our Book example slightly, but significantly. Let's assume that a small enhancement requested is that Books are categorized hierarchically. We introduce a new Category class, as indicated in Figure 9-3.

Figure 9-3. Extended example model, Book with association (UML representation).




+Category()
+add(category:Category):void
+remove(category:Category):void
+find(name:String):Category
0..1
parent

A Category is just an example. What matters is that the persistent class Book now has a relationship to another persistent class. You can easily imagine other cases; indeed, most real-world domain models have various associations between persistent classes. Persistent objects thus are nodes in a graph, with links to other persistent objects.

And that's where it starts to get really interesting: With the approach of direct serialization of formerly persistent objects, which worked fine for the simple case above, we would now potentially serialize a huge data stream: The entire closure of instances would normally be serialized along.

Referenced instances would also have to be made transient; retrieve() and makeTransient() would have to be manually invoked for the entire graph of reachable instances, because that is what serialization does. The makeTransient() call by itself is not propagated across the graph, although it would be possible to write graph traversal helper functions using reflection and semi-automating such procedures.

However, the real question is this: Do we actually want the entire Category hierarchy (examine how the Category instances themselves hold references to a parent and subcategories) to be serialized and sent over the wire with each Book object? Presumably not, and the same would apply to other similarly structured real-world domain models.

It could seem that there are workarounds for some of these issues, for example:

- Code could explicitly *nullify* references after making the instance transient.
- The domain model could be changed to make it "serialization-friendly" by taking advantage of JDO's ability to declare a field as transient for the purpose of Java serialization, yet persistent for the purpose of JDO. This is technically possible by specifying the Java transient keyword on the attribute, together with explicitly marking the field with persistence-modifier="persistent" in the JDO XML metadata descriptor.
- We could change the domain model to make it "serialization-friendly" by attempting to "direct" associations in the "right" way, for the above example, making the relationship from Category to Book (as a Collection) instead of from Book to Category so that Book can be serialized.

However, all such approaches could rapidly defeat the purpose of an object-oriented transparent domain model. Using explicit data transfer objects and assemblers are usually a better choice for a true service-oriented architecture, albeit requiring extra coding. Attributes of more complex data transfer objects should always be simple types such as String, double, and so on, or other data transfer objects, all created by an Assembler.

In the case of the above example, we could model a data transfer object (name it value object if you are more used to that terminology) for a persistent Book class that deliberately omits the link to Category, and instead contains less data, according specifically to the use-case that requires this service. For example:

```
public class Category {
   //
   // JDO persistent-capable class that will be enhanced!
   //
   private Category parent;
   private List subCategories;
   private String name;

   // ...
}

public class Book {
   //
   // JDO persistent-capable class that will be enhanced!
   //
   private String author;
   private String isbn;
   private Category category;

   // ...
}

public class BookVO1 implements Serializable {
    private String author;
    private String isbn;
    private String category;  // String, not Category!
    public BookVO1(String bookAuthor, String bookISBN,
                String bookCategoryName) {

        author   = bookAuthor;
        isbn     = bookISBN;
        category = bookCategoryName;  //  !
    }

    public String getAuthor()   { return author;   }
    public String getISBN()     { return isbn;     }
    public String getCategory() { return category; }

    // Note: Deliberately no setters yet!  See below.
}

// ...
   public Collection getBooks() {

   // Business logic method would not use retrieve() etc.
   //
      {
         BookVO dtoBook = new BookVO1( pcBook.getAuthor(),
                         pcBook.getISBN(),
                         pcBook.getCategory().getName());
```

In summary, the basic model is the same as has been recognized as an EJB best practice for some time already, with explicit data transfer or value objects returned from inside session beans.

9.2.4.2 Return-trip ticket: Parameters

Another issue worth mentioning is what we may call the "return trip" of value objects from a client tier: Most real-world applications offer remote clients business methods related to updating data. In their simplest form, such methods could look like this:

```
void updateBook(String bookISBNKey, String newAuthor);
```

However, given that we have already defined a data transfer class for Books above, it could make sense to allow a client to change instances of these on the client (introducing setter methods) and send them back to a service:

```
public class BookVO2 implements Serializable {
    private String author;
    private String isbn;

    public String getAuthor()  { return author;  }
    void setAuthor(String _auth) { author = _auth; }

    public String getISBN()     { return isbn;     }
    void setISBN(String _isbn)  { isbn = _isbn;    }
}

public void updateBook(BookVO2 updatedBook) {
    // 1) Find persistent book
    // 2) Update persistent instance
}
```

Either way, the basic pattern is the same: First, find the persistent object, and then change its attributes either to values passed as arguments to the business method or from the DTO received as arguments.

What differs is how to find the persistent object. Let's remind ourselves that we really do want to find an existing object and change it. From update-type methods, we never want to do something like this:

```
Book pcBook = new Book(isbnCode);  // wrong here!
pm.makePersistent(pcBook);         // wrong here!
pcBook.setAuthor(newAuthor);
```

The above code snippet would invariably lead to the creation of a new persistent instance, not an update of existing information. In order to find a persistent object to change, we need to decide how the identity of the object to be changed is known to the service implementation. Several options exist:

- If the domain model uses JDO application identity and the key fields are part of the DTO, e.g., a persistent book class that would use the ISBN number as application identity and define a corresponding BookID class, the update code for the above example would look like this (issues around change of existing persistent object's application identity are ignored for simplicity):

```
public void updateBook(BookVO2 updatedBook) {
    PersistenceManager pm = null;

    try {
        BookID objId;
        Book pcBook;

        pm = jdoPMF.getPersistenceManager();
        objId = new BookID(updatedBook.getISBN());
        pcBook = (Book)pm.getObjectById(objId, true);
        pcBook.setAuthor(updatedBook.getAuthor());

    } catch (Exception ex) {
        ejbCtx.setRollbackOnly();
        throw new EJBException(ex);
    } finally {
        try {
            if (pm != null && !pm.isClosed())
                pm.close();
        } catch (Exception ex) {
            // Log it
        }
    }
 }
```

- If the domain model uses JDO datastore identity, the persistent Book's object identity needs to be specified either via an extra parameter to the update-type method, or included in the DTO. Either way, this object identity should be transformed into a String for traveling across tiers, because using a parameter or member variable of type Object would introduce a dependency on the respective JDO implementation's class for datastore identity. So an extended DTO could look like this:

```
public class BookVO3 extends BookVO2 {
    private String oid;

    public BookVO3(String _oid) {
        oid = _oid;
    }

    public String getOID()  { return oid; }
}
```

  Although it is tempting at first to obtain the persistence-capable object's ID from within the DTO constructor, this would create a dependency on JDO in the DTO, and thus on the client, and should be avoided. Instead, either use a separate assembler class or pass the OID as parameter to the constructor as shown above, obtaining it via PersistenceManager.getObjectId(pcBook).toString() when constructing the DTO. The PersistenceManager.newObjectIdInstance() method then proves helpful for converting such an OID string back into a JDO identity in the service:

```
public void updateBook(BookVO3 updatedBook) {
  PersistenceManager pm = null;

  try {
      Object objId;
      Book pcBook;

      pm = jdoPMF.getPersistenceManager();
      objId = pm.newObjectIdInstance(Book.class,
                         updatedBook.getOID());
      pcBook = (Book)pm.getObjectById(objId, true);
      pcBook.setAuthor(updatedBook.getAuthor());

   } catch (Exception ex) {
       ejbCtx.setRollbackOnly();
       throw new EJBException(ex);
   } finally {
       try {
            if (pm != null && !pm.isClosed())
                pm.close();
        } catch (Exception ex) {
            // Log it
        }
    }
}
```

- If the object to update is part of a larger DTO, then none of the above is needed, because we already have an object reference to the respective persistent object, without having to go through ID objects. Care must be taken, however, to repeat the basic "find persistent instance, and then update it" cycle for dependent DTO instances as well, in order to avoid assigning transient DTO objects to reference fields of persistent objects.

Various enhancements are possible around the approaches shown here. For example, optimizations to only update "relevant" persistent fields, usually a mechanism to recognize those that have actually been changed in data transfer objects. The basic pattern always stays the same, however.

Web Services

The latest EJB specification standardizes how stateless session beans may be exposed as Web services; this is termed a Web-service client view on a Web-service endpoint interface. So if you already have stateless session beans and an application server that supports the standard, this should purely become a matter of configuration at deployment.

However, to implement Web services, you don't necessarily need EJB 2.1 or EJBs at all. Several application-server-independent third-party tools exist to build Web services around "plain vanilla" Java classes.

In both scenarios, data-transfer objects returned from services (and similarly service input arguments) are marshaled into XML data format according to a WSDL document in order to be packaged up in a Web-service SOAP response.

9.3 Message-Driven Beans and JDO

A message-driven bean is a server-side message consumer that has, according to the EJB specification[3] the following characteristics:

[3] http://java.sun.com/products/ejb

- Executes upon receipt of a single client message.
- Is asynchronously invoked by container on arrival of messages, never called directly by a client.
- Can be transaction-aware.
- Is relatively short-lived and is stateless. (A typical EJB container provides a scalable runtime environment to execute a large number of message-driven beans concurrently.)

The EJB 2.0 specification supported only JMS message-driven beans. The EJB 2.1 specification extends the message-driven bean contracts to support other messaging types in addition to JMS.

Transaction demarcation for message-driven beans is similar to the session beans as shown above. Again, both BMT and CMT are possible. In the case of CMT, only the Required and NotSupported transaction declarations make sense and are allowed.

Transaction management in the context of JDO for persistence operations is identical to what has been discussed earlier for session beans: Obtaining a PersistenceManager inside a CMT bean onMessage() method ensures that it is always correctly associated and enlisted in the respective J2EE transaction.

9.3.1 Example code

The following code snippet shows how a message-driven bean might look:

```
public class OrderBooksBean
        implements MessageDrivenBean, MessageListener {

   private MessageDrivenContext ejbMsgCtx;
   private PersistenceManagerFactory jdoPMF;

   public void ejbCreate() {
   }

   public void setMessageDrivenContext(
           MessageDrivenContext messageDrivenContext)
                           throws EJBException {

       ejbMsgCtx = messageDrivenContext;
       try {
           String jndiName = "java:comp/env/jdo/PMF";

           Context jndiInitCtx = new InitialContext();
           Object o = jndiInitCtx.lookup(jndiName);
           jdoPMF = (PersistenceManagerFactory)
                   PortableRemoteObject.narrow (o,
                     PersistenceManagerFactory.class);

       } catch (NamingException ex) {
           throw new EJBException(ex);
       }
   }

   public void ejbRemove() throws EJBException {
   }

   /**
    * Updates orders in persistent books.
    * Finds a book with the message.bookISBN, and
    * if its price is lower than message.maxPrice,
     * then adds a new order for the message.clientNum.
     **/
    public void onMessage(Message message) {
        PersistenceManager pm = null;
        try {
            MapMessage mapMsg = (MapMessage)message;
            String bookISBN = mapMsg.getString("bookISBN");
            double maxPrice = mapMsg.getDouble("maxPrice");
            long  clientNum = mapMsg.getLong("clientNum");

            pm = jdoPMF.getPersistenceManager();
            BookID oid = new BookID(bookISBN);
            Book book = (Book)pm.getObjectById(oid, true);

            if ( book.getPrice() <= maxPrice ) {
                pcBook.newOrder(clientNum);
            }
            else {
                // Don't order... reject?
            }

        } catch (Exception ex) {
            ejbMsgCtx.setRollbackOnly();
            throw new EJBException(ex);
        } finally {
            try {
                if (pm != null && !pm.isClosed())
                    pm.close();
            } catch (Exception ex) {
                // Log it
            }
        }
    }
}
```

If persistent objects were to be included in messages, e.g., using javax.jms.ObjectMessage, then the points raised earlier in the context of session-bean parameters (serializability or DTO, graph of objects, object ID as String, and so on) would similarly all apply.

Why Use Message-Driven Beans?

Using message-driven Enterprise JavaBeans offers advantages such as declarative transaction management, declarative messaging system parameters (e.g., JMS topics and destination), and most notably a generally scalable and clusterable container.

However, it is worth mentioning that it is technically not necessary to use a message-driven bean to listen to a JMS queue; a plain vanilla Java class might do just as well for a given application. The point is this: JMS and EJB are orthogonal to each other, so using JDO and JMS together does not necessarily require the use of EJB!

9.4 Entity Beans and JDO

Having discussed session and message-driven enterprise beans above, let's now turn to another type of enterprise bean, entity beans. As per the EJB specification[4] , an entity enterprise bean has the following characteristics:

[4] http://java.sun.com/products/ejb/

- Provides an object view of data in the database.
- Allows shared access from multiple users.
- Can be long-lived (lives as long as the data in the database).

Two different types of entity beans exist: container-managed persistence (CMP) and bean-managed persistence (BMP) entity beans. Both types of entity beans can be created and removed, and queried with a language named EJBQL. They also have an identity, their primary key. Does this all seem remarkably similar to what you have learned about JDO so far? Well, you're not the only one. Indeed, the debate about when and whether to use entity beans versus JDO arose with the inception of JDO, and the debate is active and ongoing.

One possible motive may be the idea of directly modeling remotely accessible persistent components. As we have seen earlier, JDO itself does not include any notion of remote invocation, but focuses purely on JVM local persistence issues. Entity beans may thus seem tempting at first for this purpose, instead of developing the required calling, pooling, and so on, infrastructure from scratch. However, most of today's EJB literature highly recommends using the so-called *session façade pattern,* wherein entity beans are accessed from other local enterprise beans only, and although technically possible, are not actually exposed to remote clients. An often-cited and well-documented reason for this architectural choice is performance issues with fine-grained remotely accessed entity beans. So if you use entity beans only locally from session and message-driven beans anyway, why would you not rather forget about entity beans altogether and simply use the JDO API from within session and message-driven beans to implement persistence?

Another rationale could be existing entity beans that need to be reused. Given that these entity beans are usually persisted in a relatively straightforward relational schema when using entity beans, it is certainly worth at least evaluating whether it is possible to migrate to an O/R mapping-based JDO implementation that could reverse engineer the existing relational schema while preserving existing legacy data. Chapter 12 provides more information on such a road.

A related possibility would arise should you have a project that uses existing entity beans that do not map data to and from a relational database, but use another, e.g., an EIS-type or TP monitor, backend datastore. Although JDO technically allows and the market aims at providing JDO implementations for such datastores, it may not be available for your EIS of choice yet. It may be worth talking to the EIS vendor about JDO, and checking with independent JDO implementations about the EIS, in such a specific case. However, this is a somewhat theoretical possibility, because most real-world integration projects join together using a higher-level API, likely service-oriented, not on an entity level. A similar and slightly more common scenario is if you have existing entity beans with BMP that do O/R mapping entirely through, e.g., PL/SQL-stored procedure calls rather than direct access to the underlying relational tables. Again, check your JDO implementation for support of stored procedures.

Last but not least, you could be on a project that actually requires an architecture with distributed and transactional persistent components. Read again, distributed transactional persistent objects, not distributable services. If you really are working on an application like this, the remotely accessed entity beans may be a fitting solution.

In short, with the advent of JDO, it is hard to find good reasons to continue using entity beans instead of JDO for modeling and working with persistent objects in new J2EE projects.

9.4.1 Bean-managed persistence (BMP) entity beans and JDO

Bean-managed persistence (BMP) entity beans are components that respect the entity bean API contract, but implement actual persistence code themselves within the BMP implementation class, instead of relying on "transparent" declarative persistence handled by the EJB container, as is the case for container-managed persistence (CMP) entity beans.

Reasons for using BMP instead of CMP enterprise beans could be limitations in (or in early containers, a simple lack of) container-managed persistence (CMP) entity beans, or non-relational datastores.

Should a specific application have a good reason to adopt such an architecture, it is technically feasible to use JDO to implement the ejbCreate, ejbRemove, ejbActivate, ejbPassivate, ejbLoad, and ejbStore methods that a BMP entity bean needs to provide. The JDO 1.0 specification roughly outlines how this could be done in its Chapter 16.2.1.[5]

[5] If you are interested in looking into this issue in more detail, be aware that the initial JDO 1.0 specification from Spring 2002 mistakenly stated that "The ejbActivate() method acquires a PersistenceManager from the PersistenceManagerFactory...." This is wrong because the EJB 2.0 specification does not specify that ejbActivate()

will be called within the context of an open transaction. Because persistence managers must be obtained within such a context in order to be bound to the J2EE transaction, acquiring the PM from the PMF should be done in the ejbLoad() method only, with the ejbActivate() method remaining empty. The v1.0.1 maintenance release of the JDO specification has corrected this detail.

However, as argued above, the real question in this context is not "How do you do this?" but rather "Why would you do this?" The authors of this book believe that for most applications, the answer is simply "You wouldn't do this!" We thus do not provide examples of how to use JDO from BMP entity beans in this book.

Using JDO to Implement CMP

Another issue is whether JDO could be used as the underlying technology to implement container-managed persistence (CMP) entity beans in a J2EE application server. The answer is definitely yes; JDO could be used for that purpose. This would, however, be an AppServer implementation decision and is unrelated to the implementation of J2EE applications per se. This scenario, of course, hides JDO features that have no CMP equivalent.

Indeed, at least one (the SunOne AppServer) is reported to work like this. Interestingly, the O/R mapping toolkit that JBoss uses internally to implement CMP is also reported to likely become exposed for local (non-EJB!) persistence, although at the time of this writing, it is not clear whether this AOP-based service will be accessible via a JDO-compliant API.

Note also that using the standard JDO 1.0 API alone leads to some minor issues complicating this slightly in practice; however, vendor extensions could accommodate these. For example, seamless translation of some EJBQL to JDOQL queries may be difficult due to the lack of bi-directional (inverse, managed) relationships as well as projections in JDO queries. See Chapters 24.7 and 24.17 of the JDO 1.0 specification.

9.5 To Use EJB or Not to Use EJB?

Now that we have explored the details of JDO integration with EJB in this chapter, an often-asked question may be on your mind: "Should I use EJB in a given application?" The answer, as happens so often, is "It depends."

Before proceeding, let's quickly recall what has been said in this chapter already. Enterprise JavaBeans come in three flavors: entity, session and message-driven beans. The authors of this book believe that JDO clearly offers a better alternative over EJB entity beans and advocate using JDO as primary persistence API within the service methods of session- and message-driven beans. Alternatively, directly using JDBC or possibly a combination of JDO and JDBC may also be applicable.

However, as we have seen above and as readers familiar with the matter know, EJB-based development even with just two remaining bean types of interest represents a certain inherent complexity. The real question, then, is this: "Should I use EJB session- and message-driven beans at all?"

Session- and message-driven beans are components in the EJB architecture, which essentially provides a distributed component model, plus features such as security, transaction management, and scalability. Another J2EE API provides similar features: the Servlets API. Servlets represent another kind of J2EE component architecture. It is worth comparing the two in more detail with respect to the features of interest to help decide whether EJB is appropriate for a given application:

- **Component distribution:** Does your application require fully distributed components that can be invoked from other remote components and dislocated plain old Java objects on other Java virtual machines? Then EJB session beans are probably a good choice. Or does your application mostly use co-located Plain Old Java Objects (POJO) "components" (i.e., well-defined public interfaces with underlying implementation possibly spanning multiple classes that the component consumer should not directly access) with a few coarse-grained services, including human-readable Web pages, offered to the outside world? Then Servlets may be enough.[6]

  [6] A Servlet-only based architecture is still "service-oriented" from an external system's point of view: Servlets offer HTTP-based (only) services, input arguments are the request parameters, and the returned documents contain the service result, either an HTML page if the service-consumer is a human, or as a SOAP-based XML response (or simpler format, e.g., plain text string) if the service-consumer is a machine.

- **Service protocol:** If remotely accessible services need to be invoked only via HTTP, then Servlets are sufficient.[7] If RMI and possibly IIOP for CORBA interoperability are a requirement, then EJB session beans are the obvious choice. If cross-firewall support is an issue, Web services often make sense again, though. A similar argument applies if non-Java clients need access to the service, e.g., a .NET application. If performance is a concern, processing time for features such as SOAP marshalling and unmarshalling overhead could be worth a thought. In short, it really depends on the big picture.

  [7] It is worth mentioning briefly that the Servlet API technically provides for non-HTTP Servlets, leaving the door open for support of other request/response style protocols in the future. This option, however, is almost never used in practice, and Servlet support in most application servers today really means HTTP Servlet support only.

- **Security:** Security is a broad subject, encompassing authentication, authorization, encryption, and other topics. The relatively simple method-level declarative security model that EJB containers offer requires role-based authorization. Servlets can also be protected with role-based access control in the J2EE standard, albeit at an even cruder class-level only because they do not expose individual business methods by design. However, many real-world applications today require more advanced security authorization anyway. Other security aspects, such as authentication and encryption (SSL, via HTTPS in Web containers), are provided by both Web and EJB containers.

- **Stateful architectures:** Both the EJB and Servlet component models offer the possibility for stateful architectures via stateful session beans on one hand and the HTTP session variables on the other hand. However, if you need a stateful distributed component architecture, the EJB model is possibly more appropriate.

- **Message-oriented architectures:** If an application is essentially an asynchronous message-processing "sink," then relying on EJB 2.0 message-driven beans probably makes sense. The Servlet component model itself does not offer any comparable message awareness; however, plain vanilla Java classes can act as JMS destinations as well. Further discussion of this topic is outside of the scope of this book.

- **Efficient multi-threading and request dispatching:** While an inherent feature of EJB containers, a Web container also provides features such as retrieving a thread from a thread pool, calling the correct method to handle the request, and so on. This is not necessarily an argument in favor of the EJB component model. An in-depth discussion of scalability with clustering follows below.

- **Transaction demarcation:** EJB offers declarative transaction demarcation via CMT, for which there is no standard alternative in the Servlet component model. Servlets can, however, use explicit demarcation via JTA, a standard J2EE API that EJB also uses when operating in BMT mode. As we have seen earlier, JDO fully integrates with JTA and offers strong support for ACID transactions on persistent data, even when used outside of EJBs. (In a Web-based architecture, including Web services, transaction demarcation can often be centralized in relatively few classes, e.g., in Servlets, a Servlet filter, MVC-type controller Action classes, or similar piece of code.)

- **Transaction context propagation:** If your service is invoked remotely as part of a larger "value chain," i.e., in an Enterprise application integration (EAI) scenario, then EJB definitely makes sense because the transaction context with the currently active transaction can be propagated across service method invocations of yours and other services, and all related work can take place within one transaction. The Servlet API and Web service's standard do not (yet) offer any such possibility. EAI-type applications also frequently interact with existing EJB-based services, i.e., other session beans or possibly directly with entity beans, and using fully EJB here definitely makes sense.

Concluding briefly, the first (distributed components) and the last (transaction context propagation) issues are often the decisive ones in favor of EJB if an application requires such.

For most other systems, most notably the common enterprise information systems (EIS) and productivity applications with mainly a Web-centric presentation tier and straightforward transactional requirements, a Servlet-only based architecture is generally simpler and makes more sense.[8]

[8] The presentation and backend/business logic tiers can and should still be clearly separated in the class design when the business service using JDO is co-located in the same VM as the Servlet tier, instead of in a remote EJB tier. Such an approach is shown, e.g., by the Sun Java Blue Prints Adventure Builder 1.0 (Early Access Release at the time of writing), albeit with direct JDBC instead of JDO, but the principle is the same.

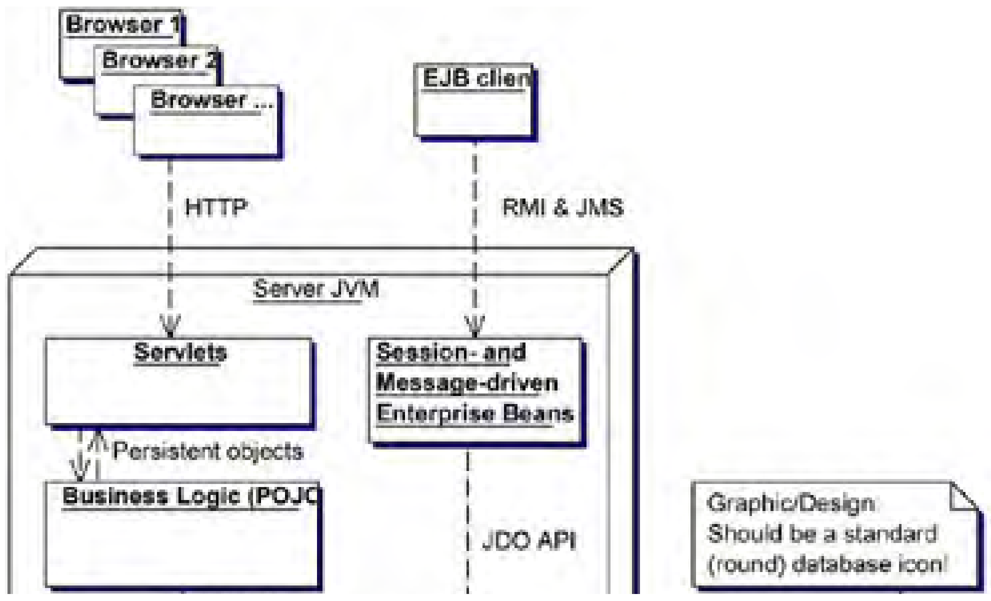
If, while going through the above bullet points, you conclude "mixed" results for your application, a "mixed" approach may well be what you need: EJB session beans for remote invocation via RMI, plus a few thin and simple Servlets for HTTP-based access. In this scenario, the Servlets may use the EJB 2.0 local interfaces to forward requests to co-located session beans, or both session beans and Servlets could possibly access Plain Old Java components.

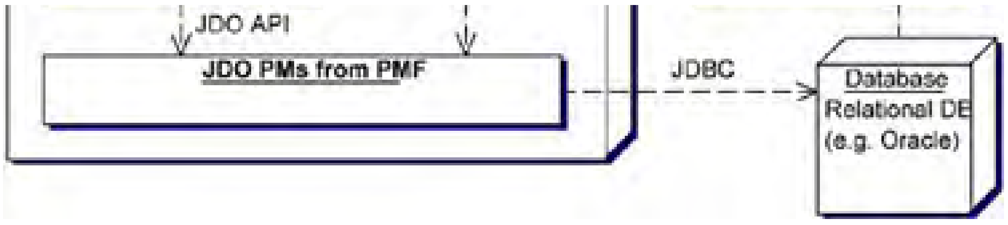
Last but not least, a point must be made that this discussion centered specifically around the usage of the EJB session and message-driven beans and their APIs, and not J2EE application servers in general. Indeed, the J2EE universe and a compliant application server offers many other APIs such as JMS, Servlets, JSP, JCA, JDBC, JNDI, JavaMail, JTA with a JTS-compliant transaction manager for possibly distributed transactions, JMX, and so on. Modern J2EE application servers often also provide other services such as connection pooling and monitoring, hot deployment, clustering, cluster-related distribution and update features, security and user administration, generic monitoring, and management features. All of these other APIs and features are also available to and integrate with JDO-based enterprise applications on a server that chooses not to use the EJB component model.

9.5.1 Scalable JDO-based applications with clustering

If scalability requirements are the only reason to develop an application based on the EJB component model, then a second look at alternatives is time well spent. Figure 9-4 is an illustration of a typical non-clustered scenario, which will be extended next.

Figure 9-4. Simple system architecture with one server and no clustering.





Scalability in this context usually refers to clustering mechanisms where requests can be redirected by software (or a hardware load-balancer) to different physical Java virtual machines, often on physically different hardware boxes, depending on load and availability of servers. A clustered-system architecture is a goal generally for one or both of the following reasons:

- High availability, a way to ensure "fail-over" to a standby server.
- Load-balancing, distributing incoming requests for improved scalability.

Such features are often provided by J2EE application servers and are thus far unrelated to JDO. Enter JDO into the picture, and with the extensive caching that JDO implementations usually perform, a minor problem for clustering architectures appears: If nodes in a cluster are not aware of each other, and if each keeps an in-memory cache of persistent data, write access to a persistent object on one node quickly renders cached instances on other nodes out of date and thus invalid. Furthermore, concurrent modification of identical instances on different nodes arise as a related issue.

Two basic approaches exist to counter the issue and make clustering possible:

- Turn off caching altogether, and rely on datastore transactions only, with no JDO optimistic or locally optimized transactions. This resolves the problem, but implies a performance penalty that would largely outweigh the basic advantage that clustering tried to achieve in the first place. This configuration would, in a way, use only the underlying native datastore as cache instead of JDO.
- Use a JDO implementation that provides a distributed PersistenceManagerFactory-level cache synchronization feature.

Implementations of such cache synchronization features vary widely between JDO implementations, for those that offer such a feature at all, and include anything from simple broadcast UDP messages, to IP multicast-based solution, to a JMS-based one, or an HTTP-based one, possibly using SOAP or even RMI.

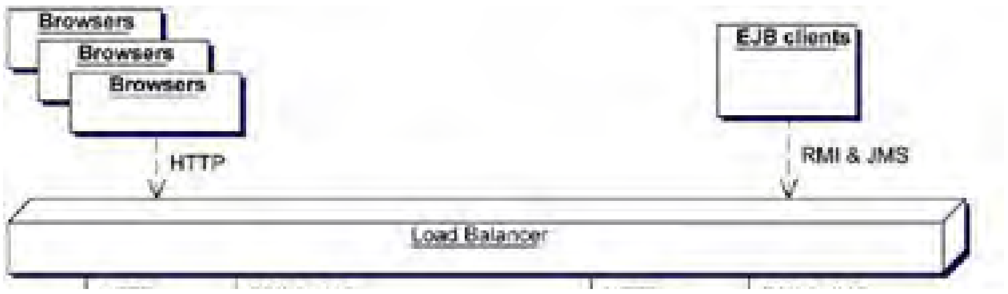
This feature is not standardized in the JDO 1.0 specification, but is available from several JDO implementations. This type of feature is generally regarded as an implementation differentiator between vendors, and may not need standardization under the specification. Relevant implementations naturally do not interoperate. Configuration of a particular feature is highly dependent on the respective JDO implementation.

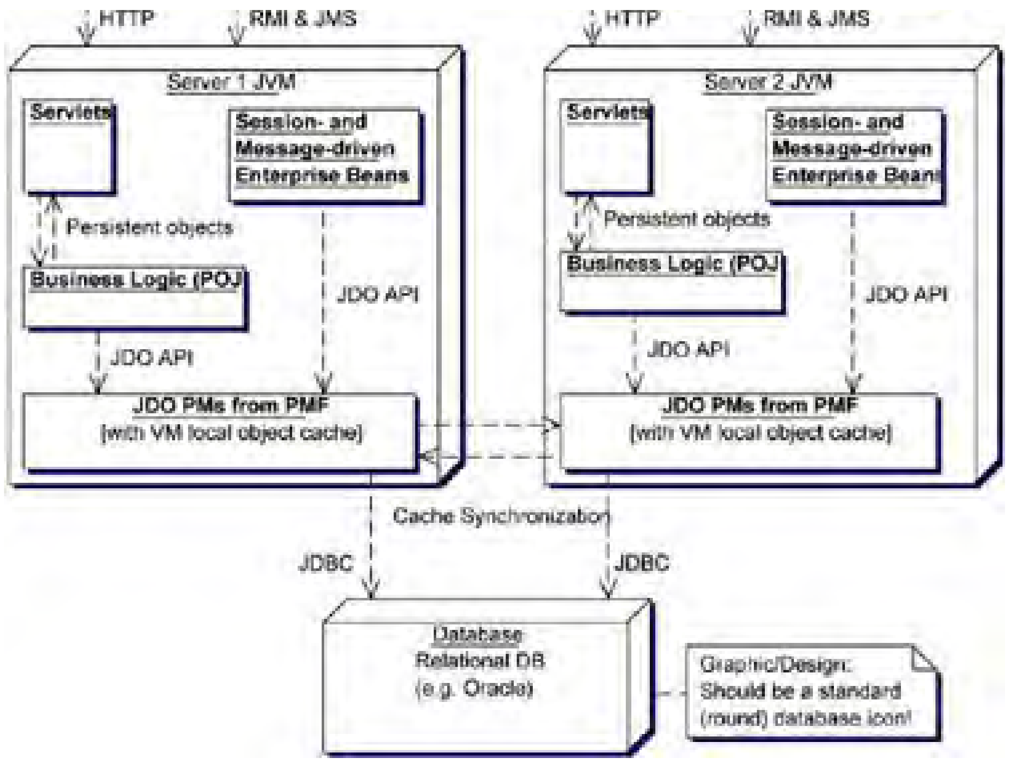
However, such a clustered deployment configuration should be completely transparent to application logic, and using an implementation-specific cache synchronization feature may not be a major portability issue, because changing to another JDO implementation (ideally) only changes the runtime deployment configuration, but no code.

In fact, the architecture used here is one of application-level clustering, as opposed to individual object-level clustering: The objects representing business entities, the business logic, caches, and the container are all co-located within one VM. Each node in such a cluster persists data directly to the underlying datastore. The nodes then use some sort of distributed caching coherency strategy akin to classical data replication among themselves. This architecture can perform particularly well for read-often, change-rarely data scenarios.

Note that clustering using this approach can be applied both when using JDO from within EJB and when using JDO directly from, for example, the Web tier. Figure 9-5 visualizes such an architecture.

Figure 9-5. Cluster with two server nodes running JDO implementations with cache synchronization on one database server. Note how the JDO implementations on both servers directly access the datastore.




HTTP
RMI & JMS
Server 1 JVM
Servlets
Session- and Message-driven Enterprise Beans
Persistent objects
Business Logic (POJ
JDO API
JDO API
JDO PMs from PMF [with VM local object cache]
HTTP
RMI & JMS
Server 2 JVM
Servlets
Session- and Message-driven Enterprise Beans
Persistent objects
Business Logic (POJ
JDO API
JDO API
JDO PMs from PMF [with VM local object cache]
Cache Synchronization
JDBC
JDBC
Database
Relational DB (e.g. Oracle)
Graphic/Design: Should be a standard (round) database icon!

9.5.2 Round-tripping approaches

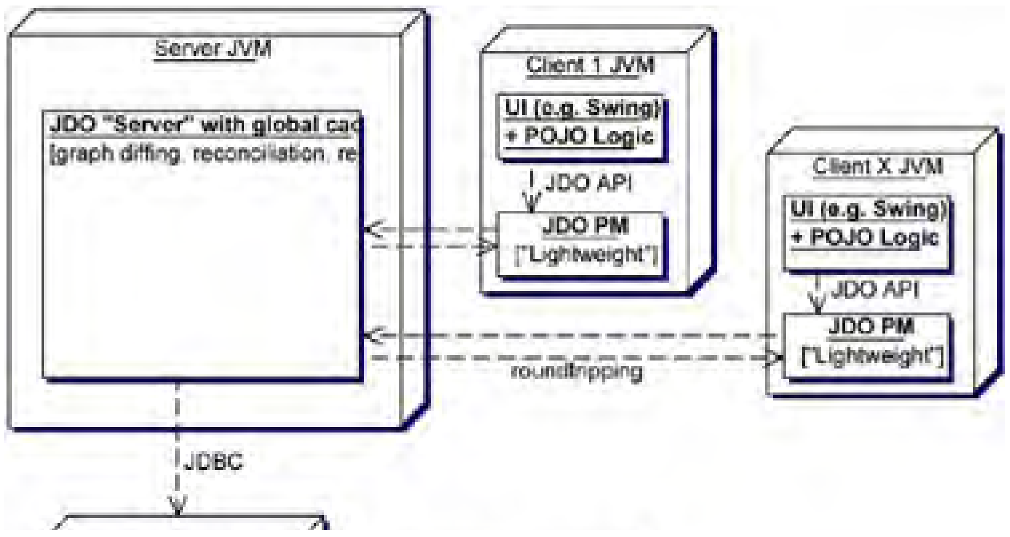
A slightly different approach to the one above is what is often known as "round-tripping" in its various forms. The basic idea is to provide a more or less lightweight client-side JDO PersistenceManager without access to the underlying datastore.

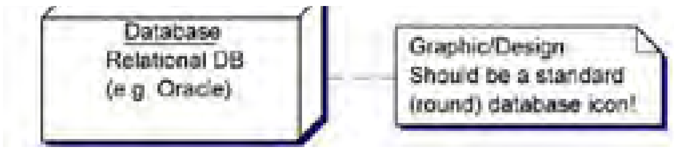
Such a local PM can transparently request copies of objects from a master server, e.g., via a generic service with getAllObjects(Query q, String[] fields) and updateObjects(Object[] changedObjects) sort of methods, but provided by the JDO implementation and transparent to the application.

Issues such as object graph diffing and instance reconciliation, relationship graph management, and instance insertion ordering have to be addressed by such implementations.

In a less transparent and more explicit variation, an EJB session bean could send objects and partial object graphs that are copied to the remote machine, manipulated there, and then sent back for resynchronization with the original persistent objects. Such features are not standardized in the JDO specification, but are available in different shapes in some JDO implementations. Figure 9-6 visualizes such an architecture.

Figure 9-6. Architecture of "Distributed JDO" with a round-tripping approach. Note how clients interact only with a server that accesses the datastore.


Server JVM
JDO "Server" with global cac
[graph diffing, reconciliation, re
Client 1 JVM
UI (e.g. Swing) + POJO Logic
JDO API
JDO PM ["Lightweight"]
Client X JVM
UI (e.g. Swing) + POJO Logic
JDO API
JDO PM ["Lightweight"]
roundtripping
JDBC



The architecture of such round-tripping approaches (sometimes referred to as "Distributed JDO") is generally more in line with optimistic/long-running transactions. It could typically be well suited for Swing-based fat GUI clients, for example, if security and code distributions implications are not a concern. Such approaches are not service-oriented in the traditional sense. Scalability of architectures based on such features should be evaluated in more detail before embarking fully on them.

9.6 Summary

This chapter looked at the EJB component model and its relationship to JDO. Various implementation topics were addressed with extensive source code examples. Higher-level conceptual and architectural issues were also discussed.

In summary, if your application requires distributed and transactional persistent components, use EJB entity beans. In this case, if you are happy with the CMP entity bean implementations that your container provides, JDO is not for you. If you are not, but your design really does require distributed and transactional persistent objects, BMP entity beans implemented using JDO may be the answer. For the rest of us, and that's the vast majority, JDO provides a better alternative over EJB entity beans.

If you are looking at implementing a service-oriented architecture accessible from remote EJB or CORBA fat clients, then EJB session beans are a great way to go. This architecture requires the passing of data transfer objects between tiers, as always, but JDO can easily be used to retrieve and update persistent state. The chapter presented the practical details and options in such an approach.

If message-oriented features are of interest, the message-driven EJB component type can possibly be helpful. This chapter presented the integration of JDO in this sort of bean as well.

However, if you are looking at implementing the average Web-based enterprise application and would like a simple yet effective architecture, using JDO in plain Java objects that are VM local to the Servlet tier is a perfectly acceptable architecture. If Web-service support is of interest, this can be supported as well, without EJB.

As we have seen, even without the use of EJB, a reasonable scalability can be achieved by using a clustered-system architecture with a distributed cache synchronization with a JDO implementation that supports such an approach.

Chapter 10. Security

"The ultimate security is your understanding of reality."

—H. Stanley Judd

Complex applications bring a collection of security requirements to today's data management. These requirements begin at application development time to protect different layers in the software architecture against each other and end at deployment time when production systems have to be protected against malicious access or data corruption. The first part of this chapter defines those different levels of security, their requirements, and their objectives. A controversial part of JDO, the so-called *reference enhancer*, is discussed under security aspects. The rules and measures defined by JDO that keep up the Java sandbox security and secure an application against foreign or malicious code are also examined. The last part of this chapter is about application-level security and how J2EE and JDO work together regarding security.

10.1 Security Levels

Today's database applications become increasingly complex, which is a reason why JDO has been introduced to make their development simpler. On the other hand, this complexity demands a higher security level, because of higher data volumes, more users, and more complex data models. When applications provide their services and data over the Internet, a multi-tier system can process or exchange some one hundred megabytes per second. Special precautions must be made to protect sensitive data. By clever planning and partitioning into application layers, the protection overhead against reluctant access or data corruption can be reduced. The following paragraphs define the different levels or domains of security and their objectives.

10.1.1 Field- and class-level security

The Java programming language already provides some security properties, so code can be written to encapsulate data and deny foreign code access to inner data structures of classes. By using the well-known keywords "private" and "public," field, method and class access can be extended or limited. Unfortunately, there is no difference between "reading" and "modifying," so programmers often implement set...() and get...() methods to restrict field access. The C++ const qualifier is also missing.

The objective of field and method access restrictions is primarily to make code more structured, not to protect data access.

An example for this security domain can be found in the collections framework, java.util.Collection, and friends. Modification or deletion from collections should be prevented or at least detected while another part of the application still iterates the collection. Some collection class implementations throw a ConcurrentModificationException when this case happens. If the framework developers had allowed access to next and previous references of a list, such detection could not be implemented.

