Java™ Data Access

JDBC™, JNDI, and JAXP

Todd M. Thomas

Features coverage of JDBC 3.0

Updates and sample code
at the Companion Web Site

Visit us at mandtbooks.com

PROFESSIONAL MINDWARE™

Java Data Access—JDBC, JNDI, and JAXP

Todd M. Thomas

Published by
M&T Books
An imprint of Hungry Minds, Inc.
909 Third Avenue
New York, NY 10022
http://www.hungryminds.com/

Library of Congress Control Number: 2001092891

ISBN: 0-7645-4864-8

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

1O/RR/QR/QS/IN

Distributed in the United States by Hungry Minds, Inc.

Distributed by CDG Books Canada Inc. for Canada; by Transworld Publishers Limited in the United Kingdom; by IDG Norge Books for Norway; by IDG Sweden Books for Sweden; by IDG Books Australia Publishing Corporation Pty. Ltd. for Australia and New Zealand; by TransQuest Publishers Pte Ltd. for Singapore, Malaysia, Thailand, Indonesia, and Hong Kong; by Gotop Information Inc. for Taiwan; by ICG Muse, Inc. for Japan; by Intersoft for South Africa; by Eyrolles for France; by International Thomson Publishing for Germany, Austria, and Switzerland; by Distribuidora Cuspide for Argentina; by LR International for Brazil; by Galileo Libros for Chile; by Ediciones ZETA S.C.R. Ltda. for Peru; by WS Computer Publishing Corporation, Inc., for the Philippines; by Contemporanea de Ediciones for Venezuela; by Express Computer Distributors for the Caribbean and West Indies; by Micronesia Media Distributor, Inc. for Micronesia; by Chips Computadoras S.A. de C.V. for Mexico; by Editorial Norma de Panama S.A. for Panama; by American Bookshops for Finland.

For general information on Hungry Minds' products and services please contact our Customer Care department within the U.S. at 800-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

For sales inquiries and reseller information, including discounts, premium and bulk quantity sales, and foreign-language translations, please contact our Customer Care department at 800-434-3422, fax 317-572-4002 or write to Hungry Minds, Inc., Attn: Customer Care Department, 10475 Crosspoint Boulevard, Indianapolis, IN 46256.

For information on licensing foreign or domestic rights, please contact our Sub-Rights Customer Care department at 212-884-5000.

For information on using Hungry Minds' products and services in the classroom or for ordering

examination copies, please contact our Educational Sales department at 800-434-2086 or fax 317-572-4005.

For press review copies, author interviews, or other publicity information, please contact our Public Relations department at 317-572-3168 or fax 317-572-4168.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 978-750-4470.

Credits

Acquisitions Editor
Grace Buechlein

Project Editor
Michael Koch

Technical Editor
Terry Smith

Copy Editor
S. B. Kleinman

Editorial Manager
Mary Beth Wakefield

Senior Vice President, Technical Publishing
Richard Swadley

Vice President and Publisher
Joseph B. Wikert

Project Coordinator
Nancee Reeves

Graphics and Production Specialists
Sean Decker Melanie DesJardins
Laurie Petrone
Jill Piscitelli
Betty Schulte
Jeremey Unger

Quality Control Technicians
Laura Albert
David Faust
John Greenough
Andy Hollandbeck
Angel Perez

Proofreading and Indexing
TECHBOOKS Production Services

Cover Image
© Noma/Images.com

About the Author

Todd M. Thomas is an Oracle Certified DBA and Java developer with more than six years' experience in the IT industry. He currently works as an independent consultant, software developer, and trainer focusing on Java database and network programming. His work has ranged from building and managing data warehouses to architecting enterprise Java server applications. His most recent assignment was helping develop the network and database interface layers for BlueMoon, Airtuit Inc.'s wireless application gateway. His education includes a B.S. in Industrial Engineering and an M.S. in Engineering Science from the University of Tennessee in Knoxville.

About the Contributors

Johennie Helton has been an architect, developer, and software consultant on numerous *n*-tier–distributed systems and products. She has worked with databases and database design and implementation since 1990. Her database design projects include an application to make available automobile inventory online, a material management application for the health-care industry, and an application for customer coupon redemption for the grocery and coupon industries. During her career, her focus has been on creating applications with leading-edge technology, including application modeling, database design and implementation, and using J2EE and Java technologies to provide enterprise solutions to customers. She has a strong background in object-oriented analysis and design as well as in hypermedia systems. She has an M.S. in Computer Science from the University of Colorado.

Steve Nobert got his first taste of programming in 1983, when he took a mathematics class in his senior year in high school. The class entailed developing computer programs on an Apple IIe to solve math problems. He hasn't looked back since. As much as Steve loathed school, he still went on to receive his Associate of Science degree from Northern Virginia Community College in 1988, his Bachelor of Science degree from James Madison University in 1990, and his Master of Science

degree from George Mason University in 1996, all of them in Computer Science. He has more than twelve years of professional programming experience involving myriad heterogeneous computing languages, operating systems, and state-of-the-art technologies. Steve's primary career focus has been developing GUI applications on various flavors of UNIX. In 1996, Steve recommended that an unmaintainable and internationalized cross-platform application be rewritten using Java technology. He taught himself Java to develop the application/applet, and Java technology instantly became his primary career focus and has been so ever since. He has been involved in database technology since 1990. Steve has lived most of his life overseas and then in the northern Virginia area for almost fifteen years. Steve and his wife, Misti, reside in Knoxville, Tennessee.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Part I: Getting Started with Java Data Access

Chapter List

Chapter 1: Introducing Java Data Access Development

In This Chapter

- Understanding how enterprises use and store data
- Using the Java technologies for accessing the different enterprise data stores

Now more than ever, Java developers need to understand how to create data− centric applications. Data is an important commodity and organizations now try to capture, store, and analyze all the information they generate. As a result, many different forms of data exist and an equal number of different methods exist to store it. As a Java developer, you will likely face the challenge of writing an application that enables an organization to effectively use its data stored in either a single source or multiple sources.

Your chances of having to build an application that accesses enterprise data increase because Java continues to gain market share as the language of choice for creating server applications and the J2EE platform become increasingly popular. In addition, most server applications require access to data stores for information. As an example, an EJB component may need to update inventory levels in a database or send XML messages to other applications. As a result, your knowing how to access the different data stores is paramount in enterprise development.

However, client applications also need access to enterprise data stores. For example, a human−resources application that tracks employee vacation time must retrieve and store information from a database. In addition, you now have mobile clients that need access to enterprise data stores. Writing data−centric applications for these devices is challenging, as they operate with little memory, minimal processor speeds, limited power supplies, and intermittent network access.

Fortunately, Java provides a robust set of data−access technologies that enables you to access the most common types of enterprise data. Using these same technologies you can create both server−side components and client−side applications. The technologies consist of APIs for accessing databases, naming and directory services, and XML documents.

This chapter introduces the most common types of data enterprises used in their operations, from simple text files to complex specialty databases. This chapter also covers the various Java−based technologies that you can use to access the data stores.

Taking Stock of Enterprise Data Stores

As you know, enterprises rely on data to make business decisions, generate revenue, and run daily operations. For example, managers generate sales forecasts based on historical sales data stored in a data warehouse. Companies also build online stores using live inventory levels that sell directly to their customers. Accounting departments use financial database applications to generate payroll checks and track accounts receivables. These are only a few examples of how enterprises use data.

As you also know, data can take many forms. Figure 1−1 illustrates some of the more common kinds of data

an enterprise uses and how it stores them. It also shows how clients access the information residing in the data stores.

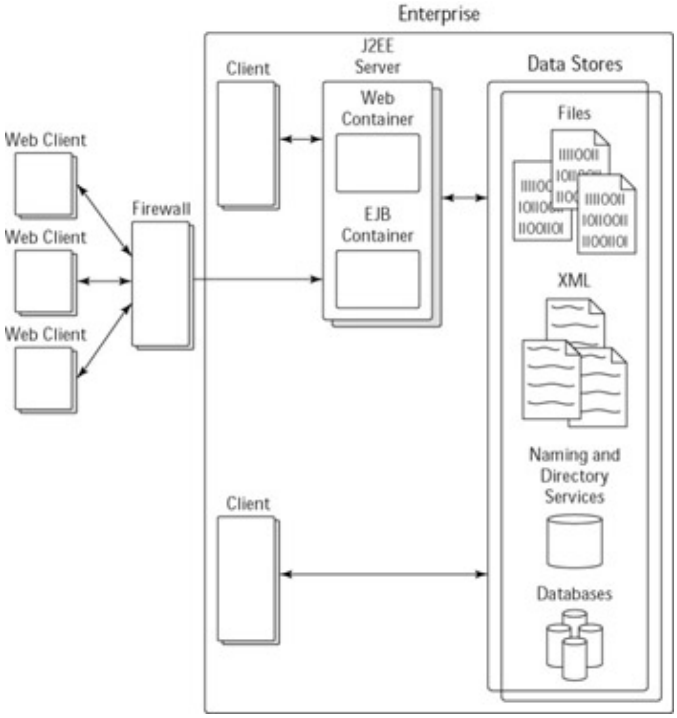

Figure 1−1: This figure shows an example of the more common kinds of data an enterprise uses and how it stores them.

For example, data most commonly takes the form of files stored in file systems on central servers or individual workstations. There are as many different forms of data files as there are applications. However, some categories include word− processing documents, spreadsheets, multimedia (graphic, sound, and video), and XML documents.

Most companies also use databases to store information and model business processes. Databases enable corporations to store, organize, and retrieve large amounts of data. Some organizations use them for data warehouses containing hundreds of gigabytes of information. Others may use databases to support high−volume transactional applications such as an airline−reservation system. Databases also offer a lot of flexibility in terms of how you interact with them. Almost all have proprietary data−access tools as well as mainstream APIs such as JDBC drivers for you to use.

Other forms of data exist as name−value pairs stored in a naming or directory service. These services store data in a hierarchical database system optimized for lookups. In addition, some organizations may use a directory service as an object repository. Distributed applications use the directory service to locate and download objects. This minimizes the problems associated with distributing updated code because applications always have access to the latest version.

When an organization uses different forms of data, it requires you, as a developer, to use different access methods as well. For example, most file access occurs across a LAN and so the network software and operating system handles the communication details. However, retrieving data from a database or directory service requires additional components. You will likely need special drivers or APIs. In addition, some organizations let clients access their data stores over the Internet. You must consider security issues as well as client−activity levels if you decide to do this.

As a developer, your job is to create applications that enable people, or processes, to interact with any form of data that contains the information they need. Therefore, you should understand the many different forms and how enterprises typically store them. In addition, you need to consider how clients access the information as it affects your application as well.

The following sections describe the most common data stores enterprises use to house their information.

Database systems

Next to file systems, enterprises use databases to store most of their information. This enables centralized information storage, meaning that both clients and server have one data source. That is, everyone — onsite staff, field employees, and Web clients — looks at the same data. Centralizing data storage also enables administrators to perform maintenance routines such as data updates and backups more frequently and reliably.

Today's databases can store more than just simple character or numerical data. The Internet has pushed database vendors to support more varied forms of data. For example, most database systems now enable you to store multimedia data such as sound and video. In addition, support for persisting native programming objects, such as those used by Java, also exists. Vendors developed this support because of the difficulty of combining object−oriented programming models with standard referential database systems.

There are many types of databases, including hierarchical, relational, object, and object−relational. Each has its strengths and weakness. However, by far the most popular type of database is the relational database. It is used by almost all enterprises employing database solutions.

The relational database gained popularity by providing the following benefits:

- **Data integrity** — Relational databases incorporate integrity rules to help protect against data corruption, duplication, and loss. You can use the built−in integrity rules or define your own.
- **Common access language** — SQL provides a universal access language for relational databases. The language enables you to build database structures, model business processes, and to add, delete, modify, and retrieve data. The core SQL language works with most relational database systems.

XRef For more details on relational databases and how to interact with them, refer to Chapter 2, "A Relational Database Primer."

Because of their popularity, you should familiarize yourself with relational− database theory, SQL, and access techniques. Chances are that you will need them at some point as a developer.

Different Database Types

Many different databases are available to meet an organization's data−storage needs. For example, some companies may need to persist Java objects. Others may want to model business processes or create databases optimized for retrieving data.

The following list describes the different database types available:

- *Relational database* — Stores all data in tables, among which you can define relationships in order to model most real−world processes. By default, relational databases have entity (table) and referential (relationship) constraints to protect against data loss or corruption. Relational databases are the most

widely used database system.

- *Hierarchical database* — Stores data in records. Only parent−child relationships can exist between records. This creates a hierarchy wherein each record can participate in only one parent−child relationship, which makes it hard to model complex processes. Hierarchical databases provide fast data retrieval, but slow write operations. Directory services often use hierarchical databases.
- *Network database* — Similar to hierarchical databases except that they enable you to model more complex relationships. Network databases support many−to−many relationships among records.
- *Object database* — Supports storage of native programming objects and custom data types. Many object databases support object−oriented programming concepts such as inheritance, polymorphism, and encapsulation of the user−defined data types. Some support SQL while others have proprietary access languages.
- *Object−relational database* — A cross between an object database and a relational database. Most often, object−relational databases are relational databases that treat objects as new data types.

Naming and directory services

Naming and directory services are hierarchical (not relational) databases optimized for read (not write) operations. Therefore, you should not use them where significant insert, update, or delete activities occur.

Naming services store objects using a simple name−value format. A common example is a file system whose objects are files. As a naming service, the file system associates a name, the filename, with a value, the file handle. A user requests a file by its name and the operating system retrieves it by the associated file handle. An RMI Registry provides another example of a naming service. In this case, the name is the object identifier, and the value is the object itself.

A directory service extends the capabilities of a naming service by allowing you to attach attributes to objects. An example of a directory−service application is an employee directory stored in an LDAP−enabled directory service. In this example, an employee is an object and can have attributes in addition to his or her name. For example, you may attach attributes such as department, e−mail address, and phone number to each employee. In addition, you can search a directory service for objects based on attribute values.

XRef Chapter 13, “Accessing Enterprise Data with JNDI,” provides more detail on naming and directory services as well as how to interact with them using the Java Naming and Directory Interface (JNDI) API.

The Lightweight Directory Access Protocol (LDAP) is often associated with naming and directory services. Contrary to popular belief, LDAP does not define a data−storage model or schema. Instead, it defines a communication protocol for interacting with directory services. Vendors use LDAP for communications and store data however they wish.

However, unlike with relational databases, with naming and directory services you cannot easily model processes or protect data using integrity constraints. Naming and directory services also lack a common data−access language like SQL and you usually rely on a vendor’s API for access. Fortunately, Java’s JNDI API addresses this lack of a standard access method by providing a common interface to many different naming and directory services.

Nonetheless, naming and directory services provide you with a powerful tool for retrieving data. In addition, they are useful when you do not need the overhead of hardware and DBAs to run a relational database.

XML services

The eXtensible Markup Language (XML) enables you to create self−documenting data. Enterprises now use XML as the standard for exchanging data and messages with other organizations or among applications. In addition, organizations use it in conjunction with XSLT to develop a single source of online content viewable from a variety of devices. As a result, most enterprise applications use some form of XML−service.

An XML−service is an application, whether EJB components or specific application classes that consume or generate XML. These services are quickly becoming a major component of distributed architectures and applications. Some examples of XML−services include:

- Processing configuration files such as EJB deployment descriptors
- Transforming data from one format to another
- Exchanging messages with other applications using JMS

Java provides significant support for XML. In fact, both technologies appeared in the mid−1990s and have grown together. During this time, many developers created numerous free Java tools for working with XML documents. Now the Java JDK and JRE distributions include many of these same tools, such as the SAX parser.

XML provides many benefits that have boosted its adoption rate. The following is a partial list of its advantages:

- **XML is an open−standard** — The World Wide Web consortium controls the XML specification, and therefore no one industry or company can control its direction.
- **XML is text−based** — XML documents are text files. As a result, you can read and edit them using text editors.
- **XML is self−describing** — An XML document can contain information about itself, meaning that it is self−contained. Other applications can use the document without any extra information.
- **XML has free tools and processors** — A multitude of Java tools exist to help you create, manipulate, read, and exchange XML documents.

Along with relational−database knowledge, a solid understanding of Java−XML technologies will help you significantly as you work with enterprise data using Java. Mastering both technologies definitely won’t hurt your career either.

Introducing Java Data Access Technologies

As I previously mentioned, the Java platform plays a dominant role in server−side application programming, as demonstrated by the recent increase in J2EE applications. Java succeeds because it has the right combination of tools and technologies that enable developers to create robust and scalable server applications. Corporations now use Java technologies for tasks such as providing the presentation layer for Web sites, the business logic on application servers, as well as creating custom client−server solutions.

Part of being a successful server−side technology is the ability to interact with data. Fortunately Java has this ability. It provides numerous APIs that help you access the different types of data stores. Figure 1−2 shows the role Java Data Access Technologies (JDATs) play in the enterprise environment.

From the figure you can see that JDAT includes JDBC, JNDI, JDO, and the XML APIs. You use the JDBC API to access data stored in a SQL database. The JNDI API gives you access to naming and directory services such as iPlanet's Directory Server or Novell's NDS. JNDI also supports LDAP so you can communicate with LDAP−enabled directory services. The JDO API provides a tool you can use to persist Java objects into a data store such as a relational database. This API does not compete with JDBC but rather complements it. Lastly, the XML APIs enable you to perform many different tasks with XML. For example, you can employ the APIs to use XML for inter−application messaging, making remote procedure calls, as well as parsing and transforming XML documents using SAX, DOM, and XSLT.

Notice in Figure 1−1 that any application can use the JDAT. Java Server Pages (JSP) technology, servlets, Enterprise Java Beans (EJBs), and stand−alone clients can take advantage of the APIs. Therefore, you do not need to rely on J2EE applications to access enterprise data stores. An application written using the Java 2 Standard Edition (J2SE) has equal access. The JDBC API provides a good example. A considerable amount of JDBC development does not reside in a J2EE container. Most often developers create clients that use the API to access relational databases without involving J2EE.

The remaining sections provide more details on the different Java data−access technologies shown in Figure 1−2.

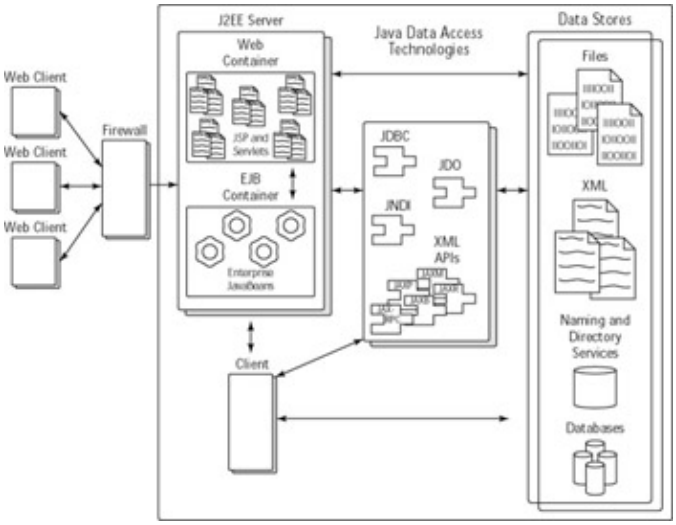

Figure 1−2: The role Java Data Access Technologies (JDATs) play in the enterprise environment.

JDBC 3.0

JDBC, the Java database−access technology, is probably the most widely known Java data−access API. Many books and Web sites exist to help educate you on its use. In addition, JDBC has wide industry support and you can find JDBC drivers for most databases on the market.

Note Here's a piece of trivia. Contrary to popular belief, JDBC is not an acronym. It is a trademarked name by Sun Microsystems that represents a type of technology. It does not stand for "Java Database Connectivity" as most people think.

When Sun Microsystems released the original JDBC API 1.0 in 1997, the API had very limited features. It provided only a basic call−level interface to SQL databases. JDBC 2.0 touted more practical features such as scrollable cursors and batch updates. Along with the JDBC 2.0 release came the 2.0 Optional Package, which added a new package, javax.sql, and provided advanced features such as connection pooling, distributed transactions, and the RowSet interface.

Now at release 3.0, JDBC adds a few missing features to the API, such as transactional savepoints and more support for SQL99 data types. In addition, the core package, java.sql, and the optional API, javax.sql, are included with the Java 1.4 distribution. No longer must you separately download the javax.sql package to gain access to its features.

Where to Get the JDBC API

Both the 1.4 JDK and JRE distributions include the JDBC 3.0 API. You can download both of these software packages at the JDBC homepage: www.javasoft.com/products/jdbc. However, the API only includes the JDBC−ODBC bridge driver, which enables you to use an ODBC driver to access a database. As a result, to use the bridge you must have an ODBC driver compatible with your database.

The API does not ship with drivers for other databases. You should contact your database vendor to determine if they have a JDBC driver available. If they do not, most likely a third party provides an implementation. You can find a list of driver vendors on the JDBC homepage.

In short, JDBC provides database developers with a powerful and flexible toolbox. It enables you to write database applications using one database interface. The methods defined in JDBC operate independently of the underlying database. For example, you use the same programming techniques to do any of the following tasks whether you are using DB2, Oracle, Informix, SQLServer, mSQL, or any other database:

- Open a connection
- Call stored procedures
- Add, modify, or delete data using SQL DML statements
- Execute DDL statements
- Handle database errors

As you can see, having one programmatic interface provides many benefits. If each vendor ignored JDBC and built its own API, dozens of different database programming paradigms would exist. Working with multiple databases would then require learning multiple interfaces. Fortunately, industry's widespread adoption of JDBC has helped make it a standard.

JNDI

The JNDI API 1.2 gives you a tool for accessing a variety of naming and directory services using Java. The JNDI API, like JDBC, also provides a single, consistent programming interface for access naming and directory services. Most enterprises use a directory service of some type. They may use one included with an operating system, such as Microsoft's Active Directory Services, or a commercial product like iPlanet's Directory Server.

JNDI plays an important role in the J2EE platform, as it enables you to locate and use objects stored in naming and directory services. For example, a J2EE deployment may use a naming or directory service as a repository to store objects like EJBs or JDBC DataSource objects. JNDI gives you the tools to create applications that can manage, retrieve, and use those objects from the services.

The JNDI API architecture consists of two components. The first, or core, API, provides the methods and properties you work with when writing client applications. You use this same API regardless of your target

naming or directory service. Vendors use the second component, the Service Provider Interface (SPI), to develop drivers, creatively named service providers, for accessing naming and directory services. Just like a JDBC driver, the service provider abstracts the communication details used to interact with a naming or directory service. A service provider also plugs into the core API, which enables you to easily add support for different naming and directory services.

The JNDI 1.2 API ships with Java 1.3.1 or greater and includes the following service providers:

- LDAP
- Domain Name Service (DNS)
- CORBA's Common Object Services (COS naming)
- RMI Registry

In addition, from Sun's JNDI Web site (www.javasoft.com/jndi) you can also download other service providers created by Sun and third parties for the following naming and directory services:

- Sun's Network Information Service (NIS)
- Novell's Network Directory Service (NDS)
- File system
- Directory Services Markup Language (DSML)

The SPI architecture makes it easy for naming and directory vendors to create service providers. Because of the increased role these data stores are beginning to play in enterprises, you will probably see more service providers appearing — especially ones focused on high−performance LDAP access.

JDO

Java Data Objects (JDO) is a relatively new technology intended to make it easier to persist Java objects. Yes, you can easily persist objects by serializing them to disk, but no easy method exists for storing them in transactional systems like databases.

As of this writing, JDO is in development and only a reference implementation, version 0.95, exists. Thus, I do not cover the details of JDO in the book. However, the available code enables you to transparently store and retrieve Java objects from a B−tree storage structure stored on a hard drive. Support for relational databases should appear soon.

JDO fills a void that has plagued Java developers for some time. Often a developer needs to persist the state of an object in order to be able to resume at the same point later or share the object's state with other components. Unfortunately, simple serialization does not enable you to use objects in transactions. Storing the objects in a database can alleviate the problem; however, doing so requires significant planning, as you must map an object's fields to database fields. It also creates a maintenance challenge because object changes require database−schema changes as well.

JDO mitigates these problems. It provides an API with consistent interfaces that enable you to persist Java objects in different data stores. It also provides mechanisms for using stored objects in transactions as well as for searching for objects that meet certain criteria.

To obtain the latest information and JDO distribution, visit the JDO homepage at access1.sun.com/jdo.

Java XML APIs

Java has strong support for XML. In fact, both have quickly positioned themselves as indispensable players in server−side development and messaging applications. To support XML, Java has a suite of APIs that enable you to create, transform, and exchange XML documents. In addition, an API also exists that enables you to create Java classes representing an XML document based on its schema. The resulting classes handle parsing and transforming the XML document without you needing to worry about using the SAX parser and DOM trees.

The Java XML APIs also provides the foundation for the Web−services programming model. Using the APIs you can create distributed components that use XML to exchange information or look up application and business services.

As I mentioned earlier, Java provides several XML−centric APIs. The following list provides a partial list of what's available:

- **Java API for XML Parsing (JAXP)** provides support for parsing and transforming XML documents. It supports SAX and DOM parsing as well as XSLT.
- **Java API for XML−Based Remote Procedure Calls (JAX−RPC)** enables you to use XML to make remote procedure calls across a network.
- **Java API for XML Messaging (JAXM)** gives an application the ability to send and receive XML−based messages. Based on Simple Object Access Protocol (SOAP) 1.1 with Attachments.
- **Java Architecture for XML Binding (JAXB)** provides a mechanism for creating Java classes based on XML schemas. Objects instantiated from the classes parse XML documents and enforce schema constraints.
- **Java API for XML Registries and Repositories (JAXR)** is a tool that enables you to search for registered business services listed in well−known XML business registries.

At the time of this writing most of the previous APIs were available as reference implementations or were under specification development. However JAXP, the parsing and transformation engine, is available. At version 1.1, it provides a fast, reliable, and flexible tool for working with XML documents. Visit www.javasoft.com/xml for more information.