On the other hand, the JVM can provide real security by code verification and different ClassLoader instances. This kind, the so-called sandbox security, protects non-public fields and methods against foreign code. Compared to C++, a cast of some reference cannot be used for direct memory access. An exception is the Reflection API, which not only allows field access, but also provides metadata about classes. Since Java 2, the Reflection API is protected by a security policy and must be enabled for code bases that need metadata access. Special considerations must be taken to disable extension of packages by foreign code, which would withdraw package-level security. By "sealing" and signing a JAR file, the simple implementation of another class in the same package can be refused.

10.1.2 Datastore-level security

The restriction of database operations on columns, rows, tables, or even complete databases is covered by this layer of security. Database systems often implement their own user administration and group, role, and rights management. Usually, users can be assigned to groups that have a set of rights. For example, an administrator group may create or delete tables, may make backups, and so on, while the employee management may modify salaries or read other sensitive data.

Basically, the following operations are available:

- Read
- Insert
- Modify
- Delete
- Query

These operations can be applied to the following entities:

- Columns
- Rows
- Tables
- Users

- Rights/roles
- Databases

Under certain circumstances, the rights management can lead to a particular database design—for instance, if there are no column-level access rights definable. The test of access rules can have strong performance effects and may also lead to special database designs.

Using cryptography can be necessary either to protect the data against unauthorized copying (fixed disks and backups), or it can protect data that is sent via public networks.

10.1.3 Application-level security

Implementing application-level security can create a complex, flexible and powerful access management, which can hardly be realized by Java or database-level security. A good design is important and must be able to prevent inadvertently implemented security holes. In the past years, many Web sites were found that let intruders execute SQL code directly from outside. Some developer had overseen a niche where the JDBC user/password pair could be somehow tracked. Often, this security hole is introduced by String operations in the code instead of a reasonable encapsulation—for instance, when input parameters are put into SQL queries.

Essentially, JDO provides an outstanding option to implement application-specific security checks by transparent persistence. A user/group/rights management or access control list can be implemented in short time.

Role-based security is also provided by the J2EE authentication concept and access restrictions can be declared for EJB interfaces on a method-level. How JDO and J2EE work together in a secure context is explained later.

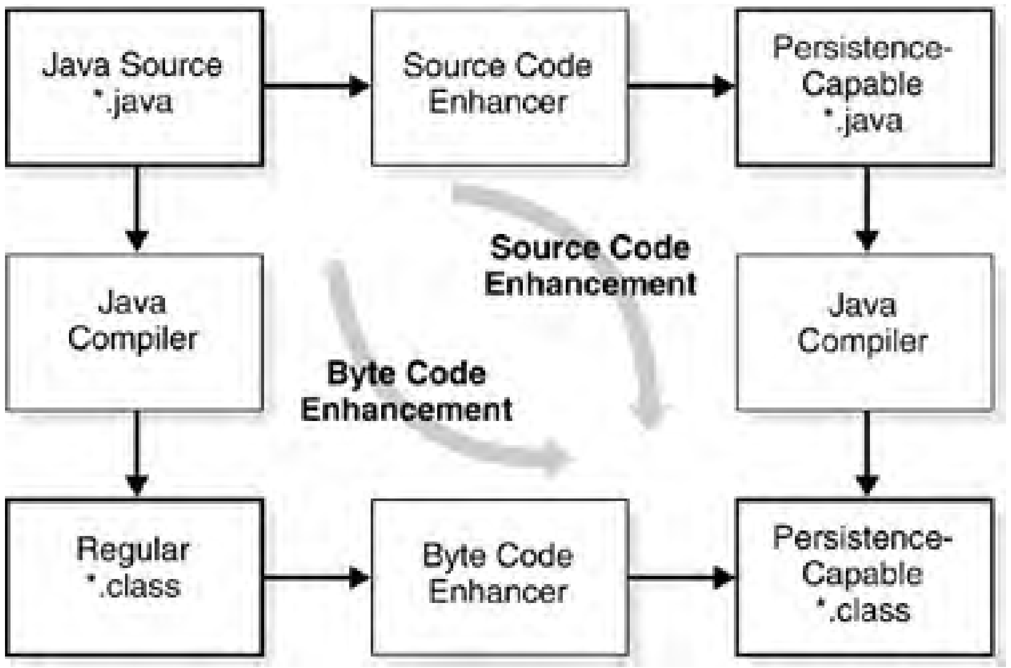
10.2 Implementing PersistenceCapable

To make transparent persistence possible, the JDO specification defines the contract between PersistenceCapable, the interface that all persistent classes have to implement, and PersistenceManager, the central interface of a JDO implementation. But don't worry, how a JDO implementation works inside is not explained in this chapter. Because JDO gets direct access privileges for fields and classes, the contract is examined under security aspects. On the one hand, the PersistenceManager must be able to read and write fields and gather information about persistence-capable classes, like field types. This is deliberately not implemented using Java Reflection API, but by a more direct interfacing to get highest performance. On the other hand, it should not be possible for anybody to read or modify persistence-capable instances. Restrictions such as "private" and so on must be taken into consideration.

The code necessary for persistence-capable classes may be added to the Java source code by tools before a class is compiled or after compilation by modifying byte code. It is a misunderstanding that JDO implementations are required to modify byte code. In essence, the JDO expert group has focused high portability and support of various kinds of possible JDO implementations.

Using source code enhancement, as illustrated in Figure 10-1: Source code and byte code enhancement, the Java source files are processed like a C-preprocessor on a text basis. To implement such a source code enhancer, a full Java parser is required because field access and reserved word must be recognized. A simple line-wise processing is not sufficient. This procedure has the advantage that one can look at the resulting source code and debugging is easy, but today's IDEs do not really support source-code preprocessing well enough. It's like early Java Server Page (JSP) development: Without the right hooks in the development environment, edit-compile-debug cycles become a pain.

Figure 10-1. Source code and byte code enhancement.



Byte code enhancement uses the Java compiler's output and processes *.class files. This standardized format, which all Java compilers have to create and which can be interpreted by all Java virtual machines, is modifiable much simpler than source code. It is even possible to enhance the byte code at class load time of the application, like JSPs are compiled at Web page load time by the JSP engine.

A third approach creates persistence-capable classes completely from other sources than Java code. This might be from a database table layout or from XML schema declarations (XSD). The automatically created source code can be persistence-capable inherently.

10.2.1 The reference enhancer

The reference enhancer has been an idea to provide an umbrella for different technologies that existed before 1999. Essentially, the API that was needed to read, modify and create persistence-capable instances could have been vendor-specific, but JDOs were not accepted by a broad developer community in that case. During the first phase of the JDO specification, it was determined that differences in persistence technologies were insignificant and that the reference implementation could also provide a reference enhancer. By specifying the PersistenceCapable interface and the contract between persistence-capable classes and the PersistenceManager, the JDO specification achieves the goals outlined in the following sections.

10.2.1.1 Binary compatibility

This enables classes to be used by different JDO implementations, after they are enhanced. Persistence-capable applications can change the JDO persistence service without re-compilation or re-enhancement. The JDO compatibility test kit (TCK) takes this into account: Some special tests explicitly check those modifications, which make a class persistence-capable, and compare the byte code with reference classes. A JDO implementation can additionally extend the classes, which allows vendors to tune performance for a specific database system or add other vendor-specific features. The reference enhancer is the smallest common denominator.

10.2.1.2 Simplicity

The class must be enhanced by applying minimal code changes. The algorithms defined to make code persistence-capable or persistence-aware are restricted to exchanging only single opcode[1] in the byte code or simply adding a method to a class. No complex flow-analysis or structural changes should be needed to make a class persistence-capable.

[1] Java byte code is made of different sections for strings, constants, methods or debug information. In a method's section of a class, instructions start commonly with a single byte instruction or opcode, therefore the name "byte code."

10.2.1.3 No reflection permission

Reflection is needed for JDO implementations only to instantiate an application class. This is done by calling `ClassLoader.getClass(String name)`. All other meta-information about the classes' fields, key classes, and so on are provided by a separate interface that is inaccessible other than by the JDO implementation classes.

10.2.1.4 No set/get methods or persistent base class

The application code is not required to implement any methods to become persistence-capable. The idea is to tag a class as persistent, and the rest is done like Java serialization. A persistence-capable class is also not required to derive from a special, persistent class or interface. The inheritance hierarchy doesn't have to be changed.

10.2.1.5 Field navigation by natural Java syntax

JDO defines names for set and get methods and the rules to convert field access in applications to method invocation. To make a class persistence-aware, field access is changed automatically from this:

```
String title = book.title;
```

to this with help from a tool:

```
String title = book.getTitle();
```

10.2.1.6 Automatic tracking

The application is no longer responsible for detecting modified data and updating the data in a datastore or for propagating state changes to the JDO implementation. This is done in JDO `set()`-methods, which are added automatically at enhancement. These methods contain the required code to detect modifications, set flags, and propagate state changes.

10.2.1.7 Support of Java field attributes

The application class may use regular field modifiers, such as `private`, `public`, or even `transient` for any persistent fields. In XML, the metadata default attribute for fields may be overridden.

10.2.1.8 High performance

Compared to other persistence technologies, which may use Java Reflection to read or modify persistent instances, JDO provides great performance. As long as an instance is transient or after valid data is loaded from the datastore, it performs like any other Java instance. A factory inside of the persistence-capable class is used to create new (hollow) instances without the need for Java Reflection.

10.2.1.9 Same classes for different JDO implementations

The persistence-capable application code can be used together with different PersistenceManager instances at the same time (not the same persistent instances, but the code).

10.2.1.10 Debugger support

A JDO enhancer implementation is required to modify the byte code in such a way that any Java debugger can still be used to debug the application code.

10.2.2 Principles of operation

What has all this to do with security? The main problem is allowing a JDO implementation to access private fields, while protecting other code from doing the same. That is the reason for an essential principle in the contract between the persistence-capable class and the PersistenceManager defined in the JDO specification: "double dispatch." The next paragraphs shed some light into the implementation details of this contract.

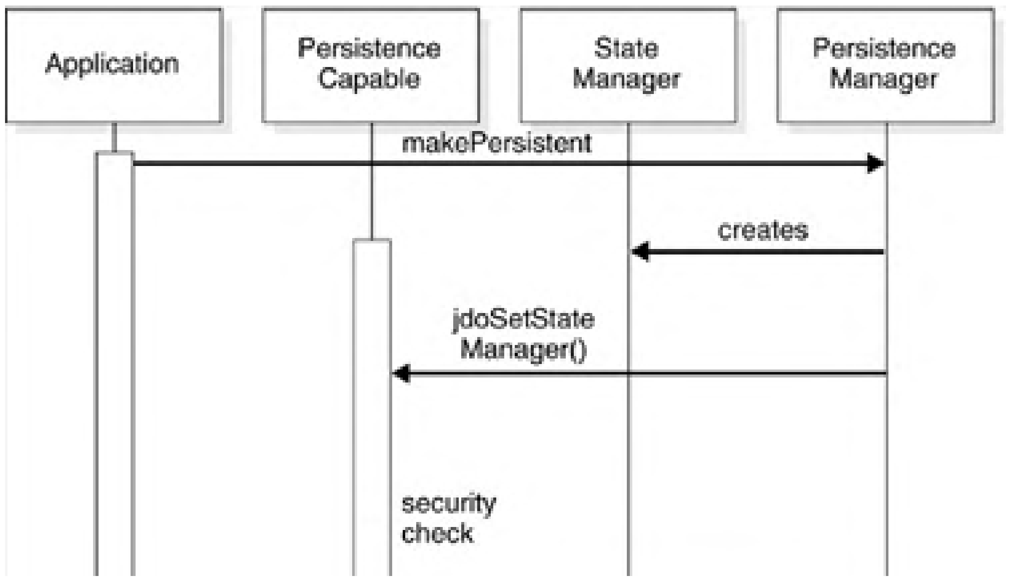
10.2.2.1 Double dispatch

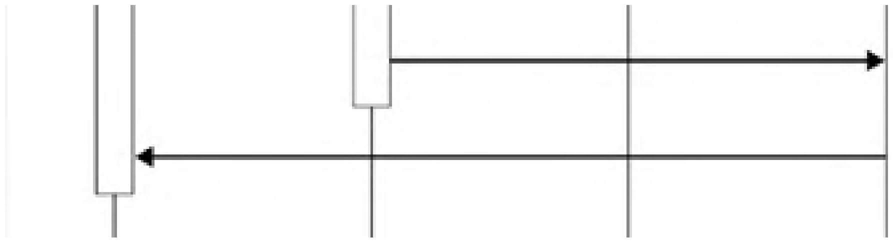
Every persistence-capable class has a StateManager reference, which is the mediator between itself and the corresponding PersistenceManager. The StateManager is a JDO service provider interface used for dispatching. Principally, the StateManager's methods could have been implemented in the PersistenceManager or in the PersistenceCapable interface.

10.2.2.2 Making instances persistent

At the moment that a transient instance becomes persistent by a call to pm.makePersistent(pc), the JDO implementation first must get the right to access persistent instances to read out fields and store the data persistently. The transactional behavior of the persistent instance must be controlled by the JDO implementation as well, because the previous state of the instance must be restored at transaction rollback. At this point, the StateManager of the persistent instance is set, and control is granted to the JDO implementation. Just at this point, a security check is made through Java's regular security API, to check the JDO implementation's access permissions. If the JDO's code base is allowed to set the StateManager reference at the persistent instance, it also gets access to all the rest, as illustrated in Figure 10-2: Setting the StateManager. This ensures that only authorized implementations can use the StateManager interface.

Figure 10-2. Setting the StateManager.





The JDO implementation checks the initial setting of the StateManager through JDOPermission("setStateManager").

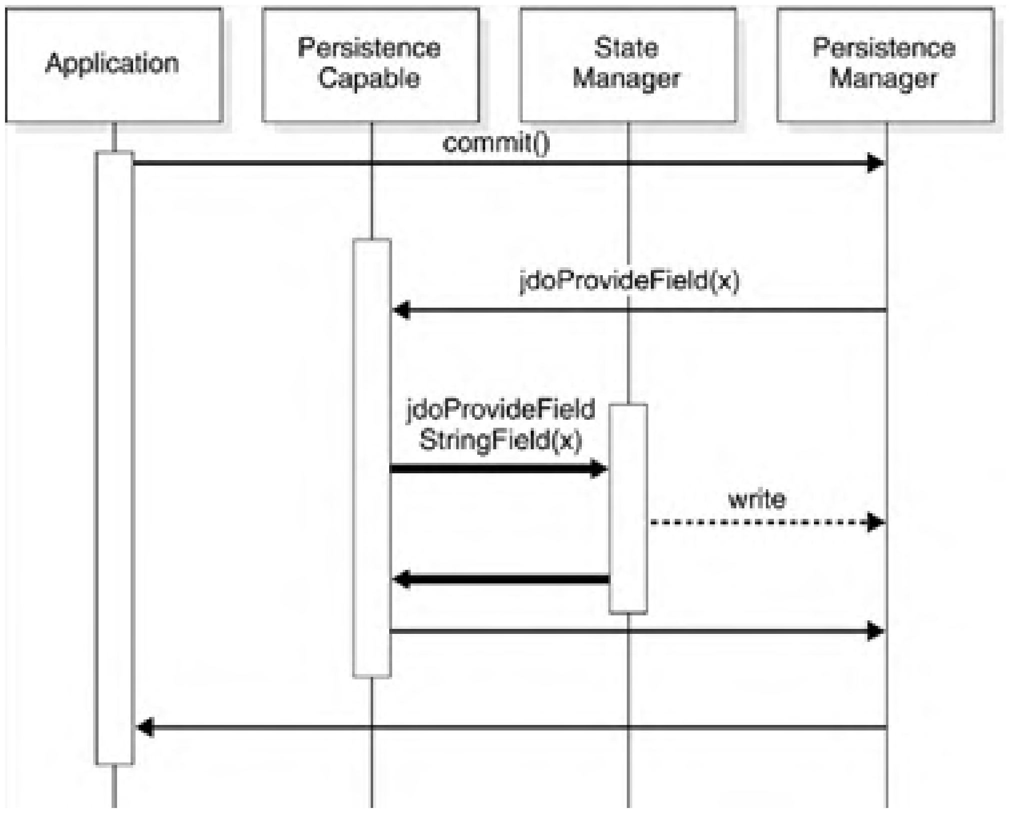
The implementation of the StateManager interface is not defined by JDO. In principle, there can be one StateManager instance per persistence-capable instance, one instance for each persistence-capable class, or one instance per PersistenceManager instance.

10.2.2.3 Reading fields from persistent instances

When the instance is made persistent, the PersistenceManager may need to copy fields from the instance into some internal data cache or into the storage. This may happen to save the field data for later restoration in case of rollback or at commit time. The PersistenceManager is not allowed to read directly, because any other class could do the same. Instead, it may trigger the operation, and field data is provided via the StateManager interface.

As indicated in Figure 10-3: Reading fields, the PersistenceManager first calls the PersistenceCapable jdoProvideFields() method to trigger the read-out by the StateManager, which then receives the data from the persistence-capable instance. The idea is based on the protection of the StateManager: If foreign code cannot set the StateManager reference of the persistence-capable instance, it cannot access fields. The costly Java security check is necessary only once, when a transient instance becomes persistent or when a hollow instance is created.

Figure 10-3. Reading fields.



The StateManager interface provides all methods to get or provide data from or to the persistent instance. Some other methods are called for state changes and modification tracking, which are covered in the next section. The interface might support other applications as well, for instance the Java XML Data Binding (JAXB), which also reads and writes data from/to Java objects (called *marshaling* and *unmarshaling*).

10.2.3 Tracking field access

Another important aspect is tracking field modifications and initial access of hollow instances. Resolving hollow instances on demand and modification tracking makes database application development with JDO extremely simple. It can be a horror to find a missing line of code in a huge application, which causes some data to not be written back into the datastore, or to track down performance problems because data is written too often.

Because many developers are afraid of byte-code post-processing, the actual code modifications to accomplish persistence awareness are explained here. First, any field access must be converted into a method invocation. As mentioned earlier, the basic idea is to change automatically a line like the following:

```
String title = book.title;
```

into this by a tool:

```
String title = book.getTitle();
```

Add a method getTitle() that reads the corresponding data from the datastore on demand, and the state changes from hollow to persistent-clean in this case. To make persistence-aware classes compatible between different JDO implementations, the name is defined by JDO and the method signature is not the same:

```
String title = Book.jdoGettitle(book);
```

The rule is to add the prefix jdoGet, add the field name, and declare it as a static method with the same access modifiers in the class. The set methods are similar:

```
book.title = "new title";
```

In a persistence-aware class becomes:

```
Book.jdoSettitle(book,"new title");
```

To make things clearer, the code can be put into a small method, containing just the above lines. The code can be "disassembled" by the javap tool after it has been processed by the enhancer. Before enhancement is applied, the field assignment disassembles to the following:

```
0 aload_0
1 getfield #2 <Field Book book>
4 ldc #3 <String "new title">
6 putfield #4 <Field java.lang.String title>
9 return
```

This is the same modification method after the enhancer modified the code to look like this:

```
0 aload_0
1 getfield #2 <Field Book book>
4 ldc #3 <String "new title">
6 invokestatic #5 <Method void
          jdoSettitle(Book,java.lang.String)>
9 return
```

The mutation of the code does not change its length. Both return statements are at position 9, and debugging information will not change because mapping of code addresses and source lines is not altered. During access of persistent fields, the timing is changed a bit. So what happens inside those set and get methods? The JDO specification suggests a default implementation, but it is not mandatory. The goal is to run as fast as possible in case of transient, persistent-clean, and persistent-dirty for fields in the default fetch group.

Default Fetch Group (DFG)

A fetch group declares a number of fields retrieved from a datastore together. Often, data is organized in sections, clusters, or tables, and values can be fetched in a group more efficiently. JDO provides two different types: fields in the default fetch group and fields in other fetch groups. The advantage of the DFG is primarily a flag field that is checked at field access. If that flag is set, the field value is returned directly without calling the JDO implementation. It is not specified how these other groups are declared in the XML metadata, because it is strongly implementation-specific. For example, the Book class might contain a reference to some complex class that has subclasses (**Category, CategoryList**, and so on). The retrieval of that reference can become expensive in a relational database, because a join query is needed. By declaring the reference in another fetch group, retrieval is delayed until the reference is directly requested by the application.

If a persistent instance needs to be filled with data from a datastore, the time necessary to fill the instance is much lengthier than the code in the methods. The proposed jdoGettitle() method might be implemented like this:

```
final static String
jdoGettitle(Book b) {

  if (book.jdoFlags <= READ_WRITE_OK) {
    return book.title;
  }

  StateManager sm = book.jdoStateManager;

  if (sm != null) {

    // hollow ?
    if (sm.isLoaded(book,jdoInheritedFieldCount+1))
      return book.title;

    return sm.getStringField(book,
              jdoInheritedFieldCount+1,book.title);
  } else {

    // transient
    return book.title;
  }
}
```

First, the method checks jdoFlags for fields in the default fetch group. If the instance is already filled with valid data from the datastore, the instance returns the field. If the StateManager reference was not set yet, the instance is transient and the field can be returned (else part). If a StateManager is set, the field might be read and returned, or the field is completely managed by the JDO implementation and access is always mediated. Here is a summary:

1. The instance is "clean" or "transient."
2. There is no StateManager.
3. The instance is hollow, or the field has not yet been read. Access causes the field to be read.
4. Field data is kept in some "shadow" memory, and access is always mediated.

The jdoSettitle() method works similarly and can be implemented in the following way:

```
final static void
jdoSettitle(Book b, String value) {

  if (book.jdoFlags == READ_WRITE_OK) {
    book.title = value;
    return;
  }

  StateManager sm = book.jdoStateManager;

  if (sm != null) {
    sm.setStringField(book,
              jdoInheritedFieldCount+1,
              book.title,
              value);
  } else {
    // transient
    book.title = value;
  }
}
```

Again, the set method checks jdoFlags and simply returns whether the instance is persistent-dirty already. In this case, the title field can be set directly. If a StateManager is set, the appropriate method is called, or else the instance is transient and the field can just be set.

The sequence of calls between PersistenceCapable and StateManager allows for flexibility for various kinds of implementations. One extreme can have all fields in the default fetch group. Reading a field causes all other fields to be filled with valid data. Any further read access simply returns the corresponding field. The other extreme delegates every access to its JDO implementation. O/R mapping tools probably support a flexible configuration somewhere amid, depending on the mapping of fields to tables. An object-oriented database might always read and write entire objects.

10.2.4 Metadata access

The JDOImplHelper class helps to gather information about persistence-capable classes. It is another service class and is not intended for application use. At the point when a class is initially loaded by the JVM, it registers itself in JDOImplHelper. Later, JDO implementations can query the JDOImplHelper singleton for persistent classes and their field attributes. Again, this indirection is needed because of security issues.

The JDOImplHelper.getInstance() method checks for JDOPermission("getMetadata") rights and executes only the code of JDO implementations that have been granted access. If this check was not made, anyone can get information about field names or object identity classes that can be compared to RuntimePermission("accessDeclaredMembers").

10.2.5 Policy file

The JDO implementation must be granted privileges to get access to the PersistenceCapable.setStateManager() and JDOImplHelper.getInstance() methods by adding the following lines to a policy file:

```
grant codeBase "file:/home/myJDOImpl" {
  permission javax.jdo.spi.JDOPermission
       "getMetadata";
  permission javax.jdo.spi.JDOPermission
       "setStateManager";
};
```

10.2.6 Security problems

Unfortunately, the PersistenceCapable interface opens up some security gaps. Because any persistence-capable instance is able to return its PersistenceManager by the following:

```
PersistenceManager pm =
     JDOHelper.getPersistenceManager(book);
```

any other application part, even some code that has nothing to do with persistence, can operate on the datastore. Some code may simply delete the instance from the datastore, although it had never been intended. Additionally, other parts of the application can peek at the datastore because Transaction, Extent, Query, and PersistenceManagerFactory are easily reached via the PersistenceManager.

This may sound a bit paranoid. On the other hand, JDO leads developers to use persistent objects throughout the application. Compared to direct JDBC programming, which often leads to home-grown persistence layers, persistence-capable instances pass their database connection around. Maybe such a security gap does not really allow malicious code to be executed, but an API that's too exposed can lead to an application with messed up control paths. To prevent such a mess, making instances transient after transaction completion is recommended. Under some circumstances, it can be an option to pass around JDO object identities instead of persistent instances.

Another security problem can come from JDOQL: JDO allows querying private fields outside of a class or even outside of a package. For example, if the Book class had a private field purchasePrice, a simple binary search could find the price for a book. Taking the reference of a persistent book instance, the search method can get the PersistenceManager, create Extent and Query instances, and run queries. All this has to take place in a running transaction, or a new Persistence-Manager must be instantiated from the PersistenceManagerFactory.

10.3 Application Security

The reasons mentioned in the previous section lead to requirements for a clean application design and, if applicable, to copy instance data or to make instances transient before passing them to other application layers. These are the results of the previous section:

- Access modifiers public, private, and protected are not changed by the JDO reference enhancer. Fields are still not accessible by other classes, although JDO implementations are authorized to do so by security policies.
- The PersistenceManager is reachable from persistent instances. Those instances can be deleted from or inserted into the datastore.

- The other JDO interfaces, Transaction, PersistenceManagerFactory, Extent, and Query can be reached, and other operations can be performed than those available at the persistent instance.
- Queries can be executed with expressions that contain private fields outside of the owning class.

JDO does provide some basic application security, which can be used to prevent some flaws mentioned above.

10.3.1 User/password security

When JDO is used in a non-managed environment, the application can provide the user and password to open a connection to an underlying datastore. The code looks like this:

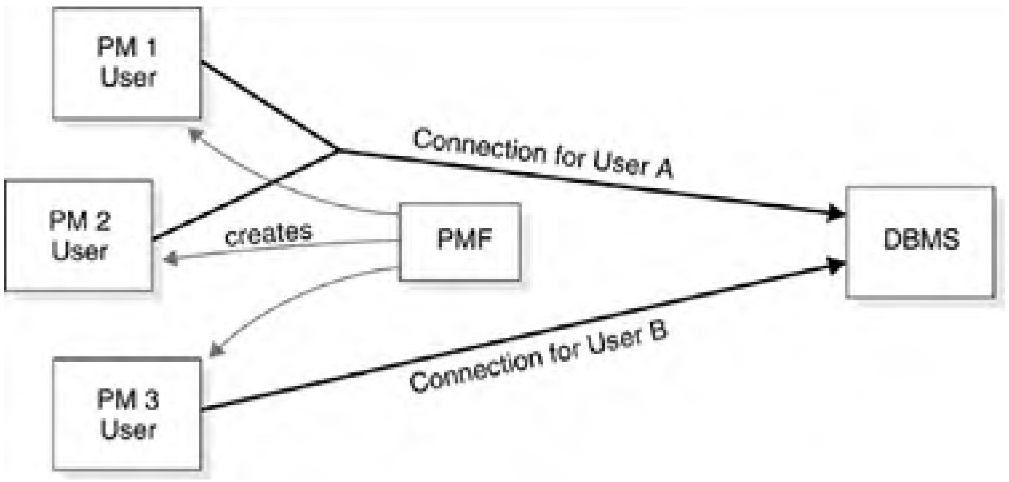
```
Properties props = new Properties();
props.setProperty("javax.jdo.option.ConnectionUserName",
                  username);
props.setProperty("javax.jdo.option.ConnectionPassword",
                  password);
... other properties ...
PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(props);
```

Additionally, the PersistenceManagerFactory can create instances of PersistenceManager with different user contexts. As mentioned in Chapter 5, it is not possible to change the properties of a PMF after a first PersistenceManager has been created.

```
Properties props = new Properties();
... other properties ...
PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(props);
PersistenceManager pm = pmf.getPersistenceManager(
      username, password);
```

In the second variant, JDO implementations must keep track of connection pooling and user/manager mapping. If different PersistenceManager instances are used with the same user identity, the same database connection might be used. As illustrated in Figure 10-4: User context and connections, the PersistenceManagerFactory created three PersistenceManager instances, two of them sharing the same user context.

Figure 10-4. User context and connections.



How this mapping is done is not defined by JDO.

10.3.2 Managed environments

In managed environments, the user or session can be authorized by the container, and PersistenceManagerFactory instances are returned by a JCA resource adapter, as explained in Chapter 9:

```
PersistenceManagerFactory pmf =
   (PersistenceManagerFactory)new InitialContext().
        lookup("java:comp/env/jdo/example")
```

The resource adapter is responsible for providing the security context or getting the security credentials from the container. This means that a session can share its access rights, user information, and so on with different services in the container, like sharing a transaction manager across multiple, distributed transactions. Likewise, security checks can be more restrictive in a managed environment, so that only specific classes are allowed to access the persistence service. This would also prevent code from taking the PersistenceManager of a persistent instance and creating new connections to a datastore, as mentioned before.

10.3.3 Application-specific authorization

Some application types require a different layer of security on top of the data-store's native security layer. For example, to manage Internet chat room users by database user administration would be overkill. In this case, implementing a simple user administration by means of transparent persistence is recommended. Whenever rights, groups, and users should be managed by the application itself, it is advised to do so. It provides datastore independency and keeps the domain object model free of vendor-specific code.

10.4 Summary

From the Java language point of view, JDO does not introduce any security gaps. The PersistenceCapable interface is well designed, optimized for speed, and effectively requires the same security checks as Java Reflection does. The JDO programmer is not forced to implement any public set/get methods or to derive from an abstract persistence class. On the other hand, an application developer needs to keep in mind that a persistent instance lets anyone access the underlying data-store connection, as long as the instance is not made transient. To prevent that, a service-oriented architecture (SOA) is required for security-sensitive applications, as explained in Chapter 9, Section 9.2.4.

From the database point of view, JDO only provides a simple username/password authentication. JDO does not define any other access control list (ACL) management. By defining fetch groups, it can be possible to read only fields of persistent instances, to which the current user is allowed. This part is completely unspecified by JDO.

Lastly, J2EE managed environments (Servlet, EJB) or application-specific access implementations provide a greater flexibility. They can be used to declare access privileges on a method, class, field, or operational basis, but they require a higher implementation effort.

Chapter 11. Transactions

"A mistake is to commit a misunderstanding."

—Bob Dylan

The ability for tasks to be completed predictably is an integral part of any real-world application. The transaction facilities in the middleware or server-side applications are responsible for ensuring this predictability. In this chapter, we begin by examining some basic concepts surrounding transactions and how transactions are used in Java. We then look at the concept of transactions in JDO and the more complicated scenarios resulting from integration into managed environments. (Refer to Chapter 1 for more information on managed environments.)

11.1 Transaction Concepts

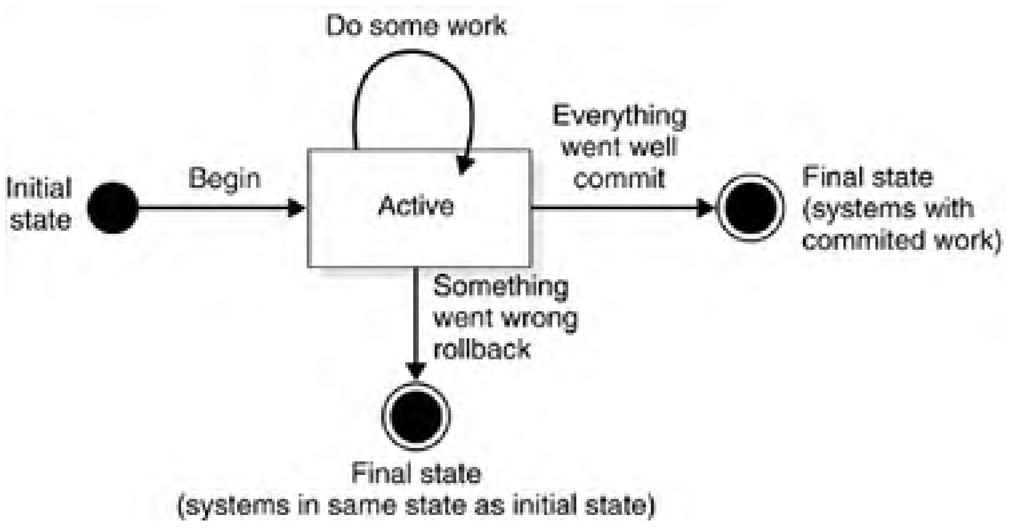
A transaction can be described as a *unit of work* that is comprised of several different operations acting on one or more shared system resources and has a predictable outcome. Additionally, this unit of work needs to be governed by *ACID* properties. The acronym ACID[1] stands for Atomic, Consistent, Isolated, and Durable. These four classical characteristics essentially define the concept of a transaction:

[1] Andreas Reuter, *Transaction Processing: Concepts and Techniques* (ISBN 1-55860-190-2) Morgan Kauffman.

- **Atomic:** The unit of work is atomic in that it is either performed in its entirety or not performed at all. This "all or nothing" properly ensures that in case of failure, changes that occur due to the unit of work are either made permanent (committed) or reversed (rolled back) to restore the original state.
- **Consistent:** Consistency requires that after the execution of the unit of work, the system is in a stable state. If there is any inconsistency, then the work done must be reversed to restore the original state. For example, the referential integrity of the data must be maintained upon completion.
- **Isolated:** The notion of isolation is simple; a transaction should execute as though it is running by itself, meaning that the correctness of the unit of work performed in a transaction should not be compromised when another concurrent transaction accesses the same set of resources.
- **Durable:** The changes made by the unit of work constituting the transaction should be permanent and should not be lost because of system failure.

From a design perspective, a transaction usually represents a level of abstraction in the context of a business operation. A transaction consists of business operations that have application semantics and a scope greater than a method invocation or database operation. For example, suppose that the transaction consists of an account withdrawal and an account deposit; the application code to realize this may consist of multiple databases and classes. However, the overall transaction is still governed by the ACID properties, which require that all the work done in the transaction be completed as one entity, or in the case of failure, no work is done at all, with data integrity being maintained in both cases. This all-or-nothing *flat* transaction model is characterized with the states shown in Figure 11-1. The transaction is begun and goes into an active state where some work is done. If the work was successfully performed, the transaction is committed and the resources accessed in that transaction are updated with the results of that work. If something goes wrong, the transaction is rolled back and the resources are restored to their initial state.

Figure 11-1. State transitions in a transaction.


Do some work
Initial state
Begin
Active
Everything went well commit
Final state (systems with commited work)
Something went wrong rollback
Final state (systems in same state as initial state)

Nested Transaction Model Not Supported

The nested transaction model allowing transactions to be nested inside each other is not supported by J2EE or JDO.

11.1.1 Parties involved

In any transactional system, the following multiple parties work in unison for the model to work smoothly:

- **Transactional object:** This object initiates or demarcates a transaction. It can be any user application or in an application server environment J2EE component like EJBs. In the case of EJBs using container-managed transactions, the application server may initiate and end the transaction; however, it still does that on behalf of the EJB.
- **Transaction manager:** This is the implementation of a transaction service and is responsible for coordinating and managing the transactions.
- **Resource adapter:** This is the system-level software "driver" that application code uses to connect to the underlying resources, e.g., JDBC drivers, JMS providers.
- **Resource manager:** This manages a set of resources. A transactional resource manager can participate in transactions that are externally controlled and coordinated by a transaction manager. A resource manager is typically in a different address space or on a different machine from the resource adapters and clients that accesses it, e.g., a database system, a mainframe transaction processing system, and so on.
- **Resource:** This is the actual underlying resource, e.g., a database instance.

Depending on the shared resources being accessed by the transactional object, the transaction can be characterized as a local or distributed transaction.

11.1.2 Local transactions

A transaction that involves a single resource manager is known as a local transaction. In local transactions, the transaction manager simply delegates management of the transaction to the underlying resource manager. For example, in an application server when a transaction object like a J2EE component interacts with a single database, the transaction manager creates a transaction object, associates it with the current thread, and invokes `begin()`, `commit()`, and `rollback()` on the underlying database connection.

11.1.3 Distributed transactions

A distributed transaction is a transaction that interacts with multiple and possibly heterogeneous resource managers in a coordinated manner. For example, a transaction manager needs to coordinate the transaction involving a database and a JMS queue in the same method execution to ensure that these resources work together coherently.

Distributed Transactions

From a JDO perspective, the concept of distributed transactions is important if the application attempts to use JDO-based persistence alongside application code that uses other persistence mechanisms.

11.1.4 Two-phase commit

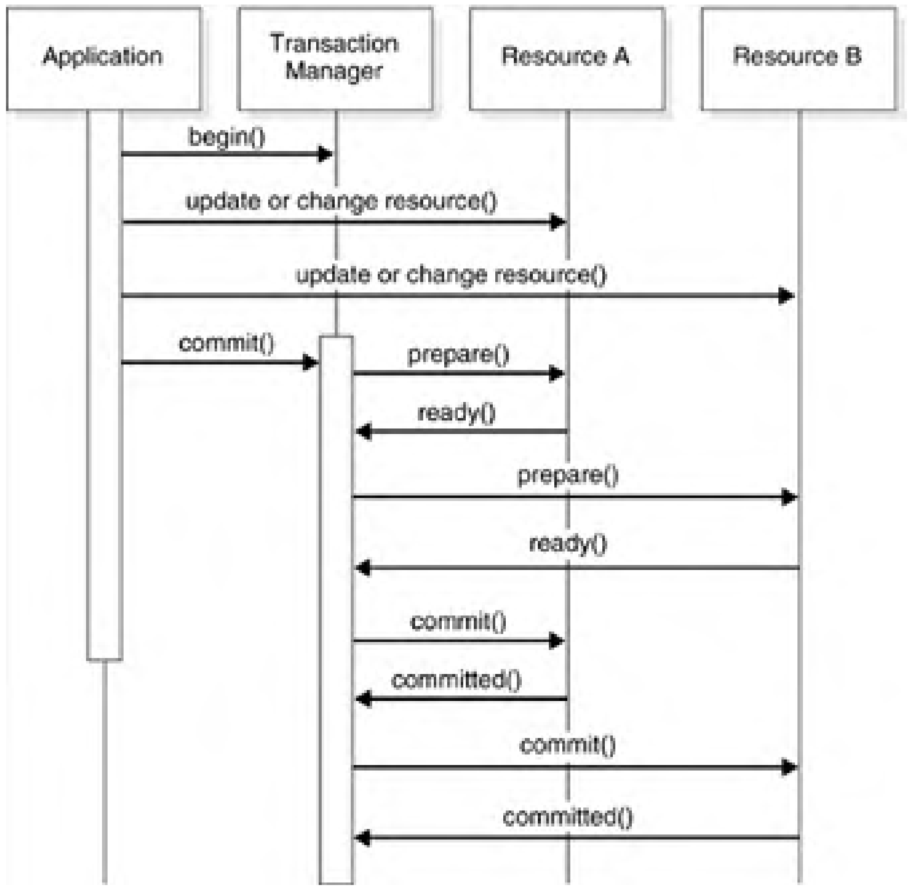
In order to maintain the atomicity of a distributed transaction, the transaction managers use a standard two-level recovery mechanism known as a two-phase commit protocol (2PC). The two-phase commit protocol guarantees data integrity by ensuring that transactional updates are committed in all the participating resources, or are fully rolled back out of all the resources, reverting to the state prior to the start of the transaction. To understand how 2PC works, look at what happens in such a transaction:

1. The resources are enlisted with the transaction manager.
2. The transactional object initiates a transaction.
3. The transaction manager begins the transaction.
4. Application code updates resource A.

5. Application code updates resource B.
6. Application code requests a commit form the transaction manager.
7. The transaction manager executes the two-phase commit.

The transaction manager, as the name suggests, completes the transaction in two phases. A simplified view of the interaction is shown in Figure 11-2.

Figure 11-2. The two-phase commit execution.



Distributed Transaction Standards

The most widely accepted distributed transaction processing model is that specified by the Opengroup consortium in the XA specification. The XA specification defines the API in the form of an interface—the XAInterface that serves as a contract between distributed transactional components. See http://www.opengroup.org/products/publications/catalog/s423.htm.

- **Phase 1:** The transaction manager sends a *prepare-to-commit* request to the resource managers. The resource managers perform internal checks and store information about the requested updates in some durable storage (e.g., such as a disk). When the transaction manager receives acknowledgements from all the resource managers, it stores the information somewhere, such as in a file system.
- **Phase 2:** The transaction manager sends a commit message to all the resource managers. The resource managers perform the requested update on their end and send an acknowledgement of the execution back to the transaction manager. After the transaction manager receives all the acknowledgements, it informs the transactional object about the result of the transaction.

The transaction manager issues a rollback message to the resource managers if either of these conditions is true:

- Any of the resource managers is unable to prepare itself (e.g., locks could not be obtained) and returns a failure message in Phase 1.
- Any of the resource managers is unable to complete the task and returns a failure message in Phase 2.

The two-phase commit protocol in distributed transactions can sometimes give rise to fatal error conditions called *heuristics* that violate the *atomic* properties. A typical case called a *heuristic rollback* is as follows: In Phase 1, the transaction manager signals the resource to prepare itself within the context of a transaction. The acknowledgement from the resources is, in fact, a commitment that later on when the transaction manager signals a commit or rollback, it will be able to do either. However, sometimes it may break this promise—for example, the transaction manager goes away and the resource is waiting in a prepared state tying up other locks. Even though the resource does not know the outcome of the transaction, it makes a preemptive decision, called a *heuristic decision,* based on some internal timer or other administrative command. Typically, this guess is to rollback the transaction, release internal locks, and return itself to a consistent state. If the sub sequently transaction manager returns with a request to do some work (e.g., commit) on the same transaction, the resource throws a heuristic exception.

These are other similar heuristic conditions:

- **Heuristic commit:** A resource preemptively decides to commit its part of a distributed transaction.
- **Heuristic mixed:** Due to one or more heuristic decisions, a transaction has different branches in conflicting states, some committed and some rolled back.

From a developer's perspective, any indication of a heuristic condition is a signal to examine the application deployment and architecture. Typical causes may include a lack of system resources on the database or resource manager and excessive consumption of resources (e.g., locks or memory) by a particular transaction.

11.2 Transactions in Java

Now that we have introduced the concept of a transaction and distributed transactions, let us now look at how transactions are supported and used in existing Java applications.

11.2.1 JDBC transactions

Java applications can include transactional constructs at different levels. An application interacting with a datastore using JDBC can use JDBC level transactions. A JDBC transaction is controlled by the transaction manager of the underlying datastore and is demarcated using the commit and rollback methods of the java.sql.Connection interface. For example:

```
Connection conn =DriverManager.getConnection
                    ("jdbc:oracle:thin:@jdo:Oracle");
conn.setAutoCommit(false);
try{
   Statement stmt = con.createStatement();
   stmt.executeUpdate("UPDATE USERS SET username='corejdo'
                    WHERE ACCOUNTID=1829");
   stmt.executeUpdate("UPDATE ACCOUNTS set ACCOUNTTYPE=09
                    WHERE ACCOUNTID=1829");
   conn.commit();
  }catch(Exception e){
         conn.rollback();
    }
```

JDBC transactions, although simple, are not suitable for controlling transactions that span multiple application components. The connection-based demarcation is local to the component in which it is used, and the transaction context is not propagated. For distributed environments such as J2EE, a different framework is needed that caters to transactional resources in general, not just databases. This is where JTA and JTS come in.

11.2.2 JTA and JTS

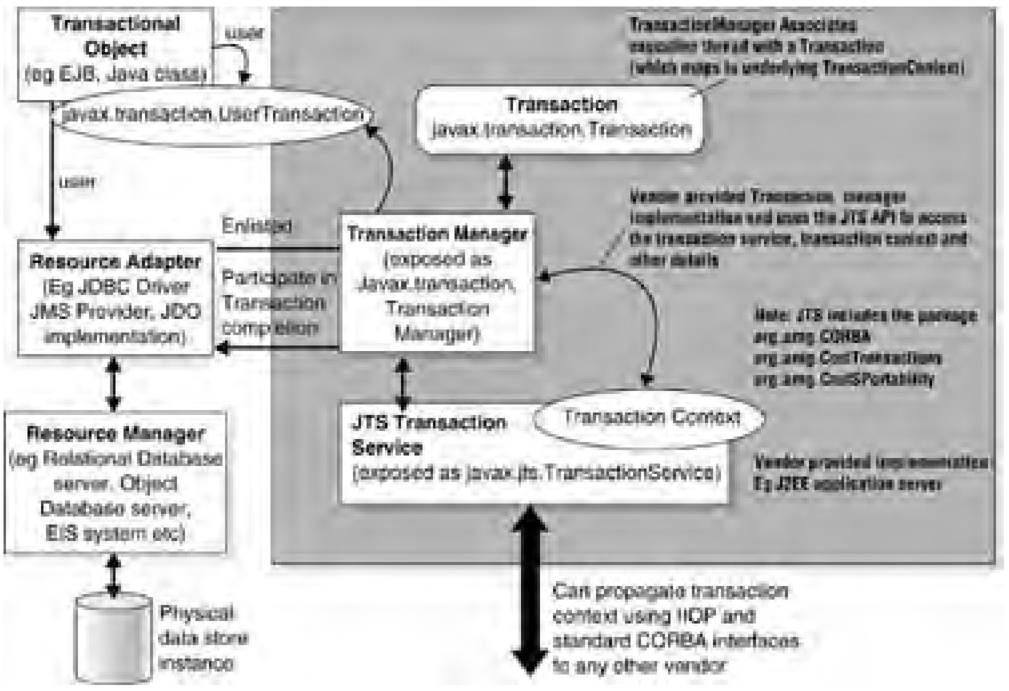
The Java Transaction API (JTA) and Java Transaction Service (JTS) provide the necessary and generic framework for applications to use transactions across multiple components and datastores.

The OMG Object Transaction Service (OTS) specifies how a transaction service can be implemented and the associated IDL interfaces. OTS is the transaction service used in CORBA to provide the transaction-related plumbing. JTS is the Java mapping of the IDL interfaces that defines how the service should be implemented in Java and the functionality that it must expose. Vendors (such as J2EE application server vendors) use and provide the implementation of the JTS in their products; developers never work directly with JTS.

Java Transaction API (JTA) provides the API and the standard Java interface for developers to access the facilities of a transaction service (i.e., the JTS implementation). It provides an interface for the transaction manger, which manages the underlying transaction service. The following list summarizes (and Figure 11-3 illustrates) how JTA and JTS work together:

- Transactional objects like EJBs and Java classes obtain a reference to the instance of the JTA UserTransaction interface and demarcate transactions.
- Vendors provide the JTS implementation and a transaction manager implementation. This is exposed using the JTA TransactionManager interface.
- When a transaction is begun by the transactional object, the transaction manager creates an instance of the transaction object, represented by the JTA Transaction interface, and associates it with the calling thread.
- When the transactional object uses the transactional resource, the resource is enlisted with the transaction manager.

Figure 11-3. Different entities in JTA.


Transactional Object (eg EJB, Java class)
uses
javax.transaction.UserTransaction
uses
Transaction javax.transaction.Transaction
Enlisted
Resource Adapter (Eg JDBC Driver JMS Provider, JDO implementation)
Participate in Transaction completion
Transaction Manager (exposed as Javax.transaction.Transaction Manager)
Resource Manager (eg Relational Database server, Object Database server, EIS system etc)
JTS Transaction Service (exposed as javax.jts.TransactionService)
Transaction Context
Physical data store instance
Can propagate transaction context using IIOP and standard CORBA interfaces to any other vendor
Note: JTS includes the package org.omg.CORBA org.omg.CosTransactions org.omg.CosTSPortability
Eg J2EE application server

Important Definitions

Enlistment refers to the process of making a resource a participant in a specific transaction and occurs in two flavors:

Static enlistment: The application server automatically enlists the resource in any transactions that it handles before any application code executes, e.g., at start-up.

Dynamic enlistment: Some container implementations are smart enough to realize that enlistment is not needed until the resource is used, and they enlist the resource dynamically when it is accessed.

The code below shows how the UserTransaction can be used in an application:

```
UserTransaction  tx =// lookup user transaction object
tx.begin();
 // do some work across multiple components and datastores
tx.rollback();
```

In the above code, the reference to the UserTransaction can be obtained from the JNDI context of the J2EE application or by using the EJB context if the component is an EJB. For example:

```
Properties p= new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,"some.vendor.factor.class");
p.put(Context.PROVIDER_URL, "applciation serve URL);
InitialContext ic = new InitialContext(p);
UserTransaction    tx = UserTransaction)ic.lookup("javax.transaction.UserTransaction");
//Or in EJB's
UserTransaction    tx = ejbContext.getUserTransaction() ;
```

The UserTransaction is simple and is used only to mark the transactional boundary. The application code never has access to the actual Transaction object. In this sense, the UserTransaction name is a misnomer because it works as more of a user transaction manager. It does not represent a single transaction, but the application's ability to begin and commit transactions. Table 11-1 lists some of the important methods in the UserTransaction interface.

Table 11-1. Methods in the UserTransaction

UserTransaction Method	Description
void begin()	Tells the transaction manager to create a new transaction and to associate it with the current

	thread.
void commit()	Commits and makes permanent the changes made in the transaction.
int getStatus()	Gets the status of the transaction associated with the current thread.
void rollback()	Rolls back and undoes the changes made in the transaction.
void setRollbackOnly()	When this is invoked, the transaction manager is informed that the only possible outcome of the transaction is to undo and roll back the work done in the transaction.
Void setTransactionTimeout (int seconds)	Sets the timeout value.

The interface javax.transaction.TransactionManager has methods identical to the UserTransaction with the additional ability to return the Transaction object. The TransactionManager also has two additional methods that allow the transaction to be suspended and resumed. These are listed in Table 11-2. J2EE does not mandate that a container expose the TransactionManager interface to application code (application code always works with UserTransaction); therefore, some containers may not expose this. Typically, containers bind the TransactionManager to a specific JNDI name for their product.

Table 11-2. Additional Methods in the TransactionManager

TransactionManager Methods	Description
Transaction getTransaction()	Returns the actual Transaction object associated with the current thread.
Transaction suspend()	Suspends the current transaction.
void resume(Transaction tobj)	Resumes execution of the current transaction.

The javax.transaction.Transaction object is used by the TransactionManager as an abstraction to the actual executing transaction. Application code almost never accesses the actual Transaction object. It is here that the resource enlistment and de-enlistment takes place. Table 11-3 summarizes the methods in this interface.

Table 11-3. The Methods in the javax.transaction.Transaction interface

Transaction Methods	Description
boolean enlistResource XAResource xares)	Invoked by the TransactionManager to enlist the resource with the transaction.
boolean delistResource XAResource xares,int flag)	Invoked by the TransactionManager to delist the resource with the transaction.
void registerSynchronization (Synchronization sync)	An object implementing the javax.transaction.Synchronization interface can be registered with the Transaction to receive callbacks. The TransactionManager invokes the beforeCompletion() method in the passed object before starting the transaction commit process and the afterCompletion() method when the transaction has completed.

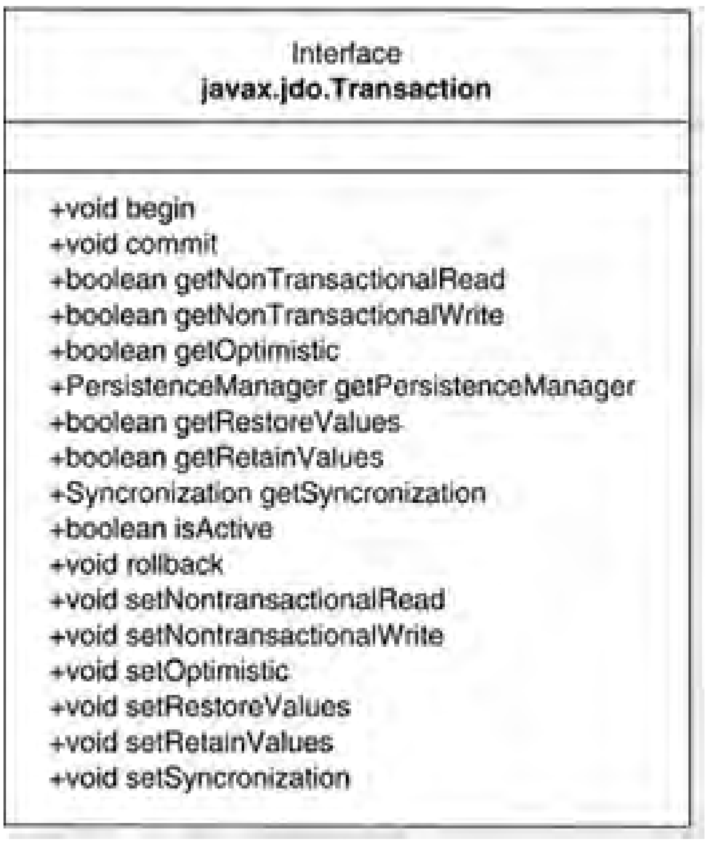
Now that you have an overall understanding of how transactions work and how they are abstracted by JTA and JTS for use in Java and J2EE, look at how JDO deals with transactions.

11.3 Transactions in JDO

Earlier in this chapter, we briefly mentioned that, in a flat transaction model, the calling thread is associated with a transaction. In JDO, the PersistenceManager represents the application's view of any persistent data. Logically, it is here that this association must occur. To separate out the functionality from the PersistenceManager itself, JDO defines a javax.jdo.Transaction interface. This Transaction has a one-to-one relationship with the PersistenceManager, meaning that if parallel transaction execution is required in an application, then multiple persistence managers must be obtained from the factory. (Refer to Chapter 1 for a discussion of transient and persistence instances.)

The Transaction interface by itself is quite simple, and the class diagram is shown in Figure 11-4. It contains a few different get and set methods that can be used to access and set different properties of the Transaction instance. These methods are discussed in subsequent sections. The begin(), commit(), and rollback() methods are the primary methods that application code uses to demarcate the transaction.

Figure 11-4. The javax.jdo.Transaction interface.


Interface
javax.jdo.Transaction
+void begin
+void commit
+boolean getNonTransactionalRead
+boolean getNonTransactionalWrite
+boolean getOptimistic
+PersistenceManager getPersistenceManager
+boolean getRestoreValues
+boolean getRetainValues
+Syncronization getSyncronization
+boolean isActive
+void rollback
+void setNontransactionalRead
+void setNontransactionalWrite
+void setOptimistic
+void setRestoreValues
+void setRetainValues
+void setSyncronization

Any persistence-related work or interaction with persistent objects done by the application must be performed in the context of a JDO transaction as shown below:

```
 PersistenceManager pm = pmf.getPersistenceManager();
 Transaction tx = pm.currentTransaction();
 tx.begin();
// Do some work here with persistent objects
  tx.commit();
// OR tx.rollback()
```

To preserve isolation semantics (the "I" in the ACID), a transaction "locks" the data in the underlying datastore using different concurrency control techniques. The locking, which happens under the covers and is completely transparent to application code, ensures that multiple operations on the same dataset do not interfere with each other. This is analogous to the way the synchronized keyword operates in the Java language. By using different locks (e.g., read locks, write locks, and so on), the transaction ensures that the serializable characteristic or the appearance that the transactions are acting upon the dataset one after the other is maintained.

The Four Isolation Levels

Most resource managers allow four isolation levels for a transaction, which are listed in increasing order of strictness:

- **Read uncommitted:** If another concurrently executing transaction writes data to the underlying resource without committing it, the data is visible in the executing transaction.
- **Read committed:** Committed data from another concurrently executing transaction is available.
- **Repeatable read:** Data read, which has been committed by another concurrently executing transaction, is guaranteed to retain the same values when read again.
- **Serializable:** This guarantees that two transactions operating on the same data execute serially one after the other.

J2EE and Isolation

The only way to specify isolation levels in J2EE (including EJBs) is by using a resource manager-specific API such as the JDBC Connection API. Most relational databases, by default, utilize the repeatable read or serializable isolation levels. Serializable isolation solves the problem of unrepeatable reads (application reads data and works on it, but when re-read, the dataset is different because of modification by another concurrently executing transaction) and phantom reads (application reads dataset, but when re-read, there is an addition to the dataset due to modifications by another concurrently executing transaction), but it can degrade performance due to the large number of locks required.

Based on the locking strategy and isolation level used, a transaction can be categorized into two groups:

- **Pessimistic transactions:** The data that is being used in this transaction is locked in the underlying resource and cannot be modified by anyone until the transaction has completed. This corresponds to the serializable isolation level and is also the strategy that the EJB model assumes. Pessimistic transactions are the default model for JDO that all vendors must implement. When a transaction executes, the data reflected in the persistent objects is locked by the underlying datastore.

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
tx.begin();
Author author = new Author("Keiron McCammon");
pm.makePersistent(author);
// do some other work here
// the data in the datstore that represents the author
// object is locked
tx.commit()
// OR tx.rollback()
```

- **Optimistic transaction:** This model assumes that it is unlikely for the data being used to be modified by any other transaction in the application. If at commit time, the datastore detects a collision, then the transaction is rolled back. This is an optional strategy for JDO vendors to implement. The JDO transaction strategy can be specified using the setOptimistic(boolean optimistic) method. When invoked with a true, the transaction is set to be optimistic; a false results in the default pessimistic strategy.

  To understand this further, look at the traditional producer-consumer example built on the code introduced in Chapters 3 and 5. The "hungry" consumer shown in Listing 11-1 runs in an infinite loop and searches the datastore for a particular object, and the transaction type is set with an argument. The producer creates the object instance, saves it, and reads the data back instantly as shown in Listing 11-2.

Listing 11-1 The hungry consumer

```
import com.corejdo.examples.chapter4.model.Author;
import javax.jdo.*;
import java.util.*;
import java.io.FileInputStream;

public class Consumer {
  public static void main (String args[]) throws Exception{
    boolean txtype= new Boolean(args[0]).booleanValue();
    Properties p = new Properties();
    p.load(new FileInputStream("config.properties"));
    PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory( p );
    PersistenceManager pm = pmf.getPersistenceManager();
    System.out.println("Reader searching..");
    while(true){
      Transaction tx = pm.currentTransaction();
      tx.setOptimistic(txtype);
      tx.begin();
      Query query = pm.newQuery(Author.class);
      String filter = "name ==(\"Keiron McCammon\")";
      query.setFilter(filter);
      Collection authors =(Collection) query.execute();
      Iterator it = authors.iterator();
      while (it.hasNext()) {
             Author author = (Author) it.next();
             String name = author.getName();
             System.out.println("Author name = " +name);
             }
           tx.commit() ;
      }
  }
}
```

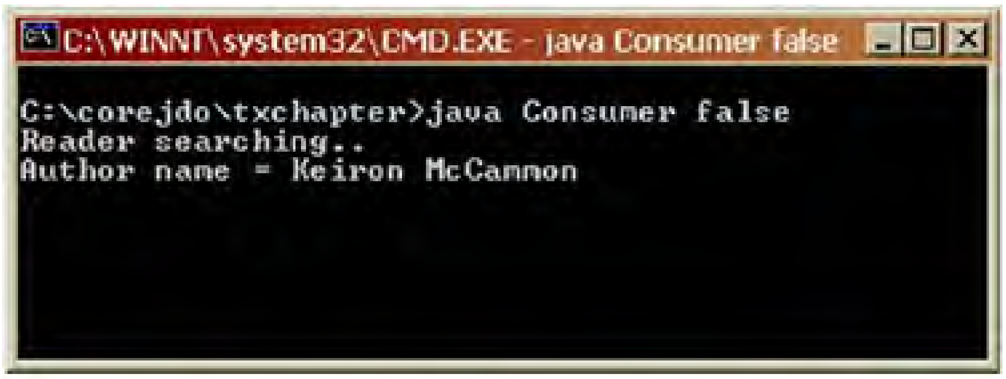
Listing 11-2 The producer

```
import com.corejdo.examples.chapter4.model.Author;
import javax.jdo.*;
import java.util.*;
import java.io.FileInputStream;

public class Producer {
  public static void main(String[] args) throws Exception {
  boolean txtype= new Boolean(args[0]).booleanValue();
  Properties p = new Properties();
  p.load(new FileInputStream("config.properties"));
  PersistenceManagerFactory pmf =
          JDOHelper.getPersistenceManagerFactory( p );
  PersistenceManager pm = pmf.getPersistenceManager();
// create and persist the object
  Transaction tx = pm.currentTransaction();
  tx.setOptimistic(txtype);
  tx.begin();
  Author author = new Author("Keiron McCammon");
  pm.makePersistent(author);
  System.out.println("Author persisted");
  tx.commit() ;

// Read the persisted object from the datastore
  tx.begin() ;
  Query query = pm.newQuery(Author.class);
  String filter = "name ==(\"Keiron McCammon\")";
  query.setFilter(filter);
  Collection authors = (Collection) query.execute();
  System.out.println("Everything is locked");
// do some other work here
  Thread.sleep(120000);
// the data in the datstore that represents the author
  System.out.println("Producer wokeup");
  tx.commit() ;
  System.out.println("Producer closed");
  pm.close();
    }
}
```

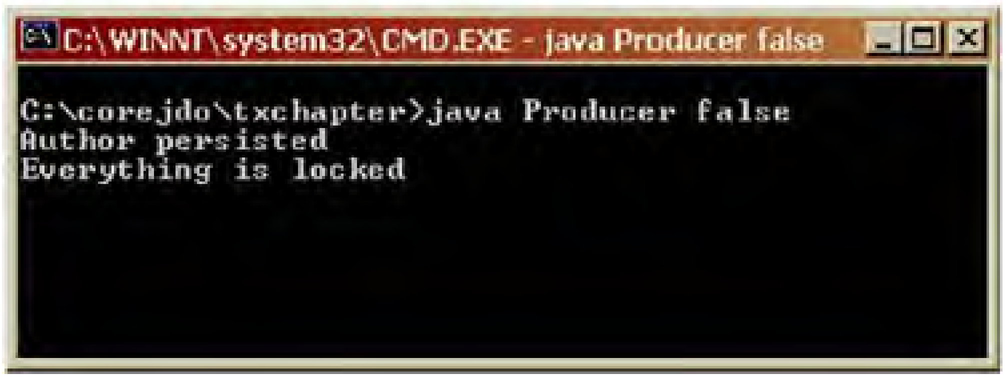
To execute the example, the steps outlined in Chapter 3 need to be followed (i.e., the classes compiled and enhanced, and the datastore set up for the implementation being used) and then the consumer started, followed by the producer.

With pessimistic or datastore transactions, the producer locks the data corresponding to the persistent objects in the underlying datastore, making it inaccessible to other transactions. The consumer can only access the data in the window of time between the two transactions in the producer or when the producer exits. This is shown in Figures 11-5a and 11-5b.

Figure 11-5a and b. The producer has locked the data in pessimistic transaction, and the consumer is waiting for the locks to be released.



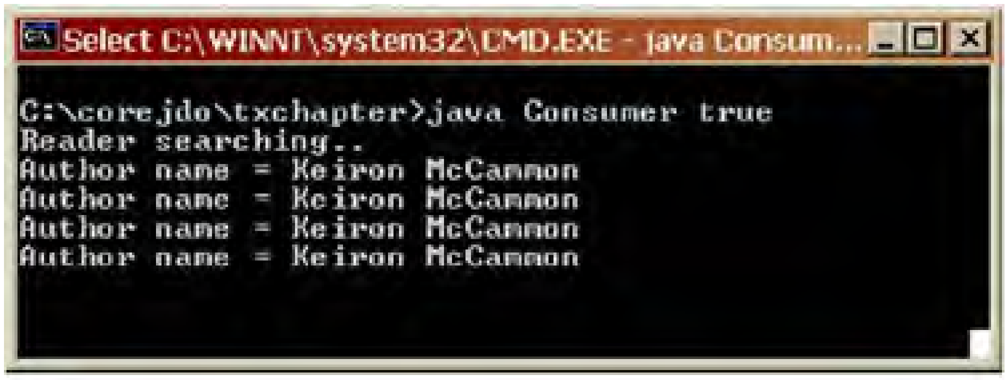
```
C:\corejdo\txchapter>java Consumer false
Reader searching..
Author name = Keiron McCammon
```



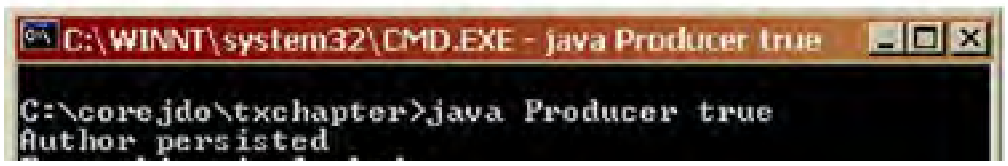
```
C:\corejdo\txchapter>java Producer false
Author persisted
Everything is locked
```

With optimistic transactions, the producer doesn't lock the data for the duration of the transaction, so the consumer is able to access the data immediately after it is persisted, as shown in Figures 11-6a and 11-6b.

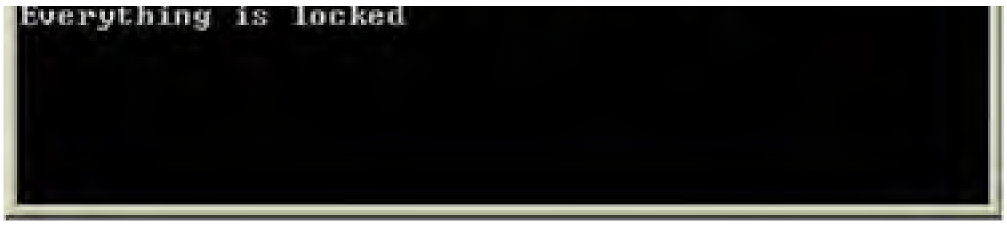
Figure 11-6a and b. The optimistic producer does not lock the data, and the consumer can access it immediately.



```
C:\corejdo\txchapter>java Consumer true
Reader searching..
Author name = Keiron McCammon
Author name = Keiron McCammon
Author name = Keiron McCammon
Author name = Keiron McCammon
```



```
C:\corejdo\txchapter>java Producer true
Author persisted
```



```
Everything is locked
```

There are distinct advantages and disadvantages to each locking technique. Tables 11-4 and 11-5 summarize the differences and approaches to use.

Table 11-4. Pessimistic Transactions Summarized

Strategy	Advantages	Disadvantages
Pessimistic: Data is locked until the transaction completes.	Provides highest degree of consistency in the data.	Does not scale well. May cause deadlocks when transactions are waiting on each other to release locks. May be detrimental to performance because of the high degree of locking.
When to use	When concurrent access is rare or not expected. When transactions are short lived. When user think-time does not affect transaction duration.	
When not to use	For long lived transactions. If optimistic transactions suffice (see Table 11-5).	

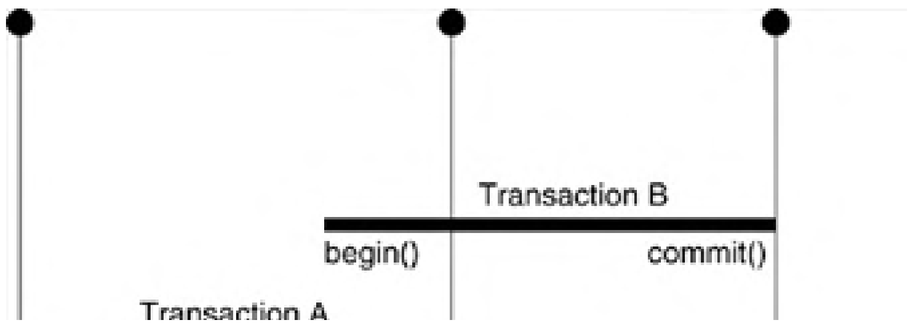
Table 11-5. Optimistic Transactions Summarized

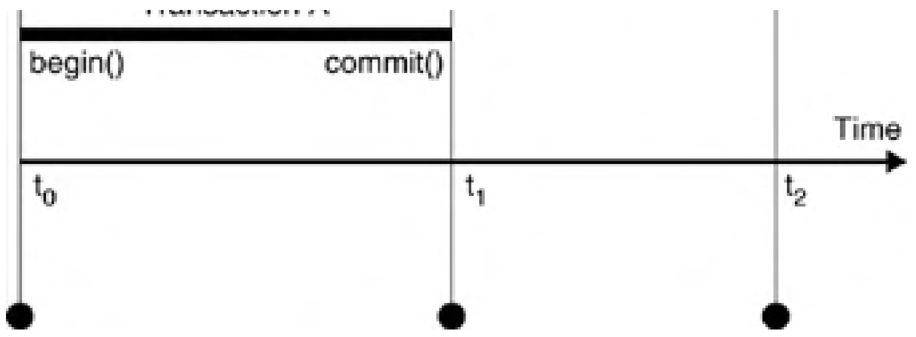
Strategy	Advantages	Disadvantages
Optimistic: Data is locked only when the transaction committed.	Performs better because locking overhead is reduced. Systems generally scale better because resource usage at the datastore level is reduced.	Requires developers to write code to support collision detection and handling such situations.
When to use	When concurrent access is expected to be frequent and high. When transactions are long lived. When the clients are remote (e.g., the JDO code on thick clients). When transaction involve user think-time.	
When not to use	For long lived transactions.	

11.3.1 Understanding conflicts with optimistic transactions

Although optimistic transactions assume that it is unlikely for the underlying data being used to be modified, such situations can occur. To understand how to resolve such potential conflicts, consider the scenario in Figure 11-7, where two overlapping transactions, Transaction A and Transaction B, affect the same underlying data.

Figure 11-7. Conflicts in optimistic transactions.





When transaction B attempts to commit, its changes at time t2 would be forced to rollback because the underlying data has been changed externally by Transaction A at t1 (the datastore detects a collision). This is often called "First Commit Wins," in which the first set of changes are not overwritten and force the second transaction to restart the work with the changed data. The exception facilitates failure detection; however, the amount of rework is increased because developers now have to explicitly account for such situations and retry Transaction B and the business logic by reloading the data from the datastore.

One strategy for reducing the occurrence of such conflicts is called "Last Commit Wins," which assumes that the consequences of overwriting data are acceptable to the application.

This strategy tries to reduce the risk of overwrites by refreshing the data from the underlying datastore in Transaction B just prior to performing the business logic. In JDO, this is achieved though the refresh() method in the PersistenceManager, which reloads the state of a particular persistent object from the datastore. For example, in the code below, the author instance is refreshed explicitly in an optimistic transaction:

```
  PersistenceManager pm = pmf.getPersistenceManager();
  Transaction tx = pm.currentTransaction();
  tx.setOptimistic(true);
tx.begin();
try {
//
    // do some work here
    //
         Author author = (Author) it.next();
         String name = author.getName();
// Just before changing the persitent object refresh it
         pm.refresh(author);
         author.setName("John Malkovich");
         tx.commit();
         }catch(Exception e){
                if(tx.isActive())
                 tx.rollback();
         }
```

11.3.2 Transactional and non-transactional objects

In Chapter 3, we mentioned how the PersistenceManager includes a caching mechanism, and in Chapter 4, we looked at the different states including transactional and non-transactional states.

You may recall that objects can be categorized as transactional and non-transactional depending on the caching behavior of the PersistenceManager. This is orthogonal to the concept of transient and persistent objects introduced in Chapter 1 and later covered as states in Chapter 4. A transactional object is an object whose *managed* fields are cached by the underlying implementation of the PersistenceManager in the context of a Transaction. For example, consider the code below:

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
tx.begin();
Author author = new Author("Keiron McCammon");
try {
    author.setName("John Malkovich");
  // do some other work here
    tx.commit()
    }catch(Exception e){
       tx.rollback();
      }
```

Persistent Fields and Transactional Fields

Object instance variables that are persisted are called persistent fields. Instance variables that participate in a transaction are called transactional fields, meaning that their values can be restored by a rollback. Both persistent and transactional fields are collectively called as managed fields because they must be managed by the JDO implementation.

Recall from Chapter 5 that individual fields can be marked as persistent or transactional in the metadata descriptor. By default, fields that are not static, final, or transient are marked as "persistent" and exhibit transactional behavior in the enhanced class. When marked as "transactional," the fields are cached and restored on a transactional rollback, but are not persisted to the datastore.

By default, the name field is transactional because the Author class is persistence-capable, and the value "Keiron McCammon" is cached in the PersistenceManagers cache. This value is restored in case of a potential rollback() and is discarded when commit() is invoked.

All this is unrelated to the fact that the Author object is transient or persistent. In this example, the instance is persistent and undergoes a state change to persistent-dirty when the setName() method is invoked. However, author could be an instance of a transient object as well, in which case the state changes would be different, as explained in Chapter 5.

In summary, both transient and persistent objects can be transactional or non-transactional. JDO requires that implementations support transactional behavior for persistent objects. A transient transactional object is one that by itself does not represent persistent data, but has fields that recognize transactional boundaries and are restored to their original values by the JDO implementation when the transaction rolls back. Support for transactional behavior in transient objects is optional, meaning that a JDO implementation is not required to cache fields for transient objects.

This is in line with the general behavior in which, between the scope of a tx.begin() and a tx.commit(), transient objects—i.e., persistence-capable instances that are made transient using the pm.makeTransient()—are not managed by the JDO implementation. Therefore, by default, transient instances are non-transactional, whereas persistent instances are managed by the JDO implementation and are transactional.

Transient Objects and State Transitions

A transient object is merely an instance that is not persisted; however, the class definition must still be persistence-capable in order for JDO to manage the state transitions.

Table 11-6. Transactional Behavior in Persistent and Transient Objects

	Transactional	**Non-Transactional**
Persistent Objects	Required	Optional
Transient Objects	Optional	Required (JVM's default behavior)

The persistence manager can be configured with two optional properties that specify how non-transactional instances are accessed and can be very helpful while using optimistic transactions:

- **javax.jdo.option.NontransactionalWrite:** When this property is set to true, the fields of a persistent non-transactional instance can be modified outside the scope of a JDO transaction, even though changes made in this manner are not automatically propagated to the datastore. Without this property explicitly sent, the default behavior is to throw a JDOUserException for any attempt to modify a persistent instance outside the scope of a JDO transaction.

- **javax.jdo.option.NontransactionalRead:** When this property is set to true, the fields of a non-transactional instance can be read outside the transaction.

The usage of both of these properties can be better understood by the example in Listing 11-3. The properties file passed to the JDOHelper must contain the above properties set to true, and the implementation must support this behavior, which is optional in the specifications. The example first creates and saves an object. It then uses the NonTrasnactionalRead property to read it outside the JDO transaction and the NonTrasnactionalWrite property to modify the object. The example also shows transactional behavior of transient and persistent instances.

Listing 11-3 A non-transactional read-write example

[View full width]

```
import com.corejdo.examples.chapter4.model.Author;
import javax.jdo.*;
import java.util.*;
import java.io.FileInputStream;

public class NonTxReadWriteExample {

 public static void main(String[] args) throws Exception{

// The properties file contains all configuration
// properties
    Properties p = new Properties();
    p.load(new FileInputStream("config.properties"));
    PersistenceManagerFactory pmf =  JDOHelper.getPersistenceManagerFactory( p );
    PersistenceManager pm = pmf.getPersistenceManager();

// create and persist the object
    Transaction tx = pm.currentTransaction();
    tx.setOptimistic(true);
    tx.begin();
    Author author = new Author("Keiron McCammon");
    pm.makePersistent(author);
    System.out.println("Author persisted");
    tx.commit() ;
// At this point the data is commited to datastore.

// Note Below line with throw javax.jdo.JDOUserException // if implementation does not
➥ support property
// javax.jdo.option.NontransactionalRead=true

   System.out.println("Author Name = "+author.getName());

// Persisted data of Keiron above will not be be
// overwitten by this even though the instance is modified // and available in memory
➥ while this application is
// executing.

// Note :Below line with throw javax.jdo.JDOUserException // if implementation does not
➥ support property
// javax.jdo.option.NontransactionalWrite=true

   author.setName("Heiko Bobzin");

// Create another object. This is transient
  Author contributingauthor = new Author("John Malkovich");

  tx.begin();
// Transient-Transactional
   pm.makeTransactional(contributingauthor);
// make some changes to object
   contributingauthor.setName("Jeanne Smith");
// objects state will be restored from cache
   tx.rollback();

// should not print out "Jeanne Smith" but initial cached
// value of "John Malkovich"
  System.out.println("Contributing Author Name = "   +contributingauthor.getName());

// the segment below shows trasnactional behavior in the // persistent objects.
    tx.begin();
    Author anotherauthor = new Author("John Doe");
// instance is transient
    pm.makePersistent(anotherauthor);
/ instance is now persistent
    anotherauthor.setName("Michael Vorburger");
    System.out.println("Author name inside tx=" +anotherauthor.getName());
    tx.rollback() ;

// The cached value of John Doe is replaced in the
// persistent object.
    System.out.println("Author name outside tx=" +anotherauthor.getName());

// Read the persisted object from the datastore
  Query query = pm.newQuery(Author.class);
  Collection authors = (Collection) query.execute();
  Iterator it = authors.iterator();
  while (it.hasNext()) {
    author = (Author) it.next();
    author = (Author) it.next();
    String name = author.getName();
    System.out.println("Author name = " + name);
    }
  }
}
```

11.3.3 Retaining and restoring values

The PersistenceManager also allows two optional settings that affect the manner in which instances are kept in memory at commit and rollback time:

- **javax.jdo.option.RetainValues:** By default, after a transaction commits, the persistent instances in the cache are evicted and moved to a hollow state as explained in Chapter 4. This allows the implementation to easily manage the cache by keeping it small. When this property is set to true, the instances are retained in the cache after a transaction commits. This can increase the cache size, but can be helpful if the same instances are reused across transactions in an application.

- **javax.jdo.option.RestoreValues:** By setting this property to false, the default transactional behavior of restoring fields to the state that they were in when the makePersistent() method was invoked can be overridden. Application code needs to explicitly restore the original values. Because the overhead of the cache is removed, the performance is generally improved with this property. This should only be set if instances are not reused after a transaction rolls back. The code segment below from the previous example explains this further:

  [View full width]

```
 Transaction tx = pm.currentTransaction();
// the segment below shows trasnactional behavior in the
// persistent objects.
tx.begin();
    Author anotherauthor = new Author("John Doe");
    pm.makePersistent(anotherauthor);
    anotherauthor.setName("Michael Vorburger");
    System.out.println("Author name inside tx="
                     +anotherauthor.getName());
tx.rollback() ;
// With javax.jdo.option.RestoreValues=false this will not // restore the value to "John
➥ Doe" since that was never
// cached and the value "Michael Vorburger" will be printed

System.out.println("Author name outside tx="
                        +anotherauthor.getName());
```

RetainValues and RestoreValues

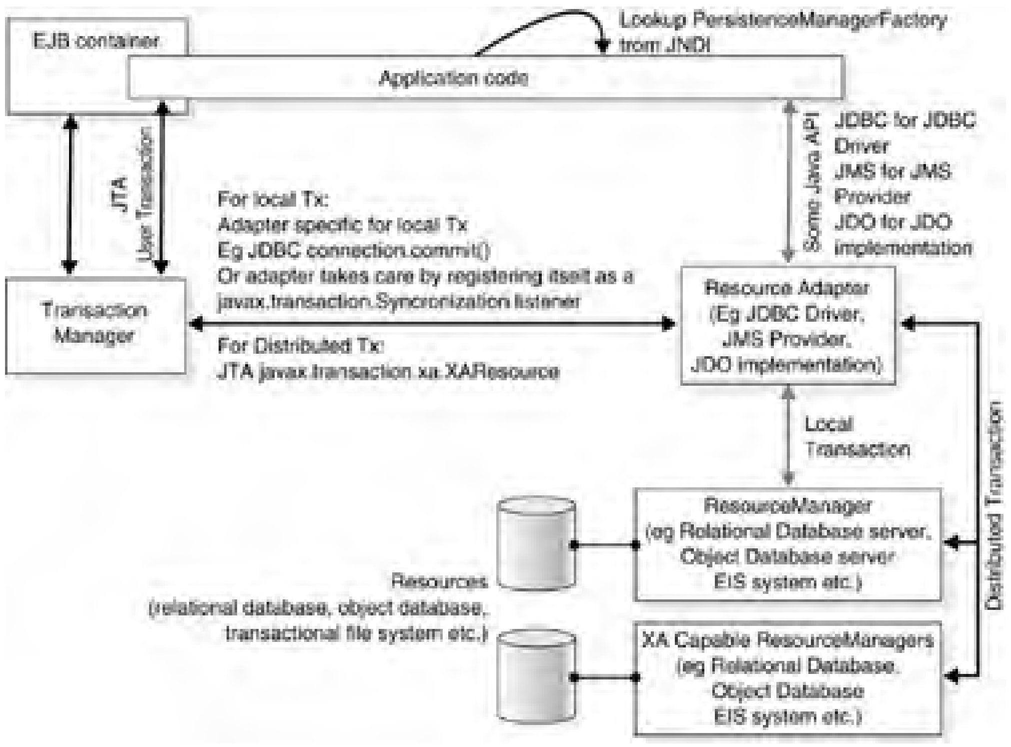
The optional features **RetainValues** and **RestoreValues** affect only how the instances are kept in memory at commit and rollback time, respectively. They do not alter the outcome of the commit or rollback on a transaction. The RestoreValues option is used if instances are not used across rollbacks, and the RetainValues option is used if instances are used across transactions.

11.3.4 JDO and transactions in a J2EE application server

In order to incorporate transactional semantics in J2EE components that utilize JDO for transparent persistence, it is important to understand how a transaction in JDO behaves in such managed environments.