Summary

This chapter presented an overview of the different types of data an enterprise may store, the structures they use to store it, and the Java technologies that can access it. In a nutshell, you will find an enterprise storing data in:

- Operating system files such as word−processing documents and spreadsheets
- Databases, including hierarchical and relational databases
- Naming and directory services such as RMI Registries and LDAP−enabled directory services
- XML documents containing self−describing data from databases or used in inter−application communication

To help you access the various types of data storage containers in the previous list, Java provides the following data−access technologies:

- JDBC 3.0, which provides support for accessing relational databases
- JNDI 1.2, which provides support for interacting with naming and directory services
- JDO, which enables you to easily persist Java objects in a variety of data stores
- XML APIs, which consist of JAXP, JAX−RPC, JAXM, JAXB, and JAXR, and that enable you to parse, send, and transform XML document

Chapter 2: A Relational Database Primer

In This Chapter

- Understanding relational database systems
- Leveraging the features and benefits of relational databases
- Understanding SQL concepts
- Using DML and DDL SQL statements

It is often said that data is a company's real asset. The data can include marketing research, sales history, customer lists, online content, or many other types of information, all of which companies use to generate revenue. In most scenarios, enterprises use database systems to store the information and provide access to it. As a result, the database has become an important component in a corporation's IT infrastructure.

Different types of database management systems (DBMSs) exist to help enterprises store data. However, enterprises most often use the relational DBMS (RDBMS), which has characteristics that provide organizations with everything they need to meet their data storage needs. An RDBMS handles large quantities of data, allows a high volume of read/write activity, provides default integrity constraints to protect data, and gives you flexibility in modeling business processes and entities.

Enterprises use RDBMSs for a variety of purposes. Some use them to store hundreds of gigabytes, or even terabytes, of data. Others use them for CRM solutions such as field−force automation and field−sales support. They are also used for high−volume transaction processing such as handling call−center activities. Because of the pervasiveness of RDBMSs, you will likely interact with one at some point in your career.

As with most technologies, if you do not apply the theory and concepts of relational databases frequently you may forget the basics. This chapter provides a refresher on the fundamentals of relational database systems. However, the chapter does not provide information on designing schemas for RDBMSs.

I start by covering what comprises an RDBMS, its architecture and components. Next I provide a quick synopsis of the Structured Query Language (SQL), the non−procedural language used to communicate with an RDBMS. At the end of the chapter you should have a renewed understanding of relational database systems that you can use to develop JDBC applications.

The RDBMS in a Nutshell

In 1970, E. F. Codd developed the relational data model from which the RDBMS was born. The concept centered around tables, called relations or entities, to store data. Codd called the model "relational" after the name he used for tables, not the relationships you can build among them.

From his model Codd created 12 rules summarizing the features of a relational database. One rule specifies that a system must have integrity constraints to protect data. The constraints apply both to tables and table relationships. He also created a rule stating that the relational database system should have a single language that supports all the data−access and system−management needs. This rule provided the impetus for creating the universal database language called SQL.

Codd's relational model was revolutionary at the time and stood in stark contrast to existing database systems. Soon after he presented his model companies began building database systems around it. Vendors also added features, such as indexing logic and custom procedural languages to enhance their systems. As a result, enterprises quickly adopted the RDBMS and it is now the default database system.

The benefits an RDBMS provides over the other database−systems helped increase its adoption rate. The following list summarizes some of the model's advantages:

- **Data integrity** — The relational model defines integrity rules that help guard against data corruption. That is, the data you place into an RDBMS do not change, disappear, or become corrupt.
- **Flexibility** — In some respects an RDBMS server acts as an application−development platform. It provides an environment in which you can create data−centric applications. By definition, an RDBMS enables you to create entities and build relationships among them. More advanced RDBMSs incorporate a procedural language enabling you to build store procedures. Using these languages, in conjunction with entity relationships, enables you to model business processes and company workflow and store the resulting data in the database.
- **Universal data access** — SQL has evolved as the default language for interacting with an RDBMS. Although some RDBMSs extend SQL to take advantage of proprietary features, the core language still remains portable.

An enterprise RDBMS is a complex system. It must provide robust data storage, incorporate integrity rules, and include server capabilities to share data. Figure 2−1 shows a conceptual view of a relational database system in a client−server deployment. As you can see, many pieces must come together to form the system.

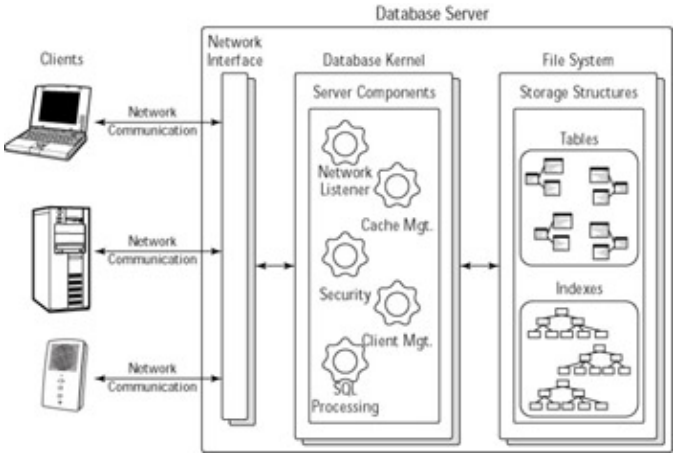

Figure 2−1: Conceptual view of an RDBMS

Most RDBMSs use a client−server architecture, an architecture wherein clients communicate with the server over a network. Most often TCP/IP is the protocol for sending requests and receiving responses. The host server houses the database kernel, which consists of several internal−server processes that manage client connections, process SQL statements, and communicate with the host's file system to manage data.

Despite the number of different enterprise RDBMSs on the market, they all share two common components: data−storage structures and data−integrity rules. The following sections provide an overview of the major components.

Understanding data storage

The purpose of a database is to safely store data. How it organizes the data on the physical storage device depends on the vendor. Regardless, an RDBMS has two basic storage structures, tables and indexes. Tables, the center of an RDBMS, hold the data. Indexes, although not technically required, improve data access performance.

A database system typically stores the tables and index data in files on a local file system. Depending upon the database, it may use either multiple files or one large file to hold the data. Some database systems also enable you to create distributive systems and store data on other hosts.

Tables

Tables consist of rows and columns. Rows represent entries like patient medical records or customer invoices. Each table row is called a *record* and is unique in a purely relational model. Without uniqueness, you cannot ensure consistent access to the same row. In addition, non−unique data can create data−corruption problems when you start creating relationships among tables.

Columns, often called *attributes*, describe each record. A table can have one or more columns. In general, columns define the information you want to track. The data type of the columns varies between databases. However, SQL standards such as SQL−92 and SQL3 define data types that most databases support. Some example data types include CHAR, NUMBER, and DATE.

As Codd mentioned, an RDBMS is based on tables. In fact, all data, even the table definitions themselves, are stored in tables. To store them, the relational database relies on system tables to completely describe itself. Storing the database metadata in tables enables database administrators (DBAs) or other authorized clients to manage the system using the same language as other users.

Indexes

Indexes help SQL queries quickly locate records. The way indexes work depends upon the index type. In general, an index ranks, or sorts, records based on a column in the table. The indexed column is known as a *key*. Indexes can also use composite keys consisting of multiple columns.

Without indexes a search for a record usually requires a table scan, which starts at the first row and looks sequentially at each entry for a match. Table scans do not provide quick access to a particular record or group of records, especially when a table has thousands or millions of records.

For example, imagine a phone book in which names and numbers were listed in the order in which the subscribers initially obtained their phone service. In this scenario it is nearly impossible to locate a person’s phone number. You have to perform a table scan, starting with the first entry, and look at every entry until you find a match. Fortunately, a phone book indexes entries alphabetically to help you locate the person you’re looking for. Although RDBMS indexes organize the data differently, the end result is the same: You can locate the information quickly.

Indexing is a science unto itself and many different types exist. Some common types include B−tree, clustered, non−clustered, and function−based indexes. However, most databases employ B−tree, or balanced−tree, indexes as the default. This type provides very quick access to data with minimal storage requirements. The actual implementation details of a B−tree index vary between vendors.

Figure 2−2 shows a conceptual view of a B−tree index for the alphabet. It consists of nodes that contain keys pointing to the location of data. The top node is called the root, subsequent nodes are branches, and the lower nodes are leaves. A B−tree index minimizes the path from the root node to any leaf node in the tree. Using the index you can locate any letter in the alphabet in three steps or fewer.

The keys in a database B−tree index point to individual records. The figure shows only three levels in the index; real indexes often have more levels.

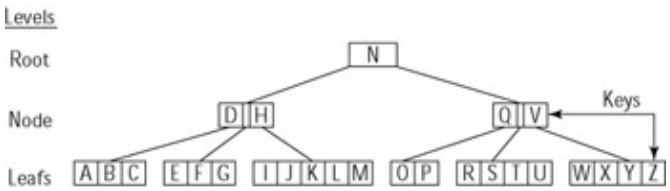

Figure 2−2: Conceptual view of a B−tree index

Note Proper indexing of a table can significantly speed up “read” operations such as SQL SELECT statements. However, too many indexes can slow down INSERT or UPDATE operations. In general, a database system updates indexes whenever the column data that they represent change. Therefore new records, and updates to existing records, can cause index updates as well. The more indexes, the more updates needed, and therefore the slower the response.

Ensuring data integrity

One of the biggest benefits an RDBMS provides is that it protects the integrity of the data. Once you place data into a relational database you can remain confident that it will not change. To help provide this protection, the relational model defines integrity rules.

Different types of data corruption can occur in a database. Some, like disk−drive corruption, an RDBMS cannot guard against because it occurs at the physical layer. A whole drive does not have to become damaged for this type of corruption to occur; one bad block can cause significant problems. A DBA usually handles this situation by restoring the database from the last good backup. Unfortunately, an RDBMS can only mitigate the effects of this type of corruption, not prevent it.

However, two types of corruption exist that the RDBMS can protect against. The first occurs when the data among related tables become unsynchronized, creating orphan records. For example, suppose a Medical_Claims table contains records for patients, which requires a link between the Medical_Claims table and the Patients table. Deleting records from the Patients table without removing the corresponding records from the Medical_Claims table will create orphan records in the Patients table. Under relational−database rules, each medical record should map to a patient. In this situation you cannot retrieve any information on the patients associated with the orphan claims records.

Duplicate table records constitute the other form of data corruption. Data duplication can lead to incorrect values generated during queries. For example, when you have two or more identical line items in an Order table, reports based on that table may be flawed. Duplicate records also create problems when you define a relationship between two tables. For instance, you cannot reliably locate the correct record that provides the link between the two tables if one of the tables has duplicate entries.

Theoretically you may create database designs to allow orphan and duplicate records. However, you will rarely encounter these designs. Remember, an RDBMS provides you with the flexibility to create good data models as well as bad ones. You should always design with data integrity in mind.

Ensuring data integrity requires both entity and referential integrity. Entity integrity pertains to individual tables while referential pertains to table relationships. The following sections describe both in greater detail.

Entity integrity

Entity integrity ensures that table data remain unique, which is a requirement of Codd's relational model. As I mentioned in the previous section, duplicate rows can lead to erroneous values on reports or in summary queries. In addition, you cannot effectively create a relationship between two tables when duplicate records exist. RDBMSs rely on data integrity as a foundation.

To ensure uniqueness a table employs a *primary key*, a single attribute that is unique for every record in a table. You may also define a *composite primary key* consisting of multiple attributes that remain unique for each record. In any given record the other columns may contain identical data, but the column(s) representing the primary key must be unique. In general, a primary key, when combined with the table name, acts as a pointer to a particular record.

Figure 2−3 illustrates the primary key for the Employees table I use as an example throughout this book. I have defined the column SSN, which represents an employee's Social Security number, as the unique identifier for each row. No two rows can have the same value as this column.

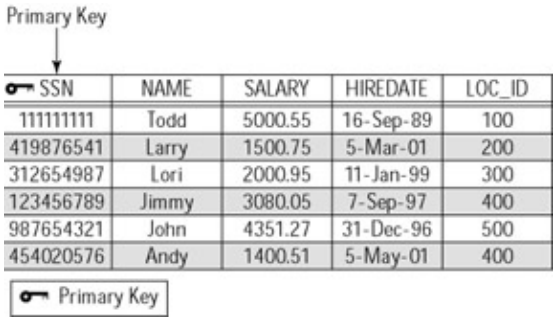

Figure 2−3: Example of a primary key

A column that represents a primary key carries some additional constraints. For example, you cannot store NULL values in these columns. The column(s) that define a primary key must contain meaningful data. NULL values represent uncertainty. In addition, you must always specify a primary key value when adding a new record to a table. Not doing so is equivalent to adding a NULL value.

> Note A database NULL value represents unknown values or indecision. Contrary to what many believe it does not represent the number zero or an empty character string.

Referential integrity

You may build RDBMS applications that contain multiple unrelated tables. However, this creates a database application good for looking up information in only one table at a time. Most RDBMS applications consist of multiple tables related in some manner to model real−world objects and processes.

To relate two tables you must create a link from one table to the other table. *Foreign keys* provide that link. Foreign keys are primary keys from one table used in another table. Because they are primary keys, they act as pointers to unique records in the other table.

For example, consider the Employees and Location tables in Figure 2−4. The Employees table has a column, Loc_Id, which holds the code from the Location table that indicates the home−office location. In this example

the Loc_Id column in the Employees table is a foreign key. Each entry points to a unique record in the Location table. Regarding nomenclature, the Employees entity is called the *relational* table and the Location table is known as the *base* table.

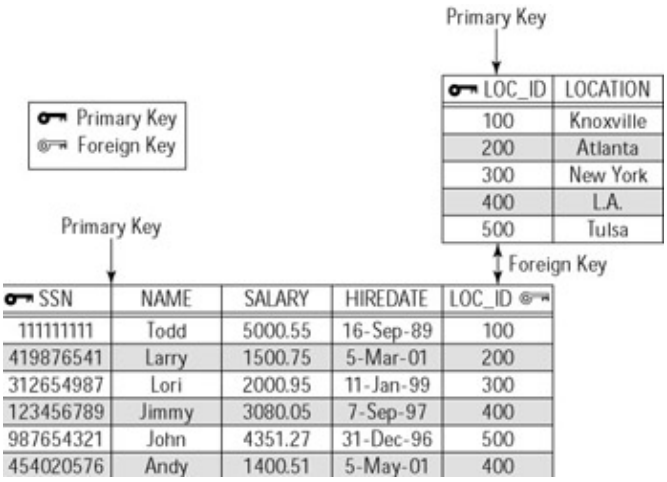

Figure 2−4: Example of a foreign key

Foreign keys also limit data duplication, which minimizes your database's size. If you reference the primary key in a base table you do not have to include the whole record. The foreign key points to it. For example, when querying the Employees table you can easily retrieve the location by joining the two tables in a SELECT statement. I provide an example of this operation in the section "Reviewing SQL Basics" later in this chapter.

Like primary keys, foreign keys have usage rules. For instance, foreign keys must contain valid values in the base table. However, unlike primary keys, a foreign key can also contain NULL values. A table may even have multiple rows containing NULL values for the foreign key. Why? In some situations a record may not have valid value in the foreign key's base table.

Figure 2−5 provides an example of a situation in which you might store a NULL value in a foreign key. Notice that in the Family table the foreign key, Pet_Type, references a Pet lookup table. In some cases a family may not have a pet, in which case a NULL value is appropriate.

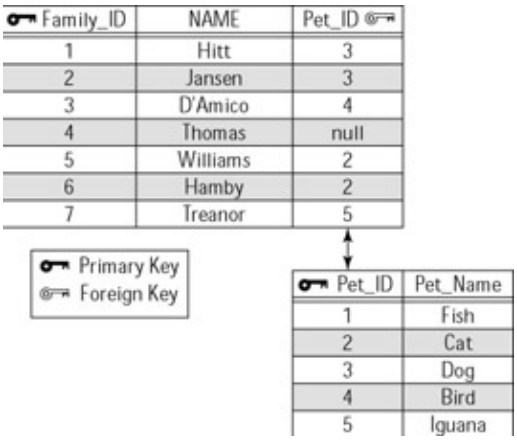

Figure 2−5: Example of a NULL value for a foreign key

However, a NULL value introduces ambiguity because it represents the unknown. In the previous example, if you encounter a NULL value you don't know whether the family doesn't have a pet, forgot to answer the

question, or has a type of pet that is not listed. Good database design suggests including a "no pet" or "not listed" entry in the Pet table to deal with this problem.

Table relationships

As I mentioned above, the ability to relate entities enables you to model systems and processes. Table relationships describe how records in one table map to records in another table. When creating the relationship you rely on primary and foreign keys to glue the entities together. Because the keys have integrity constraints associated with them, they also help to ensure that the relationship remains valid.

The following sections present the different table relationships with examples explaining them.

One−to−one relationships The one−to−one relationship is the simplest. It specifies that only one record in a table maps to only one record in another table. This relationship seldom occurs in database applications. However, you will need it in some obvious situations.

Figure 2−6 shows an example of a situation in which you might encounter a one−to−one relationship. In this case, it describes the relationship between an automobile and an engine. As you know, an automobile can only have one engine and vice versa. In the model, both the Automobile and Engine tables have composite primary keys: Auto_ID and Engine_ID, respectively. The Automobile table stores data about the vehicle, such as color and model. The Engine table stores engine specific information. The two tables share a one−to−one relationship based on their primary keys. In this example the one−to−one relationship is obvious.

Figure 2−6: Example of a one−to−one relationship

You can also use the one−to−one relationship to split a table into two parts. For example, suppose you have a table with a large number of columns. To make administration and documentation easier, or circumvent system limitations, you may want to split the table into two smaller entities. When splitting a table you keep the same primary key for each table and create the one−to−one link based on it.

Splitting a table also makes sense if it contains both confidential and non− confidential data. You can keep the confidential data in a separate table and allow only authorized personnel to access it. Again, identical primary keys in each table create the link.

One−to−many relationships The most common entity relationship is the one−to−many relationship. It occurs when a record in one table has zero, one, or many matching records in another table. You may also hear this relationship appropriately called a parent−child or master−detail relationship.

The relationship frequently occurs when a relational table includes information from a lookup table. Figure 2−7 shows an example of a one−to−many relationship used in the data model for an online store that sells computer systems. All the lookup tables — Mouse, Monitor, and Keyboard — contain unique inventory items for the different components.

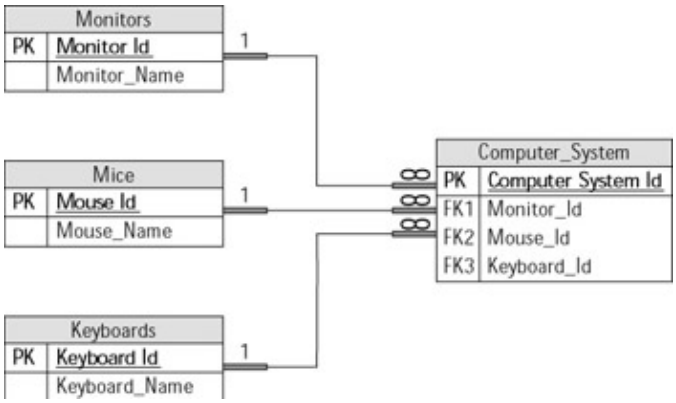

Figure 2−7: Example of a one−to−many relationship

The lookup tables have a one−to−many relationship with the Computer_System table. That is, every record in a lookup table can have zero, one, or many records in the Computer_System table. For example, a certain style of mouse is used in many different systems. The same holds true for keyboards and monitors.

Many−to−many relationships Two tables share a many−to−many relationship when a record in Table A has many matching records in Table B, and a record in Table B has many matching records in Table A.

Unfortunately, the relational model does not enable you to directly implement this relationship between two tables. To implement the relationship you need to use a third table that indirectly join the two other tables to each other. The third table, often called a joining, linking, or association table, has a one−to−many relationship with each table.

Figure 2−8 illustrates this relationship between an Order and a Products table. A single product can exist on many different orders, and a single order can contain many different products. To uniquely identify a specific product−order combination you need an association table, Order_Details. It has two one−to−many relationships, one with the Products table and another with the Order table. Each entry in the Order_Details table represents a specific order−product combination.

As you can see, many−to−many relationships can become complex and confusing. This underscores the importance of understanding the data model before developing an application.

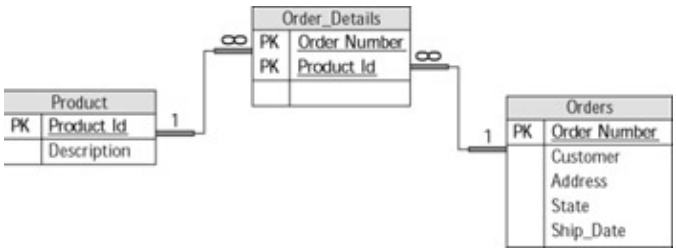

Figure 2−8: Example of a many−to−many relationship

Reviewing SQL Basics

The Structured Query Language (SQL) is the foundation for interacting with an RDBMS. The language enables you to build a database and the structures that hold data, as well as to insert, delete, and retrieve data from it. All RDBMSs use SQL. In fact, the universal acceptance of SQL has helped make the relational

database the most widely used database system. As a result, an understanding of SQL will enable you to work with many different databases with relative ease.

SQL is a non−procedural language. You cannot create a sequence of statements and execute them as a whole. When using SQL you submit one statement, or *command*, at a time and wait for the result. A query provides a good example. You submit a SELECT statement and the database server returns the results. It is not possible to chain two statements together and execute them simultaneously.

However, some database vendors have developed procedural languages around SQL to create a programming environment. With a vendor's language you can create stored procedures, which are mini−programs equivalent to Java methods. Stored procedures enable you to sequentially execute multiple statements as well as to control the flow of execution.

ANSI published the first SQL standard in 1986. The specification defined the basic functions and syntax for database interaction. The next major version, SQL92, published in 1992, provided the major language components we use today. In a nutshell, it gives us language constructs for manipulating schemas and database administration. SQL3, adopted in 1999, provides support for custom data types and supports certain object−oriented programming concepts, such as polymorphism and inheritance. Using SQL3 you can create your own object hierarchy within the database using custom data types.

Although an SQL standard exists, different database vendors have modified it slightly to exploit their different proprietary features. Think of the national language of a country. It's understood everywhere, but different dialects exist in different regions. It's the same with the modern RDBMS. Each vendor uses SQL, but each has its own dialect.

For example, Oracle developed an SQL version called PL/SQL with extensions that apply to Oracle−specific features. SQLServer uses T−SQL, or Transact−SQL, in the same manner. Nonetheless, if you understand SQL concepts, you should have few problems adopting an individual database's SQL dialect.

SQL has two major components with which you can work: the Data Manipulation Language (DML) and the Data Definition Language (DDL). DML is the language for interacting with the data. With it you can add, change, or remove table data. DDL is the language for building the data−storage structures within the database. It enables you to create or destroy structures such as tables, indexes, or even a database itself.

The following two sections describe DML and DDL. I've also provided some examples to illustrate the concepts.

Using Data Manipulation Language (DML)

DML is the name given to a subset of SQL commands used for manipulating records in a relational database. It defines commands for retrieving, inserting, updating, and deleting table data. As a developer you will likely use DML statements more often than any other SQL statement type. I list the four most commonly used DML commands and a brief description of each in Table 2−1.

Table 2−1: Data Manipulation Language (DML) Commands

SQL DML Command	Description
SELECT	Retrieves records from one or more tables in a relational database.
UPDATE	Modifies existing columns for one or more records in a single table.
INSERT	Adds a new record into a single database table.
DELETE	Removes one or more records from a single table.

You use the SELECT command to retrieve data from one or more tables. It does not affect the underlying data, it only returns it. The other statements — UPDATE, INSERT, and DELETE — alter the underlying table data. Improperly using these commands can cause data loss or corruption.

However, most RDBMSs protect against misuse or mistakes by using transactions. Generally, a database system buffers your changes until you issue the COMMIT statement, which tells the database to make the changes permanent. If you decide not to apply changes, and have not yet issued the COMMIT statement, you can use the ROLLBACK statement to undo them.

> Caution JDBC automatically performs commits after each DML statement. You can change this behavior by using the Connection.setAutoCommit() method to change the auto−commit state to false. This will require you to explicitly commit changes with the Connection.commit() method. Your driver may function differently, so you should check the documentation provided with it.

Not only do the UPDATE, DELETE, and INSERT statements affect table data, they also affect indexes. Generally, a database system will update an index whenever a record changes or you add new data. For example, when you add a record to a table, the database adds an entry to the appropriate index. The same holds true for updating and deleting records; the index entry is modified appropriately.

The following sections provide more detail on the SELECT, INSERT, UPDATE, and DELETE DML statements. Figure 2−9 provides a schema for the examples used in this section.

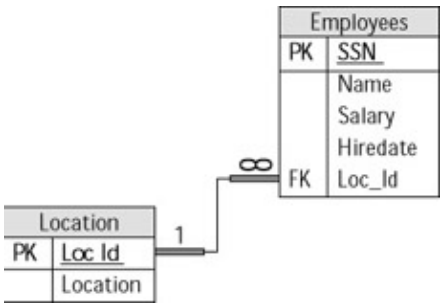

Figure 2−9: Employee−Location entity−relationship diagram

SELECT statements

The SELECT, or query, statement is used to retrieve data from the database. Almost all your database programming work will use this statement in some capacity. Besides being a vendor−specific tool for bulk extraction of data, the SELECT statement provides the only way to retrieve and view records. The general form of a SELECT is as follows:

```
SELECT column_1, ... , column_N
FROM table_1, ... , table_N
[WHERE condition]

[]= optional
```

The column names after the SELECT keyword indicate the table attributes you want to retrieve. You may include one or more of the table columns in the query. You indicate the tables you want to query after the FROM keyword. You can also retrieve data from multiple tables by specifying more than one table in a comma−delimited list. However, the list must contain related tables.

The optional WHERE clause acts as a filter to limit the number of rows returned. You use it to evaluate a column's content against some criteria. Table 2−2 lists the conditional symbols you can use with a WHERE clause. You may also use the WHERE clause with the UPDATE and DELETE statements to limit their effects.

Table 2−2: WHERE Clause Evaluation Symbols

Evaluation Symbol	Description/Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to
LIKE	Special keyword that allows wildcard searches using % to match any possible character. Example: LIKE Name='T%' retrieves all names starting with T.

The following snippets provide some examples of the SELECT statement:

```
SELECT Ssn, Name
FROM Employees
```

Query output:

```
SSN         NAME
---------   ----
111111111   Todd
419876541   Larry
312654987   Lori
123456789   Jimmy
987654321   John
454020576   Andy
```

Notice that the previous query returns the SSN and Name for all the rows in the table. Using the WHERE statement enables you to select which rows to return. For example, the following SQL statement retrieves only my record:

```
SELECT Ssn, Name
FROM Employees
WHERE Name= 'Todd'
```

Query output:

```
SSN         NAME
---------   ----
111111111   Todd
```

If you want to list all the columns you have two choices. First, you can specify every table column in the SELECT statement. This might be impractical if you have a lot of columns. The second option is to use the asterisk (*), to return all the columns. The following SELECT statement demonstrates this option:

```
SELECT *
FROM employees
WHERE name= 'Todd'
```

Query output:

```
SSN        NAME                SALARY     HIREDATE  LOC_ID
---------- ------------------- ---------- --------- -------
111111111  Todd                5000.55    16-SEP-89 100
```

I mentioned earlier that you can retrieve data from multiple tables using the SELECT statement. This enables you to pull records from related tables. To build this type of query you must use create join between the tables involved.

Joins enable you to combine two tables based on the values in specific columns. In general, if you want to join two related tables, you use the foreign-key columns to create a join. It is the inclusion of the foreign key in a relational table that creates the link with the base table. When using a join you usually want to retrieve descriptive data from a lookup table based on the foreign key in the relational table.

For example, suppose I want a list of the home offices for all the records in the Employees table. From Figure 2-9 and the preceding query, you can see that the Employees table has a foreign key, Loc_Id, that is the primary key of the Location table. As a result I will use the Loc_Id column to create the join between the tables. The following query retrieves the information I want:

```
SELECT Employees.Name, Location.Location
FROM Employees, Location
WHERE Employees.Loc_Id = Location.Loc_Id
```

Query output:

```
NAME                LOCATION
------------------- ---------------
Todd                Knoxville
Larry               Atlanta
Lori                New York
Jimmy               L.A.
John                Tulsa
Andy                L.A.
```

One problem can arise when using joins. Because you include columns with identical names from both tables, the SQL processor may have difficulty identifying to which table a column belongs. As a result, you need to uniquely identify the column names. Prefixing them with table names solves the problem. Notice that in the previous query I qualified the columns with the table name and a dot (.) to remove any ambiguities associated with the foreign-key name.

INSERT statements

The INSERT statement enables you to add data to a table. The general form is as follows:

```
INSERT INTO table_name (column_1, ... , column_N)
VALUES (value_1, ... ,value_N)
```

You can add only a single record to a table with each INSERT statement. With the column list you can specify which columns of the new record you want to contain data. You may specify individual columns or all the columns in the table. The entries in the VALUES list must correspond one−to−one with the entries in the column list.

If adding values for all the table columns you do not need to include the column list. (This is a shortcut for tables with a large number of columns.) An INSERT statement of this type has the following form:

```
INSERT INTO table_name
VALUES (value_1, ... ,value_N);
```

When using the previous statement you need to supply values for all columns in the table. In addition, the order of the values must match the order of the columns in the table definition. Here's an example of an INSERT statement that adds another office location to the Location table:

```
INSERT INTO Location
VALUES (600, 'London')
```

Unlike the SELECT statement, which does not modify data, the INSERT statement carries some restrictions on its use. Therefore, when using the INSERT statement you must follow some guidelines. For instance:

- You must always specify values for primary keys and columns designated as NOT NULL.
- New data must match the respective column's data type.
- When using foreign keys you must include valid values in the relational table from the base table.

The RDBMS performs checks on the data you insert to ensure that these rules are not broken. If you issue an illegal INSERT statement the database will throw an error and not allow the change.

UPDATE statements

UPDATE statements enable you to modify one or more column values for existing table data. You can apply the change to all the rows in the table, a related subset, or a single row. The UPDATE statement has the following form:

```
UPDATE table_name
SET column_1 = value_1
[,column_N = value_N]
[WHERE condition]

[]=optional
```

As with the INSERT statement, several restrictions apply to the UPDATE statement. In fact, they share several of the same constraints. First, the data type of the new value must match the data type of the table's column. However, some drivers, or databases, will coerce the data types for you. Nonetheless, don't rely on this behavior. Another restriction is that you can update only a single table with the UPDATE statement. This statement does not support joins between tables.

As with the SELECT statement, you can use the optional WHERE clause to limit the rows affected. This enables you to selectively apply changes. For example, the following snippet gives all the employees in the Employees table a 3 percent raise:

```
UPDATE Employees
SET Salary = Salary * 1.03
```

However, you can use the WHERE clause to select the records you want to update. I'm greedy, so I will use the following statement to give myself an additional 10−percent raise:

```
UPDATE Employees
SET Salary = Salary * 1.10
WHERE Name = 'Todd'
```

The UPDATE statement also enables you to modify columns containing either a primary or foreign key — as long as you follow the general integrity rules I discussed in the previous section.

DELETE statements

As you might expect, the DELETE statement removes one or more records from a table. It has the following form:

```
DELETE table_name
[WHERE condition]
[]=optional
```

You will find that using the statement is straightforward, as the following snippet illustrates:

```
DELETE Employees
WHERE Name = 'Todd'
```

The previous statement deletes my record from the Employees table. Dropping the WHERE clause from the DELETE statement removes all the records in the table. Use this form with caution, as it completely removes a table's contents. This DELETE statement takes the form:

```
DELETE table_name
```

Just as with the other statements, you must abide by the entity and referential integrity rules when using the DELETE statement. For example, you cannot delete a record from a base table containing a value used as a foreign key in another table. You must first break the relationship using DDL or remove the dependent record from the relational table.

Using Data Definition Language (DDL)

DDL enables you to build, manipulate, and destroy database structures. With this subset of SQL you create the database infrastructure, such as tables and indexes, in which to store data. You also use DDL statements to build integrity constraints, such as primary and foreign keys to define table relationships.

DDL statements differ drastically from DML statements because they do not participate in transactions. As a result, you cannot roll them back. DDL statements work directly with the data dictionary and changes occur immediately. If you issue a DROP TABLE statement, the database system removes the table immediately. You cannot get it back. Therefore, use these commands, especially the ALTER and DROP statements, with

care — you do not have the safety net of transactions.

You usually need privileged security permissions to execute DDL statements because they affect the database infrastructure. Typically you need DBA, database "super−user," or schema−manipulation rights.

As a developer you are not likely to use DDL statements very often. When you do, you will likely only use the CREATE, ALTER, and DROP statements. Table 2−3 presents a quick listing of these statements, along with a brief explanation of each. The following sections provide an overview of the statements as well.

Table 2−3: Data Definition Language (DDL) Commands

SQL DDL Command	Description
CREATE	Builds or makes new database structures. Used specifically for tables, indexes, and integrity constraints.
ALTER	Modifies an existing database structure. Used typically for adding new columns to tables or adding integrity constraints.
DROP	Removes a database structure.

CREATE statements

You use the CREATE statement to build all database structures. For example, to build a database, index, and table you use the CREATE DATABASE, CREATE INDEX, and CREATE TABLE statements, respectively.

Most RDBMSs use the CREATE keyword to build custom proprietary structures as well as the standard SQL ones. The syntax for the CREATE statement varies according to the structure you want to build. However, the general form is as follows:

```
CREATE OBJECT attributes
```

For a specific example, examine the following CREATE TABLE syntax:

```
CREATE TABLE table_name
(
coulumn_name1 datatype [constraint,]
[coulumn_name2 datatype] [constraint,]
[coulumn_name3 datatype] [constraint]...
)
[]=optional
```

In this example the keyword TABLE and the column definitions identify the OBJECT and the attributes, respectively. The column definitions consist of two components. The first, which is mandatory, is the database data type. You must give a valid type for your database system because it may not support all ANSI SQL data types or use different nomenclature specific to the system. The second component, which is optional, specifies any constraints you want to place on the column. Some examples are PRIMARY KEY , NOT NULL, and UNIQUE.

> XRef Chapter 7, "Understanding JDBC Data Types," provides more information on both Java and SQL data types.

The following snippet shows the CREATE statement used to build the Employees table shown in Figure 2−9:

```
CREATE TABLE Employees
(SSN number(9) CONSTRAINT PK_EMP PRIMARY KEY,
Name VARCHAR(20),
Salary number(9,2),
Hiredate DATE,
Loc_Id NUMBER(3) CONSTRAINT FK_LOC REFERENCES Location(Loc_Id)
)
```

In the preceding example, each column includes the data type and constraints I need to define the table. The two columns with constraints are the Ssn and Loc_Id columns, which have the PRIMARY_KEY and REFERENCES, or foreign−key, constraints, respectively.

As I mentioned earlier, the structures you can build with the CREATE statement vary according to your database. You should check your documentation to determine your options.

ALTER statements

The ALTER statement modifies an existing database structure, much like the UPDATE DDL statement. As with the CREATE statement, the syntax for this command depends upon the target−database structure and has the following general form:

```
ALTER OBJECT attributes
```

For example, when the OBJECT is a table, the ALTER command enables you to add a new column or change the data type of an existing one. The following form adds a column:

```
ALTER TABLE table_name
ADD column_name datatype
```

This form modifies an existing column:

```
ALTER TABLE table_name
MODIFY column_name datatype
```

For an example of the ALTER statement, the following snippet adds a ZIP_CODE column to the Location table:

```
ALTER TABLE Location
ADD Zip_Code VARCHAR(10)
```

When using the ALTER statement, you still need to adhere to the integrity constraints defined in the database. For example, you must exercise caution when modifying the data type of a primary key column, especially if you use it as a foreign key in another table. In addition, you cannot change a column data type to an incompatible data type, such as changing a DATE to a DECIMAL.

DROP statements

The DROP statement is analogous to the DELETE DDL statement used to remove table records. However, this statement removes an entire database structure and any reference to it in the data dictionary. So use this command with caution, as you typically cannot undo the changes.

The syntax for the DROP statement is simple:

```
DROP object
```

The following snippet removes the Employees table from the database:

```
DROP Employees
```

As with the ALTER command, you need to respect the integrity rules of the database before issuing the DROP statement. For example, you cannot drop a base table that provides foreign keys to a relational table. Consider the Employees−Location relationship shown in Figure 2−9. You cannot drop the Location table without incurring a database error, as doing so breaks the referential integrity constraint. To remove the Location table you need to either break the relationship or drop the Employees table first.

To break the relationship you can use the CASCADE CONSTRAINTS keyword along with the DROP statement. This keyword removes the integrity constraints associated with the table so you can remove it. Therefore, to remove the Location table you issue the following command:

```
DROP Location CASCADE CONSTAINTS
```

Because the DROP statement permanently removes a structure, use it with caution. Contact your DBA to ensure a valid backup exists before issuing the DROP statement against any mission−critical table. That table does not have to be a production table; development tables are just as valuable. Especially once you get everything working the way you want, if that ever occurs.

Summary

The RDBMS plays a major role in enterprises today by storing mission−critical data used to make business decisions or generate revenue. As a developer you will likely build an application that will interact with database in some manner.

This chapter serves as a refresher on relational database systems. In it I covered:

- The basics of the RDBMS architecture
- Entity and referential integrity constraints
- SQL, DML, and DDL statements

Although not a treatise on SQL, this chapter should provide the essentials to help revive your SQL knowledge. Now let’s move on to JDBC programming.

Part II: Understanding JDBC Programming Basics

Chapter List

Chapter 3: Setting Up Your First JDBC Query

In This Chapter

- Understanding JDBC configuration issues
- Obtaining and installing JDBC drivers
- Identifying common JDBC components
- Creating a simple JDBC application
- Understanding the steps used in a JDBC application

Database programming may seem daunting at first glance. After all, it encompasses many facets, such as client/server communications, drivers, APIs, incompatible data types, and SQL statements. You may think you need to know about all these issues before you can start to develop database applications. Frankly, you do need to know *something* about them in order to create reliable database applications; however, JDBC minimizes the learning curve for those just getting started.

The JDBC API abstracts much of the work needed to create robust database applications. Its core components consist of simple, intuitively named objects that do the work for you. To create an application, you just configure and assemble the components in the correct order. However, if you move into JDBC enterprise development, things change a little. You use different objects for opening database connections, but their functionality remains the same.

JDBC programming is very methodical. Ninety percent of JDBC applications use the same objects and methods regardless of what you want to accomplish. For example, you always load a driver, open a connection, submit an SQL statement, and examine the results. The details of each step vary very little from task to task.

In this chapter, I guide you through building a simple JDBC application from start to finish. I start by discussing how to configure JDBC. Next, I identify the common components used in all JDBC applications, and then I present a sample application and cover the discrete steps involved in it. Lastly, I cover how to compile and run a JDBC application as well as how to troubleshoot any problems that may occur.

Configuring JDBC

JDBC is an API that encapsulates the low−level calls needed for database access and interaction into one common interface. Both the Java Development Kit (JDK) and Java Runtime Environment (JRE) contain the API as part of the standard distribution. The API’s interfaces and classes reside in the java.sql and javax.sql packages. The standard components are packaged in java.sql while the enterprise elements are in javax.sql.

The JDBC API differs from a JDBC driver. The API defines the interfaces and methods vendors implement when writing a driver. If you examine the API source code you will find it consists mainly of interfaces. As a result, before you can write a JDBC application, you need to obtain and install a JDBC driver, which implements the interfaces. However, a single JDBC driver does not enable you to access different “brands” of databases. In other words, you cannot access an SQL Server database using an Oracle driver. You must use a driver specifically targeted for your database.

To help you understand the role of a driver, Figure 3−1 depicts how JDBC and a Java application interact. All communications with a database must go through the JDBC driver. The driver converts the SQL statements to a format the database server understands and makes the network call using the correct protocol. The JDBC driver abstracts the database−specific communication details from you. All you need to learn in order to create a database application is SQL and JDBC.

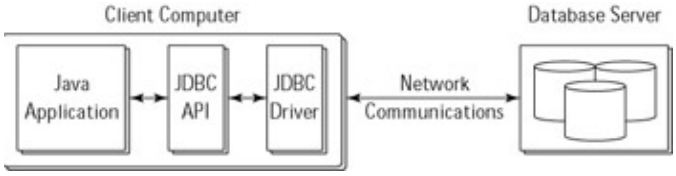

Figure 3−1: JDBC−Java relationship

In the following two sections I explain how to obtain and install a JDBC driver for your database. The process is straightforward, as you will see.

Obtaining JDBC drivers

As I mentioned in the previous section, you must obtain a JDBC driver for your target database before you start to write database applications. Most often your database vendor can supply a JDBC driver. If not, you can likely find a third−party implementation that works with your database. Regardless of how you obtain a driver, the point to remember is that it must target your database.

To help you get started more quickly, both the JDK and JRE contain a JDBC−ODBC bridge driver that enables you to use ODBC drivers for database access.

Figure 3−2 illustrates how JDBC and ODBC function together. Database calls in an application still use JDBC methods. However, instead of communicating directly with the database, the JDBC driver communicates with the ODBC driver, which in turn communicates with the database. As a result, you still need an ODBC driver for your database. Again, your database vendor, or a third party, will likely have an ODBC driver available for your use.

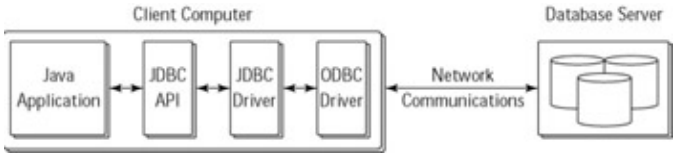

Figure 3−2: JDBC−ODBC bridge architecture

Sun’s Web site, http://java.sun.com/products/jdbc/, provides information on JDBC drivers and vendors. It also features a search engine to help you locate a driver to match your database and application needs.

Installing the JDBC driver

Once you obtain a JDBC driver, you must install it. Fortunately, installing JDBC drivers is identical to installing other Java APIs: Just add the driver path to the classpath when running or compiling the application. If you mistype the path or forget to add it, numerous errors will occur.

This step might sound trivial but neglecting it often creates frustration in new JDBC programmers. They often think they have faulty code when they really have classpath issues.

When you are using the JDBC−ODBC driver the classpath requirement does not apply. Sun has built the driver into the distribution, so you need not worry about the classpath settings. However, to use the JDBC−ODBC bridge you must meet different requirements.

One requirement, as I mentioned previously, is that you have an ODBC driver for your database. The JDBC−ODBC bridge will not operate without it. Second, you must configure a valid ODBC Data Source Name (DSN) before you can run an application. Chances are that you already have one configured for your database if you do any work with ODBC. If not, your ODBC driver documentation should contain instructions on configuring a DSN.

However, you might opt to forgo the JDBC−ODBC bridge and use pure Java instead. This approach gives you the luxury of not having to ensure that the ODBC driver and DSN exist on all workstations. This is a real benefit when it comes to deploying and maintaining the application.

Examining the Common JDBC Components

You can use JDBC to create very diverse database applications. For example, you can write an application as an EJB component that manages inventory or processes customer orders for an online store. Or you can create a JDBC application to help DBAs manage their databases. Regardless of the purpose, all JDBC applications have similar requirements.

First, the application must be able to communicate with the database. This means that it must understand the protocol and low−level language the database server uses when communicating with the client. Second, the application must be able to establish a connection with the database server in order to create a communication channel for sending SQL commands and receiving results. Finally, the program must have a mechanism for handling errors. Database applications use complex operations and numerous opportunities for failure exist — such as intermittent networks and malformed SQL commands.

To meet these requirements the JDBC API provides the following interfaces and classes:

- Driver — This interface handles the communications with the database server. It encapsulates the "know−how" for interacting with a database. Very rarely will you interact directly with Driver objects. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- Connection — Instantiated objects of this interface represent a physical connection to the database. You can control result set behavior and transaction operations using Connection objects.
- Statement — You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- ResultSet — These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- SQLException — This class handles any errors that occur in a database application.

Regardless of the JDBC application, you always work, directly or indirectly, with these four components. Once you understand them, writing JDBC applications becomes easier. The next section shows the components in action as I demonstrate the steps required to create a simple JDBC program.

Writing Your First JDBC Application

Now that you have your JDBC driver, know how to install it, and have familiarity with the common JDBC components, I'll provide an example of how to create a simple JDBC application. This will show you how to open a database connection, execute a query, and display the results. This example can serve as a template when you need to create your own JDBC application in the future.

You will find JDBC programming very much like following a recipe. To accomplish any one programmatic task requires repeating the same steps. For example, you must always load a JDBC driver and open a database connection. Each operation has its own set of steps, which remain the same regardless of the application. As a result, you may choose to build classes to take care of most of the drudgery.

Creating the sample application

For the example, I'll use the database schema I built in Listing 5−1 of Chapter 5, "Building JDBC Statements." You can either run that example first to create the schema, modify this example to fit your environment, or just follow along and pick up the concepts.

> XRef Part III, "Using Java Data Access Design Patterns," covers object−oriented programming techniques that reduce the repetitiveness of JDBC programming. Design patterns focus on building robust, scalable, and reusable programs using object−oriented concepts.

Figure 3−3 illustrates the six steps required to make a connection, submit a query, and retrieve data. Unless you use a different query, such as one that accepts parameters, these basic steps do not change.

From the figure you can see that the major steps you must complete include:

1. *Import the packages* — Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice.
2. *Register the JDBC driver* — Requires that you initialize a driver so you can open a communications channel with the database.
3. *Open a connection* — Requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database.
4. *Execute a query* — Requires using an object of type Statement for building and submitting an SQL statement to the database.
5. *Extract data from result set* — Requires that you use the appropriate ResultSet.get*XXX*() method to retrieve the data from the result set.
6. *Clean up the environment* — Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

A little later in the chapter I'll provide the details of each step shown in Figure 3−3. For now, let me present a simple example I use to illustrate the steps. For your convenience, I provide the example in Listing 3−1 before beginning the discussion.

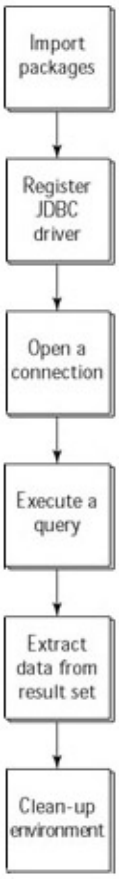

Figure 3−3: The six steps involved in building a JDBC application

In brief, the application opens a connection with the Employee database, submits an SQL query asking for data from the Employees table, and finally displays the results. As I mentioned earlier, I am keeping the example simple to illustrate the distinct steps involved in JDBC programming.

Before continuing let me also mention exception handling. Most JDBC methods throw an SQLException. Those that don't generally don't throw any exceptions. For this reason you need to catch the SQLException in your code. In my example I place all the commands in a try−catch block and explicitly handle the SQLException error.

Also notice that in the example I initialize the Connection object before entering the try−catch block. This enables me to access the variable in a finally clause to ensure it is properly closed.

Listing 3−1: FirstQuery.java

```
package Chapter3;

//STEP 1. Import packages
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.Date;
import java.sql.SQLException;
public class FirstQuery {

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

```
//Define Connection variable
Connection conn = null;

//Begin standard error handling
try{

   //STEP 2: Register JDBC driver
   String driver = "oracle.jdbc.driver.OracleDriver";
   Class.forName(driver);

   //STEP 3: Open a connection
   System.out.println("Connecting to database...");
   String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
   String user = "toddt";
   String password = "mypwd";
   conn = DriverManager.getConnection(jdbcUrl,user,password);

   //STEP 4: Execute a query
   Statement stmt = conn.createStatement();
   String sql;
   sql = "SELECT SSN, Name, Salary, Hiredate FROM Employees";
   ResultSet rs = stmt.executeQuery(sql);

   //STEP 5: Extract data from result set
   while(rs.next()){
      //Retrieve by column name
      int ssn= rs.getInt("ssn");
      String name = rs.getString("name");

      //Retrieve by column index as an example
      double salary = rs.getDouble(3);
      Date date = rs.getDate(4);

      //Display values
      System.out.print("SSN: " + ssn);
      System.out.print(", Name: " + name);
      System.out.print(", Salary: $" + salary);
      System.out.println(", HireDate: " + date);
   }

   //STEP 6: Clean-up environment
   rs.close();
   stmt.close();
   conn.close();

}catch(SQLException se){
   //Handle errors for JDBC
   se.printStackTrace();

}catch(Exception e){
   //Handle errors for Class.forName
   e.printStackTrace();

}finally{
   //finally block used to close resources
   try{
     if(conn!=null)
       conn.close();
   }catch(SQLException se){
     se.printStackTrace();
```

```
         }//end finally try

      }//end try

      System.out.println("Goodbye!");

   }//end main
}//end FirstQuery
```

The output from Listing 3−1 is as follows:

```
Connecting to database...
SSN: 111111111, Name: Todd, Salary: $5000.55, HireDate: 1995−09−16
SSN: 419876541, Name: Larry, Salary: $1500.75, HireDate: 2001−03−05
SSN: 312654987, Name: Lori, Salary: $2000.95, HireDate: 1999−01−11
SSN: 123456789, Name: Jimmy, Salary: $3080.05, HireDate: 1997−09−07
SSN: 987654321, Name: John, Salary: $4351.27, HireDate: 1996−12−31
Goodbye!
```

Examining the output from Listing 3−1 shows that the query retrieved the SSN, Name, Salary, and HireDate from the Employees table. Now, let me give you the details of each step required to make it happen.

Step 1: Import the packages

As with all Java applications you must import the packages that contain the classes you need for your program. Fortunately, all the JDBC interfaces and classes exist in either the java.sql or javax.sql package.

The java.sql package contains the JDBC core libraries. You use classes or interfaces from this package in every JDBC application. The javax.sql package contains the classes and interfaces that support enterprise−level JDBC programs. You need this package only when dealing with connection pooling, rowsets, distributed transactions, or other advanced features.

XRef Part IV, “Taking It to the Enterprise,” provides more information on using the javax.sql package. It covers connection pooling, rowsets, distributed transactions, and more.

The following is a list of the classes and interfaces I import in the example:

- java.sql.DriverManager — Manages JDBC drivers. Maintains an internal collection of Driver objects and provides them as needed for database communications.
- java.sql.Connection — Represents the physical connection to the database. Objects instantiated from this interface also control transaction levels and the types of result sets created when a query returns.
- java.sql.Statement — Sends SQL statements to the database. This interface enables you to send only static statements. The java.sql. PreparedStatement interface defines methods that allow you to use SQL statements that accept parameters.
- java.sql.ResultSet — Holds the SQL query results and provides an iterator so you can traverse the ResultSet object’s data.
- java.sql.Date — JDBC data type identifier that maps to the SQL DATE data type.
- java.sql.SQLException — Handles database errors and JDBC programming exceptions.

Now that I have provided the import statements so my application can find the JDBC components, I can move on to registering a JDBC driver.

Step 2: Register a JDBC driver

As I mentioned earlier, the driver contains the "know−how" for communicating with the database. Therefore, in your application you must always register a driver when working with JDBC. You can do this using either:

- DriverManager.registerDriver(Driver *driverClassName*) method
- Class.forName(String *driverClassName*) method

The Class.forName() method affords more flexibility because it accepts a String parameter representing the driver class name. This enables you to dynamically obtain driver values at runtime from the command line or a properties file. Using DriverManager.registerDriver() method requires a parameter of type Driver. Generally you must hard code the fully−qualified path name for the parameter, which limits your flexibility.

The following snippet from Step 2 of the example illustrates my use of the Class.forName() method to initialize a driver:

```
//STEP 2: Register JDBC driver.
String driver = "oracle.jdbc.driver.OracleDriver";
Class.forName(driver);
```

For this example I use Oracle's 8.1.7 JDBC driver; the String variable driver represents the fully qualified name of the class. Although I use the Class. forName() method, DriverManager still manages the driver in the background. Per the JDBC specification, all objects implementing the Driver interface must self−register with DriverManager. As a result, examining the objects in memory shows an instance of DriverManager even though you use the Class.forName() method.

XRef Chapter 4, "Connecting to Databases with JDBC," provides more information on registering JDBC drivers using the Class.forName() method and the DriverManager object.

A note on exceptions: The Class.forName() method throws a ClassNotFoundException if the driver specified by the parameter cannot be located during runtime. In the example, I handle this error by catching a stan−dard Exception. The DriverManager.registerDriver() method throws an SQLException if a problem occurs during driver registration.

Now that I have initialized my driver I can begin setting the connection parameters and open a database connection.

Step 3: Open a database connection

Most database applications operate in a client−server environment. JDBC applications act as clients and to work with the server they must establish a physical connection to it. This is true regardless of whether the application resides on the database host or on a different host.

As mentioned, Connection objects represent a physical connection with a database. As a result, Connection objects provide the conduit for server communications. If the object is not directly involved, it indirectly participates.

For example, you don't execute a query using a Connection object. However, it does act as a factory to produce a Statement object, which references it for server interaction. Here's how it works: When you call the Statement.executeQuery() method, the Statement object uses the Connection object to submit the query to the database. The results come back through the Connection object and into the Statement object, which in turn

populates a ResultSet object.

To open a database connection you call the DriverManager.getConnection() method. Before calling the method you must set several DriverManager parameters used to open the database connection. The number of parameters required varies among databases. However, you usually supply the following parameters:

- **JDBC URL** — Specifies the database location as well as driver configuration information. This parameter's format depends upon vendor and driver requirements.
- **Username** — Indicates the database login account you want to use when opening the connection.
- **Password** — Specifies the password associated with the username parameter.

The following snippet from Step 3 of the example shows how I set these parameters:

```
//STEP 3: Open a connection.
System.out.println("Connecting to database...");
String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
String user = "toddt";
String password = "mypwd";
conn = DriverManager.getConnection(jdbcUrl,user,password);
```

Of all the parameters required to make a connection, the JDBC URL generates the most confusion. It represents three pieces of information. The first part of the URL, JDBC:oracle:thin, gives the driver specific information about how to connect to the server. This information means more to the driver than it does to you or me.

The next part, @localhost:1521, tells the driver the database server's location, using the server's host name and port. In this example, it resides on the localhost and listens on port 1521. The last piece of information, ORCL, indicates the target database name as used by the database server. This is not the database server's name. Note that these JDBC−URL settings apply to the Oracle 8.1.7 JDBC driver. If you use a different driver, you will need to use a different format. Check your driver's documentation for details.

XRef Chapter 4, "Connecting to Databases with JDBC," covers the intricacies of opening connections using DriverManager and working with Connection objects.

With the parameters set, I call the DriverManager.getConnection() method to open a database connection. If the connection attempt fails, an SQLException occurs. The Connection object returned represents a physical connection to the server. I will use this object throughout the example to interact with the server, such as when executing a query.

Okay, I have established a database connection, now I can submit my SQL query to retrieve information.

Step 4: Execute an SQL query

You can start preparing to issue SQL commands once you establish a database connection. Using JDBC you can execute any SQL command you wish. You can use DDL to build database structures or DML to manipulate and retrieve data. However, you must ensure you have the appropriate security privileges within the database to execute the command. If you do not have proper privileges, an SQLException will occur.

Querying the database requires two objects. The first object implements the Statement, PreparedStatement, or CallableStatement interface. Each of these interfaces has a different purpose, as outlined in Table 3−1. In brief, use the Statement interface for simple, static SQL statements, the PreparedStatement interface for parameterized SQL statements, and the CallableStatement interface to execute stored procedures in the

database.

XRef Chapter 5, “Building JDBC Statements,” provides more detail on how to use Statement, PreparedStatement, and CallableStatement objects.

Table 3−1: The Family of Statement Interfaces

Interface	Description
java.sql.Statement	Enables you to submit static SQL queries or commands.
java.sql.PreparedStatement	Works with SQL statements that accept parameters. The parsed form of the SQL command remains cached, which speeds the next execution.
java.sql.CallableStatement	Allows you to access and execute database stored procedures.

The second required object is a ResultSet. This object holds the query results and provides an iterator with which you can traverse the result set and view the row and column data returned.

XRef Chapter 6, “Working with Result Sets,” covers the different ways to use ResultSet objects to view and update query results.

In this example, I use a basic Statement object to submit a simple, static, SELECT statement. The following snippet, from Step 4 of the example, illustrates how I create a Statement object and execute a query:

```
//STEP 4: Execute a query
Statement stmt = conn.createStatement();
String sql;
sql = "SELECT SSN, Name, Salary, Hiredate FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
```

To instantiate a Statement object I call the Connection.createStatement() method. As I mentioned in the previous section, the Connection object provides the main interface with the database, and objects that interact with the server originate from the Connection object.

Next I define a String variable, sql, to hold the SQL query. The next statement, stmt.executeQuery(), executes the query. The method returns a ResultSet object, which I store in the variable rs. As usual, any errors that occur generate an SQLException.

At this point I have successfully executed my query. Now I can view the data residing in the ResultSet object.

Step 5: Display the query results

As I mentioned before, the ResultSet object holds the data returned by an SQL SELECT statement. It stores the data as a table of rows and columns. The rows of a result set fulfill the query’s criteria. For example, if I limited the query to people with the last name Thomas, all the rows in the result set would meet that criterion.

The result set columns map one−to−one to the attributes listed in the SELECT statement. In my example I chose the SSN, Name, Salary, and Hiredate column from the Employees table. The data type of the result set columns is the same as the data type on the server. The ResultSet.get*XXX*() method used to return column data

coerces the database data into a type compatible with Java.

XRef Chapter 7, “Understanding JDBC Data Types,” covers SQL and Java data type issues and their compatibility.

The ResultSet object uses a cursor to point to rows in the result set. To access the result set data, you must move the cursor from row to row and retrieve the data from each row’s columns. Moving through the result set is simple: Just call the ResultSet.next() method, which advances the cursor to the next row.

However, two special cursor locations exist, one Before the First Row (BFR) and one After the Last Row (ALR). These areas do not contain data and trying to retrieve information from them throws an SQLException. After initially populating a ResultSet object, the cursor points to the BFR position. As a result, you must advance the cursor to a row containing data before calling a get*XXX*() method.

Generally you call the ResultSet.next() method to position the cursor on the first row. However, you can use any cursor movement method to position the cursor.

Now that I have explained the basics of using a ResultSet object, let me illustrate how to apply them with Step 5 from the example:

```
//STEP 5: Extract data from result set
while(rs.next()){
   //Retrieve by column name
   int ssn= rs.getInt("ssn");
   String name = rs.getString("name");

   //Retrieve by column index as an example
   double salary = rs.getDouble(3);
   Date date = rs.getDate(4);

   //Display values
   System.out.print("SSN: " + ssn);
   System.out.print(", Name: " + name);
   System.out.print(", Salary: $" + salary);
   System.out.println(", HireDate: " + date);
}
```

Step 5 lists all the records in the Employees table, along with the column values. To do so I use a while−loop with the rs.next() method as the conditional statement. The method returns true when the cursor moves into a valid row, and false when it enters the ALR position, which causes the loop to terminate.

The operations within the while−loop extract the result set data. I retrieve data from the columns by calling the ResultSet.get*XXX*() method, where *XXX* maps to the Java data type of the variable I want to assign a value. To illustrate the different get*XXX*() methods, I purposely included many of the same data types (NUMERIC, DATE, VARCHAR) you may encounter when writing your applications.

The get*XXX*() method requires one parameter, either a String or an int, that identifies the column from which to retrieve data . You can use a String to identify the column by name or an int to specify the ordinal position.

Tip The parameter i in the ResultSet.get*XXX*(int i) method represents the ordinal position of the target column. Unlike Java arrays’ numbering system, the column’s numbering system starts with one, not zero. For example, to access the third column you use get*XXX*(3), not get*XXX*(2).

The while−loop exits once the cursor moves through all the result set rows. At this point I have finished with my application. However, one important task remains: I have to clean up the JDBC environment.

Step 6: Clean up your environment

Properly closing all the database resources is an important step in JDBC programming. Keeping resources open, such as a Connection object, requires client, server, and network resources. As a responsible programmer, you should try to minimize your application's impact on these resources, as it will negatively affect other users and processes.

Fortunately, closing database resources is simple: Just call the close() method of each JDBC object. The following snippet from Step 6 of the example illustrates:

```
//STEP 6: Clean−up environment
rs.close();
stmt.close();
conn.close();

}catch(SQLException se){
   //Handle errors for JDBC
   se.printStackTrace();

}catch(Exception e){
   //Handle errors for Class.forName
   e.printStackTrace();

}finally{
   //finally block used to close resources
   try{
      if(conn!=null)
      conn.close();
   }catch(SQLException se){
      se.printStackTrace();
   }//end finally try
}//end try
```

Notice that I close all ResultSet, Statement, and Connection objects. In general, closing the Connection object also closes the ResultSet and Statement object. However, depending upon the vendor's driver implementation, this behavior may not occur. Therefore, I explicitly close the objects in accordance with good programming practice. Lastly, notice that I use a finally block to ensure the proper cleanup of the Connection object. I do this in case an Exception occurs and the explicit close() methods are skipped.

Compiling and running the application

If you know how to compile and run standard Java applications, you should have no problem doing the same with JDBC programs. As I mentioned at the beginning of the chapter, the most common problem has to do with the classpath setting. You *must* include the path to your JDBC driver in the classpath when compiling and executing an application.

How you configure the classpath setting does not matter. You can create a CLASSPATH environment variable to hold the setting. This technique is useful if you have a static environment and can rely on program locations remaining constant. You can also set the classpath using the classpath option when running javac or java. This method provides more consistency in how you start an application, because it self−documents the startup command.

In my example I need only to set the classpath when running the application. Because I am using the Class.forName() method, which uses a String parameter representing the driver, the compiler does not try to reference the driver class. Instead, the class is dynamically loaded at runtime.

The following examples show how to compile the sample application on Windows NT/2000 and Linux using the JDK1.4:

On Windows NT/2000:

```
E:\jda\code>javac Chapter3\FirstQuery.java
```

On Linux:

```
$ javac Chapter3\FirstQuery.java
```

Executing the application does require setting the classpath properly. The following are examples of how to execute the application on Windows 2000 and Linux:

On Windows NT/2000:

```
E:\jda\code>set ORACLEJDBC=d:\oracle\ora81\jdbc\lib\classes12.zip
E:\jda\code>java −classpath .;%ORACLEJDBC% Chapter3.FirstQuery
```

On Linux:

```
$ export ORACLEJDBC=/oracle/ora81/JDBC/lib/classes12.zip
$ java −classpath .:$ORACLEJDBC Chapter3.FirstQuery
```

Troubleshooting the sample application

JDBC programming has lots of room for errors. I guarantee that SQLException errors will occur while you're debugging and testing your applications. Fortunately, most errors are relatively simple to fix. The JDBC API is very robust and most mainstream driver implementations are equally solid.

To understand what can go wrong, consider the operations that must occur to retrieve data from a database. First you must make a connection to the database server over a network. Client−server programming is not trivial and lots of failure points exist. Next you must issue a correctly formatted SQL statement to the database. This requires understanding your target schema so you can specify the correct table and column names. In addition, you need to know the column data types so you can use the correct get*XXX*() method to retrieve the data. Any problems that occur in these operations can generate an SQLException.

Table 3−2 provides you with a quick reference for troubleshooting JDBC problems. Most often your problem will be the result of an incorrectly set classpath or an invalid SQL statement.

Table 3−2: Common JDBC Problems and Solutions

Problem	Possible Solutions

Cannot compile or execute Java application.	Ensure that the classpath contains the location of the the JDBC driver.
	Ensure you have spelled the fully qualified name of the driver correctly. Some driver names are large and easily misspelled.
Cannot connect to server.	Check the server name.
	Ensure that you have a network connection.
	Check your username and password.
	Ensure that you have used the correct JDBC URL format for your driver.
	Check the database name and port.
SQL commands fail.	Check the SQL syntax.
	Ensure that you are using the correct column and table names.
	Ensure that you have sufficient rights to execute the SQL statement
	Ensure that your SQL statement does not violate integrity constraints.

Summary

In this chapter I showed you how to write a simple JDBC application. Now you should have a basic understanding of the components and steps required to start JDBC programming.

In particular this chapter covered:

- Understanding the JDBC environment
- Working with JDBC drivers
- Importing the correct Java packages for JDBC programming
- Opening database connections
- Executing SQL queries and viewing the results
- Troubleshooting problems you may encounter while creating JDBC applications

Chapter 4: Connecting to Databases with JDBC

In This Chapter

- Choosing a JDBC driver type
- Loading and managing JDBC drivers
- Using DriverManager objects
- Opening and closing database connections
- Understanding the role of the Connection object in JDBC

Chapter 3, “Setting Up Your First JDBC Query,” provided the basics on connecting to a database and on retrieving and updating information. In this chapter, I provide more details on the steps involved in connecting to a database — such as registering JDBC drivers, opening database connections with DriverManager, and working with the Connection objects. To begin the chapter, I’ll explain the architectural differences among the four JDBC driver types and when you would use each. Next, I’ll illustrate how to register and manage the drivers with DriverManager. Finally, I’ll show you how to connect to your database with DriverManager objects, and some good programming practices associated with them.

Understanding JDBC Drivers

Just as a country has its own language for communication among its citizens, a database system has its own language or protocol for communication between the server and the client.

Most modern databases are client−server implementations using either single, two−, or three−tier architectures. Single−tier systems, such as applications that execute on the database server host, house both the client and the server on the same host. Daemons and other automated tasks, like database maintenance programs, may be implemented as a single−tier system.

However, you will likely find most deployments using either two− or three− tier architectures. Figure 4−1 shows an example of the two− and three−tier architectures. In two−tier systems, the client and server generally reside on different hosts and communicate with each other using their own protocol over a network. These implementations are most common. Standard Oracle and SQL Server implementations use a two−tier architecture.

A three−tier system uses a server, commonly known as the application server, between the client and DBMS. This server location is often called the middle tier. Developers often place business logic on the servers to help minimize the deployment of code to the client, which makes code maintenance easier. If the code containing the business logic changes, developers can make updates to the software on the application server without affecting the client configurations. J2EE and web servers often play the middle tier role.

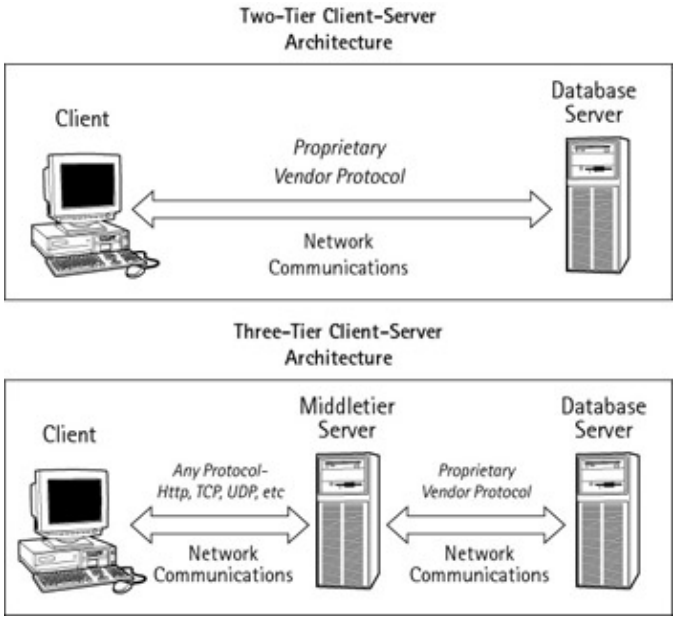

Figure 4−1: Two− and three−tier architectures

Regardless of the “tier” architecture, all client−server systems essentially function the same way. Clients package SQL or database commands to the DBMS for execution in the language, or protocol, they both understand. Once the server completes the request, it sends the client the response using the same protocol. The response may contain data that meet the criteria defined in an SQL query, or it may contain a success or failure result code.

As you can see, if you want to write a client database application you must communicate with the server. You can do this in two ways:

- Provide a custom implementation of the DBMS communication protocol. To use this approach, you must understand the details of the communication protocol used by the database system. You package any calls you make to the database in this protocol. Once you receive a response, you unpackage and use the results. This option is hardly practical, because implementing a full protocol stack requires a tremendous programming effort.
- Use the database vendor’s or a third−party provider’s implementation of the communication protocol. Such implementations are called software drivers or Application Program Interfaces (APIs). This is the more popular option.

A database API, or *driver*, defines methods and properties that enable you to send, retrieve, and obtain status information about the database as well as extract data. You can obtain database drivers from a variety of sources. Most often, database distributions, either commercial or open−source, provide drivers you can use. In addition, third parties develop drivers for popular database platforms such as Oracle, SQL Server, and DB/2.

To address some of the previously mentioned challenges associated with writing database applications, and to avoid the chaos that would inevitably result if every database vendor provided a proprietary API, Sun Microsystems defined a standard API to provide a consistent interface for driver writers to use when writing their drivers. Sun named this API, or the whole database connectivity technology in general, JDBC. Oddly, JDBC is not an acronym. But it is a registered trademark used to identify the Java database connectivity technology as a whole.

What are JDBC drivers?

JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server. For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

Where do you get a JDBC driver? Database vendors and third parties can produce them. However, chances are that you will use the driver supplied by your database vendor. Once you obtain a JDBC driver you should only have to worry about registering it using DriverManager objects and creating the proper JDBC URL in order to use it.

JDBC driver types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Thus, Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, whose characteristics vary greatly. Figure 4−2 provides a high−level overview of the various types.

Types 1 and 2 rely heavily on additional software (typically C/C++ DLLs) installed on the client computer to provide database connectivity. Java and JDBC use these components to interact with the database. Types 3 and 4 are pure Java implementations and require no additional software to be installed on the client, except for the JDBC driver. Fortunately, packaging the JDBC driver with your software distribution is trivial.

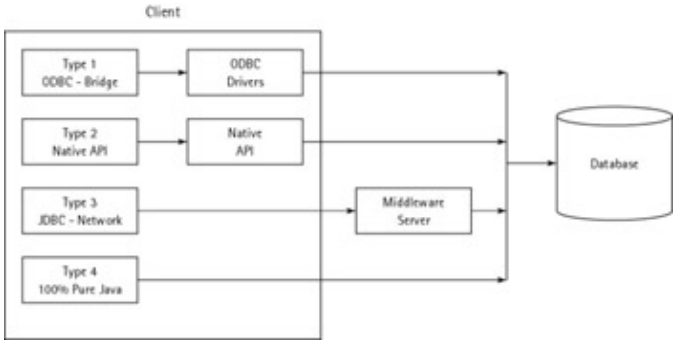

Figure 4−2: JDBC driver types

Type 1: JDBC−ODBC bridge This category works with ODBC drivers supplied by your database vendor or a third party. Although you may find ODBC implementations for UNIX, they are usually used with Microsoft Windows. To use the bridge, you must first have an ODBC driver specifically for your database and any additional software that you need for connectivity. Figure 4−3 shows how a client interacts with the database using the JDBC−ODBC bridge.

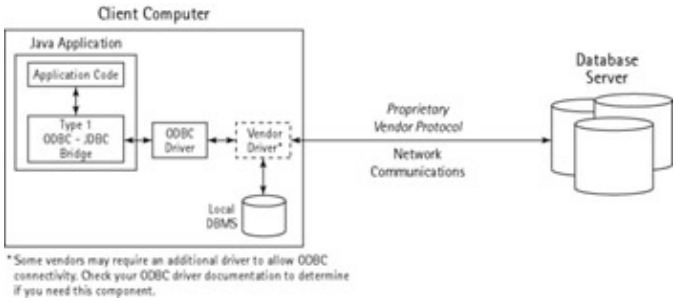

Figure 4−3: JDBC Type 1 driver: JDBC−ODBC bridge

Using ODBC also requires configuring on your system a Data Source Name (DSN) that represents the target database. You may find solutions using this driver type difficult or impossible to implement if your application requires dynamically downloading and running Java programs such as applets. In addition you may find it harder to package and distribute software relying on ODBC. Not only must you ensure that the ODBC driver installs properly, but you must also create the proper DSN.

However, Type 1 drivers do have some uses. For example, you may save some development and deployment costs by using inexpensive desktop or workgroup databases such as Microsoft Access with Type 1 drivers. In addition, ODBC bridges may be extremely useful in (Windows−based) two− and three−tier applications in which only a small number of clients need access to the database.

Type 2: JDBC−native API This category requires an operating system–specific API that handles the database communications. Usually the vendor implements the API using C/C++ so it will be optimized for the client's environment. As a developer, you do not have to worry about the details of the native API, a Type 2 driver handles the interaction. Figure 4−4 illustrates a Type 2 JDBC driver configuration.

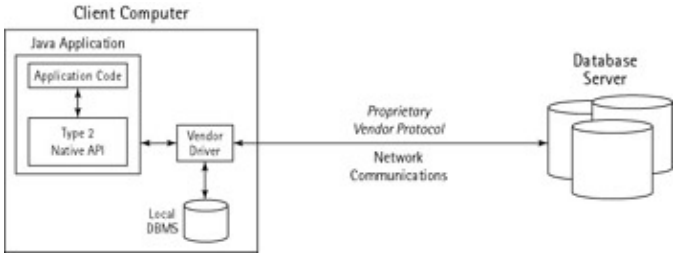

Figure 4−4: JDBC Type 2 driver: native API

The Type 2 driver is like the Type 1 driver in that JDBC calls are converted by the native API into the format recognized by the target database. One interesting note is that you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead. However, as with ODBC, you still bear the burden of properly installing the vendor's API on each of your client's computers.

Type 3: 100% Pure Java, JDBC−network Type 3 drivers use a three−tier approach to accessing databases. J2EE deployments often implement this architecture. Clients use standard network sockets to communicate with an application server. The socket information is then translated by the application server into the call format required by the DBMS, and forwarded to the database server. Figure 4−5 shows a typical three−tier configuration that uses a Type 3 driver for database connectivity.

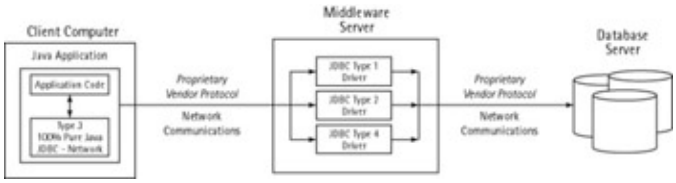

Figure 4−5: JDBC Type 3 driver: 100% Pure Java, JDBC−network

You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type. For example, because your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Nonetheless, you may find this approach most flexible. Application server vendors often provide support for multiple database back−ends. This feature may enable you to write a single code base that supports numerous databases. The middle tier will handle the SQL syntax and data−type nuances that exist among databases.

Type 3 drivers make deployment easier than types 1 or 2 because the client does not need any additional software, other than your application, to access the database.

Type 4: 100% Java This driver is implemented entirely in Java and encapsulates the database−specific network protocols to communicate directly with the DBMS. As with the Type 3 driver, clients do not need additional software. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers. Figure 4−6 illustrates how a Type 4 driver operates on a client.

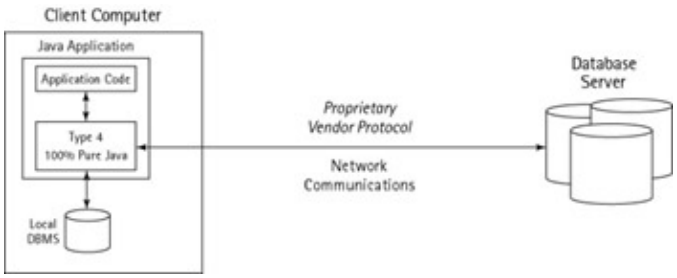

Figure 4−6: JDBC Type 4 driver: 100% Pure Java

Each type of driver has its benefits and weaknesses. Your job as a developer is to understand them so you can make good design decisions. For your convenience, Table 4−1 provides an overview of the pros and cons associated with each driver type.

Table 4−1: JDBC Driver Types

Driver Type: Category	Pros	Cons
Type 1: JDBC−ODBC bridges	Can access local databases such as Microsoft Access and FoxPro.	You must set up and maintain ODBC data sources.
	No enterprise−level database required.	Slow. The extra ODBC layer necessitates additional processing.
	Useful for testing basic JDBC features on stand alone Windows computers.	May require additional client software such as database network connectivity components.
		Not usable when deployment requires automatic downloading and configuration of applications.
Type 2: Native API	Faster than Type 1 because the ODBC layer is removed.	A vendor−specific API must be installed on client computer.
	Native code optimized for your platform and DBMS	Not usable when deployment requires automatic downloading and configuration of applications.
Type 3: JDBC−Network	No additional client software required.	Middleware product or application server required.
	Application server may give you access to multiple DBMSs.	May require additional configuration for Internet use.
Type 4: 100% Java		

	No additional client software required. Direct communication with database server.	May require additional configuration for Internet use.

Which JDBC driver type to use?

Before you can choose the correct driver type, you must know how your solution's intended deployment will influence your choice of driver types. The primary question is: Do I have control of the computer that will need database connectivity? The computer may be the client in some cases or an application server in others.

If you answer no, then you are limited to using Type 3 or 4 drivers. This is not necessarily bad. These drivers enable you to focus on the implementation without considering the client configuration because they do not require you to install additional components like ODBC drivers and DBMS software.

Note If your application must access a database across the Internet, using any driver type becomes challenging. Database servers do not communicate with their clients using standard Internet protocols, such as HTTP and FTP. The vendor's proprietary protocols can create security and firewall issues. If you need to provide access to your database via the Internet, you are better served by embedding JDBC logic into JavaServer Page (JSP) technology and servlets to provide database access. These topics are covered in Chapter 13, "Building Data−centric Web Applications."

JDBC driver vendors

Most DBMS vendors provide JDBC APIs for their product. Oracle, for example, has JDBC drivers available for each version of its database. Microsoft, however, does not supply a commercially available JDBC driver for SQL Server. You must purchase a third−party vendor's package. In addition, several open−source JDBC drivers are available for open−source databases such as mSQL and PostgreSql.

Note Sun provides a JDBC−ODBC bridge that enables you to connect to ODBC data sources. However, the bridge is currently not production−quality and should only be used for testing purposes. You will find the bridge in the sun.java.odbc package. To use Sun's JDBC−ODBC bridge, include the following import statement in your code:

```
import sun.jdbc.odbc.JdbcOdbcDriver.
```

Refer to Sun's JavaSoft Web site (http://industry.java.sun.com/products/jdbc/drivers) for information about driver vendors for your database platform. The site contains plenty of details about the type and feature sets of most JDBC drivers. It also has links to vendor Web sites so you can find more information, such as price and availability.

Using your JDBC driver

This section explains how to use JDBC drivers in your applications. Before getting started, you need to determine the exact class name for the driver from your vendor. Typically, vendors follow the Java package−naming convention when naming their drivers. For example, Oracle's JDBC driver is: oracle.jdbc.driver. OracleDriver.

Driver and DriverManager

The java.sql.Driver interface and java.sql.DriverManager class provide the tools for working with JDBC drivers. Figure 4−7 shows a UML class diagram illustrating the relationship between the DriverManager class and the Driver interface.

XRef Appendix D, “UML Class Diagram Quick Reference,” provides information on how to interpret UML class diagrams.

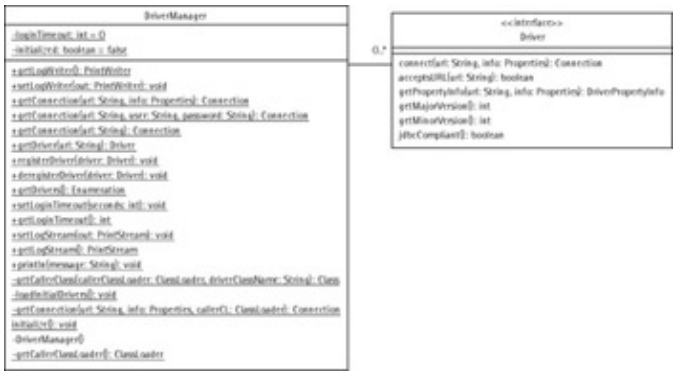

Figure 4−7: UML class diagram of Driver and DriverManager

All JDBC−compliant drivers must implement the Driver interface. In the real world, however, you may find that working directly with a Driver object is not very useful. It does provide a method for obtaining connections, but DriverManager provides more flexible connection alternatives.

In addition, a DriverManager object manages JDBC drivers for you. With this class you can explicitly register, select, or remove any JDBC−compliant driver from your application. This gives you the flexibility to store numerous drivers programmatically and choose the one you need at runtime. Factory design patterns benefit from this feature.

Registering JDBC drivers

To use your JDBC driver you must first register it with the DriverManager object, which, as you might expect, has a driver−registration method. However, two alternate techniques are available for registering JDBC drivers. The following is a list of the three different ways to register a JDBC driver:

- Class.forName(String *driverName*).newInstance()
- DriverManager.registerDriver(Driver *driverName*)
- jdbc.drivers property

Using Class.forName(String *driverName*).newInstance() is the most common way to register a JDBC driver. Besides instantiating a new JDBC Driver object, it also allows the object to register itself with DriverManager. Well how can this happen when DriverManager is not involved? The JDBC specification requires Driver objects to register themselves with DriverManager via a static initializer that calls the DriverManager.registerDriver() method. This mechanism allows for dynamic driver registration at runtime regardless of how you register your driver. The following code snippet provides a template for using Class.forName(). newInstance():

```
String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
try {
```

```
   Class.forName(driver).newInstance();

} catch(ClassNotFoundException e) {

   //Thrown if the driver class
   //is not found in classpath.
   e.printStackTrace();

}
```

The Class.ForName() method provides a lot of flexibility because it takes a String object as a parameter. You can define the parameter at runtime using a variety of methods. One technique is to supply the driver name via the command line when the application starts. Another is to read it from a properties file. Either method enables you to change the driver at startup.

Tip To ensure that your Driver is properly registered with DriverManager, always use the Class.forName().newInstance() method. This ensures that the Driver object is instantiated and the required static initializer is called.

As you already know, DriverManager provides the DriverManager. registerDriver(Driver *driverName*) method for registering JDBC drivers. The following code sample shows you how straightforward this method is; notice that you create a new instance of a Driver object and pass it as a parameter:

```
try {

  DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver());

}
catch(SQLException e) {

  e.printStackTrace();

}
```

Tip The DriverManager.registerDriver() method limits your flexibility during runtime because it requires a Driver object as a parameter. Class.forName().newInstance() accepts a String value that you can obtain from a command−line parameter or a properties file.

Setting the jdbc.drivers.property system property works great for testing. You can use the −D option on startup to specify a valid driver when starting the application and it will be automatically registered with DriverManager. This technique may prove useful in production as well. For example, in UNIX operating systems you can control database access by setting environment variables to specific drivers based on a user's login. This enables you to point individual users to specific databases. You may also set this value in a system property file on a client. The downside of this technique is that you must control your client's environment. Here is an example of using the jdbc.drivers.property at startup:

```
D:>java −cp −Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver MyJDBCProg
```

Note Remember to place the jar file containing your driver in your CLASSPATH. Neglecting to do this will cause compilation and execution errors. See Chapter 3, "Setting Up Your First JDBC Query," for more details.

Selecting and de−registering JDBC drivers

During the course of your application you may need to select or remove specific drivers from the DriverManager list. These methods are useful if, for example, you implement a Factory design pattern to support multiple databases.

Within your Factory class you could use DriverManager to pre−register the JDBC drivers you need for connecting to the supported databases. When a request for a Connection object occurs, your factory selects the proper driver from DriverManager and then creates the Connection object. In this manner you can support multiple databases and provide the correct Connection object to the client based on its request.

XRef Part III, “Using Java Data Access Design Patterns,” covers the Factory pattern and other standard design patterns that you may apply to JDBC.

The two methods for selecting drivers are DriverManager.getDriver() and DriverManager.getDrivers(). The first returns a specific Driver object when you pass it the JDBC URL identifying your database. (I cover JDBC URLs in the next section.) The second returns an Enumeration object so that you can iterate through a list of Driver objects until you find the one you want. The following snippet provides an example of how to use the DriverManager.getDrivers() method:

```
//Assume valid JDBC drivers have been registered

//Enumeration object needed to obtain registered drivers
Enumeration driverEnum = DriverManager.getDrivers();

//List the drivers.
while(driverEnum.hasMoreElements()) {

  //Cast to a Driver object

  Driver driver = (Driver)driverEnum.nextElement();

  String str = "Driver name is: " + driver.getClass().getName();
  System.out.println(str);

}
```

Although I just list the driver’s name in the snippet, you can use the Driver object retrieved from the Enumeration object to connect to a database or use the driver with DriverManager.deregisterDriver() to explicitly remove it from the DriverManager object’s internal list.

XRef I cover JDBC URLs in the next section, “Opening connections.” In brief, a JDBC URL provides instructions to the JDBC driver on how to connect to your database.

If for any reason you need to remove support for a database you can call the DriverManager.deregisterDriver() method to remove a registered Driver from DriverManager. The method is similar to the registerDriver() method in that it takes a Driver object as a parameter.

Working with Connection Objects

In JDBC, an instantiated Connection object is a physical connection to the database. The Driver.connect() method does the work of supplying you with a Connection object. You can use the Driver object directly, but DriverManager wraps the call with its getConnection() method.

For a number of reasons, using DriverManager is the preferred way to open database connections. To name one, if you have registered several drivers DriverManager will determine the appropriate driver to connect with. In addition, the getConnection() method is overloaded to provide you with a variety of ways to open connections.

The previous UML class diagram in Figure 4−7 shows that each DriverManager. getConnection() method has the common String parameter, url. This value is called a JDBC URL and has special meaning.

Understanding JDBC URLs

JDBC requires a naming system so you can connect to your database. The JBDC URL provides this system. Here is the general structure:

`jdbc:<subprotocol>:<subname>`

When creating a JDBC URL you need to supply values for the <*subprotocol*> and the <*subname*> placeholders. The value of <*subprotocol*> indicates which vendor−specific protocol to use when connecting to the database. Some DBMS vendors use multiple proprietary protocols to communicate with the database server.

The <*subname*> value indicates the data source, or database, you want to connect with. Some servers may hold more than one database and use logical names to represent each one. In general, the <*subname*> value is the logical name of the database on your database server.

Tip The exact <*subprotocol*> and <*subname*> values for your JDBC URL depends on your driver. Drivers from the same vendor may have different subprotocols. There is no standard format for either parameter. Consult your documentation for the correct format.

The following two examples may help you understand the JDBC URL format better:

- *JDBC−ODBC example*

  String url = "jdbc:odbc:MyDB";

  In this example, the <*subprotocol*> value is odbc and you want to connect to the ODBC DSN called MyDB, which is the value provided for <*subname*>. Remember that when using ODBC you are responsible for properly configuring the DSN on the client's computer before using your application.
- *Oracle example*

  String url = "jdbc:oracle:thin:@dbServerName:1521:ORCL";

  In this example, the <*subprotocol*> value is oracle:thin. The oracle portion is standard for Oracle's driver. The next part, thin, refers to the Oracle−specific connection mechanism to use. Some vendors may encapsulate many different network protocols for connecting to their databases in the driver. This

is what Oracle does with its Type 4 driver. Finally, the <*subname*> value, @dbServerName:1521:ORCL, tells the Oracle driver which host, port, and database instance to connect to.

Note The value for <*subprotocol*> is unique. Driver vendors must register their <*subprotocol*> names with Sun Microsystems, which acts as an informal registrar.

Opening connections

Now, let me return to the DriverManager.getConnection() methods. For easy reference, let me list the three overloaded DriverManager.getConnection() methods:

- getConnection(String *url*)
- getConnection(String *url*, Properties *prop*)
- getConnection(String *url*, String *user*, String *password*)

The first method requires one parameter, a JDBC URL, to make a connection. Notice that the security information is lacking in this method. Chances are that if you use this method your database does not directly provide user−authentication services. However, some databases will assume that if you can log onto the client you should have the rights to use the system.

The next method takes an additional parameter, a Properties object, besides the String url. Use this method if you need to pass specific information to your database when connecting. Just set the appropriate name−value pairs in the object and supply it as a parameter when you call the method.

The last method, the most common one, is fairly straightforward. It requires three Strings, the url, user, and password.

When you call the getConnection() method, DriverManager returns a valid Connection object. Behind the scenes, DriverManager passes your parameters to each registered Driver object in its list. DriverManager begins with the first JDBC driver and tries to make a connection; if it fails, DriverManager tries the next one. It repeats this process until it finds a driver that can connect to the database described by your JDBC−URL.

Using the getConnection() method in your code is also straightforward. Here is the basic format for using DriverManager to open an Oracle database connection:

```
String url = "jdbc:oracle:thin:@myServer:1521:PROD";
String user = "boss";
String pwd = "bosspwd";
Connection conn = DriverManager.getConnection(url,user,pwd);
```

If the connection fails, DriverManager throws an SQLException containing database−specific error messages. As a result, it is helpful to know something about the semantics involved in connecting with your database so you can interpret the error messages.

You may wonder what happens if you register two drivers that can connect to your database. The answer is simple. DriverManager chooses the first driver that makes a successful connection.

Note As J2EE gains more popularity in the enterprise, the DataSource interface is becoming the preferred way to open database connections. Usually a DataSource interface works in conjunction with the Java Naming and Directory Interface (JNDI) to provide the connection. DataSource objects do not require you to supply the driver name to make the connection. It also enables connection pooling and distributed

transactions.

XRef I discuss the DataSource interface in Chapter 14, "Using Data Sources and Connection Pooling," and Chapter 19, "Accessing Data with Enterprise JavaBeans."

You can override this behavior by retrieving the particular driver you want from the DriverManager object's list. (The previous section shows you how to do this.) Once you get the desired driver, just call its connect() method to obtain a connection.

Similarly, you can instantiate a Driver directly and use it to connect. The following code snippet shows you how to make the connection using a newly created Driver instance:

```
Driver drv = new sun.jdbc.odbc.JdbcOdbcDriver();
Properties prop = new Properties();
Connection conn = drv.connect("jdbc:odbc:authors",prop);
```

The prop variable is used to hold a set of parameters, such as user and password, that you need in order to connect to your database. I leave it empty in this case.

To demonstrate, Listing 4−1 shows an application that registers a JDBC−ODBC driver and obtains a connection to a local DSN using each of the three getConnection() methods.

Listing 4−1: JdbcConnect.java

```
package Chapter4;

import java.sql.*;
import java.util.Properties;

public class JdbcConnect {

  public static void main(String[] args) {
    //Define 3 Connection variables
    Connection conn1 = null;
    Connection conn2 = null;
    Connection conn3 = null;

    //Begin standard error handling
    try {

      //Load a driver with Class.forName.
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

      //Define JDBC URL, user, and password
      String jdbcUrl = "jdbc:odbc:authors";

      String user = "toddthomas";
      String pwd = "mypwd";

      System.out.println("Opening database connections...");

      //Use first getConnection method using only a JDBC URL.
      conn1 = DriverManager.getConnection(jdbcUrl);

      //Test to see if connection succeeded.
      if (conn1!=null) {
        System.out.println("Connection 1 successful!");
```

```
  }

  //Use the second getConnection method. This requires
  //a Properties object to store specific information.
  Properties prop = new Properties();
  prop.put("user",user);
  prop.put("password",pwd);

  conn2 = DriverManager.getConnection(jdbcUrl,prop);

  //Test to see if connection succeeded
  if (conn2!=null) {
    System.out.println("Connection 2 successful!");
  }

  //Use third getConnection method that requires three
  //String parameters; JDBC URL, user, and password.
  conn3 = DriverManager.getConnection(jdbcUrl,user,pwd);

  //Test to see if connection succeeded.
  if (conn3!=null) {
     System.out.println("Connection 3 successful!");
  }

  //Explicitly close all connections. ALWAYS DO THIS!
  conn1.close();
  conn2.close();
  conn3.close();

  //Test to see if connections are closed
  System.out.println("Closing all connections...");
  if (conn1.isClosed()) {
     System.out.println("Connection 1 is closed");
  }

 if (conn2.isClosed()) {
     System.out.println("Connection 2 is closed");
  }
  if (conn3.isClosed()) {
     System.out.println("Connection 3 is closed");
  }

} catch(SQLException se) {
  //Handle errors for DriverManager
  se.printStackTrace();
} catch(Exception e) {
  //Handle errors for Class.forName and all other exceptions
  e.printStackTrace();
} finally {
  //finally block
  try {
   if (conn1!=null)
     conn1.close();
   if (conn2!=null)
     conn2.close();
   if (conn3!=null)
     conn3.close();
  }
  catch(SQLException se) {
     se.printStackTrace();
  } //end finally try
```

```
    } //end try
  } //end main

} //end JdbcConnect class
The output from Listing 4−1 follows:
Opening database connections...
Connection 1 successful!
Connection 2 successful!
Connection 3 successful!
Closing all connections...
Connection 1 is closed
Connection 2 is closed
Connection 3 is closed
```

Closing JDBC connections

In Listing 4−1 I explicitly close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on garbage collection, especially in database programming, is very poor programming practice. You should make a habit of always closing the connection with the close() method for a couple of reasons.

> Note The purpose of the Connection.isClosed() method is misleading and does not provide you with the information you might expect. The method does not test for a valid open connection; it is only guaranteed to return true if you close the connection with the close() method. You would be better off trying a JDBC operation and catching the exception to determine if the connection is actually valid.

First, it will ensure that your client session is properly closed on the database server. Some databases behave erratically when a user session ends improperly. Second, the DBMS may assume that the user session crashed and roll back any changes you made during your program. For example, Oracle databases will roll back uncommitted statements if a user's session ends in the middle of a transaction. Explicitly closing the connection ensures that the database cleans up the server−side client environment the way you expect it to.

Secondly, explicitly closing a connection conserves DBMS resources, which will make your database administrator happy. For example, if your database licensing is based on open sessions, failing to close a session can keep others from using the system. Also, each open connection requires some amount of RAM and CPU cycle time. Unclosed connections unnecessarily consume database−server and client resources. When you close connections properly you free these resources for other use.

To ensure that a connection is closed, you could provide a finally block in your code. A finally block always executes, regardless if an exception occurs or not, so it will ensure that the database resources are reclaimed by closing the connection. Listing 4−2 demonstrates how to use the finally block to close connections.

Listing 4−2: Finally.java

```
package Chapter4;

import java.sql.*;
import java.util.Properties;

public class Finally {
```

```
  public static void main(String[] args) {
    //Define Connection variable
    Connection conn = null;

    //Begin standard error handling
    try {

      //Load a driver with Class.forName.
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

      System.out.println("Opening database connection...");
      conn = DriverManager.getConnection("jdbc:odbc:authors");

    //Handle errors for JDBC
    } catch (SQLException se) {
      se.printStackTrace();

    //Handle other exceptions
    }catch(Exception e) {
      e.printStackTrace();

    //finally block used to close connection
    } finally {
      try {
        if (conn!=null) {
          String msg = "Closing connection from finally block.";
          System.out.println(msg);
          conn.close();
        }
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try

  }//end main
}//end Finally
```

The output from Listing 4−2 is as follows:

```
Opening database connection...
Closing connection from finally block.
```

Summary

Registering JDBC drivers and opening database connections are the foundations of Java database programming. The most important items to remember from this chapter are:

- Use the right JDBC driver type for your application.
- Use the Class.forName().newInstance() method to load your JDBC drivers for the most runtime flexibility.
- Use the DriverManager.getConnection() method to obtain a database connection.
- Use the Connection.close() method to explicitly close your database session

Chapter 5: Building JDBC Statements

In This Chapter

- Understanding the JDBC statement family
- Working with Statement objects
- Creating prepared and parameterized SQL statements using PreparedStatement objects
- Accessing database stored procedures using CallableStatement objects

In Chapter 4, “Connecting to Databases with JDBC,” I illustrated how to connect with your database using DriverManager and the Connection object. Now that you can connect with your database you will likely want to interact with it.

How you interact with the database will vary. You may need to submit an SQL query to retrieve data from tables or build a database schema. Sometimes you may not know the values for several database fields used in a query and must collect them at runtime. In this case you will need to create parameterized SQL statements or database commands to collect unknown values. At other times you may need to use stored procedures in your database.

Whatever your task, the Statement, PreparedStatement, and CallableStatement objects provide the tools to accomplish it. This chapter covers the details of working with these objects. In particular, I cover how to create the objects, and submit commands using them. In addition I provide several examples illustrating their use.

Using JDBC Statements

The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database. This is not a trivial task.

As an example of the data type differences consider the Java int primitive data type, which cannot represent a NULL. Yet databases use NULL values extensively to represent empty data, even for numerical data. Another example is date and time types. Java’s representation of these data types are completely different from their counterparts defined by the SQL−92 standard. Nonetheless, the statement interfaces define methods that enable you to convert the data types from Java to JDBC correctly.

When you reach into the JDBC toolbox for one of the statement objects, you need to know which one to use for the job. Sometimes you may need to retrieve data and present it to the user; at other times you may need to update database information, create new tables, or even execute stored procedures in the database.

Although they are used for different tasks, the three statement interfaces have a lot of similarities. Examine Figure 5−1, which shows a UML class diagram of the statement interface family. Notice that Statement is the parent, that PreparedStatement extends Statement, and that CallableStatement extends PreparedStatement. Driver vendors provide classes that implement these interfaces. Without a JDBC driver you cannot create objects based on these interfaces.

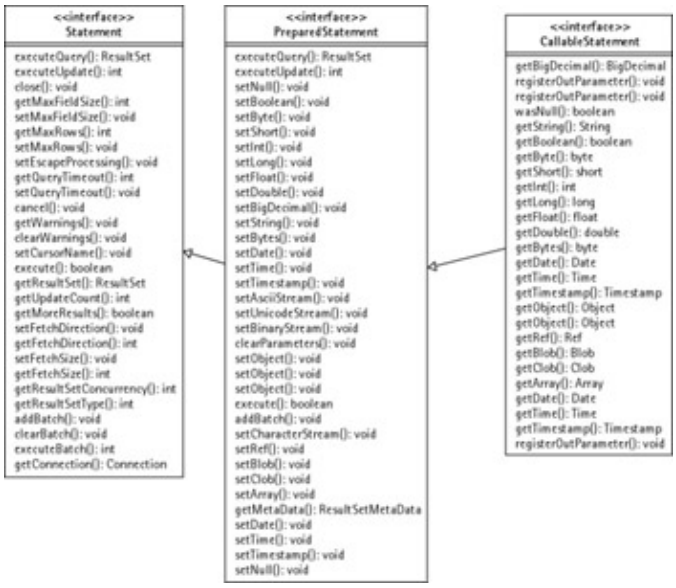

Figure 5−1: UML diagram of Statement, PreparedStatement, and CallableStatement interfaces

So how do you decide which interface to use? Table 5−1 provides a summary of each interface's purpose. Notice that both the PreparedStatement and CallableStatement can use SQL statements that can accept input parameters at runtime.

What Is a ResultSet Object?

ResultSet objects hold the data returned from SQL queries produced with the executeQuery() method. In some instances, the execute() method also returns one. ResultSet objects vary in functionality. Some only let you view the data, while others enable dynamic updating.

Table 5−1: Summary of Statement, PreparedStatement, and CallableStatement Interfaces

Interfaces	Recommended Use
Statement	Use for general−purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use when you plan to use the SQL statements many times. Quicker than the generic Statement object because the SQL statement is precompiled. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Introducing Statement Objects

This section focuses on the Statement object, the foundation of the statement interface hierarchy. A firm understanding of this object will help you understand and use the PreparedStatement and CallableStatement objects.

The Statement object provides you with basic database−interaction capabilities. However, it still gives you significant capabilities. It enables you to use all types of DML, DDL and other database specific commands. In addition, it supports batch updating, which enables you to implement transaction management.

Creating the Statement object

Instantiating a Statement object is a straightforward procedure. You create the object from a valid Connection object, as shown in the following example:

```
Connection conn = DriverManager.getConnection(url, "toddt", "mypwd");
Statement stmt = conn.createStatement();
```

Using the Statement object

A Statement object provides three methods — execute(), executeUpdate(), and executeQuery() that act as conduits to your database for sending database commands and retrieving results. Table 5−2 highlights the differences between methods.

Table 5−2: The execute(), executeQuery(), and executeUpdate() Methods

Method	Recommended Use
executeQuery	Use to query the database with SELECT statements. This method returns a result set.
executeUpdate	Use to submit INSERT, UPDATE, DELETE, or DDL SQL statements. The method returns a count of the rows affected by the statement for INSERT, UPDATE, or DELETE, or 0 for statements that return nothing such as DDL statements.
execute	Use to process any DDL, DML, or database specific command. This method can return one or more ResultSet objects or update counts depending on the statement type. In some instances it can even return a combination of both. Although this method provides the most flexibility, you may find processing the results more difficult.

The executeQuery() method, if successful, always returns a ResultSet object. Chapter 6, "Working with Result Sets," describes this object in greater detail. For now, I want to cover the executeUpdate() and execute() methods. These methods enable you to send action statements to the database, such as INSERT, UPDATE, or DELETE and DDL statements. Most of the concepts I cover apply to the executeQuery() method as well.

Working with the executeUpdate() method

Now for an example that uses the executeUpdate() method to build a small Oracle database schema that holds employee information. The database is simple. It has two tables, Employees and Location. The Employees table stores data such as Social Security Number (SSN), name, salary, hire date, and Loc_Id. The foreign key, Loc_Id, links to the Location table containing the employees' field−office locations.

Figure 5−2 provides the entity relationship diagram (ERD) for the sample database. Although small, the database contains most of the data types you are likely to encounter in the real world: DATE, NUMERIC, and

VARCHAR. However, it does not contain any advanced SQL3 data types, such as BLOB or CLOB. Nonetheless, the database provides a good starting point for demonstrating the executeUpdate() method's functionality and how to interact with different data types.

XRef For more details on data−type issues, see Chapter 7, "Understanding JDBC Data Types."

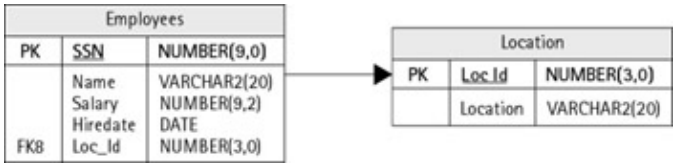

Figure 5−2: Employee database ERD

The executeUpdate() method takes a String parameter representing the SQL statement you wish to execute. You will receive the count of affected rows when the method returns. The following code snippet provides a quick demonstration:

```
String SQL = "UPDATE employee SET salary = 100000 WHERE name = 'toddt'";
Statement stmt = conn.createStatement();
int count = stmt.executeQuery(SQL);
```

If successful, the value of count should equal 1, the number of rows affected by the UPDATE statement.

Tip Checking the update count value can be a sanity check when you're executing your SQL statements. If you update a row by its primary key then you should always receive an update count of 1. Any other value may indicate an error. If you issue a broad UPDATE statement, such as applying a cost−of−living adjustment to all employees' salaries, the update count may be any value. When you issue an INSERT statement the executeUpdate() method should return a value of 1. However, DDL statements such as CREATE TABLE and CREATE INDEX always return a value of 0. Testing for the expected value can help you determine whether your statement executed as expected.

Listing 5−1 provides the code needed to create the Employee database schema. Remember, this example builds a schema in an Oracle database. The syntax used to build tables and data types is Oracle−specific. The program builds the database from scratch every time it executes. You generally use an application like Listing 5−1 as a backup if you lost your original database, writing a generic installation routine, or if you need a clean slate for the purpose of testing JDBC features.

Listing 5−1: MakeEmpDb.java

```
package Chapter5;

import java.sql.*;

public class MakeEmpDb {

  //Global Statement object
  static Statement stmt = null;

  public static void main(String[] args) {
    //Standard Connection object
    Connection conn = null;

    //Begin standard error handling
```

```
  try {
    //Register driver
    String driver = "oracle.jdbc.driver.OracleDriver";
    Class.forName(driver).newInstance();

    //Open database connection
    System.out.println("Connecting to database...");
    String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
    conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

    //Create a Statement object
    stmt = conn.createStatement();

    //Create tables and load data
    createTables();
    insertData();

    //Provide success message
    System.out.println("Employee DB created successfully.");

  //Standard error handling
  } catch(SQLException se) {
    //Handle errors for JDBC
    se.printStackTrace();

  } catch(Exception e) {
    //Handle errors for Class.forName
    e.printStackTrace();

  } finally {
    try {
      if(conn!=null)
        conn.close();
    } catch(SQLException se) {
      se.printStackTrace();
    }//end finally try
  }//end try

 System.out.println("Goodbye!");

}//end main

public static void createTables() throws SQLException {

  //Create SQL statements
  String locationSql = "CREATE TABLE Location "
       + "(Loc_Id number(3) CONSTRAINT PK_LOC PRIMARY KEY,"
       + "Location VARCHAR(20))";

  String employeeSql =  "CREATE TABLE Employees "
       + "(SSN number(9) CONSTRAINT PK_EMP PRIMARY KEY,"
       + "Name VARCHAR(20), Salary number(9,2), Hiredate DATE,"
       + "Loc_Id NUMBER(3) CONSTRAINT fk_Loc "
       + "REFERENCES Location(Loc_Id))";;

  try {

    String dropSql = "DROP TABLE Location CASCADE CONSTRAINTS";
    stmt.executeUpdate(dropSql);

  } catch(SQLException se) {
```

```
    //Ignore Oracle DROP table error.
    if(se.getErrorCode()==942) {
      String msg = se.getMessage();
      System.out.println("Error dropping Employees table: " +  msg);
    }

  }

  //Build the Location table
  if(stmt.executeUpdate(locationSql)==0)
    System.out.println("Location table created...");

  try {
    String dropSql = "DROP TABLE Employees CASCADE CONSTRAINTS";
    stmt.executeUpdate(dropSql);

  } catch(SQLException se) {
    //Ignore Oracle DROP table error.
    if(se.getErrorCode()==942) {
      String msg = se.getMessage();
      System.out.println("Error dropping Employees table: " +  msg);