As mentioned in Chapters 1 and 8, the JDO specifications define the relationship and integration between the JDO and the managed J2EE environment by requiring the vendors to satisfy the connector (Java Connector Architecture, or JCA) contracts. In short, the JDO implementation is integrated into the J2EE container as a JCA connector (i.e., a RAR file). The components use JDO by obtaining the PersistenceManagerFactory from a JNDI tree by name and then use it to obtain the PersistenceManager as usual. Figure 11-8 summarizes the transactional aspect of this integration. Although JDO allows the implementation to provide its own adapter or to use third-party adapters, most vendor implementations today provide their own adapters and package their implementations as RAR files for deployment in different application servers.

Figure 11-8. JDO and J2EE transactional integration.


Lookup PersistenceManagerFactory from JNDI
EJB container
Application code
JTA User Transaction
JDBC for JDBC Driver
JMS for JMS Provider
JDO for JDO implementation
Some Java API
For local Tx:
Adapter specific for local Tx
Eg JDBC connection.commit()
Or adapter takes care by registering itself as a javax.transaction.Syncronization listener
Transaction Manager
For Distributed Tx:
JTA javax.transaction.xa.XAResource
Resource Adapter (Eg JDBC Driver, JMS Provider, JDO implementation)
Local Transaction
Distributed Transaction
ResourceManager (eg Relational Database server, Object Database server EIS system etc.)
Resources (relational database, object database, transactional file system etc.)
XA Capable ResourceManagers (eg Relational Database, Object Database EIS system etc.)

In order to connect to the resource, the adapter can manage its own connections or have them managed by the container. Details about connection management and JCA integration are covered further in Chapter 8. In this section, we concern ourselves with transactional aspect for now. In JCA, a resource adapter can be classified in three distinct levels that inform the container about its transactional capabilities. These are summarized in Table 11-7. In order for the application to use distributed transaction capabilities in managed environments, not only must the underlying datastore expose the XAResouce (e.g., the JDBC driver must support the javax.sql.XAConnection), but also this must be exposed at the resource adapter level so that the TransactionManager in the container can manage the distributed transaction.

Table 11-7. Transactional Behavior for Resource Adapters

Transaction Level of Connector	Work Performed as Part of JTA Transaction	Distributed Transactions or Combination with Other Transactional Resources Allowed
NoTransaction	✗	✓
LocalTransaction	✓	✗
XATransaction	✓	✓

At the simplest level, components such as Servlets and JSPs can use JDO and the javax.jdo.Transaction in much the same way as the standalone code discussed until now to support local transactions. Rather than using the JDOHelper, the PersistenceManagerFactory is obtained from the JNDI tree. The example in Listing 11-4 shows a Servlet that obtains the PersistenceManagerFactory from JNDI at initialization time. The PersistenceManager obtained from this factory is then used for persisting data while servicing client requests. If the implementation supports pooling, the close() method returns the PersistenceManager instance back to the pool.

JNDI Binding

So show does the factory get bound by name in JNDI? The JCA adapter is required to provide the **javax.resource.spi.ManagedConnectionFactory** interface. This represents the outbound connectivity information to the EIS instance from an application via the resource adapter instance and includes. Along with this factory, the adapter specifies the interface supported by this factory and the implementation class as a part of the adapter's **ra.xml** configuration. For example:

```
<resourceadapter>
<managedconnectionfactory-class>
    some vendor class </managedconnectionfactory-class>
<connectionfactory-interface>
         javax.jdo.PersistenceManagerFactory
                    </connectionfactory-interface>
```

```
<connectionfactory-impl-class>
      some vendor class </connectionfactory-impl-class>
<connection-interface>
    javax.jdo.PersistenceManager</connection-interface>
<connection-impl-class>
      some vendor class </connection-impl-class>
<!--other JCA elements -- >
</resourceadapter>
```

When the adapter is deployed, it is bound to a JNDI name in an application server-specific manner like the **deploytool** in the J2EE reference implementation. When application code looks up JNDI by that name, based on the **ra.xml** file, the container returns a **PersistenceManagerFactory.**

Listing 11-4 A Servlet using JDO and local transactions

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.jdo.*;
import java.util.Collection;
import java.util.Iterator;
import com.corejdo.examples.chapter4.model.Author;

public class JDOServlet extends HttpServlet {
    PersistenceManagerFactory pmf;

    public void init(ServletConfig config) throws ServletException {
        try {
            super.init(config);
            InitialContext ctx = new InitialContext();
            pmf = (PersistenceManagerFactory) ctx.lookup("JDOPMFactory");
        } catch (NamingException e) {
            throw new ServletException(e);
        }
    }

    /** Processes requests for both HTTP <code>GET</code>
        and <code>POST</code> methods.
     */
    protected void service(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, java.io.IOException {
        response.setContentType("text/html");
        java.io.PrintWriter out = response.getWriter();
        String authorname = request.getParameter("authorname");
        if (authorname == null) {
            out.println("Invalid Request");
            out.close();
            return;
        }

        PersistenceManager pm = pmf.getPersistenceManager();
        Transaction tx = pm.currentTransaction();
        tx.begin();
        Author author = new Author("authorname");
        pm.makePersistent(author);
        tx.commit();

        out.println("<html><body>");
        out.println("<h2>Authors in this Example are</h2>");
        // Read the persisted object from the datastore
        Query query = pm.newQuery(Author.class);
        Collection authors = (Collection) query.execute();
        Iterator it = authors.iterator();
        while (it.hasNext()) {
            author = (Author) it.next();
            out.println("Author name = " + author.getName());
        }
        pm.close();

        out.println("</body></html>");
        out.close();
```

```
   }
}
```

Although the above usage of transactions is suitable for simple applications, applications using multiple components together in a single transaction or requiring distributed transactions require more. In J2EE, the container makes the JTA UserTransaction available to components like Servlets or JSPs using JDNI and to EJBs using the EJBContext object (EJBs can invoke the getUserTransaction() method). When the UserTransaction is used to demarcate the transaction, a mechanism is needed to inform the JDO implementation to flush its cache and the datastore and to synchronize itself with the transaction. For this purpose, JDO specifications require that the vendor implement the javax.transaction.Synchronization interface in the Transaction object.

The container is responsible for invoking the beforeCompletion() and afterCompletion() callbacks in the Synchronization listeners (the JDO implementation in this case) when the transaction completes. This is analogous to the way javax.ejb.SessionSynchronization works in EJBs.

When the transaction is demarcated using the UserTransaction, the PersistenceManager should be obtained inside the transaction. For example:

```
UserTransaction utx=
            ctx.lookup("java:comp/UserTransaction") ;
tx.begin() ;
// do some work
PersistenceManager pm=factory.getPersistenceManager();
// do some JDO work here
pm.close();
tx.commit() ;
  }
```

In situations like the above, when the persistence manager is obtained inside an active JTA transaction, using methods of the JDO transaction throws a JDOUserException. This is done to allow the implementation to synchronize itself with the JTA transaction. Therefore, when demarcating transactions using the UserTrasnaction as above or when using container-managed transactions, you should not use the JDO Transaction methods.

Transaction Boundaries in EJBs

In bean-managed transactions, the EJB code uses the **UserTransaction** object to demarcate the transactions boundaries explicitly using **begin()** and **commit()**. For EJBs utilizing container-managed transactions, the container (not the bean code) sets the boundaries of the transactions. Container-managed transactions can be used with session, entity, and message-driven beans, while bean-managed transactions cannot be used with entity beans.

Although it is rare for Servlets and JSPs to use JDO and other JDBC (or even multiple JDO implementations in the same container) collectively as a part of a single transaction, it may be common for JDO and JMS to be used together. Both of these cases would require additional care. For example, consider the example in which the service method of a Servlet contained code that interacts with some pre-existing databases or EIS resources and JDO in the same invocation.

```
  protected void service(HttpServletRequest request,
                  HttpServletResponse response){
// do some JDO persistence work here
// update an existing JDBC or EIS resource or put some
// messages on a JMS topic/queue
  }
```

Such interactions with multiple datastores would need the services of a JTA transaction manager and a UserTransaction to coordinate the *distributed* transaction.

```
protected void service(HttpServletRequest request,
                  HttpServletResponse response){
// get the JTA Transaction
UserTransaction utx=
            ctx.lookup("java:comp/UserTransaction") ;
tx.begin() ;
// do some JDO work here
// update an existing JDBC or EIS resource
tx.commit() ;
  }
```

Let's explore this further with an example of an EJB designed for bean-managed persistence and container-managed transactions utilizing distributed transaction capabilities. Listing 11-5a shows the remote interface with an updateAuthors method, and Listing 11-5b shows the home interface. The session bean is shown in Listing 11-5c and contains the implementation for updateAuthors. The BMP EJB uses JDO to save the Author objects and also updates a table in an existing ISBN database with the new author names. This EJB does not use JDO or JTA transactions explicitly, but uses container-demarcated transactions as a part of its deployment descriptor. For example, the ejb-jar.xml descriptor contains something like this:

```
<ejb-jar>
<!--other elements here-->
  <container-transaction>
      <method>
        <ejb-name>TxEJB</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

</ejb-jar>
```

Non Transactional Resources

Using transactions makes sense only if the underlying resource is transactional. For example, including an action such as sending an email as part of a transaction or declaring an EJB that sends mail as transactional merely causes overheads.

Listing 11-5a The remote interface

```
package com.corejdo.examples.transactions;

import java.rmi.RemoteException;
import java.util.Vector;
import javax.ejb.*;

public interface AuthorEJB extends javax.ejb.EJBObject {
 public boolean updateAuthors(Vector authors, String isbn)
          throws RemoteException, EJBException;
}
```

Listing 11-5b The home interface

```
package com.corejdo.examples.transactions;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface AuthorHome extends javax.ejb.EJBHome {
 public AuthorEJB create() throws CreateException,
             EJBException, RemoteException;
}
```

Listing 11-5c The EJB implementation using JDO and JDBC together

[View full width]

```
package com.corejdo.examples.transactions;

import com.corejdo.examples.chapter4.model.Author;
import javax.jdo.*;
import java.rmi.RemoteException;
import java.util.Vector;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.sql.Connection;
import javax.sql.DataSource;
import javax.ejb.*;
import java.util.Iterator;
import java.sql.PreparedStatement;
```

```
public class AuthorBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext ctx;
    private PersistenceManagerFactory pmf;
    private InitialContext ictx;

    private static String AuthorXADataSource = "AuthorXADataSource";

    public void setSessionContext(javax.ejb.SessionContext context) throws RemoteException
➥ , javax.ejb.EJBException {
        ctx = context;
        try {
            ictx = new InitialContext();
            pmf = (PersistenceManagerFactory)ictx.lookup("java:comp/env/jdo/authorsb");
    System.out.println("AuthorBean : Located PMF jdo/authorsb");
        } catch (NamingException e) {
           System.out.println("AuthorBean : Exception locating PMF jdo/authorsb " + e);
            throw new EJBException(e);
        }
    }

  public void ejbActivate() throws EJBException{}
  public void ejbPassivate() throws EJBException{}
  public void ejbRemove() throws EJBException {}
  public void ejbCreate() throws CreateException,
                                  EJBException {}

   /**
     * @input A vector of Author objects
     * @output The ISBN number
     */
 public boolean updateAuthors(Vector authors, String isbn){
// initialize member variables
    Connection con=null;
    String authornames="";
    PreparedStatement ps =null;
    PersistenceManager pm=null;

// do some JDO work
        try {

          pm = pmf.getPersistenceManager();
            Iterator it = authors.iterator();
            while (it.hasNext()) {
                Author author = (Author)it.next();
                authornames += author.getName() + " , " ; // build the string for next part
                pm.makePersistent(author);
            }
            pm.close();
    // do some work on another EIS resouce
    DataSource ds = (javax.sql.DataSource) ictx.lookup("AuthorXADataSource");
           con = ds.getConnection();
          ps = con.prepareStatement("update isbntable set authors = ? where isbn = ?");
            ps.setString(1, authornames);
            ps.setString(2, isbn);
            ps.executeUpdate();
            return true;
        } catch (Exception e) {
        System.out.println("An exception occured in the distributed tx " + e);
            // mark the distributed Tx for rollback only
            ctx.setRollbackOnly();
            return false;
        } finally {
            try{
            ps.close();
            con.close();
            pm.close();
            }catch(Exception e){} // do nothing here
        }
    }
}
```

In summary, keep these guidelines in mind while working with the J2EE components and JDO:

1. When using only JDO from Servlets and JSPs (i.e., local transactions), the JDO Transaction can be used to demarcate the transaction.
2. When using JDO from Servlets and JSP *alongside* other transactional resources like JDBC and JMS, the UserTransaction must be used to demarcate the distributed transaction.

3. When using only JDO in EJBs with bean-managed transactions to handle local transactions, use the UserTransaction and JDO automatically synchronizes itself.
4. When using JDO with other transaction resources in bean-managed transactions, use the UserTransaction to demarcate the distributed transaction.
5. When using container-managed transactions, do not use the JDO Transaction object in the EJB code.
6. Application components that need to execute outside the J2EE container should use the javax.jdo.Transaction.

11.3.5 Transaction callbacks synchronization

We have looked at how the JDO Transaction object acts as a receiver of callbacks from the JTA transaction by having the vendor implementation of the interface register itself and implementing the javax.transaction.Synchronization interface. The Trasanction object can also act as a supplier of callbacks to developer-written listeners that are registered with it. For example, developers can write a class that implements the Synchronization interface and register it with the JDO Transaction object, as shown here:

```
        Transaction tx = pm.currentTransaction();
    tx.setSynchronization(myobject);
// myobject is an instance of a developer written class
```

This allows the application to receive notifications on the lifecycle of the transaction in which they participate. When the JDO implementation invokes the afterCompletion(int status) and beforeCompletion() methods, the application can trap these events and perform necessary work or restore other states. This can be very useful in the J2EE environment in which the session state is being maintained in the HttpSession. For example, HttpSession instances do not have a mechanism that allows them to receive transaction notifications, so objects added to the HttpSession during a transaction are not removed if the transaction is rolled back. All changes to data in a session are durable despite the outcome of any executing transactions. Developers can use the Synchonization interface to perform such cleanup.

11.4 Summary

In this chapter, we started by looking at some fundamental concepts around transaction and how Java incorporates transaction management with JTA. We also looked at the transaction capabilities included in JDO and how they can be used in standalone applications as well as in conjunction with JTA in managed environments like J2EE. We also outlined some design considerations and practices that developers can use when using JDO in a transactional manner.

In Chapter 12, we look at the relationship between JDO and JDBC.

Part 4: The Conclusion

Chapter 12. JDO and JDBC

"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."

—Sir Arthur Conan Doyle

This chapter compares JDO with JDBC, another Java persistence API. JDBC is a standard designed specifically to access relational database systems. Although many readers may be familiar with the JDBC API, for those who are not or would like to refresh their memories, the first part of this chapter outlines JDBC fundamentals, including an overview of the latest JDBC 3.0 edition. The second part then compares JDO with JDBC.

12.1 JDBC 2.0 and 3.0

JDBC (Java Database Connectivity[1]) is a Java API for accessing relational database systems. Prominent examples of relational database management system (RDBMS) products include Oracle, IBM DB2, Microsoft SQL Server, Sybase or Interbase, as well as open-source RDBMS solutions like MySQL or PostgreSQL.

[1] Sun's official position is that JDBC does **not** stand for Java Database Connectivity, possibly because of trademark issues with Microsoft's ODBC, Open Database Connectivity; however, it is the generally accepted assumption and a useful memory bridge to remember the acronym.

Other approaches for database systems included ISAM, hierarchical, or document-oriented databases, as well as X.500/LDAP repositories, and more recently, object databases. JDBC was not designed to access any of these non-relational database systems.

A relational database stores data entities in rows of tables, with fields of the entities stored as columns. Column values are constrained by an SQL data type, such as VARCHAR, INTEGER, TIMESTAMP, and so on. Most tables have one column designated as primary key, which uniquely identifies the row within the table. Relationships between entities are usually expressed by storing values of primary keys in columns of other entities, sometimes using foreign-key integrity constraints.

Relational databases and their usage through JDBC are closely modeled around a classical and simple client/server scenario: A client (e.g., an application using JDBC) sends an SQL request (e.g., a SELECT query statement) to a relational database server, which processes it and returns the result in the form of a set of rows.

A query is expressed in SQL, the standardized Structured Query Language. SQL-92 currently is commonly implemented, and SQL-99 standard is the latest release. The SQL language includes keywords for querying, inserting and updating data, among others.

Although the JDBC API as such is certainly object-oriented, using interfaces and classes, objects and methods, it is important to understand that it still is a (very thin) wrapper around SQL, which is procedural in nature. Persistent data in JDBC always comes in the row and column flavor, never directly as Java objects.

Other APIs

Although JDBC currently is the most widely used standard API to access relational databases using Java, alternative proposals exist but have lost importance. The most well-known example of such an alternative API is probably SQLj[a] (also known under the name of Oracle's implementation SQL4J, and formerly called JSQL), a joint venture among IBM, Oracle, and other vendors. One of its ideas was to embed SQL code straight into Java source code, not inside Strings, and to use a pre-compiler to check that SQL and Java types match at application compile-time.

[a] http://www.sqlj.org

12.1.1 Using JDBC fundamentals

The basic steps using JDBC are always similar and evolve around the following points:

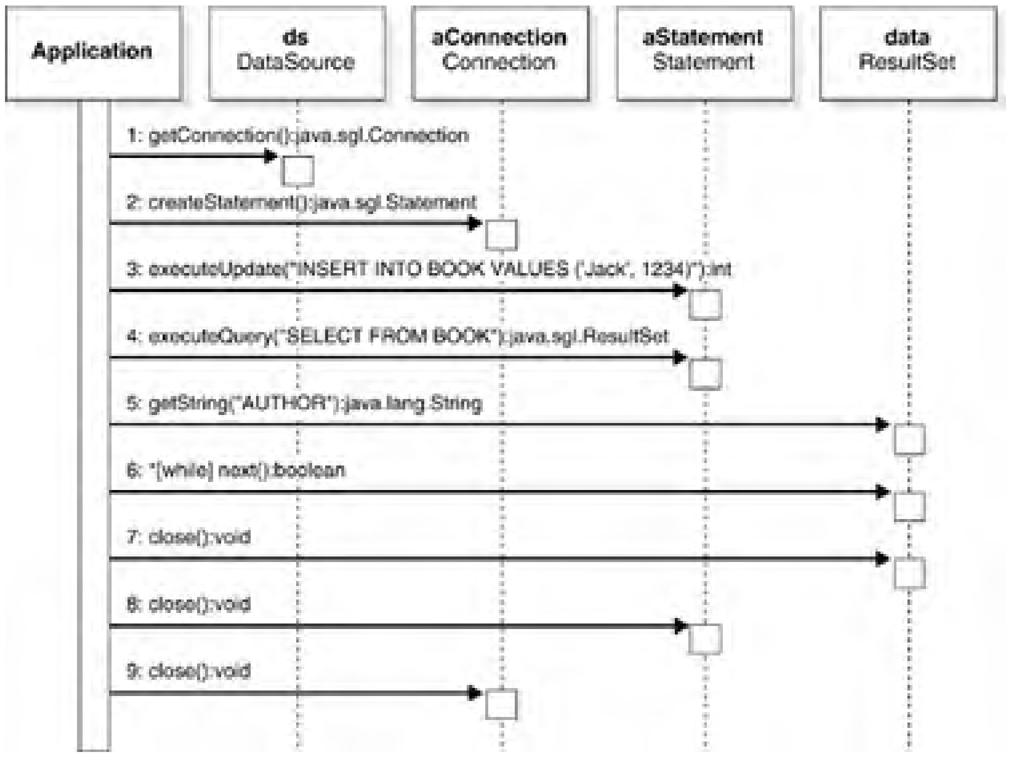
1. Obtain a java.sql.Connection, either from a java.sql.DriverManager or from a javax.sql.DataSource. Using a DataSource is the preferred method and ensures portability to application servers using connection pools.
2. Obtain a java.sql.Statement from the Connection, or reuse a PreparedStatement or CallableStatement possibly created earlier.
3. Run SQL code, e.g., via the executeUpdate() method of Statement, or via the executeQuery() method, which returns a java.sql.ResultSet. SQL can contain both Data Definition Language commands (DDL) like CREATE TABLE, and so on, as well as, more commonly, Data Manipulation Language commands (DML) like INSERT, UPDATE, DELETE, and SELECT.
4. If required, read data from the ResultSet, possibly iterating over it with the next() method if more than one row is returned in the set by the database server. Alternatively, do something else with the ResultSet, such as a moveToInsertRow(), updateXYZ(), or insertRow(). This step may also be skipped if no data (of interest) was returned from the Statement.
5. Close the ResultSet, if one was used.
6. Close the Statement.

7. Close the Connection.

Furthermore, production code must do error handling via correct SQLException treatment (throws clause, catch and abort, catch and rethrow, and so on), as well as SQL warnings via the getWarnings() method of Connection, Statements, ResultSet, and so on.

The UML sequence diagram in Figure 12-1 illustrates the relationship between the major JDBC classes and interfaces.

Figure 12-1. UML sequence diagram illustrating the major JDBC interfaces.


Application
ds DataSource
aConnection Connection
aStatement Statement
data ResultSet
1: getConnection():java.sql.Connection
2: createStatement():java.sql.Statement
3: executeUpdate("INSERT INTO BOOK VALUES ('Jack', 1234)"):int
4: executeQuery("SELECT FROM BOOK"):java.sql.ResultSet
5: getString("AUTHOR"):java.lang.String
6: *[while] next():boolean
7: close():void
8: close():void
9: close():void

12.1.2 JDBC history

The java.sql package, which provides the core of the JDBC API, was introduced as part of the core Java 1.1 API around January 1997.[2]

[2] The original JDBC API (JDBC 1.0) was initially available as an add-on to Java 1.0.

In May 1998, Sun released JDBC 2.0 with version 1.2 of the Java 2 API and added a number of new classes to the java.sql package, including scrollable and updateable result sets, batch updates, BLOB and CLOB support, and support for storing objects in their serialized form in relational database fields.[3] It also introduced the javax.sql standard extension.

[3] The ResultSet.getObject() and PreparedStatement.setObject() method do NOT perform any object/relational (O/R) mapping in any known JDBC implementation; they merely serialize and store a binary byte stream in BLOB or JAVA_OBJECT type columns, on which no query or referencing is possible.

The javax.sql package includes additional features such as the RowSet class for treating database query results such as JavaBeans, the DataSource interface used in connection with pooling and to obtain connections via JNDI. The package also provides support to allow JDBC clients to interact with the Java Transaction API (JTA) and perform units of work within existing (possibly distributed) transactions.

12.1.3 New features in the JDBC 3.0 specification

The latest JDBC specification at the time of this writing is version 3.0, which was finalized in October 2001, and is also included in the "Merlin" version 1.4 (JDK 1.4.x) of the Java 2 API. It does not aim at adding any revolutionary new concepts (such as perhaps O/R mapping, non-relational data sources, XML, or anything like that) to the JDBC API. It does, however, provide numerous detail enhancements and new capabilities that introduce more flexibility and control for developers. These include, in order of perceived importance, the following:

- New Savepoint interface, which contains new methods to set, release, or roll back a transaction to designated savepoints; e.g., Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1"); (... do some other work ...) conn.rollback(svpt1);

- New configurable properties for the ConnectionPoolDataSource interface that can be used to describe how PooledConnection objects created by DataSource objects should be pooled. Furthermore, the specification documents show how prepared statements are pooled and reused by connection pools.

- A means of retrieving values from columns containing automatically generated values. After appending the Statement.RETURN_GENERATED_KEYS flag to the execute(), executeUpdate(), or prepareStatement() methods, the method Statement.getGeneratedKeys() can be called and retrieves the value of such a key, as a ResultSet object with a column for each automatically generated key.

- Methods to alter the data contained in BLOB and CLOB objects. Also added the updateBlob(), updateClob(), updateArray(), and updateRef() methods to the ResultSet interface.

- The data type java.sql.Types.DATALINK, allowing JDBC drivers to store and retrieve references to externally managed data—for example, data in a file outside the data source.

- Support for mapping SQL-99 user-defined types (UDTs), structured and distinct types to classes in the Java programming language. The future will tell whether relational database vendors will consistently implement this in their products and JDBC 3.0 drivers. JDO implementations could leverage this new feature and transparently map a persistent JDO-enhanced class to an SQL-99 UDT. Developers wishing to use this feature today at the JDBC level have to implement the SQLData interface and write to SQLOutput and SQLInput streams manually.

- Updated and new DatabaseMetaData methods for retrieving SQL-type hierarchies.

Other minor enhancements include:

- New interface ParameterMetaData, which describes the number, type, and properties of parameters to prepared statements.

- New method getMoreResults(int) to ResultSet, which takes an argument that specifies whether ResultSet objects returned by a Statement object should be closed before returning any subsequent objects.

- Methods to allow a string to identify the parameter to be set for a CallableStatement object.

- The ability to specify the holdability of a ResultSet (cursor in DB) object.

- The data type java.sql.Types.BOOLEAN (logically equivalent to BIT).

- Methods to retrieve the object referenced by a Ref object; also the ability to update a referenced object through the Ref object.

Last but not least, the JDBC 3.0 specification now includes a chapter clarifying the relationship between the JDBC SPI (Service Provider Interface) and the Java Connector architecture JCA.[4] JDBC driver vendors can now package their drivers as resource adapters and get all the benefits of pluggability, packaging, and deployment in Connector RAR File Format, and provide wrappers to implement the JCA system contracts. For more on JCA, see Chapter 8.

[4] The Java Connector Architecture (JCA) defines a standard way to package and deploy a "resource adapter" that allows a J2EE container to integrate its connection, transaction, and security management with those of an external resource.

12.1.4 Vendor-specific JDBC API extensions

Although JDBC currently is the universally accepted API to access relational databases from Java, some vendors require API extensions that go beyond the JDBC standard. The generally accepted way of doing this is by casting an object returned from a standard JDBC method to a vendor-specific class, as in this example:

```
ResultSet resultSet;
(...)
java.sql.Blob blob = resultSet.getBlob("BINARY");
if(blob instanceof oracle.sql.BLOB)
   return ((oracle.sql.BLOB)blob).getBinaryOutputStream();
```

12.2 Example: Storing Objects in a Relational Database Using JDBC

To put theory into practice, a complete JDBC mini-application example is outlined here. It uses a domain model around books and authors similar to elsewhere in this book, and creates, modifies and displays books and authors.

The complete source code and running application are available from the book's Web site; only relevant methods of interest are shown and briefly discussed here. The physical data model used is similar to one shown in the next section and is created by the following DDL:

```
CREATE TABLE BOOK(
     pk BIGINT NOT NULL PRIMARY KEY,
     name VARCHAR(60), isbn VARCHAR(14), published DATE );

CREATE TABLE AUTHOR (
     pk BIGINT NOT NULL PRIMARY KEY,
     name VARCHAR(60) );

CREATE TABLE AUTHOR_BOOK (
     book BIGINT NOT NULL,
     author BIGINT NOT NULL );
```

First, let's write a method to insert some records into the BOOK table:

```
protected long insertBook(String name, String isbn,
       int year, int month, int day) throws SQLException {

    Connection con = null;
    try {
        con = this.getConnection();

        long pk = this.getNewPK();

        PreparedStatement createBookStmt;
        String s = "INSERT INTO BOOK VALUES (?, ?, ?, ?)";
        createBookStmt = con.prepareStatement(s);

        createBookStmt.setLong(1, pk);
        createBookStmt.setString(2, name);
        createBookStmt.setString(3, isbn);

        createBookStmt.setDate(4,
                        this.newDate(year, month, day));

        createBookStmt.executeUpdate();
        createBookStmt.close();
        con.commit();
        return pk;

    } catch (SQLException ex) {
        if (con != null) {
            try {
                con.rollback();
            } catch (SQLException inEx) {
                throw new Error("Rollback failure", inEx);
            }
        }
        throw ex;
    } finally {
        if (con != null) {
            try {
                con.setAutoCommit(true);
                con.close();
            } catch (SQLException inEx) {
                throw new Error("Rollback failure", inEx);
            }
        }
    }
}
```

Following is a method that would insert a record into the AUTHOR table and associate (join) the author with a book via the AUTHOR_BOOK table:

```
protected long insertAuthorForBook(String name,
                        long bookPK) throws SQLException {

  Connection con = null;
  try {
     con = this.getConnection();
     long authorPK = this.getNewPK();

     PreparedStatement authorStmt;
     String authorSQL = "INSERT INTO AUTHOR VALUES (?, ?)";
     authorStmt = con.prepareStatement(authorSQL);
     authorStmt.setLong(1, authorPK);
     authorStmt.setString(2, name);
     authorStmt.executeUpdate();
     authorStmt.close();

     PreparedStatement joinStmt;
     String jSQL = "INSERT INTO AUTHOR_BOOK VALUES (?, ?)";
     joinStmt = con.prepareStatement(jSQL);
     joinStmt.setLong(1, bookPK);
     joinStmt.setLong(2, authorPK);
     joinStmt.executeUpdate();
     authorStmt.close();
     con.commit();

     return authorPK;

  } catch (SQLException ex) {
     if (con != null) {
        try {
           con.rollback();
        } catch (SQLException innerEx) {
            throw new Error("rollback failed", innerEx);
        }
     }
     throw ex;
  } finally {
     if (con != null) {
        try {
            con.setAutoCommit(true);
            con.close();
        } catch (SQLException inEx) {
            throw new Error("rollback failed", inEx);
        }
     }
  }
}
```

Last but not least, here is how we would retrieve and print a list of all books, with their respective authors below them. There are several ways to achieve this in SQL, including inner joins, nested queries, or group by clauses. This method uses an inner join:

```
protected void displayBooksWithResultSet()
                                 throws SQLException {

    Connection con = this.getConnection();

    Statement stmt = con.createStatement();
    String sql = "SELECT * FROM BOOK";
    ResultSet rs = stmt.executeQuery(sql);

    while (rs.next()) {
        String name = rs.getString("name");
        Date published = rs.getDate("published");
        String isbn = rs.getString("isbn");
        long bookPK = rs.getLong("pk");

        System.out.print(name);
        System.out.print(" (" + isbn + "), ");
        System.out.println(published.toString());

        Statement authorStmt = con.createStatement();
        sql = "SELECT NAME FROM AUTHOR A, AUTHOR_BOOK AB" +
          "WHERE A.PK = AB.AUTHOR AND AB.BOOK=" + bookPK;
        ResultSet autorRs = authorStmt.executeQuery(sql);
        while (autorRs.next()) {
            String author = autorRs.getString("name");
            System.out.println("\t" + author);
        }
        autorRs.close();
        authorStmt.close();
    }
    rs.close();
    stmt.close();
    con.close();
}
```

At this point, you may wish to compare this JDBC example to JDO code achieving the same purpose shown earlier in the book. The following section discusses some of the differences in more detail.

12.3 Comparison of JDBC and JDO

After seeing a JDBC example in this chapter and JDO throughout the book, you probably agree that the two APIs definitely "feel" very different to use. So what really makes this difference?

JDO is a high-level API for transparent object persistence. The programmer can write Java code and work with Java objects as always. The data of the objects is almost automatically made persistent and can be reused in a different run of the application, can be queried, and possibly can be accessed by other applications. All this is achieved in a relatively transparent manner.

You may have noticed that the JDO API does not make any explicit reference on what the specification calls the "datastore," the place where persistent objects are ultimately stored, queried, and retrieved from. In some cases, a developer chooses to use a relational datastore. In this case, the JDO implementation generates the required SQL statements and translates JDOQL queries into SQL. This specific combination is often referred to as "object/relational mapping." The developer uses objects, and some framework transparently maps access to objects into the relational world.[5] Object/relational (O/R) mapping tools for Java existed before JDO; see the second part of Appendix E of this book. Before JDO was standardized, each tool used its own proprietary persistence API.

[5] Names of tables and columns in the relational database are usually set in an XML mapping descriptor. This allows you to easily change SQL table and column names, independently of the Java class and field name. The current JDO specification (1.0) does not standardize the format of this mapping description, in the interest of staying 100 percent datastore-neutral, and each vendor uses a similar but slightly different syntax. This may change with future releases of the JDO specification.

This datastore independence allows developers to switch to a different datastore, e.g., an object database, or an in-memory only store, or maybe a pure XML database.[6] Because JDO completely hides the SQL coding from the programmer, SQL generation is happening behind the scenes. There usually is no SQL in the business application itself.

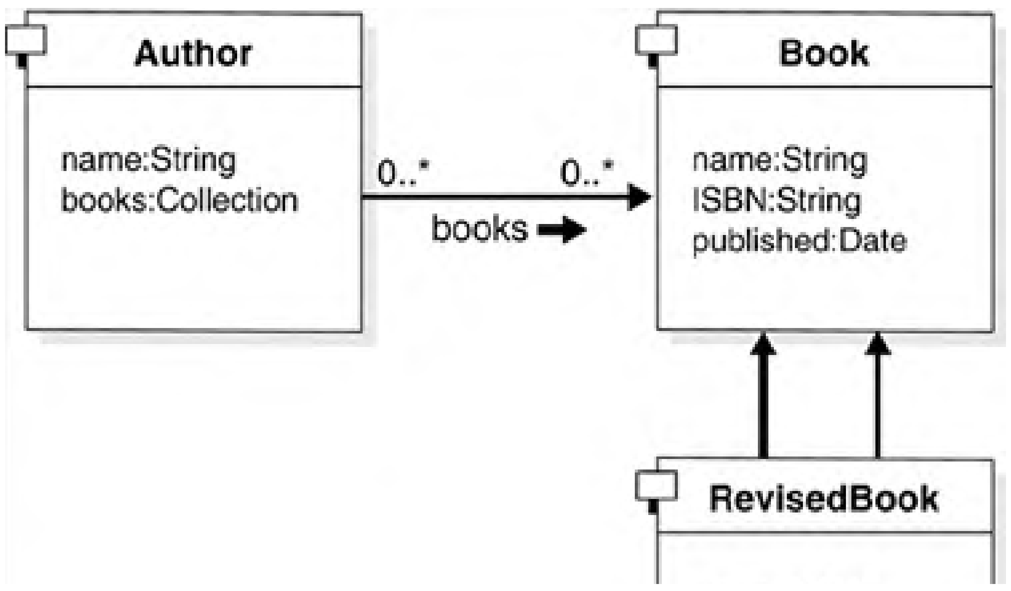
[6] At the time of this writing, the authors were not aware of any commercially available JDO implementation using a pure XML database as a datastore; however, this may be only a question of time.

JDBC, on the other hand, is a lower-level API designed specifically to access relational databases, and at no point attempts to "hide" the relational database in any way. In JDBC, developers explicitly code things like "update this information stored in this column of that table." This is done using another language: SQL. Statements written in the SQL language are spread throughout the Java code as Strings and are sent to a RDBMS via JDBC. SQL really has nothing to do with the language that was chosen to develop an application in Java. This "disjoint" is often referred to as "impedance mismatch."

The two APIs were clearly designed with different goals in mind, and although both serve to "persist data" on a high-level, in practice they use fairly different means of achieving this goal. In this way, there is limited scope for direct comparison.

The following sequence of diagrams further illustrates the difference. The UML class diagram in Figure 12-2 shows some domain classes of a typical small application.

Figure 12-2. UML class diagram.



This illustration represents the Java object-oriented view of the business domain: A Book has an ISBN number, here modeled as a field of type String, so the Book class generally has setISBN() and getISBN() methods. Some books are special; they are subclasses of general books, so there is a RevisedBook Java subclass that inherits from Book. Authors publish books, so the Author class has a books field with, for example, a getBooks() method that returns some java.util.Collection of persistent Book objects. A new author can created using the new Author() constructor, just like any other object in Java is instantiated:

```
Author author1 = new Author();
author1.setName("Donald Duck");
pm.makePersistent(author1);
```

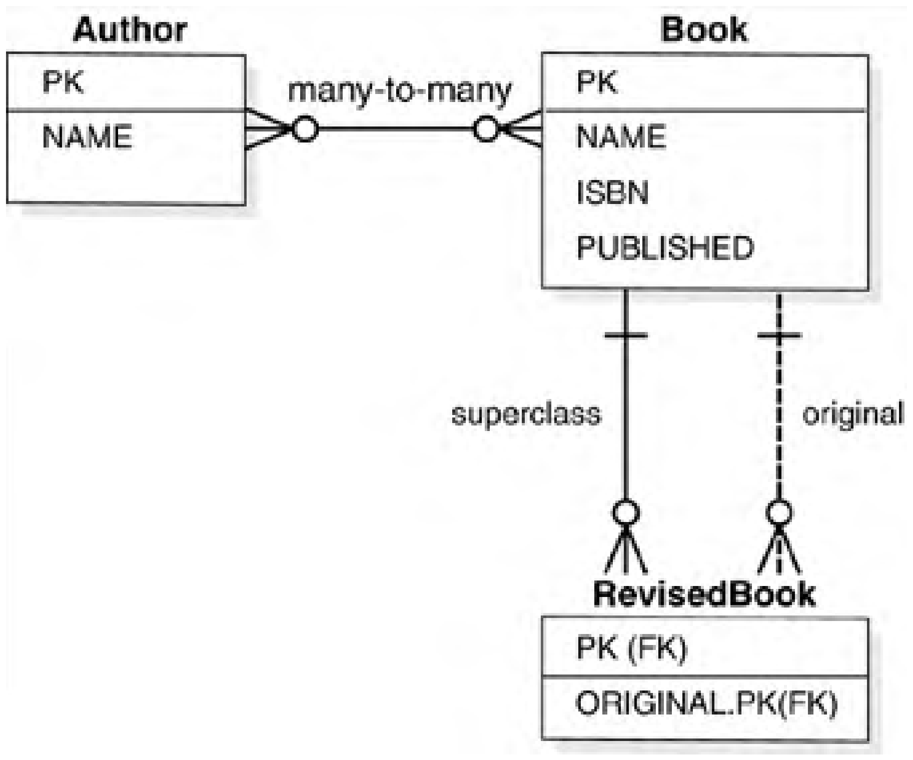
Using JDO, working with authors and books in a persistent database means working directly with these persistent domain objects. The JDO API is used for calls like makePersistent() and executing queries expressed via a query language (JDOQL) that is syntactically very close to Java code syntax. JDOQL can, for example, use navigational expressions and familiar methods like Collection.contains(), Collection.isEmpty(), String.startsWith(), and so on. For example:

```
String filter = "original.name == \"Donald Duck\"";
Query query = pm.newQuery(RevisedBook.class, filter);
Collection result = (Collection) query.execute();
revisedBook = (RevisedBook) result.iterator().next();
query.close(result);
```

With JDO, it really doesn't (have to) go much beyond the few lines shown. Write a domain model with Java classes, create new persistent instances, work with them as if they were good-old plain simple Java objects, and use JDOQL to find collections of them.

Many JDBC-only based business applications today also have domain data object classes similar to the above example in the Java application. However, to work with authors and books in a persistent database in this case means working on a different view, that of relational databases. The logical ER diagram in Figure 12-3 illustrates how authors and books live in an RDBMS.

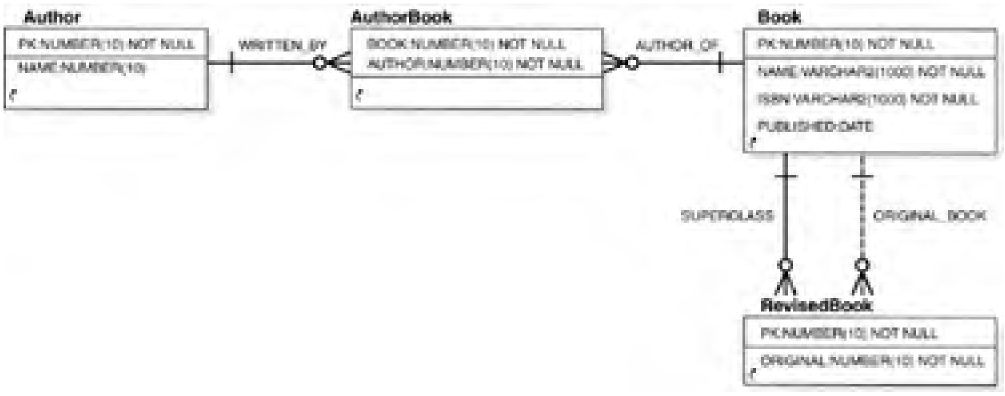
Figure 12-3. Logical ER (entity-relationship) diagram.



Although this logical diagram may look fairly similar to the UML class diagram, JDBC code works with SQL at the level of a physical ER diagram, where what's expressed as a "many-to-many" relationship in a relational model becomes the intermediate AUTHOR_BOOK table, and where Strings are VARCHARs.

In Figure 12-4, there are no object relationships or queries "following" an object reference. There are only tables, with rows and INSERT, columns and SELECT, object references that become NUMBER types, and SQL queries to be formed with JOIN and other SQL keywords. In which model would you rather work?

Figure 12-4. Physical ER (entity-relationship) diagram (e.g., for Oracle).



In review, JDO scores much higher on programmer convenience. JDO hides SQL from the programmer; that is, a developer using the Java programming language does not need to learn SQL. On the other hand, the JDBC API provides greater control by giving programmers direct access to the relational database access.

12.3.1 Feature comparison of JDBC versus JDO

This section provides a brief one-on-one feature comparison of JDBC and JDO, insofar as the two APIs can actually be compared. They were designed for different purposes, as is evident in Table 12-1.

Table 12-1. Feature Comparison of JDBC and JDO

Feature	JDBC (alone)	JDO
Transparent object persistency	No	Yes
Object references support	No	Yes
Persistence by reachability	No	Yes
Class inheritance support	No	Yes
Transactional (JTS/JTA)	Yes	Yes[*]
Lazy/deferred/partial loading	No, manual	Yes
Locking	Manual	Automatic[*]
Standard Java API	Yes, java.sql (core package)	Yes, javax.jdo (extension package)
Platforms applicable	J2SE and J2EE [‡]	J2SE, J2EE, and J2ME ready
Formally part of J2EE	Yes	No, but... not an issue[†]
Maturity	Fairly mature	Recent, but rapidly adopted
Developer community	Large	Small, but growing
Compile-time type checking	Limited	Better
Connection pooling possible	Yes	Yes
Data caching	None	Usually yes, various[*]
Scalability	Limited	Cluster with distributed cache[*]

Datastores	Relational only	Independent API, today often relational or native object, basically completely open, including EAI
Query language	SQL only	JDOQL, possibly others (SQL[*])
DDL / metadata access	SQL / yes	No, datastore dependant
RDBMS views, stored proc.	Yes, sure	No, but[*]
Renaming tables and columns	Often hard-coded	Easily changed in descriptors[*]
Persistence-related code	Often scattered and intermixed with application business logic[§]	Separate by design

[*] JDO implementation-dependent.

[‡] JDO fits into the larger J2EE picture as a JCA resource adapter.

[†] JDBC cannot be used on the Java Micro Edition (J2ME) platform. However, a similar but much simplified vaguely similar API is available on J2ME devices.

[§] Unless elaborate design patterns such as DAO are implemented. Although it is technically possible to cleanly separate persistence-related code from domain model and business logic classes, the point is that with direct JDBC programming, this requires good architecture and discipline. With JDO, the separation comes for free, out of the box.

It is only natural that JDO, as an emerging Java technology, has a smaller experienced user base than JDBC, which has been around since almost the beginning of Java and has been widely used. However, as mentioned, JDO merely standardizes an API for transparent object persistency, for which products have existed for some time. Senior enterprise developers often have knowledge of previous O/R products and are able to adopt JDO easily. Junior developers usually find JDO easy to adopt because it's intuitive and by design extremely close to "normal" Java. Sometimes people realize that they have actually been writing an in-house persistent framework and are keen to jump on a now standardized API.

In summary, for many Java applications, technologies that solve the impedance mismatch between an Object Model and a Data Model are of strong interest, and are often redeveloped in one form or the other in many projects. Transparent persistency, persistence by reachability, object/relational (O/R) mapping, lazy/deferred/partial loading, and caching are highly desirable features for many applications. JDO-compliant implementations provide these features out of the box and are an interesting alternative to direct JDBC programming.

12.3.2 Misconceptions about JDBC and JDO

Because JDO takes a different approach to persistence than JDBC, misunderstandings often come up in related discussions. The next few pages shall address some of them, namely: Is JDO a replacement for JDBC? Is JDO slower/faster than JDBC? How datastore independent is JDO really, and how easy is it to switch between different types of relational datastores?

12.3.2.1 JDO is not a replacement for JDBC

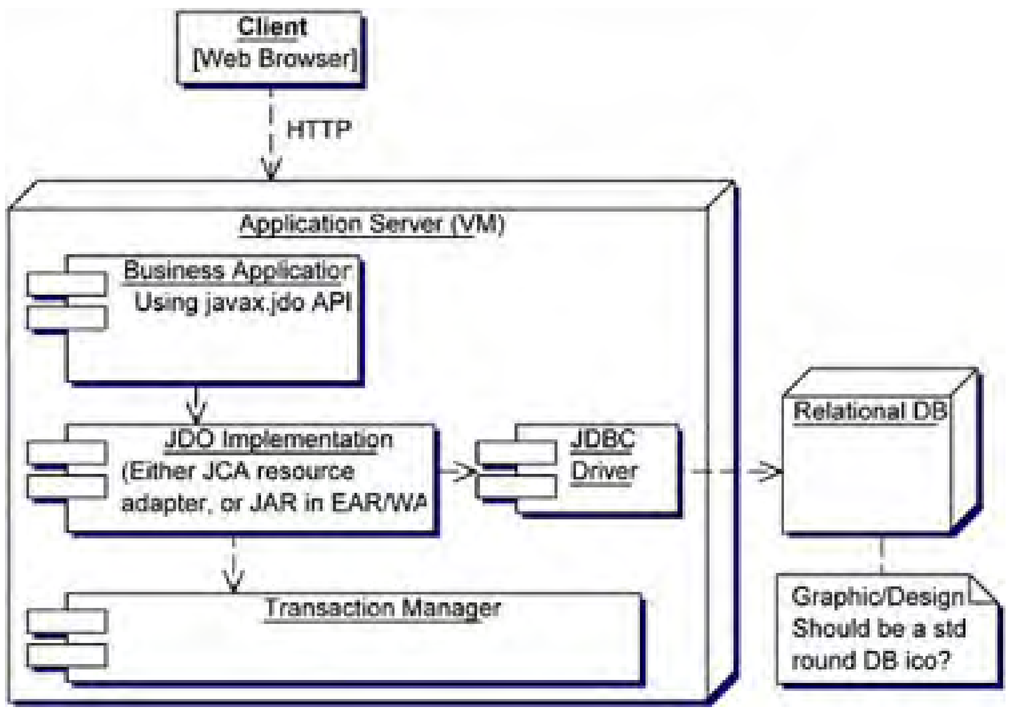
You may be thinking, "Using JDO, I can completely forget about JDBC!" We say: Be careful. Although it is true that JDO allows you to develop simple applications entirely without reference to the JDBC API in the application code, you may want to think twice before throwing away your JDBC and SQL documentation.

First, as long as you are in an O/R mapping scenario where you want to use a relational database as the final datastore for your JDO-based application, you or at least some people on a project's development team will still be dealing with Connection URLs or Data Sources, JDBC drivers, Connection Pools, and the like. In this scenario, JDO does not really replace JDBC; rather, it isolates business application developers from its details, allowing them to code at a higher level, while a JDO implementation internally still uses JDBC and a JDBC driver to access the relational datastore at the lower level. (Think of this as a bit like using JSPs at a higher level that simplifies Servlet programming, even though JSPs are still translated into Servlets; similarly, in JDO many operations ultimately lead to JDBC calls.) Of course, if you use a non-relational datastore, this will be different.

Second, for more sophisticated enterprise applications, sometimes a mix of JDO with special cases relying on some JDBC is required. You may well hang onto those PreparedStatements and ResultSets for specific modules of your applications for some time to come. Regarding mixed use, see the section "When to use JDBC" later in this chapter for more details. Last but not least, it is certainly useful to have a sound understanding of JDBC should you need to troubleshoot any issues with an O/R-based JDO implementation.

Figure 12-5 shows a typical system architecture for a solution running inside an application server, where some business logic uses the JDO API, which in turn uses the JDBC API internally.

Figure 12-5. Layers of a typical Web-based application when using JDO on a relational JDBC-based datastore within an application server.



It is best to think about JDO as a complementary technology to JDBC, both with unique strengths, usable by programmers with different skill sets and applicable to projects or modules of projects with different development objectives.

12.3.2.2 Performance

Now you may be thinking "JDBC is faster than JDO." But you'll find that is not necessarily so. To compare the performance of a JDO-based application with a JDBC-based application, much depends on the actual application scenarios, specific use-case, and details of how JDBC is used. Here are some of the factors that contribute to the performance discussion:

- **Mapping:** When reading and writing persistent objects from and to a datastore, a JDO implementation is usually required to do some sort of mapping between the Java instances and the objects in the datastore. For example, when using a relational JDBC-based datastore, at some point in time a JDO implementation will have received a ResultSet internally and will want to set the fields of a persistent object from it. Although this mapping always consumes time, what's more important is that during the mapping, data that the application may not need may unnecessarily be read or written and mapped. Depending on the sophistication of an alternative JDBC-based application, this point can actually come out in favor of JDO, instead of against it (see the next item). Furthermore, this applies to relational datastores; other JDO implementations that are, for example, using an object database as a datastore may have a simpler (or no) mapping function, and this point does not apply. In the overall performance picture, mapping does not generally lead to a significant performance penalty, taking into account that many JDBC-based applications include some sort of mapping, sometimes more hidden (i.e., spread all over the code) or more explicit (if using home written) in application code.

- **Unnecessary reading:** When reading data from a datastore, performance obviously depends on how much data is read and how. A JDBC-based application may use a specific SQL statement for each specific read access, or some more generic (SELECT *, JOIN in case we may need the data later) construct. JDO implementations, on the other hand, are able to fetch data only from a datastore that is actually required, thus minimizing database hits and network traffic. Today, this is usually implemented using a technique referred to as Deferred Reading, wherein a JDO implementation fetches some data only initially (see the default-fetch-group field of the field tag in

JDO XML metadata descriptors) and fill in other fields only later, only if actually accessed. More sophisticated mechanisms can be envisioned—e.g., a Prefetch API that is being discussed for future versions of the JDO standard. This would allow an application to specify a policy whereby instances of persistence-capable classes are prefetched from the datastore when related instances are fetched. Similarly, automatic detection of reading patterns to optimize fetching may be available in advanced JDO implementations. The bottom line of this point is "it depends." Performance could be in favor of JDO versus a simple generic JDBC application, or against JDO versus a sophisticated JDBC application with different handcrafted specific queries. However, JDO generally wins for simplicity: For the same performance, a JDO application developer does not have to be concerned with datastore fetch details. With JDBC, the developer writes the SQL code and has to think ahead about unnecessary reads.

- **Unnecessary writing:** When writing data to a datastore, most JDO implementations today are able only to write modified fields, because they automatically keep track of write access to fields of persistent objects. Furthermore, if optimistic transaction management is used, many JDO implementations are able to "batch" writes, to avoid writing fields that are changed again within the same transaction, to avoid any datastore roundtrip if a transaction is aborted, and other optimizations based on leveraging the in-memory object model and cache. These points can again contribute to minimizing database cycles and network traffic.

- **Caching:** By far, the most important aspect of performance discussions between JDO and straight JDBC is caching, as any experienced developer will confirm. Most JDO implementations keep a global cache of objects, according to different implementation-specific and often configurable caching strategies. Therefore, many JDO implementations do not read and map an object previously fetched again, because it is kept in a memory cache in the JVM. For a JDO application developer, this caching is fully transparent; there is no "load this from cache instead of from datastore," and so on, code in the application. Again, the performance impact of this depends on the actual application, amount of data, available memory, and use-case; in general, however, this leads to a huge performance benefit in favor of JDO. The usual JDBC application sends a new query statement each time data is required. Using prepared statements and similar "optimizations" are of minimal impact in the bigger picture of having an actual domain object cache in memory. There is little to no performance penalty for using a cache; however, there is an impact on memory requirements.

Of course, all the above points regarding performance could be implemented in one form or the other in a JDBC-based application as well; they are not technically specific to JDO only. However, your general JDBC applications often won't implement caching, lazy load/write, and so on, strategies. If they do, you have in fact developed an in-house persistence layer that's similar to what's now available in various JDO implementations, according to a standard API. See the "build versus buy" discussion in Chapter 2 for more about this.

12.3.2.3 Database independence

Okay, maybe you're thinking, "JDBC abstracted differences between relational databases, and now JDO abstracts even access to non-relational datastores." Let's have closer look at the details:

- Although the SQL 92 standard exactly defines the syntax of the SQL language, and a simple "`SELECT * FROM table`" certainly works across all relational databases, vendors often implement more advanced requirements differently. Sometimes seemingly trivial issues like maximum table and column name length can cause troubles in projects that need to work across different relational databases. Often, such differences end up being accounted for in the Java code of the business application or in some database abstraction layer written in-house.

  Although JDBC driver classes abstract (necessarily, because there is no standard) the product-specific client/server connection protocol details (e.g., Oracle OCI, MySQL port 3306 protocol, and so on), JDBC does not make any attempt to "translate" SQL code.[7] JDO implementations, however, translate the datastore-independent JDO query language (JODQL) according to the underlying datastore, and can and do take slight SQL differences into account and map accordingly (for example, differences in `OUTER JOIN` syntax between major relational database vendors).

  [7] The JDBC 3.0 Specification suggests that "Because scalar functions are supported by different DBMSs with slightly different syntax, it is the driver's job either to map them into the appropriate syntax or to implement the functions directly in the driver." In practice, however, most JDBC drivers—even from established vendors—do not seem to respect such requirements as of today.

- The O/R mapping-based JDO implementations often provide this database product abstraction layer; it is in fact not unusual to find some sort of implementation-specific "database type" configuration property. Most JDO implementations also list which relational databases their implementation has been explicitly tested and approved against. So to switch from one relational database to another—for example, from a simple Java embedded database running in the same VM to a full-fledged remote relational database from one of the leading vendors—the same JDO implementation, configured differently, can often be used without too much trouble. Although such claims naturally sound suspicious to anyone who has tried this with JDBC and become disappointed, this does generally work fairly well with JDO in practice.

For these reasons, using different relational databases from one project is easy with JDO, which hides possibly database vendor-specific SQL in favor of a data-store-neutral query language, than with JDBC. If non-relational datastores, such as object databases, are a possible future target, JDBC is not an option as persistence API anyway.

Switching JDO Implementation to Switch Datastore?

Sometimes a switch from one type of datastore to another one—for example, from a relational datastore to a scalable object database datastore—requires a switch in the JDO implementation that you have chosen.

As the market stands today, JDO implementations are relatively clearly split into "for relational database" and "for (often the same vendor's) object databases" products.

Of course, switching JDO implementation without rewriting code is possible only if the standard JDO API has been used; using non-standard JDO extensions "locks" you into that particular implementation. As long as only the standard API has been used, switching implementation is generally relatively easy, because the JDO specification is specific and strict, and implementations can pass a technology compatibility kit (JDO TCK) from Sun.

12.3.3 When to use JDBC after all

Some applications may need to implement use-cases where JDBC is required after all. Often, these are not your trivial example applications, but some medium to complex solutions. Broadly, this situation often falls into one of these cases:

- Specific SQL queries that cannot be expressed in JDOQL.
- Need to use existing, often vendor-specific, relational datastore features.
- Data-centric, processing intensive, performance-sensitive (batch) applications.
- Existing relational database schemas, potentially complex and not modifiable.

12.3.3.1 SQL queries

It is important to understand that the JDO standard query language, JDOQL, does *not* attempt to be a drop-in replacement for SQL. Its aim is not to be capable of expressing any SQL query one can possibly think of.

The most common and widely used SQL statements—such as INSERT, UPDATE, SELECT, and DELETE—of course have their equivalents in the JDO API, as well as many SELECT options such as WHERE and ORDER BY have equivalents in JDOQL. Similarly, the BEGIN, SET TRANSACTION, COMMIT, ROLLBACK, and related keywords have equivalents as JDO Transactions.[8]

[8] These transaction-related statements are never issued as such even in JDBC; the JDBC driver usually issues these (or uses another DB-proprietary transaction demarcation mechanism) after each individual SQL statement as soon as it is complete, if auto-commit is enabled, or when the client code calls Connection.commit() or Connection.rollback() if auto-commit is disabled on the Connection.

Other SQL constructs, however, have no direct equivalent in the JDO API or JDOQL. This is "a feature, not a bug," in the sense that JDOQL was conceptually not meant to suit such needs. And although minor extensions may be made in the future, most of such constructs probably will never be expressible using JDOQL. These include the following:

- SELECT options like GROUP BY and HAVING (and aggregate functions such as AVG, COUNT DISTINCT, MAX, MIN, SUM) have no direct equivalent in JDOQL.
- SELECT options JOIN, IN, and EXISTS as well as subqueries do not directly translate into JDOQL. However, sometimes these SQL operations, particularly IN and JOIN, may not be required to express the same goal in JDOQL, simply because it's closer to the object model. A good example for this is JOIN operations in typical relational database applications on helper tables used to express multi-multi (M:N) relationships. In a JDO model, this is expressed as Collection fields of the respective data object, and queried using contains() operations or simple navigation through a reference field, which gets translated to the correct JOIN operations in SQL, if running on a relational datastore. At the JDOQL level, explicit mention of the intermediate tables and JOINs is simply not required anymore. Still, sometimes such an SQL query cannot be translated into JDOQL.
- Unions and intersections (SELECT … UNION/INTERSECT/EXCEPT SELECT constructs) cannot be expressed directly in JDOQL. These SQL keywords are not formally part of the SQL standard. Similarly to the above comment, the same goal can often be achieved differently using JDO. Specifically, sometimes such constructions are used to "hack" in SQL what would be correctly modeled as class inheritance in a JDO application.

- Some operators, most importantly string functions like the SQL LOWER, UPPER, SUBSTRING, TRIM, and LIKE, as well as case-insensitive string comparisons, are not standardized by the JDOQL specification, which mandates only support for the String.startsWith() and String.endsWith() methods. However, O/R-based JDO implementations frequently provide extensions to use these operators. Make sure that you understand and highlight in your code when you are using such implementation-specific extensions. Future releases of the JDO specification may define more supported String methods for JDOQL query filters.

- SQL commands, like CALL for stored procedures or TRIGGER-related functions, are not covered by JDO. These statements are often database-specific.

- Last but not least, schema manipulation statements such as CREATE/ALTER/ DROP of TABLE, INDEX, and so on, are not directly supported by the JDO specification. These Data Definition Language (DDL) statements are not usually executed at the runtime of an application, but during development and build. O/R-based JDO implementations usually provide tools that create and/or upgrade the required database schema, which do issue these statements.

When looking at an existing application with SQL code and evaluating JDO migration, it is important to work on a query-by-query basis, before prematurely concluding something like "this application has queries that are too complex and that cannot be expressed in JDOQL."

If, after all, your application does requires some SQL, this is not necessarily a reason for the entire application to be written using JDBC instead of JDO. In practice, there are three basic choices to use SQL queries in JDO applications:

- Use JDBC API to execute SQL query in an otherwise JDO-based application. If transactional semantics across calls into the two APIs need to be ensured, this is possible to achieve.

- Use JDO standard to send off a pure SQL query via the JDO API. As seen in Chapter 6 about JDOQL queries, the JDO specification deliberately lets a developer specify the query language when creating a JDO Query object, and although it usually is JDOQL, several implementations available today offer the possibility to "pass through" full SQL code. (Note that the expected result set still generally must be a collection of persistent objects.)

- Use JDO implementation-specific extensions to mix JDOQL and SQL syntax in one and the same query filter. Similarly, use a JDO implementation-specific API to develop custom query expressions. For example, this can be used with some implementations to "pass through" relational datastore-specific SQL query hints or operators, if really needed.

12.3.3.2 Specific relational datastore features

Regarding the use of specific relational datastore features, often vendor proprietary SQL extensions or entire applications, consider this example: A document/content management solution wants to use JDO. It will have domain classes to represent Document, Author, Version, and so on. However, it also needs to interface to a database-specific full-text indexing engine, for which vendor-specific SQL calls are required.

This in itself is not a reason to develop the entire application in JDBC; it is perfectly legal to use JDBC/SQL for that part of your application. As explained above, the choices are either to use SQL as an alternative query language from the JDO API, if the JDO implementation chosen permits this, or simply through good old real JDBC API calls. See below for an example of how to correctly use both APIs mixed to perform transactional work.

12.3.3.3 Data-centric applications versus domain-model centric

Other than these technical issues, it is sometimes also argued that although JDO is great for domain-model centric applications, other applications lend themselves more for JDBC than JDO. For example, heavily data-centric processing, including decision support, ad hoc query, and reporting applications may want to offload processing using existing relational database features.

12.3.3.4 Existing legacy relational database schemas

Some applications are required to work on an existing legacy database schema. If such database schemas are actually complex relational database "applications"—for example, using SQL views (instead of direct SELECT on underlying tables) and stored procedures (instead of direct INSERT) to access data—raw JDBC may be a better fit than JDO. Although some implementations do claim to actually support stored procedures, for example, it is generally not possible to "map" an arbitrary complex relational schema to an object world.

However, simpler and more straightforward existing database schemas may be understood and correctly mapped by a good JDO implementation. For example:

- **Identity mapping:** Just because the standard schema generator of your JDO implementation of choice generates an automatically incrementing sequence number as primary key does not mean that it could not also use a FirstName/LastName pair that models the primary key in an existing schema. This could be modeled with a multi-field Application Identity class in JDO.

- **Simple field mapping:** Although String fields usually are mapped to VARCHAR columns, int, to NUMBER columns, and so on, most JDO implementations can be flexible about this. For example, mapping a VARCHAR(1) column with 'Y' / 'N' in a table to a Boolean is usually possible. Often, more complex mappings (using application-provided translators) can be implemented.

- **Collection field mapping:** In relational modeling, either a one-to-many or a many-to-many mapping expresses what in a Java class model is usually written as a Collection-type field. With a one-to-many mapping, the persistent objects contained in the collection are linked to exactly one containing object (or none), but never more than one; this is modeled with a foreign key on the contained records associating to the containing record and corresponds to what UML refers to as an aggregate link. With a many-to-many mapping, the persistent objects added to the Collection-type field can be linked by many of the same objects; this is modeled with an intermediate table holding the foreign keys of both records. Because a pure Java class model does not (need to) distinguish between these two cases, most O/R-based JDO implementations by default use a many-to-many mapping for Collection-type field. However, most JDO implementations are perfectly able to work with a one-to-many mapping as well. (The relational one-to-one mapping is less ambiguous and simply corresponds to a persistent object reference in the respective class.)

- **Inheritance mapping:** Another area is relational schemas, which use the same primary-key values in more than one table, aiming at expressing some sort of "this record actually is an 'extension' of that record" relationship. Although this could be mapped into two separate classes with an association, such relational structures often map well into a class model using inheritance.

- **JDO specific tables and columns:** O/R-mapping implementations often require extra tables or columns—for example, a column on each table to keep an incrementing locking value used to implement optimistic transaction management, and one holding the name of the class of a persistence object used to properly interpret subclasses. If such columns are not part of an existing schema, some implementations permit disabling their usage if the required feature (e.g., optimistic transactions or subclassing) is of no interest.

Some O/R-based JDO implementations provide schema reverse engineering tools, promising to generate Java classes and JDO metadata with mapping information from an existing database schema. This could greatly simplify (or at least support) such a mapping process as outlined above, as opposed to manual configuration.

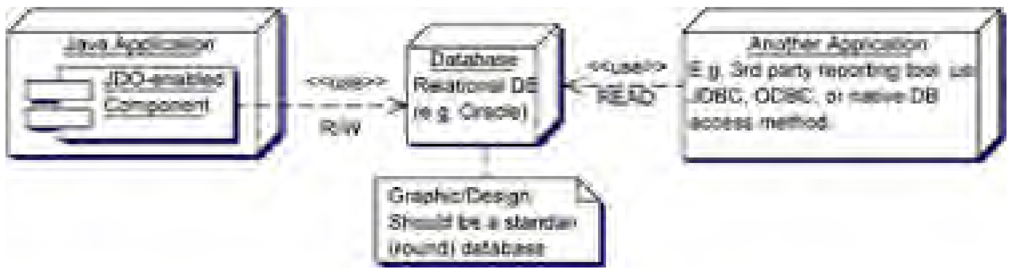
Last but not least, consider that it may be quite acceptable to enhance an existing schema for a new application without changing its structure. For example, the point on JDO-specific columns would really only apply if the legacy schema is completely non-modifiable, which may not always be the case. For example, slight additions to a database may be acceptable. Sometimes, even more profound schema changes of a legacy database may be perfectly acceptable. For example, if no legacy application continues to use it in parallel with a new JDO-based application, then the schema can be completely adapted and the data migrated, thus easily used with JDO anyway.

The above few paragraphs should have illustrated that even if a legacy schema cannot be changed, it may be worth investigating whether a JDO implementation can actually work with that schema. It should be expected that this area will see some interesting developments in the future, with JDO implementations becoming more mature and able to tackle some of the issues.

Data Access by Other Applications

Sometimes, it is not the application to be developed that is the element causing doubt between a JDO versus a JDBC choice, but rather existing external applications that should interface with the application to be written. Reporting tools are a typical candidate. Most industry-standard reporting tools today access data at the relational level, via JDBC or other relational APIs like ODBC or native drivers, and are only slowly beginning to gear up to be able to leverage object domain model representations of business data. However, this should generally not prevent a new application from being written in JDO, because if a relational datastore is chosen, other applications are still perfectly able to access the data. This system architecture is sketched out in Figure 12-6 below.

Figure 12-6. JDO application and another application both accessing relational datastore.


Java Application
JDO-enabled Component
<<use>>
R/W
Database
Relational DB (e.g. Oracle)
<<use>>
READ
Another Application
E.g. 3rd party reporting tool using JDBC, ODBC, or native DB access method.
Graphic/Design
Should be a standard (sound) database

This approach generally works well, particularly if only read access is required, as is usually the case for reporting tools. The case data structures from JDO need to be simplified for human report designers—for example, when using several tables for inheritance mapping—then creating some SQL views for the non-JDO application may simplify things.

However, if the other application requires write access to the same data, the scenario can get more complex because of cache synchronization. This issue, however, is not strictly related to JDO and would equally arise with a JDBC-based application that attempted to cache data. Possible solutions include disabled caches, which is perfectly possible with many JDO implementations, or some sort of manual cache refreshing or synchronization.

12.3.4 When to use JDO and JDBC in an application

You may sometimes wish to use JDO code mixed with some JDBC. Although such a mixed architecture should be avoided unless there is a good reason for it and it severely limits portability to another datastore (among other issues), it is technically possible. One approach is to use implementation-specific API extensions to use SQL instead of (or mixed with) JDOQL as query language, as described above. Another road, depending on why the application desires to use a mixed architecture, is to use a JDO implementation-specific method to obtain the JDBC Connection generally from a JDO PersistenceManager, among other things.

A different road, and, as opposed to the above, a portable and clean (albeit slightly more complicated) architecture is to perform work within the same transaction in a portable manner. Both JDO and JDBC calls can be made within a JTA UserTransaction in a managed environment. Chapter 11 explains this topic and its background, shows how to obtain a PM inside a JTA transaction, and presents a complete example illustrating how to use JDO alongside another transactional resource such as JDBC.

12.4 Summary

This chapter has reviewed JDBC, another API for persisting data in Java, specifically tailored to wide-spread relational databases. A brief overview of relational databases, a history of JDBC, an introduction to the new JDBC 3.0 API, and a source code case study showing the practical use of JDBC were presented in the beginning of the chapter.

The second part of this chapter then compared JDBC with JDO, discussed features and differences of each, and highlighted common misconceptions. The general conclusion drawn was that they are complementary APIs, each serving their own purpose well, and examples of applicability of each API were discussed.

On performance, it was shown that a JDO-based application need not be slower than JDBC-based ones, and in many cases they can actually be faster because unnecessary yet costly remote roundtrips to the datastore could be avoided.

On JDBC 3.0 innovations, the most interesting point from a JDO perspective may be the new support for mapping SQL-99 user-defined types (UDTs) to classes in the Java programming language. The future will tell whether relational database vendors will consistently implement this in their products and JDBC drivers, and whether JDO implementations are actually able to leverage this new JDBC 3.0 feature and transparently map persistent JDO-enhanced classes to SQL-99 UDTs, instead of having to implement the SQLData interface and writing to SQLOutput and SQLInput streams manually, as would be required with a pure JDBC 3.0 application.

Last but not least, we think that after you have read this book, the cases in which you want to use only JDBC, with no JDO, for a new usual general-purpose application, will be rare ones. Only special "processing" like applications for bulk loading, data transformation, and so on come to mind as possibly not suitable for JDO at all.

Chapter 13. Tips, Tricks and Best Practices

"Practice yourself what you preach."

—Titus Maccius Plautus (254 BC–184 BC)

This chapter provides some suggestions, usage guidelines, and best practices for using JDO successfully in typical real-world enterprise business applications.

13.1 Data Modeling

Although JDO does not prescribe any specific technical requirements for Java classes to be made persistent, such as the presence of certain methods or a common superclass, and most arbitrary Java classes can be made persistent using JDO, certain best practices for modeling persistent classes do exist.[1]

[1] As explained earlier, it is not entirely correct to state that all classes can be made persistent; certain system-type classes such as System, Thread, Socket, and File can in fact not be JDO persistence-capable. Most user-defined classes can be however, and so for most practical applications, this subtle point can be ignored. (One would rarely hold references to or subclass a Thread or similar class in a data object class.)

In essence, experience has shown that it is best to separate persistence-capable classes from the application-logic classes. One benefit with this approach is that the data model and more importantly the underlying data can be reused across applications.

13.1.1 Introduction

Translating this into a concrete coding best practice could sound like this: "Model business domain concepts as straightforward entity-type JavaBean-style[2] classes, with private fields and ubiquitous public 'getter/setter' (accessor/mutator) methods for fields." For example:

[2] Note that when we say JavaBean-style we do mean JavaBeans as per the JavaBeans specification (http://java.sun.com/javabeans/docs/spec.html), not Enterprise JavaBeans such as Entity and Session Enterprise beans. In some applications it may make sense to "façade" the plain Java data object classes with Session Enterprise beans, and a separate chapter deals with this in more detail. Whether the JavaBean-style classes have a Session Enterprise bean "façade" does not influence the design of them.

```
private String name;

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

All well-behaved JavaBeans should have a public no-arguments constructor and, as discussed earlier, so must all persistence-capable classes. Persistence-capable classes also always must have a no-argument public constructor, and often this is the only constructor. This is usually the default constructor.

13.1.2 Wrapper versus primitive types

For most applications, it is best to limit the types of fields. In addition to fields of type String as seen above, Java primitive (boolean, byte, short, int, long, float, double) or wrapper (Boolean, Byte, Short, Integer, Long, Float, Double) types are often used in persistent classes. Wrapper classes are preferable to primitive types if null values need to be expressed explicitly in the model.[3] For example:

[3] The XML metadata descriptor for such fields needs to have the null-value attribute of the field tag set to "none", which is the default for storing Java null values as null in the respective underlying datastore. (So just make sure it does not read **<field ... null-value="exception">** or **"default"**)

```
private Integer age;  // Note, CAN be null

public void setAge(Integer _age) { this.age = _age; }
public Integer getAge() { return this.age; }
```

If such fields are always set to a value with true business meaning, and never null (null as in the Java null value, not the zero '0' value), then Java primitive types may equally well be used, because support for persistence of them is mandated by the JDO specification and is well supported in implementations.

```
/** Confirmed is always 'true' or 'false' values, never
  * null. Use uppercase Boolean wrapper if tristate (true,
  * false, null) required for this application
  */
private boolean confirmed;

public void setConfirmed(boolean _c) { confirmed = _c; }
public boolean getConfirmed () { return this.confirmed; }
```

13.1.3 References to persistent objects

Last but not least, of course, references to other persistent classes are often used:

```
private Address address;

public void setAddress(Address _a) { this.address = _a); }
public Address getAddress() { return this.address; }
```

The setter/getter pair can be named either after the type of the persistent field, like Address above, to keep things simple in an obvious scenario, or after a field name clearly indicating the significance of the associated persistent object, particularly if there are several fields of the same type. For example:

```
private Book original;

public void setOriginal(Book _orig) { original = _orig; }
public Book getOriginal() { return this.original; }
```

In the above code snippets, both Address and Book are other persistence-capable classes. Note that, although fields of type java.lang.Object are technically permitted by the JDO specification and some but not all JDO implementations support this, it often makes more sense, makes code more robust, and in many cases is perfectly possible to name a concrete class, possibly a persistence-capable superclass of a completely persistent class hierarchy.

Furthermore, while the JDO specification again permits fields of Java interface types, instead of concrete implementing classes, this possibility is generally not often used. Remember that there is no notion of persistence-capable interfaces in JDO, so it is not possible, for example, to query on all persistent objects implementing a specific interface. More importantly, many JDO implementations (especially object-relational mapping-based ones) require that the type of the actual implementing class be specified in the XML persistence mapping descriptor. So, in practice, only instances of specific classes (or their subclasses) can be set for such fields; using concrete classes that do implement the interface specified by the field and method signatures will compile, but such JDO implementations will throw runtime exceptions if the classes do not match the contract established in the mapping descriptor. In summary, it is often preferable to use a concrete class, possibly a parent class in a hierarchy of persistent classes, instead of an interface.

Last but not least, although the referenced class is not strictly required to be another persistence-capable class, refer to the earlier example showing the use of JDOHelper.makeDirty() when modifying a reference to a non-persistence-capable class; most well-designed real-world domain models have only references to concrete other persistence-capable classes.

13.1.4 Collection-type fields

So far, only cases with references to other classes with "cardinality one" have been shown. However, most real-world applications will also have fields of "cardinality N" multi-references to other classes.

As seen in previous chapters, JDO offers many possibilities to model such a relationship. Java arrays (e.g., Book[]) could in theory be used, but they require special attention as we saw earlier. In practice, some sort of java.util.Collection is usually the way to go for most applications' persistent data models. People with relational data modeling backgrounds often find a Set the most familiar type, and support for the HashSet implementation is required in all implementations by the JDO specification.

However, note that a JDO-based persistence model can technically use Lists (which retain order) and Maps (key/value pairs) for associations as well. Support for Set is a required JDO feature and is supported by all implementations, while List and Map are optional. For example:

```
private Set books = new HashSet();

public void setBooks(Set _books) { this.books = _books; }
public Set getBooks() { return this.books; }
```

If the business requirements for your model mandate that insertion order of associations needs to be maintained, thus making you tempted to use a List, or you need a Map and many application domain models (especially those ported from a relational schema) do not, you may wish to read up on the documentation of and play with your JDO implementation of choice before drawing up a large data model and making a final decision in this matter.

Moreover, simple addXYZ() and removeXYZ() mutator methods are sometimes found on persistence-capable classes, mostly for convenience:

```
private Set books = new HashSet();

public void setBooks(Set _books) { this.books = _books; }
public Set getBooks() { return this.books; }

public void addBook(Book _book) { books.add(_book); }
public void removeBook(Book _book) { books.remove(_book); }
```

Note that for some JDO implementations, the exact class for a Collection-type field may be fixed. Using an assignment-compatible subclass could lead to run-time exceptions when using such implementations. A method like setBooks() could thus lead to trouble in certain cases, e.g., when an application would do something like this:

```
author.setBooks( new java.util.LinkedHashSet() );
```

If for this or other reasons a developer wanted to enforce usage of the above type-safe add/remove methods and disallow directly modifying the Collection returned by a getter, using Collections.unmodifiableCollection() in the getter and no setter could enforce this, for example:

```
private Set books = new HashSet();

public Set getBooks() {
  return Collections.unmodifiableCollection(books); }

public void addBook(Book _book) { books.add(_book); }
public void removeBook(Book _book) { books.remove(_book); }
```

Note also that an application cannot assume that it knows the exact class actually used for instances of collection-type fields. JDO implementations may choose to treat collections as second-class objects (SCO) and may substitute them by a JDO implementation-specific subclass of the respective type, although it is guaranteed that the actual class is assignment compatible.

Second-class object substitution of Collection-type attributes by JDO implementations at runtime is also the reason why it generally does not make sense (and is asking for trouble or at least for a performance penalty) to actually declare a persistent List field as an ArrayList or LinkedList, instead of just a List or a Set field (a HashSet, for example) instead of just a Set:

```
// Bad practice, why force a HashSet in field declaration?!
private HashSet books = new HashSet();
```

That's because in each case, the JDO runtime will populate a field declared as Set with an object of some internal implementation class that either extends HashSet or implements Set, respectively, yet is backed by the database. For a variety of reasons, a class that extends HashSet has fewer opportunities to optimize its database access than one that fully supplies its own implementation of Set. For example, the contents of a HashSet must always be fully loaded into memory.

In summary, it is generally a best practice to use a high-level interface-type (Set, List, and so on) instead of a specific implementation in the declaration of fields in persistent classes.

13.1.4.1 Heterogeneous versus homogenous object containers

The standard implementations of Java collection classes are heterogeneous object containers, meaning that objects of any class can be inserted into them. In JDO applications, however, the class of objects that can be added as elements to a Collection is usually constrained by an element-type field of the collection tag in the XML metadata descriptor.

This subtle difference does not usually cause any issues in real-world applications, but is an interesting point to keep in mind. Although some JDO implementations that support fields of type java.lang.Object are likely to support fully heterogeneous collections, it is not recommended to rely on such an implementation's feature for applications intended to be portable across JDO implementations. It often makes more sense, makes code more robust, and is in many cases perfectly possible, to name a concrete class, possibly a persistence-capable superclass of a completely persistent class hierarchy, as an element-type for collection fields.

In a way, this discussion is not specific to JDO as such, but really applies to developing under Java in general. If you think about it, many of your applications actually store only elements of a certain class in a collection. However, this is only implied, you know about it, don't usually make a mistake with this, may mention it in the JavaDoc, but Java does not currently provide a way to enforce this.[4] (Short of writing your own Collection wrapper classes, including type-safe not currently provide a way to enforce this.[4] (Short of writing your own Collection wrapper classes, including type-safe Iterator wrappers, with new methods, named differently because Java does not currently provide covariant return types. This is relatively cumbersome and not typically done; we generally rely on the collections framework as it comes out of the box, which, thanks to polymorphism and all objects implicitly deriving from one common system Object superclass, works fairly well. People with a background in the programming language Smalltalk often find this a non-issue and don't understand all the fuss about compile-time checking, while people with a C++ background used to template syntax and the STL find this weird.)

[4] One could technically of course write a Collection wrapper with add(), remove() and other methods overloaded to be type-safe and take the specific class as argument, instead of the generic Object. This is however rarely done in practice, one reason being that without real templates this quickly becomes cumbersome to hand-write, and another more important one making this difficult is Java's lack of "covariant return types," meaning overridden methods in subclasses are not allowed to have a return type that is a subclass of the parent method return type.

Because this is an issue with the Java language in general, preparations are under way to include a solution to this, known as Java Generics, see JSR 14,[5] in a future JDK release, possibly JDK 1.5.

[5] See http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html, and http://developer.java.sun.com/developer/earlyAccess/adding_generics/

It would be perfectly logical for JDO to take advantage of and leverage a standard type-safe collections framework for persistence.

13.1.5 Inverse relationships modeling

In Java, and in most other programming languages, associations between objects are *directed.* For example, an Author may have a books field. However, the pointed-to persistent Book objects generally do not have *inverse* associations back to the Author. This may seem an obvious and trivial statement, particularly from a Java developer's point of view; however, this is not as natural as it initially looks: In datastores such as relational databases, and thus particularly from your DBA's point of view, a relationship between two persistent entities can be navigated and identified from either way around.

It is, of course, perfectly possible to write an explicit JDOQL query on a datastore and retrieve, for example, the Authors that point to a given book. This query may even be implemented in a convenience method of the Book class, in code that would look something like this:

```
public class Author {
   private Set books;

   // ...
}

public class Book {

   // ...

   public Author getAuthor() {
      PersistenceManager pm = JDOHelper.
                          getPersistenceManager(this);
      Extent e = pm.getExtent(Author.class, true);
      Query q = pm.newQuery(e, "books.contains(thisBook)");
      q.declareParameters("the.package.of.Book thisBook");
      Collection results = (Collection)q.execute( this );

      Author author = null;
      if ( results.iterator().hasNext() ) {
        author = results.iterator().next();
        }
      q.close(results);
      return author;
   }
}
```

It is equally possible to have an explicit *mirror* field in Book for the Author (or Authors, depending on the logical model) that point to it. Such a mirror field would be redundant from a data management point of view, but it may still make sense for a completely portable JDO application. However, either an application or the model classes themselves would have to take extra care to always update the field on Book author field and the Author books field together. (This approach is similar to how a traditional double-linked list is implemented.)

```
public class Author {
   private Set books;
   // ...
   public void addBook(Book newBook) {
      newBook.getAuthor().removeBook(newBook);
      books.add(newBook);
      newBook.setAuthor(this);
   }

   public void removeBook(Book aBook) {
      books.remove(Book aBook);
   }

   // ...
}

public class Book {
   private Author author;
   // ...

   /* package-local */ void setAuthor(Author author) {
      this.author = author;
   }

   public Author getAuthor() {
      return author;
   }

   // ...

}
```

If the domain model of your application could benefit from modeling such inverse relationships and you are willing to trade in portability to other JDO implementations, it is worth reading to determine if your JDO implementation of choice already supports this out of the box. The JDO specification itself shows an example of how this may look:

```
class Employee {
   Department dept;
   (...)
}

class Department {
   Collection emps;
   (...)
}

<class name="Employee" (...)>
   <field name="dept">
<extension vendor-name="sunw" key="inverse" value="emps"/>
   </field>
(...)
</class>

<class name="Department" (...)>
   <field name="emps">
     <collection element-type="Employee">
<extension vendor-name="sunw" key="element-inverse" value="dept"/>
      </collection>
   </field>
(...)
</class>
```

Note again that, at this point, this is an implementation-specific extension. If your implementation of choice supports something like this, it most likely either generates or calls some code similar to the one shown above (likely if it is an O/R mapping-based implementation), and/or transparently implements a mirrored field (possible, if it is an object datastore) as a performance enhancement.