    }
  }

  //Build the Employees table
  if (stmt.executeUpdate(employeeSql)==0)
    System.out.println("Employees table created...");

} // end of createTables method

public static void insertData()throws SQLException {

  //Load Location table with data
  stmt.executeUpdate("INSERT INTO "
       + "Location VALUES(100,'Knoxville')");

  stmt.executeUpdate("INSERT INTO "
       + "Location VALUES(200,'Atlanta')");

  stmt.executeUpdate("INSERT INTO "
       + "Location VALUES(300,'New York')");

  stmt.executeUpdate("INSERT INTO "
       + "Location VALUES(400,'L.A.')");

  stmt.executeUpdate("INSERT INTO "
       + "Location VALUES(500,'Tulsa')");

  //Load Employee table with data
  stmt.executeUpdate("INSERT INTO Employees VALUES(111111111,"
       +"'Todd','5000',{d '1995-09-16'},100)");
  stmt.executeUpdate("INSERT INTO Employees VALUES(419876541,"
       +"'Larry','1500',{d '2001-03-05'},200)");

  stmt.executeUpdate("INSERT INTO Employees VALUES(312654987,"
       +"'Lori','2000.95',{d '1999-01-11'},300)");

  stmt.executeUpdate("INSERT INTO Employees VALUES(123456789,"
       +"'Jimmy','3080',{d '1997-09-07'},400)");
```

```
    stmt.executeUpdate("INSERT INTO Employees VALUES(987654321,"
         +"'John','4351',{d '1996-12-31'},500)");

    //Give feedback
    System.out.println("Finished inserting data...");

  }//end insertData method

}// end MakeEmpDb class
```

Output from Listing 5−1:

```
Connecting to database...
Location table created...
Employees table created...
Finished inserting data...
Employee DB created successfully.
Goodbye!
```

Note Typically you would not build database schemas using JDBC. The previous example only serves to illustrate the flexibility of the executeUpdate() method. You need to consider JDBC−to−Java data−type mappings if you want to write applications that create database schemas.

I begin the application with the standard JDBC initialization routines:

1. Register a JDBC driver with Class.forName().newInstance().
2. Open a database connection with DriverManager.getConnection().
3. Create a Statement object for submitting the DDL and INSERT statements using the Connection.createStatement() method.

After the driver registration I use two methods to build and populate the schema. The first method, createTables(), creates the table structures. Within the routine, I start by issuing a DROP statement so I can do a fresh build. Notice that I trap the SQLException "ORA−942 Table does not exist." The initial execution of the DROP statement throws this exception because the tables do not exist, therefore I ignore it by issuing a simple notification to the console. I pass any other exceptions back to the main() method and handle them there. Next I submit the CREATE TABLE statements to build the tables and the entity relationship.

Caution JDBC is not completely database agnostic. You need specific knowledge of the target database's syntax and data types before executing most DDL operations. Listing 5−1 demonstrates this fact because the DDL is Oracle−specific. However, the JDBC API methods will work with any database.

Once the createTables() method returns successfully, I insert data into the tables using the loadData() method. Notice that I populate the Location lookup table before inserting the employee data to ensure that no referential−integrity constraint violations occur.

One additional item to note is the format of the INSERT statement I use. Here is a snippet from Listing 5−1:

```
String SQL = "INSERT INTO Employees VALUES" +
             "(145985369,'Todd','5000',{d '1995-09-16'},100)";
stmt.executeUpdate(SQL);
```

I use the JDBC SQL escape sequence to set the value of parameter four, Hiredate. This format tells the driver to convert the date into Oracle's date format (*DD−MON−YY*) before sending the statement. This is one example of how JDBC helps abstract from you database data type and formatting issues.

JDBC SQL escape syntax

Most database languages provide you with a “programming” language so you can interact with the database in ways you cannot achieve just using standard DML or DDL SQL statements. The languages also generally provide you with internal helper functions that you can use to format character and numeric data as well as mathematical functions that help you perform useful calculations.

However, all databases have unique syntaxes for their programming languages. For instance, Oracle uses PL/SQL and Microsoft SQL Server uses Transact−SQL. Because of the uniqueness of each database’s programming language, JDBC provides you with access to the functions and their special features using the JDBC SQL escape syntax. When you specify a command using the syntax, the driver translates the command into the necessary database−specific format.

The escape syntax gives you the flexibility to use database specific features unavailable to you by using standard JDBC methods and properties. However, use escape clauses with caution. Overusing them can make your code database−dependent because you are using specific functions unique to your database.

The general SQL escape syntax format is as follows:

```
{keyword parameters}
```

Table 5−3 lists and describes the escape keywords.

Table 5−3: SQL Escape Keywords

Keyword	Description	Example
d, t, ts	Helps identify date, time, and timestamp literals. As you know, no two DBMSs represent time and date the same way. This escape syntax tells the driver to render the date or time in the target database’s format.	{d ‘*yyyy−mm−dd*’} where *yyyy* = year, *mm* = month; *dd* = date (for example, {d *‘2002−08−03’*} is March 8, 2002 {t ‘*hh:mm:ss*’} where *hh* = hour; *mm* = minute; *ss* = second (for example, {t *‘13:30:29’*} is 1:30:29 PM) {ts ‘*d t.f...*’} where *d* = date format; *t*=time format; *f* = optional fractional second
fn	Represents scalar functions used in a DBMS.	{fn length(*‘Hello World’*)} returns 11, the length of the character string ‘Hello World’.
escape	Identifies the escape character used in LIKE clauses. Useful when using the SQL wildcard %, which matches zero or more characters.	String sql = "SELECT symbol FROM MathSymbols WHERE symbol LIKE ‘\%’ {escape ‘\’}"; stmt.execute(*sql*);
call	Use for stored procedures.	

		For a stored procedure requiring an IN parameter, use {call *my_procedure*(?)} For a stored procedure requiring an IN parameter and returning an OUT parameter use {? = call *my_procedure*(?)};
oj	Use to signify outer joins. The syntax is as follows: {oj outer− join} where outer− join = table {LEFT| RIGHT|FULL} OUTER JOIN {table | outer−join} on search−condition.	String sql = "SELECT emp from {oj ThisTable RIGHT OUTER JOIN ThatTable on empid = '111111111'}"; stmt.execute(*sql*);

Using the execute() method

The execute() method provides the most flexible way to interact with a database because it enables you to process any type of SQL statement. For example, you may issue statements that return update counts or result sets. The executeUpdate() and executeQuery() methods can only return update counts or result sets, respectively. The execute() method can return both.

However, the execute() method's flexibility bears a price tag. Because you may know nothing about the statement type passed to the database, you may also know nothing about the result the database will return. You might receive one or more ResultSet objects, one or more update counts, or one or more of both.

Figure 5−3 is a flow chart that demonstrates how to interpret and process this command's return value.

To begin, the execute() method always returns a boolean. If it returns true, a ResultSet object was returned. At this point you call the Statement object's getResultSet() method to obtain the ResultSet object populated with data satisfying the SQL query. Once you finish processing that result set, call the Statement object's getMoreResults() method to determine if another result set exists. If the method returns true, call the getResultSet() and process that result set. Continue this loop until the getMoreResults() method returns false.

Now you must check for update counts using the getUpdateCount() method. A value of >=0 indicates that an update count exists. As I mentioned earlier, a 0 denotes an SQL DDL and anything else represents the update count of the number of rows affected by an INSERT, DELETE, or UPDATE statement or stored procedure. Continue processing update counts until the getUpdateCount() method returns −1. At this point you have processed all the results from the execute() method.

As you can see, the execute() method can be fairly complex to implement if you do not know what type of SQL statement you are processing. Fortunately, in the real world you usually know whether to expect a result set or an update count.

Listing 5−2 provides an example of processing the execute() method's return value. In the application, I submit an INSERT statement to demonstrate the case in which an update count is returned, and a SELECT statement to illustrate the case in which a result set is returned. After I execute each statement I call the method processExecute() to determine the return value and display the appropriate message.

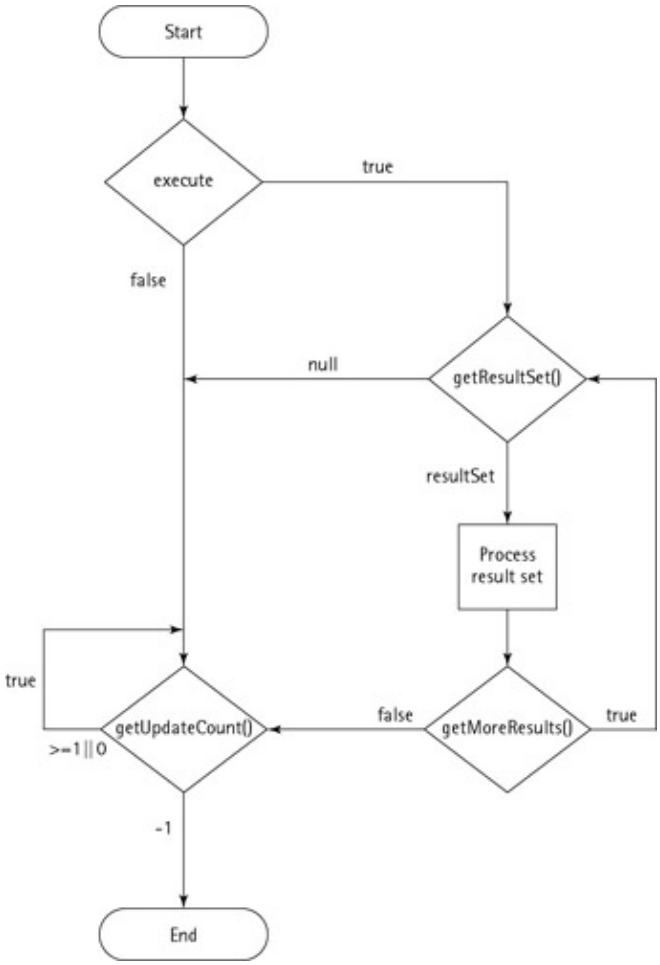

Figure 5−3: This flow chart shows how to process the results from the execute() method.

Listing 5−2: ExecuteMethod.java

```
package Chapter5;

import java.sql.*;
public class ExecuteMethod {

  public static void main(String[] args) {

    //Declare Connection, Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Holds the execute method's result
    boolean executeResult;

    //Begin standard error handling
    try{

      //Register driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection
      System.out.println("Connecting to database...");
```

```
    String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
    conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

    //Create a Statement object
    stmt = conn.createStatement();

    //Insert data and process result
    String sql="INSERT INTO Employees VALUES" +
        "(868653391,'Greg','4351',{d '1996-12-31'},500)";
    executeResult = stmt.execute(sql);
    processExecute(stmt,executeResult);

    //List employees
    sql = "SELECT * FROM Employees ORDER BY hiredate";
    executeResult = stmt.execute(sql);
    processExecute(stmt,executeResult);

  //Standard error handling.
  } catch(SQLException se) {
    //Handle errors for JDBC
    se.printStackTrace();

  } catch(Exception e) {
    //Handle errors for Class.forName
    e.printStackTrace();

  } finally {
    try {
      if(conn!=null)
        conn.close();
    } catch(SQLException se) {
        se.printStackTrace();
    }//end finally try
  }//end try

  System.out.println("Goodbye!");

}//end main

//Method to process the execute() statement
public static void processExecute(Statement stmt,
                  boolean executeResult) throws SQLException {

  //check executeResult to see what was returned
  if(!executeResult) {

    System.out.println("Update count returned...");
    int updateCount = stmt.getUpdateCount();
    System.out.println(updateCount + " row was " +
          "inserted into Employee table.");

  } else {

    //ResultSet returned
    ResultSet rs = stmt.getResultSet();
    System.out.println("SQL query issued...");

    //Table header information
    System.out.println("Listing rows for Employee table.");
    System.out.println("SSN" + "\t\t" + "Name" + "\t" + "Salary"
      + "\t" + "Hiredate" + "\t" + "Loc_id");
```

```
      //Loop through ResultSet showing all Employees
      while(rs.next()){

        System.out.println(rs.getInt("SSN") + "\t"
          + rs.getString("Name")+ "\t"
          + rs.getDouble("Salary") + "\t"
          + rs.getDate("Hiredate") + "\t"
          + rs.getInt("Loc_id"));

      }//end while-loop

    }//end if(!executeResult)

  }//end processExecute()
}//end ExecuteMethod Class
```

The output from Listing 5−2 is as follows:

```
Connecting to database...
Update count returned...
1 row was inserted into Employee table.
SQL query issued...
Listing rows for Employee table.
SSN        Name   Salary    Hiredate    Loc_id
111111111  Todd   5000.00   1995-09-16  100
987654321  John   4351.00   1996-12-31  500
868653391  Greg   4351.00   1996-12-31  500
123456789  Jimmy  3080.00   1997-09-07  400
312654987  Lori   2000.95   1999-01-11  300
419876541  Larry  1500.00   2001-03-05  200
Goodbye!
```

JDBC batch processing

The statement interface family supports batch processing that enables you to submit multiple DML statements with one call to the database. This can help you minimize the number of database calls you make and implement transactional control over your database.

For example, suppose you have an application that uses INSERT and UPDATE statements to refresh the data in a data warehouse using a text file as a source. Most data warehouse refresh files are large and you will likely process a large number of database calls that perform nearly identical tasks. With every call you are issuing either an INSERT statement to add data or an UPDATE statement to update existing data.

To minimize the number of calls, you can send a batch of statements with one call and execute them together. You can also inform the database to undo all the changes in the batch if one statement fails. This transactional approach will ensure data integrity and consistency by preventing “orphan” data from being written to the database.

JDBC 2.0 and beyond supports batch processing of INSERT and UPDATE statements, which may be useful in the scenarios I describe here. However, JDBC drivers are not required to support this feature. You should use the DatabaseMetaData. supportsBatchUpdates() method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.

XRef Chapter 8, “Mining Database Metadata with JDBC,” covers how to get and use database information with JDBC Metadata interfaces.

To take advantage of batch processing with a Statement object you must use the setAutoCommit(), addBatch(), and executeBatch() methods. The setAutoCommit() method controls when your database makes your changes permanent. I cover commits more thoroughly in the next section. With each call to the addBatch() method you add an INSERT or UPDATE SQL statement to a list for execution. When you’re finished adding all the statements, call the executeBatch() method to submit the batch to the database for execution.

The executeBatch() method returns an int[] containing the individual update counts for each SQL statement in the order in which you added them to the batch. If an error occurs while executing the batch, processing stops and a BatchUpdateError exception occurs. At this point the number of elements in the int[] equals the number of successful statements executed within the batch.

To help visualize how batch updates work, Figure 5−4 shows a flow chart that illustrates the process. Notice that auto−commit is set to false, and pay attention to the flow of the addBatch() and executeBatch() methods and the explicit commit() call.

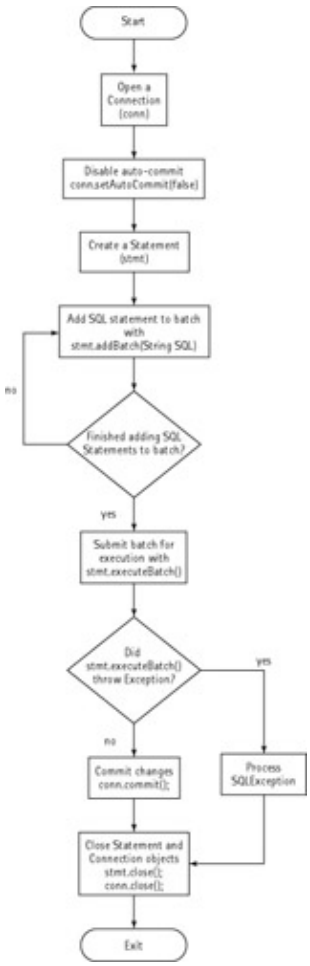

Figure 5−4: This flow chart shows JDBC batch processing.

The following code snippet provides an example of a batch update:

```
//Create a Statement object and add SQLstatements with the
```

```
//addBatch() method. Assume a valid connection.

Statement stmt = conn.createStatement();

//Set auto-commit to false
conn.setAutoCommit(false);

String SQL = "INSERT INTO Employees (Id, Name) VALUES(9517,'Jane')";
stmt.addBatch(SQL);

SQL = "INSERT INTO Employees (Id, Name) VALUES(9518,'Betty')";
stmt.addBatch(SQL);

//Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

In this example I use batch updating to add additional entries to my Employees table. Notice that the first thing I do is set auto-commit to false with a call to setAutoCommit(). Next I add two SQL INSERT statements to the batch. Then I call the executeBatch() method to execute the SQL statements. Finally, I call commit() to ensure that the changes are applied.

Note Turning off auto-commit may yield some performance increases because the number of commits is reduced. However, remember that any DML statement may force the database to lock the row, page, or even the table until you issue a commit. You may find this locking behavior undesirable, as it may prohibit other users from accessing information.

As a final comment, just as you can add statements to a batch for processing, you can remove them with the clearBatch() method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

JDBC transactions

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

For example, Figure 5-5 illustrates a banking transaction that transfers funds from a checking account to an investment account. If the investment-account credit operation fails, you need to undo the debit to the checking account. This is a simple example, but it illustrates the point. Transactions are a science unto themselves and beyond the scope of this book.

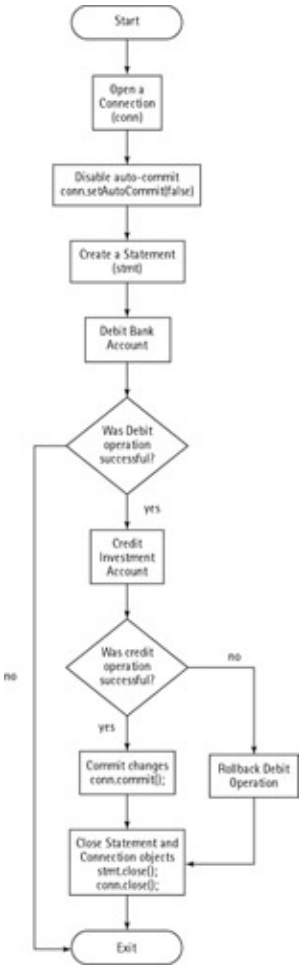

Figure 5−5: This flow chart illustrates a banking transaction.

Working with transactions has both pros and cons. For example, with transactions you can maintain both data consistency and integrity. While you make changes to a row, the DBMS prevents others from simultaneously changing the same row. This guarantees that when you execute your commit() method you actually change the data you expect to change, not data that was changed by someone else between the time you began the transaction and the time you issued the commit.

Caution Do not count on transaction control or batch update support for DDL statements. Most databases will not roll back these SQL statements once you submit them.

For the same reasons that transactions provide benefits, they can also cause problems. To prevent data from being manipulated while a transaction takes place, the database locks the data being updated. Some systems use row−level locks, which prevent others from changing the row you are currently working with. Others use page−level locks that prevent others from changing data located near yours. Some systems even lock entire tables. For obvious reasons this is undesirable.

JDBC enables you to manage transactions by manipulating the Connection object’s auto−commit mode and using its rollback() method, which undoes all changes, up to the last commit.

JDBC 3.0 Enhancement

The new JDBC 3.0 Savepoint interface gives you additional transactional control. Most modern DBMS support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints:

- setSavepoint(String *savepointName*) defines a new savepoint. It also returns a Savepoint object.
- releaseSavepoint(Savepoint *savepointName*) "deletes" a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

The following example illustrates the use of a Savepoint object:

```
try{

   //Assume a valid connection object conn
   conn.setAutocommit(false);
   Statement conn.createStatement();

   String SQL = "INSERT INTO Emp(Id, Name) VALUES (10, 'Ted')";
   stmt.executeInsert(SQL);

   //set a Savepoint
   Savepoint savepoint = conn.setSavepoint("Savepoint1");

   //Submit a malformed SQL statement that breaks
   String SQL = "TRESNI OTNI Emp(Id, Name) VALUES (10, 'Ted')";
   stmt.executeInsert(SQL);

}catch(SQLException se){

   conn.rollback(svpt1);

}
```

XRef Refer to Chapter 15, "Understanding Distributed Transactions" for more information on transactions.

Listing 5−3 demonstrates transaction management using the Connection object's auto−commit mode and rollback() method. In this example, I add a new employee and their field office location information. If the employee INSERT statement fails, which it does in my example, then the location data are removed with the rollback() method.

Listing 5−3: Rollback.java

```
package Chapter5;

import java.sql.*;

public class Rollback {

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

```
//Declare Connection and Statement objects
Connection conn = null;
Statement stmt = null;

//Holds the execute method's result
boolean executeResult;

//Begin standard error handling
try {

  //Register driver.
  String driver = "oracle.jdbc.driver.OracleDriver";
  Class.forName(driver).newInstance();

  //Open database connection.
  System.out.println("Connecting to database...");
  String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
  conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

  //Create a Statement object.
  stmt = conn.createStatement();

  //Set Autocommit = false and verify.
  conn.setAutoCommit(false);
  if (!conn.getAutoCommit())
    System.out.println("Auto-commit is set to false");

  //Insert location data.
  String sql = "INSERT INTO Location VALUES(715,'Houston')";
  stmt.executeUpdate(sql);

  //This statement will fail for invalid date.
  sql = "INSERT INTO Employees VALUES" +
      "(888653321,'Kirk','4351',{d '1996-02-31'},715)";
  stmt.executeUpdate(sql);

  //Commit data to database.
  conn.commit();

//Standard error handling.
} catch(SQLException se) {
  //Handle errors for JDBC
  String msg = se.getMessage();
  msg = "SQLException occured with message: " + msg;
  System.out.println(msg);

  //Rollback transaction
  System.out.println("Starting rollback operations...");
  try {
    conn.rollback();
  } catch(SQLException se2){
    se2.printStackTrace();
  }
  System.out.println("Rollback successfull!");

} catch(Exception e) {
  //Handle errors for Class.forName
  e.printStackTrace();

} finally {
  try {
```

```
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try

    System.out.println("Goodbye!");

  }//end main

}//end Rollback class
```

Output from Listing 5−3:

```
Connecting to database...
Auto−commit is set to false
SQLException occured with message: ORA−01839:
date not valid for month specified

Starting rollback operations...
Rollback successfull!
Goodbye!
```

Closing the Statement object

Just as you close a Connection object to save database resources, you should also close the Statement object, for the same reason. A simple call to the close() method will do the job. If you close the Connection object first it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

Working with PreparedStatement Objects

As you know, the PreparedStatement interface extends the Statement interface. Its added functionality also gives it a couple of advantages over a generic Statement object.

First, it gives you the flexibility of supplying arguments dynamically. Although you can use the Statement object to build and execute your SQL statements on the fly, the PreparedStatement object reduces your work. All you do is assign the values you want to use to the appropriate parameter placeholders.

> Caution Not all DMBSs support the concept of a PreparedStatement. In those that don’t, the database flushes the compiled SQL statement from memory after it is executed. Refer to your database or JDBC documentation for details.

Second, when you create a PreparedStatement object JDBC "prepares" the SQL statement for execution by sending it to the database, which then parses, compiles, and builds a query execution plan. This parsed statement lives in memory and remains ready to use during your database session or until you close the PreparedStatement object.

Tip PreparedStatement objects can improve performance of frequently used SQL statements. The database pre−processes the SQL statements, which saves time when you reuse the statement.

Creating the PreparedStatement object

Just as a Connection object creates the Statement object, it also creates a PreparedStatement object. The following code snippet shows how to employ its prepareStatement() method to instantiate a PreparedStatement object:

```
//Assume conn is a valid Connection object
String SQL = "Update employees SET salary = ? WHERE ename = ?";
PreparedStatement prepStmt = conn.prepareStatement(SQL);
```

What Are JDBC Parameters?

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement. The setXXX() methods bind values to the parameters. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred to by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which start at 0.

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object, which works with database stored procedures, can use all three. Here are the definitions of each:

- IN — A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the set*XXX*() methods.
- OUT — A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the get*XXX*() methods.
- INOUT — A parameter that provides both input and output values. You bind variables with the set*XXX*() methods and retrieve values with the get*XXX*() methods.

The SQL String you supply when calling the method can represent an DELETE, UPDATE, SELECT, INSERT, or DDL SQL statement. Notice too that a ? represents the unknown values that you supply at runtime.

Using the PreparedStatement object

All of the Statement object's methods for interacting with the database — execute(), executeQuery(), executeUpdate(), and executeBatch() — work with the PreparedStatement object. However, the methods are modified to use SQL statements that can take input the parameters. When using the PreparedStatement object you must bind values to all parameters otherwise a SQLException occurs.

To bind values to parameters you use the set*XXX*() methods. (*XXX* represents the Java data type of the value you wish to bind to the input parameter.) JDBC uses the set*XXX* methods to convert the Java data type to the appropriate SQL data type for your target database, as shown in the following code snippet:

```
//Assume conn is a valid Connection object
String SQL = "UPDATE employees SET salary = ? WHERE ename = ?";
PreparedStatement pstmt = conn.prepareStatement(SQL);


//bind variables
pstmt.setInt(1,"100000");
pstmt.setString(2,"toddt");
pstmt.executeUpdate();
```

Note The parameter values are not reset after you execute the prepared statement. You can overwrite them with another set*XXX*() method call or use the clearParameters() method.

Sometimes you may not know the data type of a value supplied at runtime. The PreparedStatement object's setObject() method handles this situation by taking any Java object and converting it into the appropriate JDBC data type. This method is extremely useful when you're working with SQL3 data types.

Listing 5−4 provides an example of how to use the PreparedStatement object. In this example I am simply adding a record to the Employees table. First I create the PreparedStatement object with parameter placeholders for SSN, Name, Salary, Hiredate, and Loc_Id. Next I bind these values to the corresponding parameter with the appropriate set*XXX*() method. Finally, I call the executeUpdate() method to insert the row into the table. This example only uses the executeUpdate() method, but the execute() and executeQuery() methods work in a similar fashion.

Listing 5−4: PrepStmt.java

```
package Chapter5;

import java.sql.*;

public class PrepStmt{

  public static void main(String[] args) {

    //Declare Connection object
    Connection conn = null;

    //Declare PreparedStatement object
    PreparedStatement pstmt = null;

    //Begin standard error handling
    try {


      //Register driver.
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection.
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //Create PreparedStatement object
      String SQL = "INSERT INTO Employees VALUES (?,?,?,?,?)";
      pstmt = conn.prepareStatement(SQL);

      //Bind values into the parameters.
```

```
      int randomSsn = ((int)Math.floor(Math.random() * 899999999));
      randomSsn = randomSsn  + 100000000;
      pstmt.setInt(1,randomSsn);
      pstmt.setString(2,"Andy");
      pstmt.setDouble(3,1400.51);
      pstmt.setDate(4,Date.valueOf("2002-06-11"));
      pstmt.setInt(5,400);

      //Check to ensure that the INSERT worked properly
      int updateCount = pstmt.executeUpdate();
      if(updateCount==1)
       System.out.println("Record inserted into " +
                          " \"Employees\" table.");

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");
  }//end main

}// end PrepStmt class
```

Output from Listing 5−4:

```
Connecting to database...
Record inserted into  "Employees" table.
Goodbye!
```

Streaming data with PreparedStatement objects

A PreparedStatement object has a feature that the Statement object does not: the ability to use input and output streams to supply parameter data. This enables you to place entire files into database columns that can hold large values, such as CLOB and BLOB data types. Streaming this data to the database saves you from having to assign it to a variable or an object in your application. This technique reduces the memory footprint of your program by not having to store the data in memory.

The following list explains the methods you use to stream data:

- setAsciiStream() is used to supply large ASCII values.
- setCharacterStream() is used to supply large UNICODE values.
- setBinaryStream() is used to supply large binary values.

The set*XXX*Stream() method requires an extra parameter, the file size, besides the parameter placeholder. This parameter informs the driver how much data should be sent to the database using the stream. Listing 5−5 provides an example storing and retrieving an XML file in a database.

Listing 5−5: StreamingXML.java

```
package Chapter5;

import java.sql.*;
import java.io.*;
import java.util.*;

public class StreamingXML {

  public static void main(String[] args) {
    //Declare Connection, Statement, PreparedStatement and ResultSet
    //variables

    Connection conn = null;
    PreparedStatement pstmt = null;
    Statement stmt = null;
    ResultSet rset = null;

    //Begin standard error handling
    try {

      //Register driver.
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection.
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //Create a Statement object and build table
      stmt = conn.createStatement();
      createXMLTable(stmt);

      //Open a FileInputStream
      File f = new File("employee.xml");
      long fileLength = f.length();
      FileInputStream fis = new FileInputStream(f);

      //Create PreparedStatement and stream data
      String SQL = "INSERT INTO XML_Data VALUES (?,?)";
      pstmt = conn.prepareStatement(SQL);
      pstmt.setInt(1,100);
      pstmt.setAsciiStream(2,fis,(int)fileLength);
      pstmt.execute();

      //Close input stream
      fis.close();

      // Do a query to get the row
      SQL = "SELECT Data FROM XML_Data WHERE id=100";
      rset =  stmt.executeQuery (SQL);

      // Get the first row
      if (rset.next ()){
```

```
        //Retrieve data from input stream
        InputStream xmlInputStream = rset.getAsciiStream (1);
        int c;
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        while (( c = xmlInputStream.read ()) != -1)
          bos.write(c);

        //Print results
        System.out.println(bos.toString());
      }

    //Standard error handling.
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
          se.printStackTrace();
      }//end finally try
    }//end try

    System.out.println("Goodbye!");

  }//end main

  public static void createXMLTable(Statement stmt)
throws SQLException{

    //Create SQL Statement
    String streamingDataSql = "CREATE TABLE XML_Data
(id INTEGER, Data LONG)";

    //Drop table first.
    try{
      stmt.executeUpdate("DROP TABLE XML_Data");
    }catch(SQLException se){
      //Handle Oracle DROP table error
      if(se.getErrorCode()==942)
        System.out.println("Error dropping XML_Data table:"
+ se.getMessage() );
    }//end try

    //Build table.
    stmt.executeUpdate(streamingDataSql);

  }//end createStreamingXMLTable

}//end StreamingXML class
```

The output from Listing 5−5 is as follows:

```
Connecting to database...
<?xml version="1.0"?>
<Employee>
    <SSN>123963741</SSN>
    <name>Art</name>
    <Salary>56321.87</Salary>
    <Hiredate>08-08-1988</Hiredate>
    <Loc_Id>200</Loc_id>
<Employee>
```

Notice that to use streaming data requires binding an input stream to a parameter. Any class derived from the InputStream interface will work. In this instance I read a file from a disk, but you can just as easily use a network socket as a data source.

Batch updates with PreparedStatement objects

As I mentioned earlier, PreparedStatement objects support the Statement object's executeBatch() method. The only difference between the two is that you add "parameter sets" to the batch once you supply the SQL statement.

The following code snippet demonstrates how to use the executeBatch() method with a PreparedStatement object:

```
//Assume conn is a valid Connection object
String SQL = "UPDATE employees SET salary = ? WHERE ename = ?";
PreparedStatement prepStmt = conn.prepareStatement(SQL);

//Set the variables
int sal = 150000;
prepStat.setInt(1,sal);

String name = "toddt";
prepStmt.setString(2,name);

//add it to the batch
prepStmt.addBatch();


//add more batches
.
.
.
prepStmt.addBatch();

//Now send the batch to the database
prepStmt.executeBatch();
```

All of the guidelines regarding batch updates that apply to the Statement object apply to the PreparedStatement object, particularly the auto−commit property. Remember, if you want every statement permanently applied to the database when it is executed, leave auto−commit on its default value of true. When you need transactional control, set auto−commit to false and explicitly use the commit() method to apply your changes.

Working with CallableStatement Objects

CallableStatement objects enable you to execute stored procedures located on the database from your Java application. If you look back at Figure 5−1 you can see that the CallableStatement interface extends the PreparedStatement interface. One extra feature is that the CallableStatement object not only handles IN parameters, but also has additional support for handling OUT and INOUT parameters. The CallableStatement object can use all three to adequately represent a stored procedure's behavior.

Creating the CallableStatement object

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object.

Before you can create the object you need to know about the stored procedure you want to access. Suppose, for example, that you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
    (Emp_SSN IN NUMBER, Emp_Name OUT VARCHAR) AS
BEGIN
    SELECT name
    INTO Emp_Name
    FROM Employees
   WHERE SSN = EMP_SSN;
END;
```

The following code snippet shows how to employ the Connection.prepareCall() method to instantiate a CallableStatement object based on the preceding stored procedure:

```
//Assume conn is a valid Connection object
String SQL = "{call getEmpName (?,?)}";
CallableStatement cstmt = conn.prepareCall (SQL);
```

The String variable SQL represents the stored procedure, with parameter placeholders, using JDBC's SQL escape syntax. The escape syntax tells the driver, which is database−specific, to convert the call into the correct format. As you can see, you must know the stored procedure's name and signature.

JDBC 3.0 JDBC 3.0 enables you to use named OUT parameters in the registerOutParameter() method. Prior versions only enabled you to refer to OUT parameters by their ordinal position. Enabling you to specify the name of the parameter makes the method function like the get*XXX*() methods in terms of parameter identification.

Table 5−4 shows the valid formats for the escape syntaxes you can use, depending on whether you need IN or OUT parameters.

Table 5−4: PrepareCall() Parameter Formats

Format	IN Parameter	OUT Parameter

{call *stored_procedure_name*}	No	No
{? = call *stored_procedure_name*}	No	Yes
{call *stored_procedure_name* (?, ?, ...,?)}	Yes	No
{? = call *stored_procedure_name* (?, ?, ...,?)}	Yes	Yes

As you can see, the first prepareCall() in Table 5−4 calls only the stored procedure and accepts no parameters. You would use this format to call a stored procedure that performs some internal operation within the database and does not supply feedback — for example, if you are purging historical data from a data warehouse.

The next format returns an OUT parameter at the completion of the stored procedure. The value might represent a method's success or failure flag, or a value calculated within the stored procedure.

The third format enables you to supply IN parameters. You would likely use this format to call a stored procedure to update tables with the values you supplied.

The last format enables you to supply both IN and OUT parameters. Here you supply values as IN parameters, perform calculations or query a table, then get the result as an OUT parameter.

Using the CallableStatement object

Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the set*XXX*() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The following sections describe each type of parameter and how to use each with the method.

OUT parameters

The registerOutParameter() method binds the JDBC data type to the data type the stored procedure is expected to return. This method is different from the set*XXX*() method that associates a Java data type with an IN parameter. OUT parameters require the JDBC type, which maps to SQL data types, for database compatibility.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate get*XXX*() method. This method casts the retrieved value of SQL type to a Java data type.

Listing 5−6 shows you how to access the getEmpName stored procedure I presented at the beginning of this section. Notice that it uses both IN and OUT parameters. First I bind the SSN to parameter 1 with the setInt() method. Then I use the registerOutParameter() method to set the JDBC data type for the OUT parameter. Finally, I use the execute() method to execute the stored procedure and use the getString() method to retrieve the data.

Listing 5−6: CallableStmts.java

```
package Chapter5;

import java.sql.*;

public class CallableStmt {

   public static void main(String[] args) {

    //Create Connection, Statement, and ResultSet objects
    Connection conn = null;
    CallableStatement cstmt = null;

    //Begin standard error handling
    try {

      //Register driver.
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection.
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //Create CallableStatement object
      cstmt = conn.prepareCall ("{call getEmpName (?,?)}");

      //Bind IN parameter first, then bind OUT parameter
      int ssn = 111111111;
      cstmt.setInt(1,111111111);
      cstmt.registerOutParameter(2,java.sql.Types.VARCHAR);

      //Use execute method to run stored procedure.
      cstmt.execute();

      //Retrieve employee name with getXXX method
      String empName = cstmt.getString(2);
      System.out.println("Employee with SSN:" + ssn
          + " is " + empName);

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try

    System.out.println("Goodbye!");
  }//end main
```

```
}//end CallableStmt
```

The output from Listing 5−6 is as follows:

```
Connecting to database...
Employee with SSN:111111111 is Todd
Goodbye!
```

INOUT parameters

An INOUT parameter plays the role of both an IN and an OUT parameter. Using an INOUT parameter is relatively simple. First, use the set*XXX*() method to bind a value to the parameter. This will cause the parameter to act as an IN parameter. Next, use the registerOutParameter() method to register the same parameter as OUT.

Consider the following Oracle stored procedure, which has both an IN and an INOUT parameter:

```
CREATE OR REPLACE PROCEDURE updateEmpName (Emp_SSN IN NUMBER,
Emp_Name IN OUT VARCHAR)
     AS
BEGIN
     UPDATE Employees
     SET name = Emp_name
     WHERE SSN = EMP_SSN;
     COMMIT;

     SELECT name
     INTO Emp_Name
     FROM Employees
     WHERE SSN = EMP_SSN;
END;
```

The following code snippet shows how to use the preceding stored procedure:

```
String SQL = "{call updateEmpName (?,?)}";
CallableStatement cstmt = conn.prepareCall (SQL);

//Set the IN parameter
cstmt.setInt(1,111111111);
cstmt.setString(2,"Todd Thomas");

//Set the same parameter as OUT
cstmt.registerOutParameter(2,java.sql.Types.VARCHAR);

//Execute the call to the stored procedure
cstmt.executeUpdate();

//Retrieve the result from the stored procedure.
String str = cstmt.getString(2);
```

Note Be aware of data type compatibility issues when using INOUT parameters. You should use Java and JDBC data types that map to each other for both the set*XXX*() methods and the registerOutParameter() method. For example, if you use a setBoolean() method to bind the IN parameter, you should register the OUT parameter as a JDBC BIT data type with the registerOutParameter() method. Or, if you are using JDBC 3.0, you can register the OUT parameter as a JDBC BOOLEAN data type.

Batch updates with the CallableStatement object

Because the CallableStatement interface extends PreparedStatement interface, it inherits the executeBatch() method. However, the CallableStatement object cannot use OUT an INOUT parameters in batch update operations. A BatchUpdateException is thrown if:

- you try to use either OUT or INOUT parameters.
- your stored procedure returns anything but an update count.

All other aspects associated with using batch updates remain the same as with prepared statements.

Summary

This chapter illustrated how to interact with your database once you open a connection. JDBC 3.0 provides you with several different ways to submit SQL statements and control how the database processes them. In particular you should remember:

- You can use the Statement object to submit static DDL and DML commands.
- You use the PreparedStatment and CallableStatement objects to submit SQL statements with parameters.
- You use a CallableStatement object to access database stored procedures.
- You use Connection.setAutoCommit() and Connection.commit() methods enable you to control transactions.

Chapter 6: Working with Result Sets

In This Chapter

- Creating different types of result sets
- Moving around and viewing data in a result set
- Updating data in a result set
- Retrieving data from a result set
- Handling data−type issues

In Chapter 4, “Connecting to Databases with JDBC,” and Chapter 5, “Building JDBC Statements,” I covered how to create and use Connection and Statement objects. These objects, combined with the ResultSet object, provide all the components you need to build a full−featured JDBC application.

ResultSet objects hold data from SQL queries you execute using a Statement, PreparedStatement, or CallableStatement object. In some respects a result set is a view into your database. To use the data contained in a ResultSet object you must extract it using one of the getXXX() methods defined by the ResultSet interface.

In this chapter I introduce the ResultSet object and explain its purpose in a JDBC program. I also detail the different types of ResultSet objects, what each one does, and how to decide which one to use. Finally I discuss data type differences that exist between SQL and Java; a topic that is important when you begin to extract and use data from a ResultSet object.

What Are JDBC Result Sets?

Connection objects represent database connections; Statement objects enable you to execute SQL statements against the database. These objects provide the foundation that enables you to build views of data in your database.

The term “result set” refers to the row and column data contained in a ResultSet object. This is a logical view of row and column data in your database that meets the criteria of your SQL query. A result set can have any number of rows and columns: the actual number depends on the query. If you place a WHERE clause in your SQL statement, only the row data meeting those criteria will be retrieved.

The ResultSet interface defines methods that enable you to interact with the data in the database. The JDBC driver provides the ResultSet class that implements the interface. When a Statement, PreparedStatement, or CallableStatement object executes an SQL query successfully it returns a ResultSet object.

Note Not all databases support stored−procedures that return result sets, and so the CallableStatement object might not be able to return a result set. Check the developer’s guide for your DBMS, or the documentation for your JDBC driver, to determine whether your database supports this feature.

The default ResultSet object enables you to only view the data in your database that meet the criteria of your SQL query. However, you can also create ResultSet objects that can update the data in the row you are viewing, insert a new row into the table, or even delete a row. Thus, a ResultSet object can enable you to perform DML statements programmatically without having to explicitly issue SQL INSERT or UPDATE

statements.

The following section provides an overview of the concepts you need to understand in order to work effectively with a ResultSet object.

Introducing Result Set Concepts

Although JDBC defines different types of result sets to meet different programmatic needs, many concepts apply to all types. This section is devoted to describing these common concepts. Ideally, this will provide you with a foundation for understanding the different types of result sets.

Result set cursors

Despite the number of rows in a result set, you can only access one row at a time. A ResultSet object points to this "active" row using a cursor. If you want to reference another row in the result set you must explicitly move the cursor with one of the ResultSet object's cursor−movement methods.

Figure 6−1 illustrates a result set cursor and how it moves through the data set. Notice that the cursor points to the third row of a seven−row result set. If you issue the ResultSet.next() method, the cursor advances one position to the fourth row. Remember, when working with result set data you always work with the row where the cursor points.

Two cursor positions within a result set warrant special mention. These are the "before first row" (BFR) and "after last row" (ALR) cursor positions in the result set. These areas are devoid of data and an SQLException occurs if you use the get*XXX*() or update*XXX*() methods in these locations. Figure 6−1 also illustrates these positions within a result set.

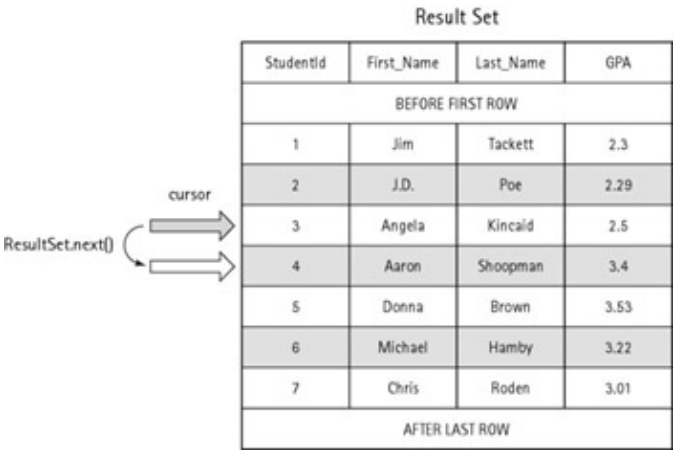

Figure 6−1: Result set cursor illustration

When a ResultSet object is initially populated the cursor defaults to the BFR position. You must explicitly move the cursor to a location that contains data before invoking any data−access methods. In addition, when scrolling through a result set you may run past the last row of data. This will place the cursor in the ALR position. Depending upon the result set type, you may or may not be able to go backwards to a valid cursor position. If you are unable to do this you must recreate your result set by executing the SQL statement again.

Result set types

JDBC provides different types of ResultSet objects that enables you to interact with the database in a variety of ways. The first, which is the default, has minimal functionality and enables you only to move forward through the result; it does not enable you to update data. The second type enables you to move forward, backward, or to any row within the result set. The third enables you to update the contents in the result set.

JDBC 3.0 Prior to JDBC 3.0, the Connection.commit() method could close ResultSet objects. The driver implementation determined this behavior. A new ResultSet interface property, holdability, enables you to specify when a ResultSet object is closed. Your options are to close it when you call the Connection.commit() method or to close it explicitly some time after a commit.

When creating either a Statement, PreparedStatement, or CallableStatement object you define the type of result set it creates by passing in parameters to the respective create statement method of a Connection object. You do not need to pass any parameters to create a standard result set.

Table 6−1 provides a quick summary of each result set type, and the following sections provide more detail.

Table 6−1 : Result Set Types

ResultSet	Description
Standard	Enables basic access to the result set data. Does not reflect changes to underlying data on server. Moves in one direction, forward. Data within the result set cannot be updated.
Scrollable	Provides enhanced movement capabilities over the standard ResultSet object. Moves forward, backward, or to any row. Can reflect data changes on the server. You set this type with the resultSetType parameter when creating the Statement, PreparedStatement, or CallableStatement object.
Updateable	Enables you to update underlying database information through the ResultSet object. You set this type with the resultSetConcurrency parameter when creating the Statement, PreparedStatement, or CallableStatement object.

Note Scrollable and updateable ResultSet objects are mutually exclusive. For example, you can have an updateable ResultSet object that is forward only, or a scrollable ResultSet object is not updateable.

Standard result sets

A standard result set is forward−only and non−updateable. This type enables you to move the cursor forward through the result set but does not enable you to update the result set data. An SQLException is thrown if you try to move the cursor backward, or insert or update the data in the result set.

You should consider using standard result sets when you need to do basic work with result set data. For example, if you need to populate a listbox to display a list of inventory items, using a forward−only result set makes sense. Or, this type will prove useful if you need to loop through the result set to count the number of rows in a database that meet some criteria defined by a WHERE clause.

Scrollable result sets

To traverse a result set you need to use a scrollable ResultSet object. Introduced in JDBC 2.0, this type enables you to go forward or backward, or to jump to a specific row in the result set. You will find this type useful when you need to move the cursor to a different row in the result set based on some event or when you present a form with a data control to allow users to scroll forward and backward through the result set.

You can also create scrollable ResultSet objects that are sensitive to changes in the underlying data on the server. As you move the cursor through the result set the ResultSet object determines if the data on the server has changed. If it has the ResultSet object updates the result set with the new data. This feature is useful when you need to provide the user with up−to−date information such as with real−time applications like airline reservation systems.

Tip Use scrollable, change−sensitive, and updateable ResultSet objects only when necessary. The overhead associated with these types may compromise performance. However, do not neglect them if they represent the best solution.

Using scrollable and change−sensitive result sets creates additional overhead because of the extra processing the ResultSet object must perform. Only use these types when you feel your application needs these features.

Updateable result sets

You can update the column data in the current row with an updateable ResultSet object. Doing this enables you to make changes without explicitly submitting additional SQL statements to the database. However, as with scrollable ResultSet objects, you will experience a performance decrease while using an updateable result set.

Retrieving result set data

Data returned from an SQL query are JDBC data types, and you must convert them to Java data types before you can use the information in your application. A ResultSet object provides the get*XXX*() method to perform this conversion.

The *XXX* placeholder in the get*XXX*() method represents the Java data type you wish to retrieve. For example, if you want to retrieve a column value as a Java int then you use the getInt() method. Each JDBC data type has a recommended Java data type. You should always try to use the recommended get*XXX*() method to decrease the risk of data corruption.

XRef Chapter 7, “Understanding JDBC Data Types,” provides more detail on JDBC and Java data types.

However, the get*XXX*() method gives you the freedom to coerce JDBC data types to different, noncorresponding Java data types. For example, you can use the getString() method to retrieve any SQL numeric data types, such as an INTEGER, DOUBLE, and NUMBER. You can also convert from numeric data types of high precision to ones of lesser precision, but this will cause you to lose data. For example, the getInt() method enables you to retrieve JDBC DOUBLE data types. During the conversion the digits to the right of the decimal point are dropped. This operation is akin to assigning a double to an int, and as a result the value is floored.

The get*XXX*() method is overloaded to provide you with flexibility with regard to retrieving the column values. The following are the ResultSet object’s methods:

```
ResultSet.getXXX(int columnIndex)
ResultSet.getXXX(String columnName)
```

The first method enables you to retrieve the result set data based on the ordinal column position. The column numbers start at 1, not 0 as Java array indices do. The second method uses the column name to retrieve the data, as shown in the following code snippet:

```
//Assume a valid Connection, conn.
Statement stmt = conn.createStatement();

//Create a ResultSet object
String SQL = "SELECT Name FROM Employees";
ResultSet rset = stmt.executeQuery(SQL);

//Retrieve by ordinal column position
String byColumnNumber = rset.getString(1);

//Retrieve by column name
String byColumnName = rset.getString("name");
```

Tip You can refer to columns in a result set by their names, which may be easier for you than having to remember the column number.

The ResultSet object also has get*XXX*() methods with which to access the following SQL3 data types: CLOB, BLOB, ARRAY, STRUCT, REF, and DISTINCT. This method gives you access to the data using an SQL LOCATOR, which is a logical pointer on the client that refers to data on the server. As a result you do not materialize the data on the client using the get*XXX*() methods. You must explicitly perform this task to retrieve the data on the client.

You may use an input stream to materialize columns that contain large amounts of binary or character data such as BLOB and CLOB. The methods getBinaryStream() and getAsciiStream() return InputStream objects, so you can control the data download to prevent extremely large values from consuming too much memory.

XRef Refer to Chapter 7, “Understanding JDBC Data Types,” for a more complete explanation of using the SQL3 data types and the ResultSet. getObject() method.

Here is an example of using an InputStream to retrieve column information:

```
//Assume a valid Statement object
Sting SQL = "SELECT Data FROM DataTable";
ResultSet rset = stmt.executeQuery (SQL);

//Loop through the result set
while (rset.next()){
  //Use an input stream to store the data
  InputStream is = rset.getBinaryStream (1);

  //Collect results from InputStream into a
  //ByteArrayOutputStream object
  int i;
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  while((i = is.read ()) != -1){
     bos.write(i);
  }
}
```

The getObject() method will enable you to retrieve any data type. The method returns an Object. If you want a more specific type you must cast it appropriately. You can also access UDTs on the server using the getObject() method. This is especially helpful for custom data mappings.

Using Standard Result Sets

A standard ResultSet object enables you to view and retrieve data in a result set. This type is the default when you do not supply parameters to the Connection object's createStatement(), prepareStatement(), and prepareCall() methods. The standard result set is forward−only and non−updateable.

How the result set is populated varies. For standard SELECT statements without sorting or grouping commands, the result set data are materialized incrementally.

That is, as the cursor moves through the result set, the data are retrieved from the server and placed in the user's application space. If someone makes a change to the data on the database server, and you have not yet viewed the data or the cursor is not on that row, you will see the changes when you move the cursor to that row. If you issue a SELECT statement that groups or sorts the result set the data is materialized on the client immediately.

A standard ResultSet object also provides the fastest access to the result set data. The other types, scrollable and updateable, must maintain additional information about the result set, and this can degrade performance.

Creating a standard result set

You do not directly instantiate a ResultSet object as the JDBC specification only defines an interface, not a ResultSet class. The Statement, PreparedStatement, or CallableStatement returns an instance of a ResultSet object when it successfully completes the execute() or executeQuery() method. As I discussed in chapter 5, the executeUpdate() method returns an update count, not a result set.

The following code snippet demonstrates how to instantiate a ResultSet object using both a Statement and PreparedStatement object:

```
//Assume a valid Connection, conn.
Statement stmt = conn.createStatement();

//Create ResultSet object with Statement
String sql1 = "SELECT Name, Salary FROM Employees";
ResultSet rset1 = stmt.executeQuery(sql1);

//Create ResultSet object with PreparedStatement
String sql2 = "SELECT Name, Salary FROM Employees WHERE Ssn = ?";
PreparedStatement pstmt= conn.prepareStatement(sql2);
pstmt.setInt(1,876349372);
ResultSet rset2 = pstmt.executeQuery();
```

That's it. The stmt.executeQuery() and pstmt.executeQuery() methods create rset1 and rset2, respectively. The result set for rset1 contains the Name and Salary information for all the employees in the Employees table. The result set for rset2 holds the same data, except for one employee. I used a parameterized query to limit the number of rows returned to one for this snippet.

XRef Chapter 8, “Mining Database Metadata with JDBC” explains how to obtain information about your ResultSet object. You can retrieve property and type information about the columns contained in a ResultSet object using the ResultSetMetaData object.

Moving data around in a standard result set

A result set is of little value unless you can move its data around. For a standard result set object you can only use the ResultSet.next() method to the move the cursor through the result set rows. As you might expect, the next() method moves the cursor to the next valid row of the result set.

When using the next() method you need to ensure you do not move the cursor to the ALR area. Fortunately, the next() method tests for this boundary condition. The method returns true if the cursor moves into a valid row. If you move the cursor into the ALR position the method returns false. This behavior enables you to process the data within a result set with a while loop by using the ResultSet.next() method as the test condition, as shown in the following code:

```
//Assume a valid Statement object stmt
ResultSet rs = stmt.executeQuery("SELECT * from Employees");

while(rs.next()){
  //process rs data
}
```

As you move through the result set you may need to determine where your cursor is located. For example, you may want ensure that the cursor is not located in the ALR position before calling a getXXX() method. A ResultSet object enables you to determine cursor position in several different ways. Table 6−2 illustrates the methods related to cursor position.