Future versions of JDO may standardize this functionality with a standard tag for the XML persistence mapping descriptor, and well-defined semantics for this feature, for example, related to updating inverse fields, and so on.[6]

[6] Note JDO Spec. Chapter 24.7, "Items deferred to the next release, Managed (inverse) relationship support: (…) The intent is to support these semantics when the relationships are identified in the metadata as inverse relationships. (…)"

13.1.6 Inheritance hierarchies

Using JDO, as opposed to some other persistence frameworks, deriving from some common "root" persistence superclass is not a technical requirement. As we have seen, most classes can be declared persistent.

However, when using inheritance hierarchies, it is a best practice to declare classes all along the entire inheritance tree persistent, up to but not including the ultimate java.lang.Object superclass. Again, in simple and clear words, and thinking both ways up and down an inheritance hierarchy: Do not create persistent subclasses of superclasses that are not persistent, and do not create subclasses that are not persistent of superclasses that are persistent. Although doing it anyway is not explicitly prohibited by JDO and will not be flagged by some enhancers, it is definitely asking for trouble.

Of course, depending on the business application being developed, you may wish to create all subclasses from one persistent domain class (application domain specific, not framework specific) root class, which in the case of a collaboration platform, for example, may have fields such as createdOn and lastModified, and so on. This is not to say that you shouldn't do that; but if you do it, make sure that the application domain specific root class is enhanced and thus persistent, and so are all its subclasses.

13.1.7 Don't rely on persistence-aware (not persistence-capable) classes

Keeping fields private and accessible only via respective public accessor and mutator methods not only adheres to the common best practice of *encapsulation* in object-oriented programming (OOP), but it also has special importance in JDO-based development: It allows developers to compile and enhance the persistence-capable classes separately from the business logic and other classes of a project.

Imagine for a moment an environment where business logic classes would make direct access to public fields of persistent classes, without using a getter or setter method of the persistent object. A build system would need to ensure that absolutely all those classes are enhanced by JDO as well, even though they do not contain a persistent application state. Otherwise, such classes may read stale fields, or attempt to access non-default fetch-group fields that have not yet been read from the datastore, or inversely modify fields without giving JDO a chance to update its (internal) dirty flag, which would ensure that the field's new value was written back to the datastore, at some point.[7]

[7] A somewhat related issue would possibly arise if code (from the application itself or in some third-party library) accessed private fields of JDO enhanced persistence-capable classes via the Java reflection mechanism. As it is possible to access private fields using reflection, given the right ReflectPermission security privilege, JDO may be "bypassed." So if you see "funny things" happening, disabling the respective permission could help to discover code that uses reflection to access private fields instead using the public accessors and mutators.

The JDO specification refers to this issue by calling the persistent entity-type classes we described above as *persistence-capable*, as we have seen, and other classes *persistence-aware,* and mandates that JDO implementations need to be able to enhance arbitrary classes to persistence-awareness without changing their semantic behavior. This works for third-party code as well, because source code is not required to enhance classes at the byte-code level. Although this could be useful in some situations and technically works perfectly well, many JDO-based applications may prefer to enhance only persistence-capable entity-type classes, and compile business logic and other classes of a project as usual, often by an Integrated Development Environment (IDE), without subsequent enhancement. This works very well as long as all access to persistent data happens through public accessor and mutator methods of the persistence-capable class (which will be enhanced) and the persistent state remains in private fields.

In summary, to keep life simple, make all your persistent classes use private fields with public accessor and mutator methods, enhance them to become persistence-capable, and then for most real-world projects, forget about persistence-aware—after you have read and conceptually understood them, of course.

13.2 JDO and Servlets

Applications with a Web-based presentation layer can use JDO in at least two different ways:

- Integration of JDO with an EJB service-oriented architecture and Data Transfer (Value) objects was discussed in depth in an earlier chapter. Such an approach could use a Web UI without JDO access from the presentation layer.
- In another architecture, the JDO API could be used directly in the Servlet layer, including JSP pages as views. This approach is discussed briefly here.

The second architecture is straightforward: The *business logic* is in plain old Java objects (POJO) co-located to the presentation tier. The business logic and the presentation layer (Servlets and JSP) can use the JDO API directly.

In a Model-View-Controller (MVC) architecture as often used in today's Web-based applications, two variations are possible with JDO:

- In a *Push MVC* model, the UI Controller (e.g., a Struts Action) would access the JDO API and then "push" certain persistent objects for display to a View. Technically, the objects are usually forwarded along as attributes stored into the request. The View would usually be implemented as a JSP or, alternatively, some template engine. The View in this variation only uses something like the useBean JSP tag; it does not access the JDO API directly.
- In a *Pull MVC* model, the Controller could forward to a View that does use the JDO API. For example, a JSP-based View could use JDO to execute and iterate through a Query, possibly prepared by a controller. Such functionality can be prepackaged into convenient JSP tag libraries.

Some of the questions that can arise in using the JDO and Servlet API together (with both MVC approaches), which are briefly addressed in the next few paragraphs, include the following:

- Should optimistic or datastore transactions be used?
- What about non-transactional reads?
- How is the PersistenceManagerFactory initialized and found? Where should the PersistenceManager be stored? Session, request, or thread-local?
- Can persistent objects be stored in the Servlet session?

Regarding the two different transaction types and other transaction configurations that JDO supports, one particular combination is often used and makes sense in the type of Web applications described here: Using optimistic transactions for write access and a PersistenceManager with non-transactional read for read-only access for display. This configuration allows a JDO implementation to optimally leverage the PersistenceManagerFactory (PMF) global in-memory cache and generally leads to good performance and scalability. For example, the MVC Actions can start and commit transactions, while the Views (in an MVC Pull model) can use non-transactional reads.

The PersistenceManagerFactory can be initialized and looked up in two ways in Servlet applications: Either an application (or proprietary framework) specific "helper" class creates, configures and holds onto the PMF. This allows Servlets to use this helper class to obtain a new PersistenceManager when required. Or, alternatively, a Servlet application could also leverage JDO's J2EE integration and obtain a new PersistenceManager from a PMF bound into JNDI by the respective JCA adapter. This allows Servlet applications to use a shared transaction across JDO, JDBC, JMS, and so on, as described in Chapter 11. Of course, all this is no different from what has been described earlier in this book, the classical managed and unmanaged JDO scenarios.

After a PersistenceManager is obtained, there are two possible architectures:

- Storing as attribute in the request context: A Servlet filter, a Struts request processor, or any similar mechanism obtains a new PersistenceManager when the processing of a request starts. Then the Controller and possibly View work with it. At the end of the request, the PersistenceManager is closed. Particular care needs to be taken that the PM is always closed and transactions rolled back even if an exception occurs rendering the view. This architecture can lead to great scalability.[8]

[8] The PersistenceManager could also be closed at the end of the controller already, before forwarding to a View, in a pure "Push MVC" model. However, with this approach a lot of issues similar to the ones raised in the EJB chapter arise, around having to make instances transient, using pm.retrieve(), and so on. The

advantage of such an architecture particularly when using optimistic JDO transactions are dubious, and it is generally not recommended.

- Storing in the session: An HttpSessionListener (or some Controller explicitly) could obtain a new PersistenceManager and store it in the Session for the lifetime of the session.

The second approach can appear to make application development somewhat easier. In this case, persistent objects (e.g., Query results) can be directly stored into the session as well, see below. However, because a PersistenceManager is not serializable (only a PersistenceManagerFactory is) this architecture actually prevents "distributable" Web applications for load-balancing and fail-over on clusters. It is recommended only for relatively simple applications with no need for session passivation and replication.

A related issue is the one of storing persistent objects as attributes into the session, if the PersistenceManager is in the session. Again, persistent objects are not generally serializable for session replication; see the respective discussion in Chapter 9.

It is generally preferable and possible to store only identities of persistent objects into the session using PersistenceManager.getObjectId() and then retrieve the object in the next request via PersistenceManager.getObjectById(). This is conceptually similar to what was discussed in Chapter 5 for stateful session beans.

Alternatively, a similar but completely stateless architecture can be used if JDO identities are passed only via URL and hidden HTML FORM INPUT fields. In this scenario, the JDO identity is converted to and from String via PersistenceManager.getObjectId().toString() and PersistenceManager.NewObjectIdInstance().

It is worth noting that obtaining a PersistenceManager for each request and looking up objects via identities (as opposed to storing the PersistenceManager and persistent objects in the session) does not necessarily lead to less performance. A good JDO implementation reuses instances and ideally does not require underlying datastore access because of caching when looking up objects via a stored identity.

13.3 Keep Domain Classes Separate from Others

As outlined in Chapter 12, persistent-domain classes and business-logic classes using them can technically be compiled separately, with only the persistent classes being subsequently enhanced.

The easiest way to practically achieve this is to keep two separate source directories for the two types of classes. This approach is not technically required, because it would be perfectly possible to simply compile and enhance all classes of a project, but it has several potential benefits, including the following:

- Increased build speed (even a well-written fast byte-code enhancer takes time on a large project when unnecessarily enhancing hundreds of persistent-aware classes that would not have to be enhanced; see above).
- Simplified and clearer CLI build infrastructure, for enhancement[9] and data-store-dependant schema utility tools (most larger projects generally have non-GUI build processes, often based on Jakarta's ant, with an IDE visual interface as alternative, if at all).

  [9] A theoretical alternative would be to keep all source files together, and configure the byte-code enhancer to only enhance explicitly named classes. This is technically possible with the Sun RI enhancer, but many implementation-supplied enhancers will simply enhance every class file found under a certain directory.

- Simplified usage from typical Integrated Development Environments (IDEs), which would typically compile all classes contained under a source path, not generally providing a hook for enhancement, and then start the application using the non-enhanced classes.

A directory structure similar to this one may make sense:

```
lib/
(...)
src/
classes/
src.model/
classes.model/
```

In this example, the following definitions are set:

- src contains the .java source files for all non-persistent business-logic, helper, and presentation-related (e.g., Servlets, and so on) classes.
- classes contains the compiled .class files for all source files in src.
- src.model contains the .java source files for all persistent-capable (not only persistence-aware) domain model classes, as well as the respective .jdo mapping descriptors for the enhancement.
- classes.model contains the compiled and enhanced .class files of the domain model, as well as copies of the respective .jdo mapping descriptors for runtime usage.

Note by the way that when splitting Java source directories like this, classes from both locations can share (parent) packages. A directory layout like this—src/com/abc/web, src/com/abc/util, and src.model/com/abc/model (where package com.abc is compiled from two locations)—does not bother Java at all.

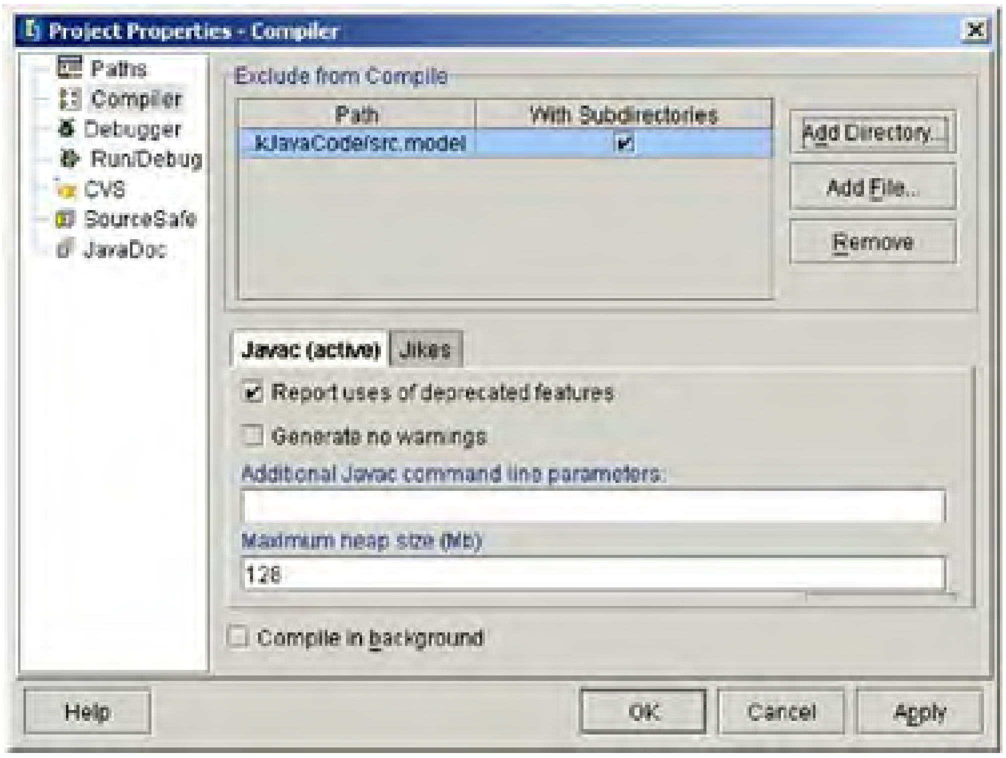
The following build order may then make sense, based on the assumption that domain model classes do not depend on (do not call any methods in) the src classes, because only the business-logic classes in src use the persistent domain model classes in src.model:

1. Compile src.model to classes.model.
2. Enhance everything in classes.model.
3. Compile src to classes (putting classes.model on the compile CLASSPATH).

The first two steps should generally be performed by an ant-based build infrastructure, unless an IDE provides specific support and hooks for this purpose. The third step can also be performed by a built-in compile feature of an IDE, as long as classes.model is on the project's CLASSPATH configuration.

Note that, although it is often convenient to edit the domain model classes in src.model in the same IDE, it is important to configure an IDE to *not* compile and mix these classes together with the project's other code in src. An example of a typical configuration from a popular IDE is shown in Figure 13-1.

Figure 13-1. Screenshot of a typical IDE's (here, IntelliJ IDEA) preferences, excluding one source directory (src.model) from compilation.



Path	With Subdirectories
..k\JavaCode\src.model	✓

[✓] Report uses of deprecated features

[ ] Generate no warnings

Additional Javac command line parameters:

Maximum heap size (Mb): 128

[ ] Compile in background

13.4 Using XML as a Data Exchange Format

This chapter looks at how XML can be used as a data exchange format (e.g., in files, or over the wire) between JDO-based applications.

13.4.1 Introduction

When using domain-centric data models as described above, there often is a need to represent the data of persistent objects, indeed entire graphs of related persistent objects, in a datastore implementation-independent, externally readable manner.

The purpose of such an external representation is usually interchanging information, and often between systems of different technology, may be using a SOAP-based Web service. However, scenarios like transferring content from one (JDO) datastore to another, or migrating data from one (earlier or different) schema model to another one, equally benefit from a neutral data interchange format. These days, XML is the universally accepted data representation format for many such applications. Various ways exist to write XML data from Java objects (marshaling) and create Java objects by reading XML data (unmarshaling).

13.4.2 Alternatives

In fact, at least one Java persistence framework (Exolab JDO, which is not a Sun JDO API implementation) comes with XML marshaling and unmarshaling abilities straight out of the box and somewhat tied into its API for orthogonal object persistence.

Although the JDO API itself does not offer a built-in XML marshaling and unmarshaling feature, and successfully focuses on binary datastore persistence exclusively, it is relatively easy to achieve if your application requires this. Several possible roads forward exist, and most are based on open-source or freely available XML toolkits, as we see.

Before delving into practice and demonstrating a concrete running example, let us clarify that the next paragraphs are *not* about how to use JDO with or as an XML database, a database architecture that uses XML structure as native storage format, allowing applications to work transactionally safe with and query possibly large sets of XML data. We are simply going to see how you can marshal and unmarshal persistent JavaBean objects to and from XML, irrelevant of the underlying datastore. However, it is probably only a matter of time until major XML database vendors provide implementations to access their own respective XML datastores using the JDO API.

13.4.3 Available technologies

The savvy Java developer by now will say, "Eh, no big deal, we can put together some code using the JAXP API using SAX or DOM and spit out or process that XML!" That's certainly true, but before you starting hacking away, consider that others have already done this job for you. The possible solutions that we can consider for this purpose are generally termed *XML data binding* products.

Most XML data binding products fall into two categories:

- Class generator-like products will generate Java classes based on a DTD or XML Schema. These classes can then be used to marshal object instances to XML documents that comply with the respective DTD or XML schema, and unmarshal XML documents to instances of the generated Java classes.
- Runtime inspection-based products will attempt to marshal and unmarshal any object complying with the JavaBeans specification to and from XML, without design/compile-time configuration, based on the Java Reflection mechanism.

Other categorizations are possible, notably around questions such as "Can any arbitrary instances be XML serialized?" or "Are XML subtleties such as exact whitespace usage or comments represented (for full object-XML round-tripping) but are of less importance for the purpose of this discussion?" However, the choice between a solution supporting the former or latter of the above approaches is one to make based on the need of your project, because both are equally applicable to JDO-based applications:

- When using a class generator-like solution, it is usually possible to subsequently enhance the generated classes and use JDO to persist and query instances. In fact, you can achieve a simple home-grown XML database yourself relatively easily.

- When using a runtime inspection-based solution, a project can use existing persistence-capable classes. These solutions usually require more or less strict JavaBean structure persistent classes, as outlined earlier in this chapter.

Below is a likely non-exhaustive list of some libraries that you may wish to check out. The following are generator-like solutions:

- Sun JAXB (JSR 31 implementation): At the time of this writing, this solution is still evolving. It is a new Java API, developed under the Java Community Process (JCP) just as JDO was. Early JAXB implementations supported DTDs, and more recent versions are based on XML schemas. At the time of this writing, it is not fully show time yet and it hasn't been for some time, but is still likely where things are going to go in the end.
- Zeus, from the enhydra.org project.
- BeanStork, from Zenaptix.
- Electric XML / XML+, from The Mind Electric.
- Castor, from Exolab, mentioned just for completeness; Castor is really a mix of persistence framework for relational databases, LDAP repositories, and schema-driven XML marshalling.

These are runtime solutions:

- JDK 1.4 Long-term bean persistence (`java.beans.XMLEncoder` and `XMLDecoder`): This is conceptually similar to JDK-standard binary object serialization, and in fact uses a similar API, except that instead of writing an undecipherable byte stream, it serializes to XML. (Developed as JSR 57 and now part of JDK 1.4.)
- JBind.
- Quick.
- Zeus.
- JiBX.
- Betwixt from the Apache Jakarta Commons project (see below).

There are many others that are usually fairly similar. The one major difference generally is whether a predefined fixed XML language (tag and field names, and so on) is used, which is the case for JDK 1.4 long-term bean persistence and looks something like the following, which is a non-JDO related example, simply illustrating XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
   <java version="1.0" class="java.beans.XMLDecoder">
   <object class="javax.swing.JFrame">
     <void property="name">
       <string>frame1</string>
     </void>
     <void property="bounds">
       <object class="java.awt.Rectangle">
         <int>0</int>
         <int>0</int>
         <int>200</int>
         <int>200</int>
       </object>
     </void>
     <void property="contentPane">
       <void method="add">
         <object class="javax.swing.JButton">
           <void property="label">
             <string>Hello</string>
           </void>
         </object>
       </void>
     </void>
     <void property="visible">
       <boolean>true</boolean>
```

```
    </void>
  </object>
  </java>
```

But if the solution is able to derive an XML language from the JavaBeans fields, usually with more or less configurable mappings, it leads to something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<book id="2">
  <authors>
    <author id="3">
      <name>Keiron McCammon</name>
      <email>mccammon@corejdo.com</email>
      <zipCode>10243</zipCode>
    </author>
    <author id="4">
      <email>tyagi@corejdo.com</email>
      <name>Sameer Tyagi</name>
      <zipCode>31484</zipCode>
    </author>
    <author id="5">
      <name>Michael Vorburger</name>
      <email>vorburger@corejdo.com</email>
      <zipCode>1234</zipCode>
    </author>
    <author id="6">
      <name>Heiko Bobzin</name>
      <email>bobzin@corejdo.com</email>
      <zipCode>3391</zipCode>
    </author>
  </authors>
  <price>39.95</price>
  <title>Core Java Data Objects Book</title>
</book>
```

13.4.4 Let's do it

Let's now see some code on how to achieve this in practice. For this example, we are going to use the Betwixt library, a useful little library from the Apache Jakarta Commons project. Betwixt is in beta stage at the time of this writing, but remaining problems are likely to be sorted out rapidly.

As a runtime approach solution, there is no need to write a DTD or an XML schema, or to generate any classes, and so on, as long as our persistent objects are JavaBeans, which the Author and Book classes are. Given a Book instance, jdoBook, with a few Authors associated, the XML snippet shown above is actually as simple to obtain as this line with Betwixt:

```
BeanWriter beanWriter = new BeanWriter( System.out );
beanWriter.write( jdoBook );
```

Reading a snippet like this could be achieved with a few lines as follows:

```
BeanReader beanReader = new BeanReader();
beanReader.registerBeanClass( Author.class );
beanReader.registerBeanClass( Book.class );
Book book = (Book)beanReader.parse("book.xml");
```

Betwixt does not require any further initialization or setup code. Although we write to a PrintStream (System.out) and read from a filename in the above simple example, Betwixt does also offer an API to write directly to a SAX ContentHandler.

It gets slightly more interesting if you would like to write and read a collection (e.g., JDO Query result) of persistent objects with Betwixt. One way to achieve this with Betwixt is to write a simple container class like the following:

```
public class BookCollection {
    private List books;

    public BookCollection() {
        books = new LinkedList();
    }
```

```
public BookCollection(Collection someBooks) {
    books = new LinkedList(someBooks);
}

public List getBooks() {
    return Collections.unmodifiableList( books );
}

public void addBook(Book _book) {
    books.add(_book);
}
```

Betwixt (at least the latest version at the time of this writing) requires the addBook(Book book) method to understand that this top-level <BookCollection> contains <Book> objects.

A more complete example scenario could look like this: It creates a few Author and Book objects, with associations from some Books to some Authors. These objects are then persisted in JDO. A real application would get some real work done at this point, and possibly lose references to these objects just created. Then we fire off a JDO Query to find all Book objects. Now we marshal all these objects into an XML representation and store them in a file. Finally, the example reads the file again and compares the unmarshaled Author and Book objects to the original objects.

This simple example implemented using Betwixt or JDK 1.4 long-term bean persistence has several severe limitations:

- It would not scale well with increasing number of instances, because all XML is held in memory.
- The JDO datastore identity is lost in the marshaling/unmarshaling. If we had used Application Identity instead of Datastore Identity, this would have come along almost for free.
- Collection fields with names different from types are not supported without providing extra information to Betwixt, because it uses the method signature to derive the XML element names.
- Class inheritance of not supported, because Betwixt is unable to map this correctly.

These problems could, however, be addressed by either extending this simple example and/or using other XML data binding technologies.

13.5 Validation

A frequently asked question is how to implement data validation with JDO. Before looking into a possible technique based on JDO instance callbacks, it is important to briefly think about the layer of an application at which data validation could take place:

- At the persistent object level.
- Before accessing the persistent objects, in a business-logic method or at UI level (simple validations can sometimes be delegated to UI clients, e.g., Web browsers with Java scripts, or Swing clients).
- Datastore level, e.g., using SQL constraints such as NOT NULL, and so on.

While some fervently argue in favor of only one of these options, e.g., "validate at the UI layer only," it often makes sense to place validation logic at several layers. The reasons can be manifold—for example, to save round-trips when validating at the browser client level in a Web-based architecture. It does, however, often make sense to validate at the domain model layer as well (or only), to ensure that validation takes place if a domain model is modified in several places from application logic.

Ideally, the validation code should be shared across layers to avoid duplication. This is sometimes possible—for example, when calling the validation logic written as Java method from both the persistent code, as well as from a UI component such as Swing listener or similar Web (e.g., Struts or JSF Action) component. Other times, this is more difficult to implement—for example, sharing validation rules expressed in Java between a JavaScript-able browser and an SQL datastore. One possible solution in that case is to express validation as higher-level "constraint" instead of directly in Java code. The next few paragraphs focus only on how to practically enforce validation logic from JDO domain model classes.

Validation logic can mean many things, such as the following:

- Certain field(s) should not be null.
- Certain numeric fields should be within a certain range.
- Certain string fields should match a given regular expression.
- Certain string fields should not be longer than *x* number of characters.
- A field should be unique within an Extent or within a Query on that Extent.
- Any arbitrary other "check" not covered by the above, e.g., "The sum of two numeric fields has to larger than another field" or "Field temp cannot be higher than the current temperature on the peak of the Matterhorn."

The JDO API does not currently offer much support to express such domain model constraints. Only the above mentioned "cannot be null" can be expressed with a declaration like this in the JDO XML mapping descriptor:

```
<field name="lastName" null-value="exception" />
```

This declaration requests the JDO implementation to throw a JDOUserException if this field contains a null value at runtime when the instance must be stored.

However, any of the more complex requirements above cannot be expressed simply by declaring something in the XML mapping descriptor. They are, however, easily written in Java code. The trick is when and how such respective Java validation code is called.

13.5.1 Leveraging InstanceCallback's jdoPreStore method

The interested reader will likely think of the JDO InstanceCallback interface and its jdoPreStore() method, covered in earlier chapters. And indeed, leveraging this mechanism often is a solution in a first go of an application, because it's simple and obvious. Although we will criticize and enhance this approach in just a few paragraphs below, let's look at how this would be implemented in practice. The example chosen for illustration here is a validation rule that enforces that an email address of a persistent Author class matches a regular expression pattern of Internet email addresses:

```
public class Author1 implements InstanceCallback {
   private String email;
   public String getEmail() { return email; }
   public void setEmail(String e) { email = e; }

   public void jdoPreStore() {
      if ( !Pattern.matches("[a-z01-9_\\-.]+@" +
                          "[a-z01-9_\\-.]+\\.[a-z]+",
            getEmail() != null ? getEmail() : ""  ) {

         throw new SampleValidationException(this, "email");
      }
   }
}
```

With the persistence-aware domain class written like this, we will always throw a non-checked SampleValidationException if the email field is not correct, as in this usage example:

```
(...)
Author1 a = new Author1();
a.setEmail("authors@corejdo.com");
pm.currentTransaction().begin();
pm.persist(a);
pm.currentTransaction().commit();

pm.currentTransaction().begin();
try {
   // Invalid email, not matching RegExp pattern
   a.setEmail("authors-corejdo;com");
   pm.currentTransaction().commit();

   throw new Error("Validation broken; should not reach!");
} catch (SampleValidationException ex) {
   // Good!
   if ( pm.currentTransaction().isActive() ) {
     pm.currentTransaction().rollback();
   }
}
pm.close();
```

Saying that "directly having jdoPreStore() call (or indeed just perform, as above) the validation usually works well" is not entirely correct. Let's take a minute to glance over what the JDO specification says regarding the preStore method: "This method is called before the values are stored from the instance to the datastore." So actually, our validation could be called multiple times on one instance in a single transaction. You don't care, not a problem, you say? We beg to differ. Think about this scenario:

1. The application begins a transaction, creates an instance, and then immediately persists the instance, but has not yet set all fields to valid values.
2. JDO implementation chooses to write modified current in-memory instances of persistent objects to the datastore, for whatever reason, and thus invokes the preStore() method as per the JDO specification.
3. The application sets all fields of the object persisted in Step 1 to valid values.
4. The application commits the transaction.

This is a perfectly valid scenario, because within a transaction we can have persistent objects in a state that would fail the validation. If we couldn't, it may not even be possible to create, for example, an Author instance without immediately setting the email field from the constructor. Although this would be possible in this example, generally speaking it would not be.

However, if a JDO implementation actually chose to also write instances to the datastore in the middle of a transaction instead of only at commit time (which would be perfectly valid according to the JDO specification), our validation check from the above example would throw the SampleValidationException in Step 2 and make it impossible for the application to get to Step 3 and beyond. Indeed, what we really want is to perform the validation at the time that the transaction commits.

13.5.2 Using Synchronization's beforeCompletion() method

The javax.transaction.Synchronization interface holds the key to implementing this second approach: JDO enables us to register a callback object implementing this interface by calling setSynchronization() on a transaction and calls this interface's beforeCompletion() method from where we can perform the validation.

So if this is the more reliable way to implement validation, why did we not do it from the start? There isn't really a catch in using transaction synchronization, but there are two added complexities that can be addressed:

- Transaction management and registration of synchronization callbacks is different in the non-managed (javax.jdo.Transaction#setSynchronization) versus the managed JTA-based (javax.transaction.Transaction#registerSynchronization) environment. In a managed environment, the application may not have (or should not have) access to the javax.transaction.Transaction object, only a javax.transaction.UserTransaction. This is not an issue if using only a non-managed environment, but it makes this approach unsuitable or more complicated for validation logic that should be usable in a managed environment as well.

- Once within the beforeCompletion() method of the callback object implementing the javax.transaction.Synchronization interface, how is the persistent object (or objects) to be validated found?

The first point is clarified in Chapter 11, but the second point can be addressed here. The basic problem is that, unlike in the InstanceCallback#jdoPreStore() method above, where we simply validate this, we lost track of which persistent object(s) to validate by the time control reaches Synchronization#beforeCompletion().

13.5.2.1 Combining both approaches

Combining the InstanceCallback idea with the Synchronization idea shows a promising approach: If a list of modified instances had been created and maintained in jdoPreStore(), but no validation is done yet, then the beforeCompletion method, having access to this list, could validate all modified instances.

So the Author persistence-capable class would change slightly to what's shown next. Note that the actual validation code moved into a newly introduction validate() method defined in the Validatable interface and that jdoPreStore() now simply delegates to an external helper method:

```
public class Author2
                implements InstanceCallback, Validatable {
   (...)

   public void validate() {
      if ( !Pattern.matches("[a-z01-9_\\-.]+@" +
                          "[a-z01-9_\\-.]+\\.[a-z]+",
            getEmail() != null ? getEmail() : ""  ) {
         throw new SampleValidationException(this, "email");
      }
   }
   public void jdoPreStore() {
      ValidationHelper.register(this);
   }
}
```

The new ValidationHelper class would then keep a list of modified persistent objects per transaction. Suppose that an instance of itself would be registered with the respective PersistenceManager and would look something like this:

```
public class ValidationHelper {
   private Set modifiedObjects = new HashSet();

   public Set getModifiedObjects() {
      return modifiedObjects;
   }

   public static void register(Validatable object) {
      PersistenceManager pm;
      pm = JDOHelper.getPersistenceManager(object)

      Object userObj = pm.getUserObject();
      ValidationHelper helper;
      if ( userObj != null ) {
         helper = (ValidationHelper)userObj;
      }
      else {
         helper = new ValidationHelper();
         pm.setUserObject(helper);
      }

      helper.modifiedObjects.add(object);
   }
}
```

Registering the ValidationHelper as UserObject of the PersistenceManager is one possibility of where to keep the list of modified persistent objects of a transaction, and is probably a much better design than using a global static hash map sort of thing associated with the current thread. The registered Synchronization object could then look something like this:

```
public class ValidationSynchronization
        implements javax.transaction.Synchronization {

   private ValidationHelper helper;

   public ValidationSynchronization(ValidationHelper vh) {
      helper = vh;
   }

   public void beforeCompletion() {
      Set set = vh.getModifiedObjects();
      Iterator it = set.iterator();
      while ( it.hasNext() ) {
         Validatable v = (Validatable)it.next();
           v.validate();
      }
   }
}
```

The ValidationHelper and ValidationSynchronization classes are separated simply for clarity here, but its methods could also be intermixed in one single class.

Usage of this solution is not entirely transparent to the application anymore and would require the registration of the ValidationSynchronization when a new transaction is started. Furthermore, the application should catch the SampleValidationException (of this example) and roll back if the validation failed, unless there is a generic when-catch(Exception ex)-then-rollback() mechanism in place. It would be possible to do this in an application-specific implementation of the JDO PersistenceManager interface that delegates to the original PersistenceManager of the JDO implementation.

13.6 Summary

This chapter presented various best practices applicable when working with JDO.

The section called "Data Modeling" covered best practices for modeling and coding the domain model of persistent classes. Advanced applications may wish to fine tune usage of certain JDO features and, for example, store some fields embedded in others, use arrays for small collections of fixed size multi-value fields, reference to non-persistence-capable classes, and so on. For the other 80 percent of all applications, however, the usage guidelines presented should prove useful.

A brief discussion on JDO and the Servlet API showed how transparent persistent could be used directly from within Web applications. Some similarities to concepts seen in the EJB chapter (regarding storing and retrieving by object identities) were mentioned.

The section called "Using XML as a Data Exchange Format" described how persistent objects could be stored and read from the XML format. This can be useful for data exchange or migration purposes.

In the section called "Validation," ways to enforce data integrity were explored.

Chapter 14. The Road Ahead

"Good things come to those who wait."

—Proverb

The JDO 1.0 standard was release in March 2002. The 1.0.1 maintenance update (scheduled during the first half of 2003) contains a number of minor corrections, clarifications and feature enhancements that have been included in this book. During 2003, work should commence on the JDO 2.0 specification with a scheduled release of sometime in 2004.

JDO 2.0 will focus on adding commonly requested features and functionality to JDO, and this chapter highlights some of those possible features. The features covered in this chapter are in no way guaranteed to make it into JDO 2.0, nor do they represent the limit of new features that may be considered. Rather, this chapter aims to give some insight into what may be in store in future revisions of the JDO specification.

This chapter covers the following topics:

- Advanced transactions semantics
- Performance optimizations
- Managed relationships
- Query enhancements
- Object mapping
- Enumeration pattern

14.1 Advanced Transaction Semantics

The JDO specification currently defines semantics for datastore transactions and optimistic transactions. For the majority of applications, these transactional semantics should prove sufficient; however, some applications may need more advanced or specialized capabilities or need to take advantage of features offered by the underlying datastore that they are using.

To this end, JDO 2.0 may add additional transaction semantics, principally nested transactions, savepoints, and explicit locking.

14.1.1 Nested transactions

With the current JDO specification, there can be only a single active transaction per PersistenceManager instance. All work done within the transaction is either committed or rolled back as a whole.

Nested transactions define the ability to begin a transaction within the scope of another transaction (often referred to as the parent transaction). Each nested transaction can be begun and committed or rolled back as per a normal transaction. However, even though committed, a nested transaction may still be subject to rollback by its parent transaction. And a parent transaction cannot be completed until all its nested transactions have completed.

Nested transactions are a way of breaking down a single "unit of work" into multiple subtasks that can be distributed and then committed or rolled back independent of each other, but still within the scope of the bigger unit of work. Supporting nested transactions within JDO will require a clear definition of the concurrency control and means of data sharing between nested transactions.

14.1.2 Savepoints

Savepoints provide a means to rollback only a portion of a transaction. In many ways, they are like a single nested transaction.

Using savepoints, an application can begin a transaction and at some point create a savepoint. At a later point, the application can decide to undo the savepoint. This essentially rolls back the transaction to the point at which the savepoint was created, undoing all the changes that were made thereafter, without having to rollback the entire transaction.

Savepoints are a useful undo mechanism for applications in which a transaction consists of a sequence of tasks. Should a given task fail, rather than having to rollback the entire transaction and redo everything, a savepoint can be used. The savepoint restores the state of any persistent objects to the beginning of the failed task, so it can be retried without affecting the other tasks that have already been successfully completed.

14.1.3 Explicit locking

The current JDO specification does not define APIs to allow an application to explicitly request a particular lock on a given persistent object. Instead, locks are implicitly managed by the JDO implementation in conjunction with the underlying datastore to ensure that concurrency control is enforced between different transactions.

Some applications may need to be able to explicitly request shared or exclusive locks on particular persistent objects. These locks can be used for synchronization of multiple, distributed applications all using the same datastore.

14.2 Performance Optimizations

JDO provides two simple ways to optimize application performance. The first is the concept of the "default fetch group," and the second are the retrieveAll() methods on PersistenceManager.

The default fetch group allows an application to determine the set of fields that should be retrieved from the datastore by default whenever an instance of a particular persistence-capable class is retrieved. By putting commonly accessed fields into the default fetch group, they will all be retrieved in a single datastore operation. In addition, less commonly accessed fields are then retrieved only when needed.

The retrieveAll() methods allow an application to retrieve a group of persistent objects potentially in a single datastore request and to avoid a single request per persistent object. This is particularly useful when iterating through the result of a query.

Beyond these two simple optimizations, JDO 2.0 may introduce a mechanism that allows an application to define policies that will govern the retrieval of graphs of persistent objects automatically. These policies may range from simple closure operations (retrieve all reachable persistent objects from a given persistent object) to more selective operations that would retrieve persistent objects reachable via certain fields, or up to a certain depth, or of a particular class only. Some JDO implementations already offer proprietary support for this feature.

The benefit of being able to specify retrieve policies in this way is that it allows optimization of datastore access without the need for additional coding. Different applications, or even different transactions within the same application, could specify different policies, depending on the application logic being performed at the time.

14.3 Managed Relationships

Relationships in JDO are modeled as normal Java object references or collections of object references between persistence-capable classes. JDO does not define any additional semantics for relationships beyond that specified by the Java language. However, unlike normal Java applications, a JDO application cannot rely on automatic garbage collection for persistent objects. Instead, persistent objects must be explicitly deleted. In addition, many datastores support varying forms of referential integrity constraints.

To simplify application development, JDO 2.0 may add support for bi-directional relationships, cascade deletion, and even persistent garbage collection.

14.3.1 Bi-directional relationships

A Java object reference or collection of object references typically represent a one-way relationship. It allows an application to navigate from one object to another, but not back again. Bi-directional relationships support navigation in both directions.

Bi-directional relationships can be modeled in Java as a pair of object references (or collection of references), one in each way. However, this requires additional application logic to maintain both sides of the relationship; if one side of the relationship is changed, then this change must also be reflected in the other side.

JDO 2.0 may add support for implicit bi-directional relationships declared in the JDO metadata. The JDO implementation would then automatically manage both sides of the relationship without additional application coding. If the application changed one side of a relationship, the JDO implementation (perhaps in conjunction with the underlying datastore) would implicitly change the other side also.

The benefit of bi-directional relationships is that they allow an application to navigate in either direction without having to explicitly code anything.

14.3.2 Cascade delete

JDO requires explicit deletion of persistent objects. Unlike the persistence by reachability algorithm that is used when objects are made persistent, JDO currently does not support the concept of deletion by reachability, or what is more commonly known as **cascade delete.** If an application needs to ensure that referenced persistent objects are deleted when their parent is deleted, then it would typically need to implement the jdoPreDelete() method defined by the InstanceCallbacks interface.

JDO 2.0 may add the ability to define in the JDO metadata that a field reference's "dependent" persistent objects (persistent objects that are not referenced by any others) and should be implicitly deleted. Upon deletion, the JDO implementation would ensure that all dependent persistent objects were also deleted, avoiding the need for the application developer to implement the jdoPreDelete() method.

14.3.3 Persistent garbage collection

Cascade delete works well for situations in which dependent persistent objects are not shared. If they are shared, a simple cascade delete approach is not going to work because it is not possible to easily determine that a persistent object is not referenced by any other and can, therefore, safely be deleted.

JDO 2.0 may add support for a persistent garbage collection mechanism in which instances of specified persistence-capable classes that are no longer referenced by any others will be deleted automatically by the JDO implementation or underlying datastore.

14.4 Query Enhancements

JDOQL is a simple query language that allows persistent objects to be found based on field values. In JDO 2.0, this query language may be extended to add better support for string operations, the ability to perform aggregate operations, and the ability to specify a projection as the result of a query.

14.4.1 Strings

JDOQL supports the String.startsWith() and String.endsWith() methods in query filters as the means of specifying datastore-specific wildcards. However, it does not support additional String methods that may generally be useful in query filters.

JDO 2.0 may add support for the String.toLowerCase() to allow case insensitive string comparisons. It may also add better support for wildcard or regular expression pattern matching beyond the startsWith() and endsWith() methods.

14.4.2 Aggregate operations

JDOQL currently does not support SQL aggregate operators like MIN, MAX, SUM, and AVG. If an application requires the summation of a field for all instances that match a specified criterion, it would have to first perform the query and then retrieve all matching persistent objects, summing their field values. Because many datastores support aggregate operators, this is not the most efficient way of solving the problem.

JDO 2.0 may add support for aggregate operators in JDOQL. This would allow an application to specify a query filter along with an aggregate operator, and rather than returning a collection of persistent objects, the query would return the aggregate result.

14.4.3 Projection

JDOQL currently does not support the ability to return certain fields of a persistent object as a result from a query rather than the persistent object itself.

JDO 2.0 may add the ability to define specific fields that should be returned from the datastore. Rather than returning a collection of persistent objects, the query would return a collection of field values instead. This is more akin to the JDBC concept of a result set.

This is useful when an application needs only the values of a single field of a persistence-capable class, perhaps to display in a selection box. Rather than having to retrieve the entire persistent object to get the field, it can be returned directly by the query, saving on further datastore requests.

14.5 Object Mapping

The current JDO specification does not define how the mapping between the Java class and the datastore schema should be specified; instead, it leaves this to the JDO implementation. This means that for each JDO implementation used, it is likely that a new set of mapping specifications would need to be defined. Of course, some underlying datastores may not require any additional mapping to be specified at all.

To aid in portability, JDO 2.0 may define standard JDO metadata elements for common mapping needs. This might allow a datastore name to be specified for a class or a field that a JDO implementation could then use when defining the datastore schema.

14.6 Enumeration Pattern

The enumeration pattern, where specific instances of a Java class represent the allowed values in an enumeration, is a useful programming technique in Java to constrain the values that can be assigned to a field.

As an example of this pattern, the following code snippet shows how the enumeration pattern can be used to represent the allowable eye colors for a person:

```
public class Person {

  private EyeColor eyes = Color.BLUE;

  /* Rest of code not shown */
}

public class EyeColor {

  public static EyeColor BLUE  = new Color("blue");
  public static EyeColor BROWN = new Color("brown");

  private String color;

  private EyeColor(String color) {

    this.color = color;
  }
}
```

JDO 2.0 may add explicit support for this pattern so that applications can use enumerations as fields of persistent objects.

14.7 Summary

This chapter has provided a flavor of what to expect in future revisions of the JDO specification. For those who cannot wait, some JDO implementations already offer support for some of these features through proprietary extensions and APIs. Of course, this results in a less portable application, at least until these features are standardized as part of JDO 2.0.

Chapter 15. Case Study: The Core JDO Library

" Few things are harder to put up with than the annoyance of a good example."

—Mark Twain

The Core JDO Library application is an implementation of a couple of use-cases to explain basic and advanced JDO features. It is also meant to explain concepts mentioned in the previous chapters such as two-tier and n-tier applications. This chapter is divided into smaller sections that can be used as cookbook-style recipes.

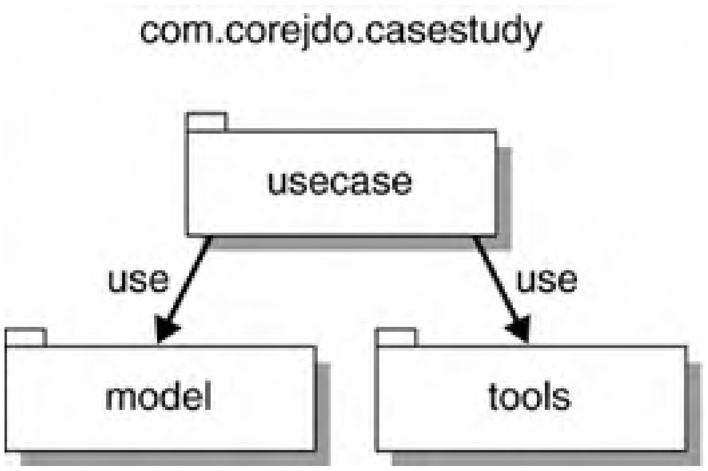
15.1 Files, Packages and Object Model

The files of the Core JDO library are in these directories:

src	Java sources
web	JSP sources and Web content
conf	Tomcat configuration
doc	JavaDoc output
classes	Compiler output

The project can be compiled by Ant (build.xml) and run either as a command-line application or as a Web application. The Core JDO Library Java sources are split into three packages, as indicated in Figure 15-1.

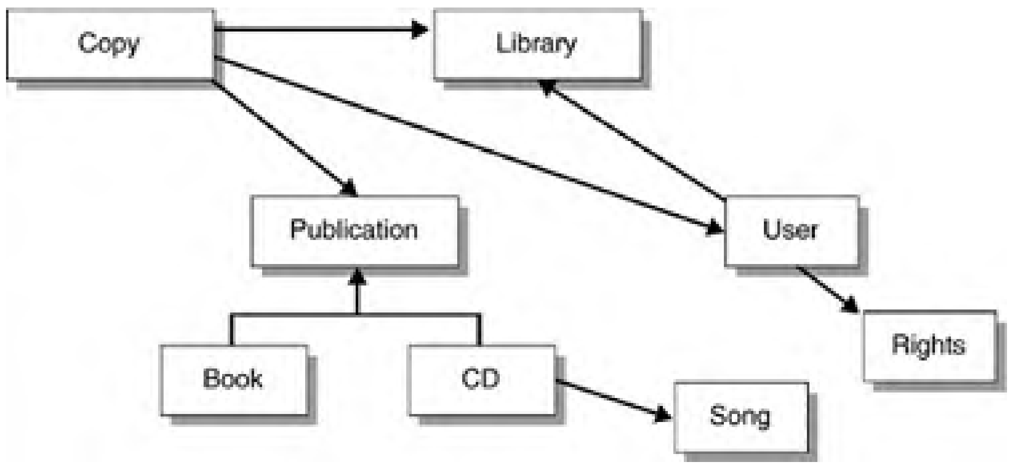
Figure 15-1. Core JDO library packages.



The usecase package contains the business logic, serves as a testbed and supports command-line invocation. Every use-case is coded in a single class and can be called either by a simple command-line interpreter or by external code. The pattern for this kind of implementation is called *Command.* An instance of the usecase classes has a single execute() method and runs a single JDO transaction for that method. The benefit of this approach is a single point in the code, where transactions are begun, committed, or rolled back. On the other hand, exception handling and error processing can become quite uncomfortable.

The model package contains all persistence-capable classes that define the class hierarchy shown in Figure 15-2: Persistent object model.

Figure 15-2. Persistent object model.



15.2 Persistent Model Package

The following sections look at each of the persistence-capable classes.

15.2.1 Publication class

This is the base class of books and CDs in the library.

```
Package com.corejdo.casestudy.model;
import java.util.Date ;
import java.util.Collection ;
import javax.jdo.*;

/**
 * This class contains generic
 * information common to all sorts of publications.
 *
 * <BR><FONT Color="red"><B>Persistent</B></FONT>
 */
public abstract class Publication
{
    /**
     * The title of the publication.
     */
    public String title;
    /**
     * Date, when the publication was published.
     */
    public Date published;
    /**
     * Date, when the publication was first published.
     */
    public Date firstPublished;
    /**
     * A copyright text.
     */
    public String copyright;

    /**
     * The librarian who maintained the publication.
     */
    public User insertedBy;

    public Publication() {
    }

    public String toString()
    {
        return "\n Title: "+title+
        "\n Published: "+published+
        "\n First published: "+firstPublished+
        "\n Copyright: "+copyright;
    }

    public String toHTMLTitle(String linkPrefix)
    {
        return "<TD CLASS=pubTitle>"+title+"</TD>";
    }

    public String toHTMLRest(String linkPrefix)
    {
        return
        "<TD CLASS=pubPubFirst>"+firstPublished+"</TD>"+
        "<TD CLASS=pubCopyright>"+copyright+"</TD>";
    }
    public abstract String toHTML(String linkPrefix);
```

The getCopies() method encapsulates the JDO query to find all the Copy instances that reference a Book instance.

Encapsulating a query in this way is a useful practice, although some may prefer to put these types of methods in a separate, associated class to avoid having any JDO-dependent code in their persistence classes.

About toHTML() methods

The classes shown here include methods like toHTML(), toHTMLTitle(), and toHTMLRest(), for the sake of simplicity and easily readable comprehensive examples.

Most real-world domain classes, however, would not include such presentation layer-related methods and put view-related code into separate classes.

15.2.2 Book class

This class represents a Book.

```
Package com.corejdo.casestudy.model;

/**
 * This class contains data about books,
 * like author, International Standard Book Number (ISBN).
 */

public class Book
    extends Publication
{
    /**
     * International standard book number.
     */
    public String ISBN;
    /**
     * Author of the book.
     */
    public String author;

    public Book() {

    }

    public String toString()
    {
        return super.toString()+
        "\n ISBN: "+ISBN+
        "\n Author: "+author;
    }
    public String toHTML(String linkPrefix)
    {
        return "<TR CLASS=Book>"+toHTMLTitle(linkPrefix)+
        "<TD CLASS=BookAuthor>"+author+"</TD>"+
        "<TD CLASS=BookISBN>"+ISBN+"</TD>"+
        toHTMLRest(linkPrefix)+"</TR>";
    }
}
```

15.2.3 CD class

This class represents a CD. It is more complicated than the Book class because it provides more functionality. It contains a song list that is returned in toString and toHTML.

It shows basic collection operations with ArrayList.

```
Package com.corejdo.casestudy.model;

import java.util.*;
/**
 *
 * <BR><FONT Color="red"><B>Persistent</B></FONT>
 */
public class CD
    extends Publication
{

    List songs = new ArrayList(); // contains Song objects
    String referenceNumber;       // CD reference number
    public CD(String referenceNumber)
    {
        this.referenceNumber = referenceNumber;
    }

    private CD()
    {
    }

    public void addSong(String composer,
                        String title, int seconds)
    {
        songs.add(new Song(composer, title, seconds));
    }

    public String songTable()
    {
        StringBuffer buf = new StringBuffer();
        buf.append("<TABLE class=song>\n");
        Iterator iter = songs.iterator();
        while (iter.hasNext()) {
            Song song = (Song)iter.next();
            buf.append("<TR class=song>");
            buf.append("<TD class=songTitle>");
            buf.append(song.title);
            buf.append("</TD>");
            buf.append("<TD class=songComposer>");
            buf.append(song.composer);
            buf.append("</TD>");
            buf.append("<TD class=songLength>");
            buf.append(song.length());
            buf.append("</TD>");
            buf.append("</TR>");
        }
        buf.append("</TABLE>\n");
        return buf.toString();
    }

    public String toString()
    {
        StringBuffer buf = new StringBuffer();
        buf.append("CD: ");
        buf.append(this.referenceNumber);
        buf.append(super.toString());
        Iterator iter = songs.iterator();
        while (iter.hasNext()) {
            Song song = (Song)iter.next();
            buf.append(song);
            buf.append(", ");
        }
        return buf.toString();
    }

    public String toHTML(String linkPrefix)
    {
        return "<TR CLASS=CD>"+toHTMLTitle(linkPrefix)+
          "<TD VALIGN=TOP>"+songTable()+"</TD>"+
          "<TD VALIGN=TOP></TD>"+
          super.toHTMLRest(linkPrefix)+"</TR>";
    }
}
```

15.2.4 Copy class

The Copy class represents a physical copy of a particular publication. For example, a book on a library shelf is represented as an instance of the Copy class. This class has a reference back to Publication. An alternative model to this would use a collection of Copy instances in the Publication class. The benefit of this approach is that if there are a large number of instances, maintaining a collection of all of them can become problematic.

```
Package com.corejdo.casestudy.model;

import java.util.Date;

/**
 * A copy is the instance of a publication in a
 * library. A copy can be located somewhere in the
 * library or it can be borrowed by someone.
 * Other states of a copy can be: missing, damaged or sold.
 * A copy cannot be reserved by someone, only
 * publications can be reserved.
 *
 * <BR><FONT Color="red"><B>Persistent</B></FONT>
 */

public class Copy
{
    /**
     * The publication of which this is the instance of.
     */
    public Publication copyOf;

    /**
     * This flag is set when the borrowing time is over.
     */
    public boolean missing;
    /**
     * If the copy is damaged, this flag is set.
     */
    public boolean damaged;
    /**
     * If someone bought the copy, this flag is set.
     */
    public boolean sold;
    /**
     * Location of the copy in the library.
     */
    public String location;
    /**
     * Reference to the borrower. If set, this publication instance is borrowed
     * by someone.
     */
    public User borrower;
    /**
     * Date when the copy was borrowed.
     * Depending on the user policies of the
     * library.
     */
    public Date borrowDate;
    /**
     * Date when the copy should be returned
     * to the library.
     */
    public Date returnDate;

    public Copy(Publication copyOf)
    {
        this.copyOf = copyOf;
        this.location = "entry";
    }

    private Copy()
    {
    }
}
```

15.2.5 User class

The User class identifies a borrower and is used to define user rights or access restrictions.

```
Package com.corejdo.casestudy.model;

/**
 * A user object represents a user of the
 * library. That may be either a
 * borrower or a librarian (or both).
 *
 * <BR><FONT Color="red"><B>Persistent</B></FONT>
 */
public class User
{
    public User()
    {
    }
    /**
     * Name of user.
     */
    private String firstName;
    /**
     * Name of user.
     */
    private String lastName;
    /**
     * Cleartext password.
     */
    private String password;
    /**
     * Internally used user id.
     */
    private String id;
    /**
     * The rights of the user.
     */
    private Rights rights;

    public String getFirstName()
    {
        return firstName;
    }
    public void setFirstName(String firstName)
    {
        this.firstName = firstName;
    }
    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }
    public String getLastName()
    {
        return lastName;
    }
    public void setPassword(String password)
    {
        this.password = password;
    }
    public String getId()
    {
        return id;
    }
    public void setId(String id)
    {
        this.id = id;
    }

    public boolean checkPassword(String pw)
    {
        return pw.equals(password);
    }

    public Rights getRights()
    {
        if (rights == null) rights = new Rights();
        return rights;
    }

    public String toString()
    {
        return firstName+" "+lastName;
    }
}
```

15.2.6 Right and Rights classes

The Right class is an example of the *Enumeration Pattern.* It implements a tri-state enumeration (ALLOWED, DENIED, DEFAULT). The Right class itself cannot be made persistent, because the JDO Implementation will not be able to return the constant references for ALLOWED, DENIED and DEFAULT, which are static objects.

```
Package com.corejdo.casestudy.model;

/**
 * A transient tri-state value.
 * It may be set either to allow, deny or default.
 * <BR><B>not persistent</B>
 */
public final class Right
{

    boolean isAllowed(Right def)
    {
        if (this != DEFAULT) return this == ALLOWED;
        if (def == null || def == DEFAULT) return false;
        return def == ALLOWED;
    }

    boolean isDenied(Right def)
    {
        if (this != DEFAULT) return this == DENIED;
        if (def == null || def == DEFAULT) return true;
        return def == DENIED;
    }

    public static Right DENIED = new Right(-1);
    public static Right ALLOWED = new Right(1);
    public static Right DEFAULT = new Right(0);

    private int code;

    private Right(int code)
    {
        this.code = code;
    }
}
```

The Rights class maps the static Right objects to appropriate integer values:

```
package com.corejdo.casestudy.model;

import javax.jdo.InstanceCallbacks;
import javax.jdo.JDOHelper;
/**
 * Defines access restrictions for users and
 * user groups of the library.
 *
 * This is an example for a type mapping:
 * Instead of defining another persistent
 * class Right, the tri-state values
 * (allow, deny, default) are coded into plain
 * integer values, (+1, -1, 0 respectively).
 * For a better programming model, the
 * Right class is exposed to the application,
 * but stored internally as integer
 * values.
 *
 * <BR><FONT Color="red"><B>Persistent</B></FONT>
 */
public class Rights
    implements Cloneable
{
    /**
     * The right to add, remove or
     * modify users of the library.
     */
    private int _modifyUser;

    /**
     * The right to borrow something.
     */
    private int _borrow;

    /**
     * The right to manage publications.
     */
    private int _manage;



    /**
     * The constructor creates a default Rights object.
     * (An unauthorized user.)
     */

    public Rights()
    {
    }

    private int rightToInt(Right r)
    {
        if (r == Right.ALLOWED) return 1;
        if (r == Right.DEFAULT) return 0;
        if (r == Right.DENIED) return –1;
        throw new RuntimeException("Illegal value: "+r);
    }

    private Right intToRight(int i)
    {
        if (i == 1) return Right.ALLOWED;
        if (i == 0) return Right.DEFAULT;
        if (i == -1) return Right.DENIED;
        throw new RuntimeException("Illegal value: "+i);
    }

    public boolean canBorrow()
    {
        return intToRight(
                   _borrow).isAllowed(Right.ALLOWED);
    }
    public boolean canManage()
    {
        return intToRight(_manage).isAllowed(Right.DENIED);
    }
    public boolean canModifyUser()
    {
        return intToRight(
                   _modifyUser).isAllowed(Right.DENIED);
    }

    public void setBorrow(Right right)
    {
        _borrow = rightToInt(right);
    }

    public void setManage(Right right)
    {
        _manage = rightToInt(right);
    }

    public void setModifyUser(Right right)
    {
        _modifyUser = rightToInt(right);
    }
    public Rights copy()
    {
        try {
            int k = this._borrow;
            Rights r = (Rights)this.clone();
            return r;
        }
        catch (CloneNotSupportedException c) {
            throw new RuntimeException("Unexpected: "+c);
        }
    }

    public String toString()
    {
        String res =
         ( canBorrow() ? "borrow" : "noborrow")+
         ( canManage() ? ",manage" : ",nomanage")+
         ( canModifyUser() ? ",modify" : ",nomodify");
        return res;
    }
}
```

15.3 Use-case Package

This package contains the implementation of the application use-cases. It uses an abstract base class based on the command pattern that implements common methods for all transactions and defines an abstract execute method. Each use-case class extends this abstract class and implements the execute method.

15.3.1 AbstractUserCase class

The base class for the use-cases implements the basic methods to get the PersistenceManagerFactory and PersistenceManager instances, begin and commit transactions, and handle exceptions.

15.3.1.1 PerstistenceManagerFactory bootstrapping

As shown in earlier chapters, PersistenceManagerFactory bootstrapping can be done using a singleton pattern and reading properties from a file to avoid hard coding connection or JDO implementation specific details.

```
/**
 * Creates or returns a JDO factory.
 * The properties are loaded
 * from a file.
 */
public static PersistenceManagerFactory
  getPersistenceManagerFactory() {

    if (pmf == null) {

      String filename =
        System.getProperty("jdo.properties");

      if (filename == null) {

        throw new JDOFatalUserException(
          "System property 'jdo.properties' not defined");
      }

      Properties properties = new Properties();

      try {

        FileInputStream file =
          new FileInputStream(filename);

        properties.load(file);

        file.close();

        pmf = JDOHelper.getPersistenceManagerFactory(
          properties);
      }

      catch (java.io.IOException e) {

        throw new JDOFatalUserException(
          "Error reading '" + filename + "'",
          e);
      }
    }
    return pmf;
  }
```

15.3.1.2 Obtaining the PersistenceManager

This method uses a singleton pattern to maintain a reference to a PersistenceManager instance for a given use-case instance.

```
/**
 * Creates or returns a PersistenceManager.
 * If a servlet helper is associated with
 * the current use case, a persistence manager
 * may be taken from the helper object.
 */
PersistenceManager getPM()
{
    if (currentPersistenceManager == null) {
        currentPersistenceManager =
           getPMF().getPersistenceManager();
    }
    return currentPersistenceManager;
}
```

15.3.1.3 Closing the PersistenceManager

This method releases the singleton PersistenceManager instance at the end of the use-case.

```
/**
 * Called by outside application management.
 *
 * In case of servlets, this method is called
 * on session timeout.
 * The command line version calls this method
 * at system exit.
 */
void close()
{
    if (currentPersistenceManager != null) {
        currentPersistenceManager.close();
        currentPersistenceManager = null;
    }
}
```

15.3.1.4 Execute use-case

This method runs the use-case; it gets the PersistenceManager instance, begins a transaction, and then calls the internal execute method to run the use-case. It then commits the transaction and handles any exceptions. If an exception is caught, the transaction is rolled back.

```
/**
 * This method creates a transaction context,
 * starts the transaction
 * and commits all work, after the <tt>_execute</tt>
 * method has been
 * successfully invoked. If the subclass throws
 * an exception, the transaction is aborted.
 */
public final void execute()
{
    if (!isAllowed())
       throw new UseCaseException("Not allowed.");
    getPM().currentTransaction().begin();
    boolean active = true;
    try {
        _execute();
        getPM().currentTransaction().commit();
        active = false;
    }
    finally {
        if (active) {
            try {
                getPM().currentTransaction().rollback();
                active = false;
            }
            catch (JDOUserException ex)
            {
                ex.printStackTrace() ;
            }
        }
    }
}
```

15.3.2 AddBooks use-case

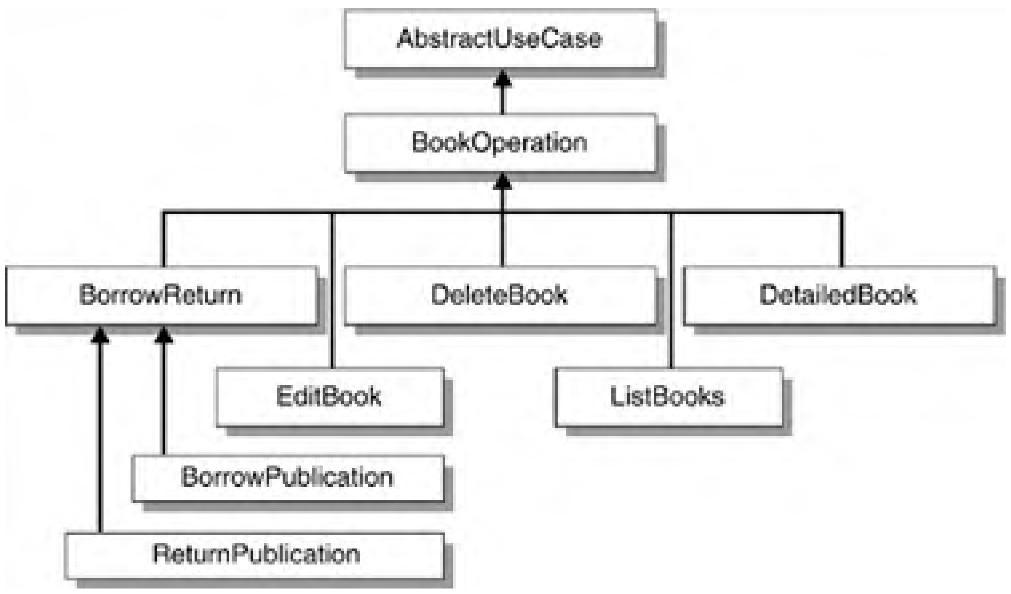
This first use-case is straightforward; it creates a new Book instance and makes it persistent.

```
Package com.corejdo.casestudy.usecase;

import com.corejdo.casestudy.model.Book;
import com.corejdo.casestudy.model.User;
import com.corejdo.casestudy.tools.CmdLine;

import java.util.Date ;
import java.text.DateFormat ;
import java.text.ParseException;

/**
 * This class adds a book to the library.
 * It implements a bean pattern, which can
 * be used simply in the JSP.
 */

public class AddBook
    extends AbstractUseCase
{
    /**
     * This is the book that gets inserted.
     * While it is transient, we
     * can set and get values directly.
     */
    Book book = new Book();

    public AddBook()
    {
    }

    public void setISBN(String ISBN)
    {
        book.ISBN = ISBN;
    }
    public void setAuthor(String author)
    {
        book.author = author;
    }
    public void setCopyright(String copyright)
    {
        book.copyright = copyright;
    }
    public void setPublished(String published)
    {
        book.published = parseDate(published);
    }
    public void setFirstPublished(String firstPublished)
    {
        book.firstPublished = parseDate(firstPublished);
    }

    public void setTitle(String title)
    {
        book.title = title;
    }

    /**
     * Adds a book via command line.
     */
    public static void main(String[] args) {
        new AddBook().runCmdLine(args);
    }

    protected void runCmdLine(String[] args) {
        setTitle(CmdLine.getStringVar("Title",""));
        setISBN(CmdLine.getStringVar("ISBN",""));
        setAuthor(CmdLine.getStringVar(
                "Author","Heiko Bobzin"));
        setCopyright(CmdLine.getStringVar(
                "Copyright","© 2003 Prentice H"ll"));
        String now = new Date().toString();
        setPublished(CmdLine.getStringVar(
              "  "Publishing d"te",now));
        setFirstPublished(CmdLine.getStringVar(
              "  "First publishing d"te",now));
        execute();
    }

    private Date parseDate(String d)
    {
        try {
            return DateFormat.getInstance().parse(d);
        }
        catch (ParseException e)
        {
            throw new RuntimeExcepti"n("Illegal dat": "+d);
        }
    }

    public boolean isAllowed()
    {
        return getContext().userRights.canManage();
    }

    protected void _execute()
    {
        // this is a really simple operation:
        User user = getUser();
        book.insertedBy = user;
        getPM().makePersistent(book);
        // set for next insert:
        book = new Book();
    }

    public String getFirstPublished()
    {
        return new Date().toString();
    }
    public String getPublished()
    {
        return new Date().toString();
    }
}
```

15.4 BookOperation Class

The command pattern can be used to share common implementations of operations. This class provides an abstract implementation for common operations on books that other commands can share. Figure 15-3 shows the class diagram for a number of the use-cases.

Figure 15-3. Inheriting from AbstractUseCase.



This class implements a number of convenience methods that can be used to do the following:

- Query generic fields by passing "field" and "field value,"
- Return found Book instance as a subclass.
- Ensure that only a single instance result is returned; else an error is thrown.
- Copies of a book can be found by "back pointer" or join query; returns all copies that belong to a book.

```
Package com.corejdo.casestudy.usecase;

import com.corejdo.casestudy.model.*;
import javax.jdo.* ;
import java.util.* ;

/**
 * Basic class of all Book operations.
 * This class can find a single book.
 */

public abstract class BookOperation extends AbstractUseCase
{

    public BookOperation()
    {
    }

    /**
     * Searches for a book that matches a string field
     * @param field name of field in the book class.
     * @param fieldValue value to match
     * @return a book
     */
    protected Book findBook(String field,
                            String fieldValue)
    {
        PersistenceManager pm = getPM();
        // find the book to update.
        Query q = pm.newQuery(Book.class, field+" == val");
        q.declareParameters("String val");
        q.setCandidates(pm.getExtent(Book.class,true));
        Collection result =
                    (Collection)q.execute(fieldValue);
        if (result.size()<1) {
            throw new RuntimeException(
                    field+" not found: "+fieldValue);
        } else if (result.size()>1) {
            throw new RuntimeException(
                    field+" not unique: "+fieldValue);
        }
        Book book = (Book)result.iterator().next();
        return book;
    }

    protected final Book findBook()
    {
        Book book = null;
        try {
            book = findBook("ISBN",this.findISBN);
        } catch (Exception e) {
            book = findBook("title",this.findTitle);
        }
        return book;
    }

    /**
     * Find all copies of a publication.
     */
    protected Copy[] findCopies(Publication publication)
    {
        PersistenceManager pm = getPM();
        // find all copies for a publication
        Query q = pm.newQuery(Copy.class, "copyOf == val");
        q.declareParameters(
            "com.corejdo.casestudy.model.Publication val");
        q.setCandidates(pm.getExtent(Copy.class,true));

        Collection result =
             (Collection)q.execute(publication);
        Copy copies[] = new Copy[result.size()];
        result.toArray(copies);
        return copies;
    }

    public void setFindTitle(String findTitle)
    {
        this.findTitle = findTitle;
    }
    public void setFindISBN(String findISBN)
    {
        this.findISBN = findISBN;
    }
    private String findTitle;
    private String findISBN;
}
```

15.4.1 ListBooks use-case

This use-case shows how to display a list of all books or books that match a specified filter.

```
Package com.corejdo.casestudy.usecase;

import com.corejdo.casestudy.tools.CmdLine;
import com.corejdo.casestudy.model.*;
import javax.jdo.* ;
import java.util.* ;

/**
 * This class lists books.
 *
 */

public class ListBooks
    extends BookOperation
{
    int startNumber = 0;
    int count = 10;
    List result;

    String filterAuthor = "";
    Query query;
    Extent extent;

    /**
     * Sets the start offset. Since JDO does not
     * support a "seek" operation on
     * extents, we implemented our own by simply
     * calling iterator.next().
     * @param i
     */
    public void setStartNumber(int i)
    {
        startNumber = i;
    }

    /**
     * Sets the number of entries
     * returned by this operation.
     * @param i number of entries
     */
    public void setCount(int i)
    {
        count = i;
    }

    /**
     * This data object is used to return the results.
     */
    public class BookDTO
    {
        public final String title;
        public final String copyright;
        public final String ISBN;
        public final String author;

        BookDTO(Book b)
        {
            this.title = b.title;
            this.copyright = b.copyright;
            this.ISBN = b.ISBN;
            this.author = b.author;
        }

        public String toString()
        {
            return "Title:  "+title+
                   "\n   Author: "+author+
                   "\n   ISBN:   "+ISBN+
                   "\n   ©: "  "+copyright;
        }
    }

    /**
     * Iterates over elements of the book Extent.
     * Called by the framework's execute() method.
     */
```

```
protected void _execute()
{
    result = new ArrayList();
    extent = getPM().getExtent(Book.class,true);
    query = null;
    Iterator iter = filter(extent);
    int i = this.startNumber;
    while (i-- > 0 && iter.hasNext()) iter.next();
    i = this.count;
    while (i-- > 0 && iter.hasNext()) {
        result.add(new BookDTO((Book)iter.next()));
    }
    if (query != null) {
        query.close(iter);
    } else {
        extent.close(iter);
    }
}

protected Iterator filter(Extent extent)
{
    if (filterAuthor != null) {
        query = getPM().newQuery(
                    Book.clas", "author == "al");
        query.declareParameters(
                «   "java.lang.String  »a l");
        query.setCandidates(exte nt);
        Collection result = (Collection)query.execute(
                    filterAuthor);
        return result.iterator();
    } else {
        return extent.iterator();
    }
}

/**
 * Command line version.
 */
public static void main(String[] args) {
    new ListBooks().runCmdLine(args);
}

protected void run(String[] args) {
    setStartNumber(CmdLine.getIntV"r("Start"at",0));
    setCount(CmdLine.getIntV"r("Co"nt",10));
    execute();
    System.out.print"n("Resul": ");
    Iterator iter = getResult();
    int i = 1;
    while (iter.hasNext()) {
        System.out.print""("""i+": "+iter.next());
        i++;
    }
}

/**
 * The resulting iterator returns BookDTO objects.
 */
public Iterator getResult()
{
    return result.iterator();
}
public int getCount()
{
    return count;
}
public int getStartNumber()
{
    return startNumber;
}
public String getFilterAuthor()
{
    return filterAuthor;
}
public void setFilterAuthor(String filterAuthor)
{
        this.filterAuthor = filterAuthor;
    }
}
```

15.4.2 DetailedBook use-case

This use-case gets the detailed information about a particular book; it encapsulates a Book instance and provides an interface to get the detailed information from it without exposing the Book instance itself.

```
package com.corejdo.casestudy.usecase;

import com.corejdo.casestudy.model.*;
import com.corejdo.casestudy.tools.*;

/**
 * This class gets detailed information about a book.
 * It is a good example for information hiding.
 * The actual book data
 * in the persistent model Book class
 * uses a User field for the
 * borrower information, but unauthorized users must
 * not be able to see detailed information about borrowers.
 */
public class DetailedBook extends BookOperation
{
    public DetailedBook()
    {
    }

    public static void main(String[] args)
    {
        new DetailedBook().runCmdLine(args);
    }

    /**
     * Command line version of the book editor.
     */
    protected void run(String[] args)
    {
        setFindISBN(CmdLine.getStringVar(
            "  "Find I"B"","928394-234-23"44"));
        setFindTitle(CmdLine.getStringVar(
            "  "Find Ti"l"",""));
        execute();
    }

    /**
     * First, looks up a book, then sets return value
     * for detailed book information.
     * The implementation of this
     * method looks quite simple because JDO does a
     * lot of things for us, like fetching other objects
     * on demand.
     */
    protected void _execute()
    {
        Book book = findBook();
        this.ISBN = book.ISBN;
        this.author = book.author;
        this.title = book.title;
        this.published = book.published.toString();
        this.firstPublished =
                    book.firstPublished.toString();
        Copy copies[] = findCopies(book);
        availableCopies = 0;

        for (int i = 0; i < copies.length; i++) {
            if (copies[i].borrower == null)
                    availableCopies++;
        }

        this.isBorrowed = availableCopies > 0;
    }

    public String getAuthor()
    {
        return author;
    }
    public int getAvailableCopies()
    {
        return availableCopies;
    }
    public String getFirstPublished()
    {
        return firstPublished;
    }
    public String getISBN()
    {
        return ISBN;
    }
    public boolean isIsBorrowed()
    {
        return isBorrowed;
    }
    public String getPublished()
    {
        return published;
    }
    public String getTitle()
    {
        return title;
    }

    String ISBN;
    String author;
    String title;
    String published;
    String firstPublished;
    int availableCopies;
    boolean isBorrowed;
}
```

15.4.3 EditBook use-case

Similar to the previous use-case, this one allows a particular Book instance to be modified. Again, access to the actual Book instance is encapsulated by this class.

```
package com.corejdo.casestudy.usecase;


import com.corejdo.casestudy.model.Book;
import com.corejdo.casestudy.tools.CmdLine;

import javax.jd o.*;

import java.uti l.*;
import java.text.DateFor mat;
import java.text.ParseExcept ion;

/**
 * This class is used to change properties of books.
 */

public class EditBook extends BookOperation
{
    public static void main(String[] args)
    {
        new EditBook().runCmdLine(args);
    }

    /**
     * Command line version of the book editor.
     *
     */
    protected void run(String[] args)
    {
        setFindISBN(CmdLine.getStringVar(
             "  "Find I"B"","928394-234-23"44"));
        setFindTitle(CmdLine.getStringVar(
             "  "Find Ti"l"",""));
        setISBN(CmdLine.getStringVar(
             "  "I"B"","928394-234-23"44"));
        setTitle(CmdLine.getStringVar(
             "  "Ti"l"","new Ti"le"));
        setAuthor(CmdLine.getStringVar(
             "  "Aut"o"","H. Bob"in"));
        setCopyright(CmdLine.getStringVar(
             "  "Copyri"h"","(C) 2003 Prentice H"ll"));
        execute();
    }

    /**
     * First, looks up a book, then updates the fields with
     * the values of this bean.
     */
    protected void _execute()
    {
        Book book = findBook();
        book.title = this.title;
        book.author = this.author;
        book.copyright = this.copyright;
        book.ISBN = this.ISBN;
    }

    public void setISBN(String ISBN)
    {
        this.ISBN = ISBN;
    }
    public void setAuthor(String author)
    {
        this.author = author;
    }
    public void setTitle(String title)
    {
        this.title = title;
    }
    public void setCopyright(String copyright)
    {
        this.copyright = copyright;
    }
    private String ISBN;
    private String author;
    private String title;
    private String copyright;
}
```

15.4.4 DeleteBook use-case

This use-case simply deletes the specified Book instance. Interestingly, it doesn't try to find all the Copy instances that reference this Book and delete them, although this would be a fairly straightforward addition. Instead, the Copy instances are left as-is, now referencing a delete persistent object.

```
package com.corejdo.casestudy.usecase;
import com.corejdo.casestudy.tools.*;
import com.corejdo.casestudy.model.*;
import javax.jd o.*;
import java.uti l.*;

public class DeleteBook extends BookOperation
{
    public static void main(String[] args)
    {
        new DeleteBook().runCmdLine(args);
    }

    /**
     * Command line version.
     */
    protected void run(String[] args)
    {
        setFindISBN(CmdLine.getStringVar(
         "  "Find I"B"","928394-234-23"44"));
        setFindTitle(CmdLine.getStringVar(
         "  "Find Ti"l"","");
        execute();
    }

    public boolean isAllowed()
    {
        return getContext().userRights.canManage();
    }

    /**
     * First, looks up a book, then delete it.
     */
    protected void _execute()
    {
        Book book = findBook();
        PersistenceManager pm = getPM();
        pm.deletePersistent(book);
    }
}
```

15.4.5 BorrowReturn class

This abstract class implements behavior common to borrowing or returning a book.

```
package com.corejdo.casestudy.usecase;

import com.corejdo.casestudy.model.*;
import com.corejdo.casestudy.tools.CmdLine;

import javax.jd o.*;

import java.uti l.*;
import java.text.DateFor mat;
import java.text.ParseExcept ion;

/**
 * The common operations "f "bor"ow"
 * a"d "ret"rn" are implemented in this class.
 */

public abstract class BorrowReturn extends BookOperation
{
    public boolean isAllowed()
    {
        if (!getContext().isLoggedIn()) return false;
        return getContext().userRights.canBorrow();
    }

    /**
     * First, looks up a book, then checks for a copy.
     * If copy exists, the book is borrowed or returned.
     */
    protected void _execute()
    {
        Book book = findBook();
        process(book);
    }

    protected abstract void process(Book book);

    public String getLocation()
    {
        return location;
    }
    public String getReturnDate()
    {
        return returnDate;
    }
    protected String returnDate;
    protected String location;
}
```

15.4.6 Borrow use-case

This use-case tries to find available copies of a book and allocates one to the borrower.

```
package com.corejdo.casestudy.usecase;

...
    public final long borrowTime
              = 1000L*60L*10L; // ten minutes

    protected void process(Book book)
    {
        Copy copy[] = findCopies(book);
        for (int i = 0; i < copy.length; i++) {
            Copy c = copy[i];
            if (c.borrower == null) {
                c.borrower = getContext().user;
                c.borrowDate = new Date(); // now.
                c.returnDate = new Date(
                      c.borrowDate.getTime()+borrowTime);
                this.returnDate = c.returnDate.toString();
                this.location = c.location;
                return;
            }
        }
        throw new UseCaseException(
          "  "No copy available. Please ask cle"k.");
    }
}
```

15.4.7 Return use-case

This use-case tries to find the copy of a book that was previously borrowed and returns it.

```
package com.corejdo.casestudy.usecase;

...
    protected void process(Book book)
    {
        Copy copy[] = findCopies(book);
        for (int i = 0; i < copy.length; i++) {
            Copy c = copy[i];
            if (c.borrower == getContext().user) {
                c.borrower = null;
                this.location = c.location;
                return ;
            }
        }
        throw new UseCaseException(
          "  "This book is not borrowed "y "+
            getContext().us"r+". Please ask cle"k.");
    }
```

15.5 Putting Things Together

The project can be compiled by ant (http://jakarta.apache.org), it but must be modified slightly to use other JDO implementations. A build.xml file can be found in the top-level directory.

15.5.1 XML metadata

The model.jdo metadata file is located in the src/com/corejdo/casestudy directory and follows the JDO 1.0 naming convention.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects
Metadata 1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
   <package name="com.corejdo.casestudy.model">
     <class name="Book">
     </class>
     <class name="CD">
     </class>
     <class name="Copy">
     </class>
     <class name="Library">
     </class>
     <class name="Publication">
     </class>
     <class name="Rights">
     </class>
     <class name="Song">
     </class>
     <class name="User">
     </class>
   </package>
</jdo>
```

15.5.2 Running in command-line mode

The command-line application can also be started by an ant target.

```
$ ant run

Use case [ListBooks]: Type name of use case here!
Start at [0]:
Count [10]:

Result:
1: Title:  Travel Guide: Sweden
   Author: Joseph, Bert
   ISBN:   7-7333-4727-6
   (C):    O'Renny
...
```

The datastore can be filled with random data by the FillDatabase command. For some operations, you have to log in. The initial user name is "su," and the password is "su." Here is how to add new users:

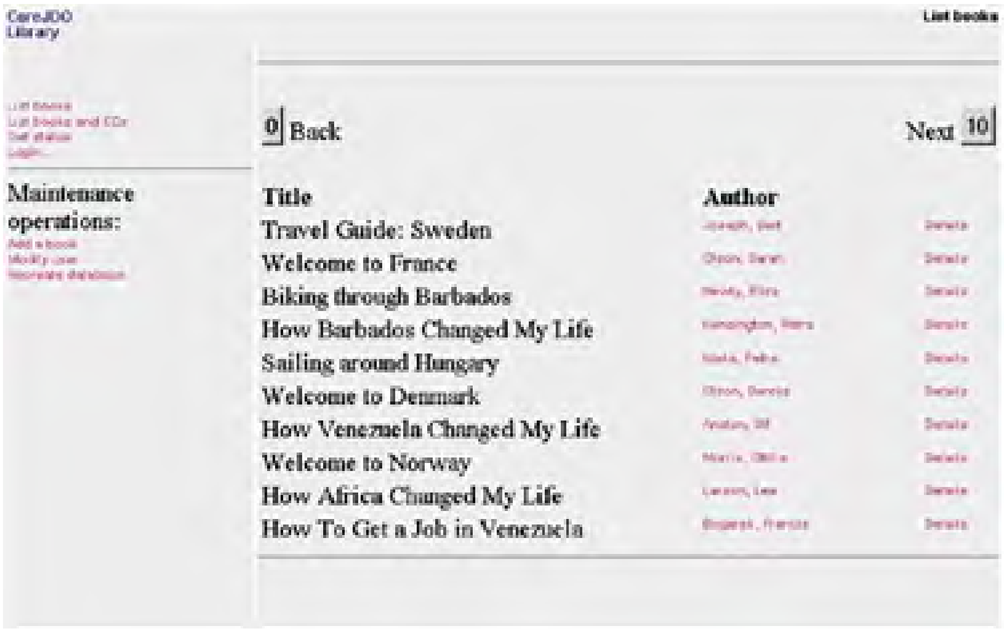
```
Use case [ListBooks]: Login
UserId []: su
Password []: su
Use case [Login]: EditUser
Find User Id [hb]: smith
User not found.
Add [false]: true
First name: John
Last name: Smith
Password: js
```

```
May borrow [false]: true
May manage publications [false]: true
May manage users [false]: true
Use case [EditUser]: _
```

15.5.3 Running the Servlets

Figure 15-4: A running Servlet shows a screenshot of a Web page that is generated from the ListBooks.jsp code. Again, ant can be used to start the tomcat Web server. The pages are compiled on demand. Please make sure that the Java SDK is set in the classpath; it is needed by the tomcat JSP engine.

Figure 15-4. A running Servlet.



CoreJDO Library

List books

List books
List books and CDs
Get status
Login...

Maintenance operations:
Add a book
Modify user
Recreate database

0 Back   Next 10

Title	Author	
Travel Guide: Sweden	[illegible]	Details
Welcome to France	[illegible]	Details
Biking through Barbados	[illegible]	Details
How Barbados Changed My Life	[illegible]	Details
Sailing around Hungary	[illegible]	Details
Welcome to Denmark	[illegible]	Details
How Venezuela Changed My Life	[illegible]	Details
Welcome to Norway	[illegible]	Details
How Africa Changed My Life	[illegible]	Details
How To Get a Job in Venezuela	[illegible]	Details

$ ant servlet starts the server.

Appendix A. JDO States

The tables on the following pages show mandatory and optional states, the transitions, and the operations that cause state changes.

A.1 How the Tables Should Be Read

Take a look at the first field in the first column (transient) and first row (makePersistent). This field means that if PersistenceManager.makePersistent is called with an instance, which state is transient, the resulting state is persistent-new. To keep the table short, not all operations are explicitly listed. For instance, PersistenceManager.makePersistentAll(Collection c) and PersistenceManager.makePersistentAll(Object[] o) have the same effect on instances contained in the collection or array, just as makePersistent()has on a single instance. The same is true for deletePersistent, makeTransient, makeTransactional, and makeNontransactional, respectively.

If a field is marked "Error!", the corresponding operation on an instance is not allowed and causes a JDOUserException to be thrown. The state will not change.

If a field is marked "Impossible", the operation cannot be performed on an instance because the initial state is not reachable. For instance, there are no instances that have the state persistent-new outside of active transactions, so "read fields outside of active transactions" cannot be performed.

If a "–" (dash) appears in a field, the state does not change.

Table A-1. State Transitions

Current State	Transient	Persistent-new	Persistent-clean	Persistent-dirty	Hollow
Method					
makePersistent	persistent-new	–			
deletePersistent	error!	persistent-new-deleted	persistent-deleted		
makeTransactional	transient-clean	–			
makeNonTransactional	error!		persistent-nontrans.	error!	–
makeTransient	–	error!	transient	error!	transient
commit	–	Hollow			–
commit (retain values)	–	persistent-nontrans.			–
rollback	–	transient	Hollow		–
rollback (retain values)	–	transient	persistent-nontrans.		–
refresh (active)	–			persistent-clean	–
refresh (optimistic)	–			persistent-nontrans.	–
evict	n/a	–	hollow	–	
read field (outside)	–	impossible			persistent-nontrans.
read field (optimistic)	–				persistentpnontrans.
read field (active)	–				persistent-clean
write field (outside)	–	impossible			persistent-nontrans.
write field (active)	–		persistent-dirty	–	persistent-dirty
retrieve (outside or optimistic)	–				persistent-nontrans.
retrieve (active)	–				persistent-clean

Table A-2. State Transitions

Current State	Transient-clean	Transient-dirty	Persistent-new-deleted	Persistent-deleted	Persistent-nontrans.
Method					
makePersistent	persistent-new		–		

deletePersistent	error!		–		persistent-deleted
makeTransactional	–				persistent-clean
makeNonTransactional	Transient	error!			–
makeTransient	–		error!		transient
commit	–	transient-clean	transient		
commit (retain values)	–	transient-clean	transient		
rollback	–	transient-clean	transient	hollow	–
rollback (retain values)	–	transient-clean	transient	persistent-nontrans.	–
refresh (active)	–				
refresh (optimistic)	–				
evict	–				hollow
read field (outside)	–	impossible			–
read field (optimistic)	–		error!		–
read field (active)	–		error!		persistent-clean
write field (outside)	–	impossible			–
write field (active)	transient-dirty	–	error!		persistent-dirty
retrieve (outside or optimistic)	–				
retrieve (active)	–				persistent-clean

Appendix B. XML Metadata

JDO requires that additional metadata be specified for any persistence-capable classes. As has been seen in earlier chapters, this metadata is contained in an XML file. This appendix provides a reference to this metadata, explaining where the metadata file should be located and what the different elements contained within the metadata imply.

This appendix covers the following topics:

- Where the metadata should be located.
- What each metadata element implies.
- The XML DTD for the metadata.
- An example of a metadata file.

B.1 Metadata Location

JDO metadata is specified in an XML file that must be available both during the enhancement process and at runtime. Because JDO can be used in many different application architecture scenarios, the location of the metadata file is quite flexible.

The metadata for a persistence-capable class can be located in a metadata file specific to the class or in a file that contains metadata for all persistence-capable classes in a given package hierarchy. A metadata file specific to a single class must be named after the class itself, but with a .jdo suffix. For example, the metadata for an Author class could be located in a file called Author.jdo. Alternatively, the metadata could be located in a file called package.jdo. This file would then contain the metadata for the Author class, as well as any other persistence-capable class in the same package hierarchy. In both situations, the metadata file should be available at runtime as a resource that can be loaded by the same class loader that loaded the class itself (i.e., it should be in the classpath). For example, the metadata for the class com.corejdo.examples.model.Author could be contained in one of the following files:

- package.jdo
- com/package.jdo
- com/corejdo/package.jdo
- com/corejdo/examples/package.jdo
- com/corejdo/examples/model/package.jdo
- com/corejdo/examples/model/Author.jdo

In addition to the above, the following locations are also supported:

- META-INF/package.jdo
- WEB-INF/package.jdo

The search order begins with META-INF/package.jdo, then goes to WEB-INF/package.jdo, followed by package.jdo, and then all the way down to com/corejdo/examples/model/Author.jdo. The first definition found for a given persistence-capable class is used. If no definition is found, a class is assumed to not be persistence capable.

The initial JDO 1.0 specification did not actually mandate the location and name of the XML metadata files. To aid in portability, the 1.0.1 maintenance release of the specification changed this to what has just been described.

Prior to this release, the name of a metadata file that contained multiple classes was the package name itself with a .jdo suffix, and it would have been located in the directory that contained the package directory. In the Author example, it would have been called model.jdo and would have been located in the com/corejdo/examples/ directory. JDO 1.0 implementations will still use this naming convention, and for backward compatibility, some JDO 1.0.1 implementations may also still allow this naming convention to be used.

B.2 Metadata Elements

The XML DTD for the JDO metadata defines the various elements that can be specified in the metadata file. In most cases, default values are defined that avoid having to specify values explicitly, except where the default is not appropriate. This keeps the average metadata file short and simple.

B.2.1 File header

A metadata file requires a header that specifies the XML version and encoding along with the location of the metadata XML DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
```

The DTD location can be the public URL as published by Sun Microsystems, or it can be a DTD file stored elsewhere. The JDO 1.0.1 maintenance release defines two standard locations for the DTD. The public location is as follows:

[View full width]

```
<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN" "http://java.sun.com
➥ /dtd/jdo_1_0.dtd">
```

A system location could be as follows:

```
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd"
```

If either of these two locations is used, then the JDO implementation may optimize access to the DTD by using its own local copy.

B.2.2 ELEMENT jdo

All other metadata elements are contained within the jdo element:

```
<jdo>
</jdo>
```

The jdo element should contain one or more package elements and can contain zero or more extension elements.

B.2.3 ELEMENT package

A package element specifies the name of a package and contains the metadata for the persistence-capable classes within that package. The name attribute is used to specify the package name:

```
<package name="com.corejdo.examples.model">
<\package>
```

The package element should contain one or more class elements and can contain zero or more extension elements.

B.2.4 ELEMENT class

A class element specifies the name of a persistence-capable Java class and contains the metadata for specific fields within the class. The name attribute is used to specify the class name:

```
<class name="Author">
</class>
```

In addition to the name attribute, the class element has a number of optional attributes that can be specified:

> **identity-type:** Can be application, datastore, or nondurable. Specifies the object identity type for the class. Defaults to the identity type of the persistence-capable-superclass class, if one is specified. Otherwise, if objectid-class is specified, this defaults to application; otherwise, it defaults to datastore.
>
> **objectid-class:** The name of the class that implements the application identity for the persistence-capable class. This attribute is applicable only when identity-type is specified as application. Defaults to the class specified by the persistence-capable-superclass class, if identity-type is application.
>
> The class name follows Java rules for naming: If no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the $ marker.
>
> **requires-extent:** Can be true or false. Specifies whether an Extent is required for the class. Defaults to true.
>
> **persistence-capable-superclass:** The name of the persistence-capable superclass of the class. If omitted, it is assumed that there is no superclass in the inheritance hierarchy that is persistence capable.
>
> The class name uses Java rules for naming: If no package is included in the name, the package name is assumed to be the same package as the persistence-capable class.

The class element may contain zero or more field or extension elements.

B.2.5 ELEMENT field

A field element specifies the name of a field of the persistence-capable class and contains metadata about that field. The name attribute is used to specify the field name:

```
<field name="books">
</field>
```

The following optional attributes can also be specified:

> **persistence-modifier:** Can be persistent, transactional, or none. Specifies whether the field should be stored in the datastore, not stored in the datastore but managed transactionally, or simply not managed at all. A transactional field's value is subject to rollback, and a field whose persistence-modifier is none is effectively ignored by the JDO implementation.
>
> Fields declared as being static, transient, or final default to none. Fields of primitive types, wrapper types, and supported system interfaces or classes (see Chapter 3 for more details) default to persistent. The same is true for fields that reference persistence-capable classes. All other fields default to none.
>
> **primary-key:** Can be true or false. Identifies the field as being part of the application identity for the class. Can be specified only for fields whose persistence-modifier is persistent. Defaults to false.
>
> **default-fetch-group:** Can be true or false. Specifies whether the field should be part of the default fetch group for the class. Can be specified only for fields whose persistence-modifier is persistent. Defaults to true for fields of primitive types, wrapper types, supported types in the java.lang and java.math packages and java.util.Date. All other fields default to false.
>
> **embedded:** Can be true or false. Provides a hint to the JDO implementation as to whether the field should be stored as part of the containing instance. Can be specified only for fields whose persistence-modifier is persistent. Fields of primitive types, wrapper types, and supported system interfaces or classes (see Chapter 3 for more details) default to true. The same is true for fields that are arrays of these types. All other fields default to false.
>
> Specifying true for a field that is a reference to a persistence-capable class implies containment—that the embedded instance has no independent existence in the datastore and is not part of its class's Extent. But this behavior is not further specified by JDO and is not portable.
>
> **null-value:** Can be none, exception, or default. Specifies how null values should be handled in the datastore. If none, then null values are stored in the datastore as null; if the datastore does not support null, then a JDOUserException is thrown. If exception, then a JDOUserException is thrown if the field has a null value when stored. If default, then a null value is converted to the datastore's default representation when stored—therefore, a null value may not be distinguishable from a default value. Defaults to none.

The field element may contain a collection, map, or array element and zero or more extension elements.

B.2.6 ELEMENT collection

A collection element can be used for java.util.Collection fields to specify additional metadata about the elements within the collection. It can specify the class name of the instances contained within the collection, as well as whether those instances should be stored as part of the containing instance:

```
<collection>
</collection>
```

The following optional attributes can also be specified:

> **element-type:** The class name of the instances contained within the collection. Defaults to Object if not specified.
>
> The class name follows Java rules for naming: If no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the $ marker.
>
> **embedded-element:** Can be true or false. Specifies whether the instances contained within the collection should be stored as part of the containing instance. Defaults to false for collections of references to persistence-capable classes and interfaces; otherwise, defaults to true.
>
> Specifying true for a collection of references to a persistence-capable class implies containment—that the embedded instances have no independent existence in the data store and are not part of their class's Extent. But this behavior is not further specified by JDO and is not portable.

The collection element may contain zero or more extension elements.

B.2.7 ELEMENT map

A map element can optionally be used for fields of type java.util.Map to specify additional metadata about the elements within the map. It can specify the class name of the instances contained within the map (both keys and values), as well as whether those instances should be stored as part of the containing instance:

```
<map>
</map>
```

The following optional attributes can also be specified:

> **key-type & element-type:** The name of the class of the key or value instances contained within the map. Defaults to Object if not specified.
>
> The class name follows Java rules for naming: If no package is included in the name, the package name is assumed to be the same package as the persistence-capable class. Inner classes are identified by the $ marker.
>
> **embedded-key & embedded-value:** Can be true or false. Specifies whether the key or value instances contained within the map should be stored as part of the containing instance. Defaults to false for maps of references to persistence-capable classes and interfaces; otherwise, it defaults to true.
>
> Specifying true for a map of references to a persistence-capable class implies containment—that is the embedded instances have no independent existence in the datastore and are not part of their class's Extent. But this behavior is not further specified by JDO and is not portable.

The map element may contain zero or more extension elements.

B.2.8 ELEMENT array

An array element can optionally be used for array fields to specify additional metadata about the elements within the array. It can specify whether those instances should be stored as part of the containing instance:

```
<array>
</array>
```

The following optional attributes can also be specified:

> **embedded-element:** Can be true or false. Specifies whether the instances contained within the array should be stored as part of the containing instance. Defaults to false for arrays of references to persistence-capable classes and interfaces; otherwise, it defaults to true.
>
> Specifying true for an array of references to a persistence-capable class implies containment—that the embedded instances have no independent existence in the datastore and are not part of their class's Extent. But this behavior is not further specified by JDO and is not portable.

The collection element may contain zero or more extension elements.

B.2.9 ELEMENT extension

Each of the elements already outlined can contain zero or more extension elements. This element allows additional metadata to be specified that is specific to a particular JDO implementation. The vendor-name attribute is used to identify a vendor:

```
<extension vendor-name="JDORI">
</extension>
```

The vendor name JDORI is reserved for use by the JDO reference implementation. The following optional attributes can also be specified:

> **key:** A string that is specific to the given JDO implementation.
>
> **value:** A string that is specific to the given JDO implementation.

An extension element may be ignored by any JDO implementation.

B.3 XML DTD

The following is the XML DTD for the JDO metadata:

[View full width]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN" "http://java.sun.com
 /dtd/jdo_1_0.dtd">
<!ELEMENT jdo ((package)+, (extension)*)>
<!ELEMENT package ((class)+, (extension)*)>
<!ATTLIST package name CDATA #REQUIRED>
<!ELEMENT class (field|extension)*>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type
  (application|datastore|nondurable) #IMPLIED>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass
  CDATA #IMPLIED>
<!ELEMENT field ((collection|map|array)?, (extension)*)?>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier
  (persistent|transaction-al|none) #IMPLIED>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field null-value (exception|default|none) 'none'>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>
<!ELEMENT collection (extension)*>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>
<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>
<!ELEMENT array (extension)*>
<!ATTLIST array embedded-element (true|false) #IMPLIED>
<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

B.4 Example

The following code snippet taken from package.jdo, contained in the com/corejdo/examples/model examples directory for the book, shows an example of an XML metadata file:

[View full width]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN"
➥ "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
  <package name="com.corejdo.examples.model">
    <class name="Address"/>

    <class name="Author">
      <field name="books">
        <collection element-type="Book"/>
      </field>
    </class>

    <class name="Book">
      <field name="authors">
        <collection element-type="Author"/>
      </field>
    </class>

    <class name="Publisher">
      <field name="books">
        <collection element-type="Book"/>
      </field>
    </class>
  </package>
</jdo>
```

Appendix C. JDOQL BNF Notation

The syntax for specifying syntax, i.e., the meta-syntax is done using a standard notion like Backus Normal Form (BNF). BNF is a widely used and widely accepted notation for specifying the syntax of programming languages such as Java and C++ and query languages such as SQL, JDOQL, and so on. In this appendix we list the BNF notation of JDOQL.

C.1 Grammar Notation

The grammar notation is taken from the Java Language Specification:

- Terminal symbols are shown in **bold** in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.
- Non-terminal symbols are shown in *italic* type. The definition of a non-terminal is introduced by the name of the non-terminal being defined, followed by a colon. One or more alternative right sides for the non-terminal then follow on succeeding lines.
- The suffix "opt," which may appear after a terminal or non-terminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right sides, one that omits the optional element and one that includes it.
- When the words "one of" follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition.

C.1.1 Parameter declaration

This section describes the syntax of the declareParameters argument.

```
DeclareParameters:

    Parameters ,opt
Parameters:
    Parameter
    Parameters , Parameter
Parameter:
    Type Identifier
```

C.1.2 Variable declaration

This section describes the syntax of the declareVariables argument.

```
DeclareVariables:
    Variables ;opt
Variables:
    Variable
    Variables ; Variable
Variable:
    Type Identifier
```

C.1.3 Import declaration

This section describes the syntax of the declareImports argument.

```
DeclareImports:
    ImportDeclarations ;opt
ImportDeclarations:
    ImportDeclaration
    ImportDeclarations ; ImportDeclaration
ImportDeclaration:
    import Name
    import Name.*
```

C.1.4 Ordering specification

This section describes the syntax of the setOrdering argument.

```
SetOrdering:
    OrderSpecifications ,opt
OrderSpecifications:
    OrderSpecification
    OrderSpecifications , OrderSpecification
OrderSpecification:
    Expression ascending
    Expression descending
```

C.1.5 Filter expression

This section describes the syntax of the setFilter argument.

Basically, the query filter expression is a Java boolean expression, where some Java expressions are not permitted. Specifically, pre- and post- increment and decrement (++ and - -), shift (>> and <<), and assignment expressions (+=, -=, and so on) are not permitted.

The description is bottom-up; in other words, the last rule Expression is the root of the filter expression syntax.

Please note that the grammar allows arbitrary method calls (MethodInvocation), where JDO permits calls only to the methods contains(), isEmpty(), and a number of String methods. This restriction cannot be expressed in terms of the syntax and must be ensured by a semantic check.

```
Primary:
    Literal
    this
    ( Expression )
    FieldAccess
    MethodInvocation
ArgumentList:
    Expression
    ArgumentList , Expression
FieldAccess:
    Primary . Identifier
MethodInvocation:
    Name ( ArgumentListopt )
    Primary . Identifier ( ArgumentListopt )
PostfixExpression:
    Primary
    Name
UnaryExpression:
    + UnaryExpression
    - UnaryExpression
    UnaryExpressionNotPlusMinus
UnaryExpressionNotPlusMinus:
    PostfixExpression
    ~ UnaryExpression
    ! UnaryExpression
    CastExpression
CastExpression:
    ( Type ) UnaryExpression
MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
RelationalExpression:
    AdditiveExpression
    RelationalExpression < AdditiveExpression
    RelationalExpression > AdditiveExpression
    RelationalExpression <= AdditiveExpression
    RelationalExpression >= AdditiveExpression
```

```
EqualityExpression:
    RelationalExpression
    EqualityExpression == RelationalExpression
    EqualityExpression != RelationalExpression
AndExpression:
    EqualityExpression
    AndExpression & EqualityExpression
ExclusiveOrExpression:
    AndExpression
    ExclusiveOrExpression ^ AndExpression
InclusiveOrExpression:
    ExclusiveOrExpression
    InclusiveOrExpression | ExclusiveOrExpression
ConditionalAndExpression:
    InclusiveOrExpression
    ConditionalAndExpression && InclusiveOrExpression
ConditionalOrExpression:
    ConditionalAndExpression
    ConditionalOrExpression || ConditionalAndExpression
Expression:
    ConditionalOrExpression
```

C.1.6 Types

This section describes a type specification used in a parameter or variable declaration or in a cast expression.

```
Type
PrimitiveType
Name

PrimitiveType:
    NumericType
    boolean
NumericType:
    IntegralType
    FloatingPointType
IntegralType: one of
    byte short int long char
FloatingPointType: one of
    float double
```

C.1.7 Literals

A literal is the source code representation of a value of a primitive type, the String type, or the null type. Please refer to the Java Language Specification for the lexical structure of IntegerLiterals, FloatingPointLiterals, CharacterLiterals, and StringLiterals.

```
IntegerLiteral: ...
FloatingPointLiteral: ...
BooleanLiteral: one of
        true false
CharacterLiteral: ...
StringLiteral: ...
NullLiteral:
        null
Literal:
    IntegerLiteral
    FloatingPointLiteral
    BooleanLiteral
    CharacterLiteral
    StringLiteral
    NullLiteral
```

C.1.8 Names

A name is a possible qualified identifier. Please refer to the Java Language Specification for the lexical structure of an identifier.

```
Name:
    Identifier
    QualifiedName
QualifiedName:
    Name . Identifier
```

Appendix D. PersistenceManager Factory Quick Reference

This appendix provides a quick reference to the standard strings used by JDO to denote optional features and to specify PersistenceManagerFactory configuration properties. This appendix covers the following topics:

- Optional features
- JDOHelper properties

D.1 Optional Features

The JDO specification includes a number of optional features that a JDO implementation may or may not implement. The supportedOptions() method on PersistenceManagerFactory can be used to determine the set of optional features that a JDO implementation supports. It returns a collection of strings, one for each supported feature.

Table D-1 provides a list of these strings.

Table D-1. Optional Features

Optional Feature Name
javax.jdo.option.TransientTransactional
javax.jdo.option.NontransactionalRead
javax.jdo.option.NontransactionalWrite
javax.jdo.option.RetainValues
javax.jdo.option.Optimistic
javax.jdo.option.ApplicationIdentity
javax.jdo.option.DatastoreIdentity
javax.jdo.option.NonDurableIdentity
javax.jdo.option.ArrayList
javax.jdo.option.Hashtable
javax.jdo.option.HashMap
javax.jdo.option.LinkedList
javax.jdo.option.TreeMap
javax.jdo.option.TreeSet
javax.jdo.option.Vector
javax.jdo.option.Map
javax.jdo.option.List
javax.jdo.option.Array
javax.jdo.option.NullCollection
javax.jdo.option.ChangeApplicationIdentity
javax.jdo.query.JDOQL

D.2 JDOHelper Properties

A PersistenceManagerFactory instance can be created via the getPersistenceManagerFactory() method on JDOHelper. This method takes a set of properties as an argument, which are then used to configure the returned PersistenceManagerFactory instance.

Table D-2 provides a list of the standard properties that can be used.

Table D-2. JDOHelper Properties

Property Name	Value
javax.jdo.PersistenceManagerFactoryClass	Class name of JDO implementation's PersistenceManagerFactory class
javax.jdo.option.Optimistic	true or false
javax.jdo.option.RetainValues	true or false
javax.jdo.option.RestoreValues	true or false
javax.jdo.option.IgnoreCache	true or false
javax.jdo.option.NontransactionalRead	true or false
javax.jdo.option.NontransactionalWrite	true or false
javax.jdo.option.Multithreaded	true or false
javax.jdo.option.ConnectionDriverName	Undefined[*]
javax.jdo.option.ConnectionUserName	Undefined[*]
javax.jdo.option.ConnectionPassword	Undefined[*]
javax.jdo.option.ConnectionURL	Undefined[*]
javax.jdo.option.ConnectionFactoryName	Undefined[*]
javax.jdo.option.ConnectionFactory2Name	Undefined[*]

[*] The value of this property is specific to the JDO implementation being used.

The only required property is javax.jdo.PersistenceManagerFactory-Class; all others are optional. In addition to these standard properties, a JDO implementation may also support its own additional properties. Any properties not recognized by a JDO implementation are ignored.

Appendix E. JDO Implementations

This chapter lists currently available JDO implementations, both from commercial JDO vendors and from the Open Source community. A second section briefly lists established non-JDO-compliant Java-based Persistence Solutions for the sake of completeness. Some of those listed may adopt a JDO API over time.

JDO Implementations

Implementations of the Sun JDO API are listed here.

Commercial JDO vendors

Vendors are sorted alphabetically by company name. Summary is by book authors, not listing common features including JCA-based J2EE integration, ant tasks, and so on. Background is usually obtained from public Web sites. Version numbers and pricing are listed as available at the time of this writing.

Exadel JDO 3.0

URL: http://www.exadel.com/products_jdoproducts.htm

Summary: O/R-mapping based. Supports the following relational datastores: IBM DB2, Oracle, and Microsoft SQL Server. Standalone Exadel JDO-Studio IDE with visual object relational mapping. Distributed caching mechanism.

Background: Exadel was founded in 1998, and is headquartered in Concord, California, in the San Francisco Bay Area, with offices in New York and Illinois. Exadel employs more than 60 software and business professionals.

Licensing model: EXADEL JDO Standard Edition: $999; Advanced Edition: $1,499; Enterprise Edition: $2,999.

Hemisphere's JDO Genie 1.3.2

URL: http://www.hemtech.co.za

Summary: O/R-mapping based. Supports the following relational datastores: Oracle, Microsoft SQL Server, Pointbase, SAP DB 7.3, Informix 9, Sybase, Interbase 6, Postgres 7.3.1, Firebird 1.0.2, and MySQL 3.23.49 and 4.0.12. Supports remote access of PM in client VM to a Genie Server VM. Comes with standalone JDO Genie Workbench IDE.

Background: Hemisphere Technologies is a professional-services firm focused on the development and deployment of Java and related applications. Positioned as the Java experts in South Africa, they combine the best people, processes, and technologies to deliver single point-of-contact solutions tailored to meet the needs of their clients.

Licensing model: JDO Genie Professional Edition: $500 per developer; JDO Genie Enterprise Edition: $2,000 per developer. Only developer licensing; no runtime costs.

Libelis's LiDO 1.4.1

URL: http://www.libelis.com

Summary: O/R-mapping based. Supports the following relational datastores: Oracle, IBM DB2, Microsoft SQL Server, Sybase, PointBase, Cloudscape, Interbase, Informix, InstantDB, mySQL, and Hypersonic. Also supports Versant OODBMS and proprietary binary files for embedded applications. Supports legacy mapping of existing schema. Comes with standalone LiDO Project Manager IDE and has plug-ins for the Eclipse/WSAD and Together IDEs.

Background: LIBeLIS is a privately held French company founded in 2000. LIBeLIS has offices in Paris (France), plus direct operations in London (UK) and Munich (Germany). LIBeLIS is distributed in Canada and in the USA.

Licensing model: LiDO Professional Edition Development License: 2,000 EUR, runtime separate. LiDO Standard Edition Development License: 600 EUR, includes 2 runtime licenses. LiDO Community Edition is free but limited to non-commercial use or educational purposes, and is only for access to Open Source RDBMS.

ObjectDB

URL: http://www.objectdb.com

Summary: Object database (ODBMS) with JDO API. Has passed Sun's JDOTCK tests. Runs embedded or in client-server mode. Comes with JDO Explorer UI. Future plans include developing editions of ObjectDB for Microsoft .NET as well.

Background: Privately held.

Licensing model: Pricing for Professional Edition and Standard Edition (limited to embedded mode only) not available at time of this writing. A Free Edition (embedded mode only) for personal non-commercial use can be downloaded.

ObjectFrontier's FrontierSuite for JDO 3.0

URL: http://www.objectfrontier.com

Summary: O/R-mapping based. Supports the following relational datastores: Oracle, IBM DB2, Microsoft SQL Server, Sybase, PointBase, Cloudscape, Informix, and mySQL. Supports distributed caching and reverse engineering. Comes with its own Development Suite, and integrates with leading modeling tools such as Rational Rose, Rational XDE, Together, Paradigm Plus, System Architect, Poseidon, and leading IDEs such as JBuilder, Forte for Java, and WSAD.

Background: ObjectFrontier, an Atlanta-based provider of products and consulting services, with an R&D Lab in Chennai, India, delivers value to clients through the appropriate combination of product and IT solutions. They provide sophisticated and powerful persistence frameworks that are Java, J2EE, and JCA compliant.

Licensing model: FrontierSuite for JDO Standard Edition: $999; Professional Edition: $1,299; Enterprise Edition: $2,299. All prices are per developer, no run-time/CPU/server fees. Support separately.

Object Industries' JRelay

URL: http://www.objectindustries.com

Summary: O/R-mapping based. JRelay Workbench, JBuilder, and Eclipse plug-in.

Background: Object Industries is based in Munich, Germany.

Licensing model: JRelay Professional Edition: $999; Enterprise Edition: $1,999. All prices are per developer; no runtime licenses required. Support separately.

Object Matter's Visual BSF

URL: http://www.objectmatter.com

Summary: O/R-mapping based. Offers a source code license.

Background: Privately held, based in Florida.

Licensing model: Professional edition binary license: $895; Enterprise edition binary license: $1,295; source code license: $3,995.

Orient Technologies' Orient 2.0e

URL: http://www.orientechnologies.com

Summary: Object database, ODBMS. ODMG and JDO API.

Background: Privately held.

Licensing model: Orient ODBMS Just Edition commercial license costs $199 per (deployment/server) license. Volume discounts available. SDK (developer license) is free. Also free for non-commercial purpose, with a database size limit of 10 MB and maximum 3 concurrent users.

Poet's FastObjects j2 / e7 / t7

URL: http://www.fastobjects.com

Summary: A classical object database, ODBMS. Different engines available: j2 is embedded pure Java; e7 is for workstations and manages objects without additional database-server processing; t7 is server-optimized for multiple parallel access and speed and availability. e7 and t7 also come with C++ API; j2 is Java only with JDO and ODMG API. t7 also has O/R mapping.

Background: Poet Software GmbH was founded in 1993, and is a wholly owned subsidiary of Poet Holdings, Inc., which is publicly traded on the Frankfurt Stock Exchange. The Company is headquartered in Hamburg. In addition, Poet maintains branch offices and strategic partners in Munich (Germany), Walldorf (Germany), London, Paris, Tokyo, and San Mateo (California).

Licensing model: j2/e7/t7 prices available upon request only. FastObjects j1 Community Edition available for free for private, academic, and other non-commercial use.

TradeCity Cybersoft RexIP JDO

URL: http://www.rexip.com

Summary: O/R-mapping based. Supports Oracle, MySQL, Microsoft SQL Server, Sybase, SAP DB, Cloudscape, and InstantDB. Integrated with RexIP Application Server.

Background: Based in Hong Kong with phone number in USA.

Licensing model: $388, unclear whether for developer and/or runtime.

Signsoft's intelliBO 3.2

URL: http://www.signsoft.com

Summary: O/R-mapping based. Supports the following relational datastores: Oracle, IBM DB2, Microsoft SQL Server, Sybase, IBM Informix, SAPDB, InstantDB, PostgreSQL, Progress, MySQL, PervasiveSQL, and JDataStore. Also supports Versant OODBMS. Support of referential integrity with statement ordering. Comes with a distributed cache using JMS or TCP. Standalone Signsoft intelliBO IDE, Borland JBuilder, and Borland Together Control Center IDE integration.

Background: Signsoft offers software solutions and services for companies that need high-scalable and high-performance applications. More than 400 companies worldwide trust in products made by Signsoft. Signsoft is located in Dresden, Germany.

Licensing model: intelliBO 3 Professional Developer (executes the development tools; license may not be published to end users): 2,450 EUR. intelliBO 3 Professional Server (used by software that is installed on a server): for 10 clients 500 EUR; for unlimited clients 1,500 EUR. intelliBO 3 Professional Local (license is used for standalone programs that are installed on separated computers): 100 users, 1,000 EUR; 1,000 users, 5,000 EUR.

Solarmetric's Kodo JDO 2.4.3

URL: http://www.solarmetric.com

Summary: O/R-mapping based. Supports the following relational datastores: Pointbase 4.2, InstantDB 3.26, IBM Cloudscape 4.0.6, IBM DB2 7.2, Oracle 8.1 & 9.1, PostgreSQL 7.2.1, Microsoft SQL Server 8, Sybase jConnect 12.5, Hypersonic hsqldb 1.7, and MySQL 3.23. Includes reverse mapping and re-engineering tools. Distributed cache synchronization via JCache-compliant coherence solution from Tangosol (third-party) using TCP, UDP, or JMS. Comes with IDE integration for Borland JBuilder, Sun ONE Studio / NetBeans IDE, and Eclipse / WebSphere Studio.

Background: SolarMetric, a company spearheaded by MIT alumni, is a global company with corporate headquarters in Washington, DC. A strong team of Java™ technology developers and experienced business leaders founded Solar-metric in 2001. The core technology team has been together since 1997, working on enterprise Web applications and networking products.

Licensing model: Kodo standard edition: $600 per developer license; Kodo Enterprise edition: $3,000 per developer license; both editions, no runtime royalties. (Add-on modules: Standard Edition Performance Pack, an additional $600 per Standard Edition license purchased, Standard Edition Query Extensions, $350 per Standard Edition license purchased.)

Versant's Judo for enJin

URL: http://www.versant.com

Summary: A JDO API to a leading object database (ODBMS) with sophisticated distributed cache and locking features, and load balancing/fault tolerance, asynchronous replication, HA backup, and online schema evolution options. Versant Developer Suite (VDS) IDE or enJin Tool Integration with IBM WSAD and Borland JBuilder.

Background: In 1988, Versant's visionaries began building solutions based on a highly scalable and distributed object-oriented architecture and a patented caching algorithm that proved to be prescient. Versant's initial flagship product, the Versant Object Database Management System (ODBMS), was viewed by the industry as the one true enterprise-scalable object database. The company is headquartered in Freemont, CA. In addition, Versant maintains European headquarters in the UK and offices in Munich, Germany, and Pune, India.

Licensing model: Prices available upon request only.

Open-source JDO projects

Projects are sorted by order of perceived maturity of JDO implementation and activity of the respective communities at the time of this writing, according to publicly available information. None of the open-source implementations has passed the JDO Technology Compatibility Kit (TCK) at the time of this writing.

Sun's JDO Reference Implementation (RI)

URL: http://java.sun.com/products/jdo

Summary: Not an open-source project, but available free of charge from Sun.

TriActive JDO 2.0 (TJDO)

URL: http://tjdo.sourceforge.net

http://sourceforge.net/projects/tjdo

Summary: Although the project is formally in beta stage, TJDO is already running successfully in a number of commercial JDO-based installations. Wraps byte-code enhancer from Sun RI. The current version has been tested successfully using Cloudscape, DB2, Firebird, MySQL, Oracle 8i, PostgreSQL, SAP DB, and MS SQL Server.

Jakarta OJB 1.0 / 2.0

URL: http://jakarta.apache.org/ojb

Summary: O/R-mapping. Feature-rich and actively maintained. JDO-compliant in combination with Sun RI. Fully independent JDO implementation being worked on for 2.0.

JBossDO

URL: http://www.jboss.org

Summary: While not much detail is publicly available at the time of this writing about a JDO implementation from JBoss, this is definitely more than a rumor now. The author has confirmed this news with Marc Fleury from the JBoss Group. A Web search by the time you read this may have more already. Here is what is known: JBossDO will be built from scratch based on the (hot!) new JBoss AOP framework in upcoming JBoss 4, not on Castor JDO. This will likely use an on-the-fly class enhancer based on the AOP framework, other implementations such as Poet's FastObjects also offer this as a development option. It will be interesting to see if this JBoss AOP-based implementation can also work with standard pre-enhanced classes. (The JBoss application server can of course be used with other existing JDO implementations, in fact many of the commercial ones listed above explicitly support JBoss as application server, but that is a different story to this one.)

XORM Beta 4

URL: http://xorm.sourceforge.net

http://sourceforge.net/projects/xorm

Summary: XORM is an O/R-mapper trying to be JDO-compliant where possible, but it is not a full JDO implementation. It implements many of the interfaces specified by JDO, but in contrast to most JDO implementations, XORM does not require you to run a class-file enhancer before deploying your persistence-capable classes. Instead, XORM allows you to specify persistence-capable classes using abstract classes or interfaces; bean-style get()/set() methods are enhanced at runtime to be managed for persistence, and requires a XORM.newInstance() instead of new SomeClass(). Also, some JDO API methods as well as some JDOQL features are not (yet) implemented.

Non-JDO-Compliant Java Persistence Solutions

This section is a listing of non-JDO-compliant Java-based Persistence Solutions. Some of those listed may adopt the JDO API over time.

Commercial vendors

TopLink

URL: http://otn.oracle.com/products/ias/toplink/content.html

Summary: Proprietary O/R mapping with many features. Long established and well recognized, originally with Smalltalk background, product has changed ownership from initial company to Webgain and lately to Oracle.

CocoBase

URL: http://www.cocobase.com

Summary: Proprietary O/R mapping with many features.

SoftwareTree JDX

URL: http://www.softwaretree.com

Summary: Proprietary O/R mapping from small vendor.

Fresher Matisse

URL: http://www.fresher.com

Summary: Matisse is a "hybrid database" that offers both native object storage as ODBMS and support for SQL 2. Comes with a unique Versioning Engine. The Matisse DBMS natively stores XML and program objects from Java, C#, C++, VB, Delphi, Eiffel, Smalltalk, and the leading scripting languages including Perl, Python, and PHP.

InterSystems Caché

URL: http://www.intersystems.com

Summary: Caché is a "post-relational database" built on a "Multidimensional Engine." It is an ODBMS also accessible via SQL, as well as a complete application server with (also) a Java API.

Progress Software ObjectStore

URL: http://www.objectstore.net

Summary: ODBMS formerly known as Excelon Javlin, formerly by ODI. Pre-JDO-compliant interface. JDO compliance planned.

Objectivity/DB 7.1

URL: http://www.objectivity.com

Summary: Objectivity/DB is another classical ODBMS.

db4o 2.6

URL: http://www.db4o.com

Summary: Light-weight ODBMS particularly targeting the embedded market.

Persistence EdgeXtend

URL: http://www.persistence.com

Summary: Persistence uses patented O/R mapping, caching, and synchronization technologies to reduce inefficiencies in query-intensive online sites. It offers built-in connectivity to modeling, messaging, and content formatting products in addition to its own component and object development tools.

Gemstone Gemfire

URL: http://www.gemstone.com

Summary: Used to be an ODBMS formerly integrated in proprietary application server. Today, it is a "live data distribution and management software that optimizes data in motion across the hardware and software boundaries of your enterprise."

Open-source persistence-related projects

Hibernate

URL: http://hibernate.bluemars.net

http://sourceforge.net/projects/hibernate

Summary: Great open-source O/R-mapping framework. Supports Oracle, DB2, MySQL, PostgreSQL, Sybase, SAP DB, HypersonicSQL, Microsoft SQL Server, Informix, FrontBase, Ingres, Progress, Mckoi SQL, Pointbase, and Interbase. Very mature and stable. It uses runtime reflection only and no code generation. ODMG 3 interface, but unfortunately no JDO implementation…yet.

Castor JDO

URL: http://castor.exolab.org

Summary: Despite the misleading name, it is not an implementation of the Sun JDO API. Has been around for some time. Robust implementation. Includes XML and LDAP/JNDI mapping as well.

JORM

URL: http://www.objectweb.org/jorm

http://debian-sf.objectweb.org/projects/jorm/

Summary: JORM (Java Object Repository Mapping) is an adaptable persistence service. It could be used to offer various personalities, such as one compliant with the JDO specification.

Cayenne 1.0b2s

URL: http://objectstyle.org/cayenne

http://sourceforge.net/projects/cayenne

Summary: Cayenne is a powerful, full-featured OpenSource O/R framework. Cayenne is written in Java and is intended for Java developers working with relational databases. Cayenne has been successfully deployed in production environments on high-volume sites. Cayenne has a growing international user and developer community.

Ozone 1.1

URL: http://www.ozone-db.org

http://sourceforge.net/projects/ozone/

Summary: ODBMS with ODMG 3.0-like interface.