Table 6−2: Result Set Cursor−Position Methods

Method	Description
isBeforeFirst()	Returns true if the cursor is in the "before−first−row" position.
isFirst()	Returns true if the cursor is on the first row.
isAfterLast()	Returns true if the cursor is in the "after−last−row" position.
isLast()	Returns true if the cursor is on the last row.
getRow()	Returns an int specifying the ordinal row number. The first row is 1, the second 2, and so on. The method returns 0 if no row exists.

You will likely use the cursor position method getRow() more than any other method. This method returns the cursor’s current row number in the result set. Notice that it returns the ordinal row number. That is, all row numbers begin with 1 and increment by one. The method returns 0 when called if the cursor is in the BFR or ALR position.

The other cursor−position methods also help you determine where your cursor is located within the result set. The isBeforeFirst() and isAfterLast() methods return true if the cursor is in the BFR or ALR positions, respectively. You can see if the cursor is on the first or last row by using the isFirst() or isLast() methods. Again, these methods return true if the cursor is located in the relevant position.

Listing 6−1 brings together the concepts presented in this section. In this example I open a database connection, create a Statement object, and submit a query to retrieve the SSN, Name, and Salary columns from the Employee table.

Next, I demonstrate several of the cursor−position methods. First I call the ResultSet.isBeforeFirst() method, which returns true because I have not advanced the cursor to the first row of data and as a result it is still in the BFR position. Next I loop through the result set using the ResultSet.next() method and print the column values to the screen. (Notice that I use the ResultSet. getRow() method to obtain the row number of the cursor location.) Before exiting the application I call the ResultSet.isAfterLast() method to illustrate its use. This method returns true because the last call to the rs.next() method places the cursor in the ALR position.

Listing 6−1: StandardRs.java

```
package Chapter6;

import java.sql.*;

public class StandardRs {

  public static void main(String[] args) {

    //Declare Connection, Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Register driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //Create a Statement object and execute SQL query
      stmt = conn.createStatement();
      String sql = "SELECT SSN, Name, Salary FROM Employees";
      rs = stmt.executeQuery(sql);

      //Variables to hold information
      int ssn;
      String name;
      double salary;

      System.out.println("\"Before−first−row\" = "
        +  rs.isBeforeFirst());

      while(rs.next()){
        //Retrieve by column name
        ssn= rs.getInt("SSN");
        name = rs.getString("Name");

        //Retrieve by column index
        salary = rs.getDouble(3);
```

```
        //Display values
        System.out.print("Row Number=" + rs.getRow());
        System.out.print(", SSN: " + ssn);
        System.out.print(", Name: " + name);
        System.out.println(", Salary: $" + salary);
      }

      System.out.println("\"After-last-row\" = " +
                         rs.isAfterLast());

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");

  }//end main
}//end StandardRs class
```

The output for Listing 6-1 is as follows:

```
Connecting to database...
"Before-first-row" = true
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5000.00
Row Number=2, SSN: 419876541, Name: Larry, Salary: $1500.00
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2000.95
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3080.00
Row Number=5, SSN: 987654321, Name: John, Salary: $4351.00
"After-last-row" = true
Goodbye!
```

Using Scrollable Result Sets

Standard result sets enable you to move in only one direction, forward. You may find this acceptable at times. However, you will sometimes need to traverse the result set's data freely in any direction.

With scrollable result sets you can move forward, backward, and to specific rows within the result set. This maneuverability proves handy when you develop an application that requires user interaction with the result set data. For example, users of an inventory-control program may need to scroll forward and backward through a result set to check current inventory data. A scrollable result set enables them to do this.

If properly configured, scrollable result sets can also recognize changes in the underlying data on the database. If you have a scrollable result set that is sensitive to changes then you will always see the most current view of the data. You might find this beneficial if your application, like the inventory−control system described earlier in this chapter, needs up−to−date information. However, result sets sensitive to changes may prove troublesome if you need to provide a “snapshot” view of the data.

Tip The get*XXX*() methods used to access data are the same for standard, scrollable, and updateable ResultSet objects.

The following section shows you how to create and use scrollable result set objects and gives an example illustrating their use.

Creating scrollable result sets

As I mentioned earlier, you specify the type of result set you want a query to return when you are creating Statement, PreparedStatement, or CallableStatement objects.

To denote a scrollable result set you must supply predefined parameter values to a Connection object’s createStatement(), prepareStatement(), or prepareCall() methods. The following are the method calls with signatures:

```
createStatement(int resultSetType, int resultSetConcurrency);

prepareStatement(String SQL, int resultSetType, int resultSetConcurrency);

prepareCall(String sql, int resultSetType, int resultSetConcurrency);
```

The first parameter, resultSetType, indicates how scrollable, and how sensitive to data changes, to make the result set when creating it. Table 6−3 lists the valid resultSetType parameter values for creating scrollable ResultSet objects.

The next parameter, resultSetConcurrency, creates an updateable result set, which I cover in the next section.

Table 6−3: Scrollable Result Set Parameter Constants

Constant	Comment
TYPE_SCROLL_INSENSITIVE	Provides a ResultSet whose cursor can move forward, backward, and to any row. Does not reflect changes to the data on the database as cursor moves through the data set.[a]
TYPE_SCROLL_SENSITIVE	Provides a ResultSet whose cursor can move forward, backward, and to any row. Provides a dynamic view of the data on the server. The most current value is always provided.
TYPE_FORWARD_ONLY	Default result set type.

	Does not reflect changes to the data on the database as the cursor moves through the data set.[a]

[a] This depends on the data type referenced. Some SQL3 data types use logical pointers, called LOCATORs, to the data on the database server. Using these data types may enable you to see changes because the data is not materialized on the client.

You create scrollable result sets using the TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE parameter values. The TYPE_SCROLL_INSENSITIVE parameter creates a result set whose cursor can move in any direction through the data but does not enable you to see changes to the data in the database — you only have a static view.

You can create a dynamic view of the data using the TYPE_SCROLL_SENSITIVE parameter. Underlying data changes on the server will be available to you as the cursor moves through the result set when you use this program.

The TYPE_FORWARD_ONLY parameter creates a default, forward−only result set. You only need to supply this parameter if you want to control the concurrency settings of a forward−only result set.

Caution Using scrollable result sets will slow down your application. The ResultSet object must do more work in order to enable you to scroll through the result set. If you also want the ResultSet object to be sensitive to data changes, even more overhead is incurred because the ResultSet object must check the database as the cursor moves through the data set. Only use these types when necessary.

Moving around scrollable result sets

A scrollable ResultSet object has methods to permit you to move the cursor forward, backward, and to a specific row. The cursor movement methods apply regardless of whether the result set is sensitive to the underlying data changes or not. Table 6−4 provides an overview of the methods used to control the cursor. All the methods return true if they complete successfully.

Table 6−4: Scrollable Result Set Movement Methods

Method	Description
next()	Moves the cursor to the next row.
previous()	Moves the cursor to previous row.
beforeFirst()	Positions the cursor in the “before−the−first” row location. Calling the get*XXX*() method immediately after this method will produce an SQLException.
afterLast()	Positions the cursor in the “after−the−last” row. Calling the get*XXX*() method immediately after this method will produce an SQLException.
first()	Moves the cursor to the first row of the result set data
last()	Moves the cursor to the last row of the result set data.
absolute()	Moves to a specific row relative to the first row of the data set.

relative()	Move to a specific row relative to the current row.
moveToCurrentRow()	Moves cursor to the remembered row.[a]
moveToInsertRow()	Moves cursor to the insert row.[a]

[a]Valid for updateable ResultSet objects. See the next section "Using Updateable Result Sets" for details.

The next() and previous() methods move the cursor forward to the next row or back to the previous row, respectively. If your cursor is on the first row and you call the ResultSet.previous() method, you will move the cursor into the BFR area. Likewise, you can move the cursor forward into the ALR with the ResultSet.next() method when the cursor is positioned on the last row.

However, the next two methods, beforeFirst() and afterLast(), explicitly move the cursor into the BFR and ALR regions. These two locations can provide you with a starting point to begin other cursor movements. For example, you might use these methods to position the cursor at the top or bottom of the data set before you start looping through it.

The ResultSet object's first() and last() methods move the cursor to the first and last rows of the result set, respectively. Calling these methods with a result set of one row produces the same result. You may also call these methods from the BFR and ALR areas to move the cursor to the respective locations.

You can use the ResultSet.absolute(int *n*) and ResultSet.relative(int *n*) methods to move the cursor to a specific row in the result set. The absolute() method moves the cursor *n* number of rows from the first row. Calls to ResultSet.first() and ResultSet.absolute(1) are equivalent. The relative() method moves the cursor forward or back *n* rows. (Positive values move the cursor forward while negative values move the cursor back.) Figure 6−2 illustrates how these methods work.

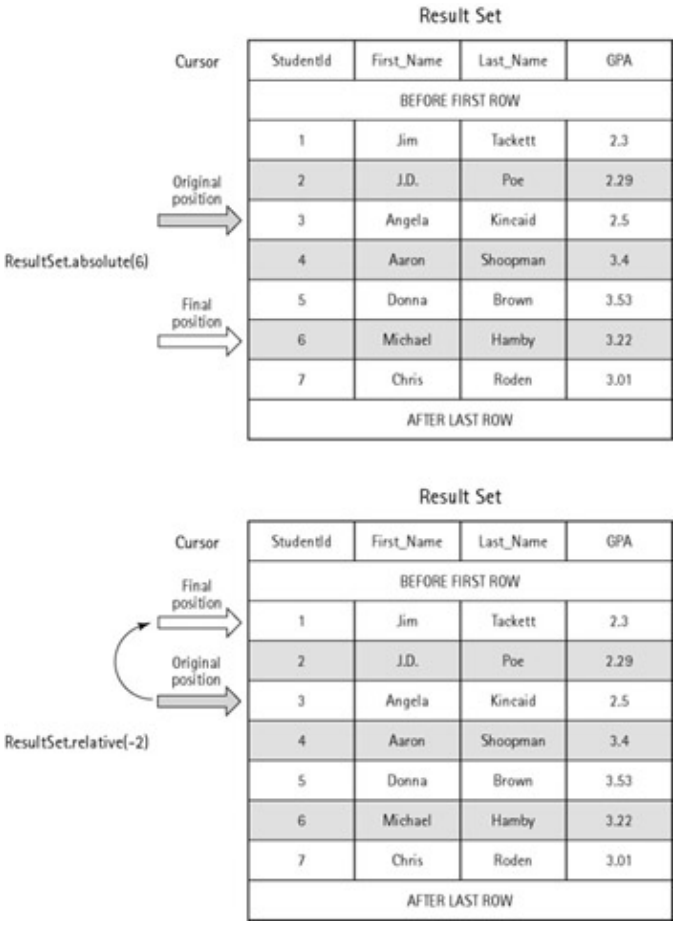

Figure 6−2: Result set cursor movement example

Listing 6−2 demonstrates the various topics covered in this section. In this example, I create a scrollable ResultSet object and illustrate the different cursor−position and cursor−movement methods.

Listing 6−2: ScrollableRs.java

```
package Chapter6;

import java.sql.*;

public class ScrollableRs {

  public static void main(String[] args) {

    //Declare Connection, Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Register driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //createStatement() method that specifies I want a
      //scrollable result set that is insensitive to changes on
      //the database. The result set is also READ_ONLY so the I
      //cannot use it to make changes.
      stmt=conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);

      String sql = "SELECT ssn, name, salary FROM EMPLOYEES";
      rs = stmt.executeQuery(sql);

      System.out.println("List result set for reference....");
      while(rs.next()){
        printRow(rs);
      }
      System.out.println();

      //Demonstrate afterLast() and beforeFirst()
      System.out.println("Move to \"After−last−row\" " +
                         "position with afterLast()");
      rs.afterLast();
      System.out.println("\"After−last−row\" = " +
        rs.isAfterLast());
      System.out.println();

      System.out.println("Move to \"Before−first−row\" " +
                         "position with beforeFirst()");
      rs.beforeFirst();
      System.out.println("\"Before−first−row\" = "
```

```
    +  rs.isBeforeFirst());
  System.out.println();

  //Demonstrate first() and last() methods.
  System.out.println("Move to first row with first().");
  System.out.println("The row  is:");
  rs.first();
  printRow(rs);
  System.out.println();

  System.out.println("Move last row with last().");
  System.out.println("The row is:");
  rs.last();
  printRow(rs);
  System.out.println();

  //Demonstrate previous() and next() methods.
  System.out.println("Move to previous row with previous().");
  System.out.println("The row  is:");
  rs.previous();
  printRow(rs);
  System.out.println();

  System.out.println("Moving to next row with next().");
  System.out.println("The row  is:");
  rs.next();
  printRow(rs);
  System.out.println();

  //Demonstrate absolute() and relative()
  System.out.println("Move to the 3rd row with absolute(3).");
  System.out.println("The row  is:");
  rs.absolute(3);
  printRow(rs);
  System.out.println();

  System.out.println("Move back 2 rows with relative(-2).");
  System.out.println("The row  is:");
  rs.relative(-2);
  printRow(rs);
  System.out.println();

//Standard error handling
} catch(SQLException se) {
  //Handle errors for JDBC
  se.printStackTrace();

} catch(Exception e) {
  //Handle errors for Class.forName
  e.printStackTrace();

} finally {
  try {
    if(conn!=null)
      conn.close();
  } catch(SQLException se) {
    se.printStackTrace();
  } //end finally try
} //end try

System.out.println("Goodbye!");
```

```
  }//end main

  public static void printRow(ResultSet rs) throws SQLException{
      //Field variables
      int ssn;
      String name;
      double salary;

      //Retrieve by column name
      ssn= rs.getInt("ssn");
      name = rs.getString("name");
      salary = rs.getDouble("salary");

      //Display values
      System.out.print("Row Number=" + rs.getRow());
      System.out.print(", SSN: " + ssn);
      System.out.print(", Name: " + name);
      System.out.println(", Salary: $" + salary);

  }//end printRow()

}//end ScrollableRs class
```

The output for Listing 6−2 is as follows:

```
Connecting to database...
List result set for reference....
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5000.0
Row Number=2, SSN: 419876541, Name: Larry, Salary: $1500.0
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2000.95
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3080.0
Row Number=5, SSN: 987654321, Name: John, Salary: $4351.0

Move to "After−last−row" position with afterLast()
"After−last−row" = true

Move to "Before−first−row" position with beforeFirst()
"Before−first−row" = true

Move to first row with first().
The row  is:
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5000.0

Move last row with last().
The row is:
Row Number=5, SSN: 987654321, Name: John, Salary: $4351.0

Move to previous row with previous().
The row is:
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3080.0

Moving to next row with next().
The row is:
Row Number=5, SSN: 987654321, Name: John, Salary: $4351.0

Move to the 3rd row with absolute(3).
The row is:
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2000.95
```

```
Move back 2 rows with relative(-2).
The row is:
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5000.0

Goodbye!
```

Determining Row Counts in a Result Set

JDBC does not provide you with a direct way to retrieve the number of rows returned from the SQL query. Nor does the ResultSet interface define a public property or method that provides this value. However, you may find a need for this information. If so, you may find the following two techniques useful: using a counter variable and using the ResultSet.getRow() method.

The counter variable technique uses a counter to determine the number of rows in a table. As you loop through the result set you increment the counter. After the loop, the variable's value will equal the number of rows in the result set.

```
//Assume a valid ResultSet object rs
int count;
while(rs.next()){
   count++;
}
```

If you have a scrollable result set you may prefer the second technique. It does not use a loop, but uses the ResultSet.last() method instead. Moving the full length of the result set populates the internal row count of the ResultSet object, as shown in the following code sample:

```
//Assume a valid ResultSet object rs
rs.last();
int count = rs.getRow();
```

Using Updateable Result Sets

Updateable ResultSet objects give you the freedom to manipulate result set data directly. You do not need to execute additional SQL statements to effect changes in the database. You can perform changes such as updating column data and inserting and deleting rows using an updateable result set.

Updating columns in a result set does not immediately change the values on the database server. The ResultSet.update*XXX*() method changes the data in the result set, not the data in the database. The changes are propagated to the database and to the result set once you commit them with the ResultSet.updateRow() method.

Be forewarned that like scrollable result sets these types create additional overhead within the ResultSet object. You should use updateable result sets only when necessary. That said, you will find that the benefits of being able to directly change data through the ResultSet object often outweigh the negatives.

However, sometimes a result set cannot be updated even if you specify the correct parameters when creating it. Some drivers will throw an SQLWarning indicating that the desired result set type could not be created. Regardless, the driver will still create a result set without the requested functionality.

Xref Appendix C, “JDBC Error Handling,” provides more details on the SQLWarning exception.

For your result set to be updateable, the SQL query must meet certain criteria. In general, to create a updateable ResultSet object the query should adhere to the following rules:

- If you intend to insert rows into a table, then the SQL query must return the primary key(s) of the table. An error occurs if you try to insert a row without specifying values for the primary key(s). However, some DBMS can auto−generate primary keys: such systems may allow this behavior. Be sure to check your driver or DBMS documentation for details.
- The SQL statement cannot use aggregation or sorting clauses such as GROUP BY or ORDER BY.
- You cannot update result sets created from SQL query statements that request data from multiple tables. Changes to these result sets would need to effect multiple tables, and this is not possible in current versions of JDBC.

The remainder of this section focuses on how to create and use updateable ResultSet objects.

Creating updateable result sets

You create updateable ResultSet objects with the same Connection object methods you use to create scrollable result sets. The following are the Connection object methods you use to create updateable ResultSet objects:

```
createStatement(int resultSetType, int resultSetConcurrency);

prepareStatement(String sql, int resultSetType, int resultSetConcurrency);

prepareCall(String sql, int resultSetType, int resultSetConcurrency);
```

You use the resultSetType parameter to specify scrollable result sets. Notice that you can create a forward−only result set that is also updateable. Also notice

that you must supply a parameter even if you want a default result set object. The last section covered the resultSetType parameter and how to use it.

The second parameter, resultSetConcurrency, defines the concurrency level you want the result set to have. This parameter has the following two options:

- CONCUR_UPDATABLE, which creates an updateable result set.
- CONCUR_READ_ONLY, which creates a read−only result set. This is the default.

The following code snippet demonstrates how to initialize a Statement object to create a forward−only, updateable ResultSet object:

```
//Assume a valid connection object.
Statement stmt = conn.createStatement(
                     ResultSet.TYPE_FORWARD_ONLY,
                     ResultSet.CONCUR_UPDATABLE);
```

Caution Like scrollable result sets, updateable result sets may decrease performance. The object must maintain additional information and make extra network calls, which can decrease responsiveness. However, the advantages of programmatically updating the result set may outweigh the disadvantages.

Concurrency is the ability to share and update information in the database with other users at the same time. You will face concurrency issues when you start allowing users to update database values. When a user wants to update data, the database locks it to prevent others from updating the same information. Other users cannot update the data until the lock is removed. The level of locking varies from database to database. Some systems only lock rows while others lock entire tables.

Concurrency can cause big problems for large systems with many users. Because of this, two types of concurrency levels exist: pessimistic and optimistic. Pessimistic concurrency assumes a lot of activity and locks the data being updated, which prevents others from updating the same data at the same time. Optimistic concurrency assumes little update activity and does not lock the data; inconsistencies between two simultaneous transactions are reconciled after any modifications are made. Generally, the last transaction to complete is the one that is applied.

Updating data with updateable result set

The ResultSet.update*XXX*() methods enable you to change database information programmatically. You can avoid executing additional SQL statements by using updateable result sets.

Calling an update*XXX*() method applies the changes to a column in the current row of the result set. The method requires two parameters. The first indicates the ordinal number of the column you want to update. (The method is overloaded so you may supply a String value for the column name as well.) The second indicates the new value for the column.

JDBC 3.0 JDBC 3.0 has new methods that enable you to update BLOB, CLOB, ARRAY, and REF data types.

To use the update*XXX*() method successfully you must follow three steps. First, you must position the cursor on the row you wish to update. Not doing so may cause you to update the wrong data. (This may sound obvious, but it can easily happen.)

Next, call the appropriate update*XXX*() method for the Java data type you are using. As with the set*XXX*() methods, the *XXX* refers to the Java programming−language data type. For example, if you are working with String object types you use the updateString() method. The JDBC driver converts the data to the appropriate JDBC type before sending it to the database.

XRef See Chapter 7, “Understanding JDBC Data Types,” for details on converting from Java data types to JDBC data types and vice versa.

Finally you must call the updateRow() method to commit the changes to the database. Failure to do this will result in your changes being lost. If you call the update*XXX*() method and then move the cursor, you will lose your changes.

Note The update*XXX*() methods do not make changes immediately. To commit all the changes you must explicitly call the ResultSet.updateRow() method.

You can undo the changes made to an updateable ResultSet object by calling the ResultSet.cancelRowUpdate() method. Using this method will undo all update*XXX*() method calls. However, you must call it before the updateRow() method to ensure that the changes are undone.

Listing 6−3 provides an example of how to use an updateable result set. The code loops through a result set of employee information and applies a cost−of−living adjustment to the employees' salaries. The changes are applied as I loop through the data. Notice the call to ResultSet.updateRow() to commit the changes to the database when I am finished with my updates. If I were to move to another record before calling this method, I would lose my changes.

Listing 6−3: UpdatableRs.java

```
package Chapter6;

import java.sql.*;

public class UpdateableRs {

  public static void main(String[] args) {

    //Declare Connection, Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Register driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //Create a Statement object and execute SQL query
      stmt = conn.createStatement();

      //createStatement() method that specifies I want a
      //scrollable and updateable result set that is insensitive
      // to data changes on the database server.
      stmt=conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
      String sql = "SELECT ssn, name, salary FROM EMPLOYEES";
      rs = stmt.executeQuery(sql);

      System.out.println("List result set for reference....");
      printRs(rs);

      //Loop through result set and give a 5.3%
      //cost of living adjustment
      //Move to BFR postion so while−loop works properly
      rs.beforeFirst();

      while(rs.next()){
        double newSalary = rs.getDouble("salary")*1.053;
        rs.updateDouble("salary",newSalary);
        rs.updateRow();
      }

      System.out.println("List result set showing new salaries");
```

```
      printRs(rs);

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");

  }//end main

  public static void printRs(ResultSet rs) throws SQLException{
    //Field variables
    int ssn;
    String name;
    double salary;

    //Ensure we start with first row
    rs.beforeFirst();

    while(rs.next()){
      //Retrieve by column name
      ssn= rs.getInt("ssn");
      name = rs.getString("name");
      salary = rs.getDouble("salary");

      //Display values
      System.out.print("Row Number=" + rs.getRow());
      System.out.print(", SSN: " + ssn);
      System.out.print(", Name: " + name);
      System.out.println(", Salary: $" + salary);
    }
    System.out.println();
  }//end printRs()

}//end UpdateableRs class
```

The output from Listing 6−3 is as follows:

```
Connecting to database...
List result set for reference....
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5544.05
Row Number=2, SSN: 419876541, Name: Larry, Salary: $1663.21
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2218.67
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3415.13
Row Number=5, SSN: 987654321, Name: John, Salary: $4824.42
```

```
List result set showing new salaries
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5837.88
Row Number=2, SSN: 419876541, Name: Larry, Salary: $1751.36
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2336.26
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3596.13
Row Number=5, SSN: 987654321, Name: John, Salary: $5080.11

Goodbye!
```

Inserting and deleting data with updateable result sets

You can also use an updateable ResultSet object to insert and delete rows programmatically using the methods insertRow() and deleteRow().

When inserting a row into the result set you must place the cursor in a staging area known as the *insert row*. This area acts as a buffer until you commit the data to the database with the insertRow() method. To move the cursor the insert row requires using the ResultSet.moveToInsertRow() method call.

Once you position the cursor in the insert row you use the update*XXX*() method to update the column data. In this case, however, you are not updating the information but creating it. Using the *getXXX()* methods after calling the update*XXX*() method will reveal the change because the result set data has changed in the insert row. However, you must call the insertRow() method to commit the changes to the database. The code snippet below demonstrates how to insert a new row into a database using an updateable ResultSet object:

```
//Assume a valid Connection object conn
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);

//build SQL string
String SQL="SELECT ssn, name, salary FROM employees";
ResultSet rs = stmt.executeQuery(SQL);

//Move to insert row and add column data with updateXXX()
rs.moveToInsertRow();
rs.updateInt("SSN",5697529854);
rs.updateString("Name","Rebecca");
rs.updateDouble("Salary",45555.77);

//Commit row
rs.insertRow();
```

Deleting a row from a result set only requires a call to the deleteRow() method. Unlike the other data−manipulation methods I've mentioned thus far, this method affects both the data in the result set and the data in the database simultaneously. Where the cursor moves to after the deleteRow() method depends upon the driver implementation. Some drivers move the cursor forward while others move it backward. You may need to experiment with this method to determine your driver's behavior.

Result Set Hints

Scrollable and updateable result sets are somewhat slower than the standard result set. However, you can supply hints to your ResultSet object to possibly increase speed. Driver vendors do not have to implement these hints. In fact, you may find that the driver is already tuned and these hints hinder, rather than help, performance.

There are two categories of hints. The first deals with fetch size and the other with fetch direction. Here is a summary of each:

- *Fetch size* — This hint sets the number of rows returned from a query. You may want to specify values for this hint if you have network bandwidth issues, such as in a wireless application, or your query retrieves a large number of results and you only need, or are able to work with, a few rows at a time. You set this hint with the Statement.setFetchSize() or ResultSet.setFetchSize() method.
- *Fetch direction* — You can set the default direction for cursor travel within the result set with this hint. You can set the fetch direction to FETCH_FORWARD, FETCH_REVERSE, or FETCH_UNKNOWN. The first is the default setting and moves the cursor forward. The second informs the result set cursor to travel backwards through its data. The third indicates that the direction is unknown. You set this hint with the Statement.setFetchDirection() or ResultSet.setFetchDirection() method.

Summary

In this chapter I covered using ResultSet objects. I explained how the result set cursor moves through the data and how to retrieve values using the ResultSet. get*XXX*() methods. I also covered the three different types of result sets — standard, scrollable, and updateable. I provided examples for each type as well.

The following are some of the more important concepts introduced in this chapter:

- The default result set type is forward−only and non−updateable.
- Scrollable result sets enable you to move forward, backward, and to a specific row in the result set.
- Updateable result sets enable you to update column values for individual rows in a result set, and to insert and delete rows.
- Trying to access data in the “before−first−row” or “after−last−row” areas of a result set throws an SQLException.
- Use the appropriate ResultSet.get*XXX*() and ResultSet.update*XXX*() method for the underlying Java data type or an SQLException occurs.

Chapter 7: Understanding JDBC Data Types

In This Chapter

- Mapping Java data types to JDBC data types
- Mapping JDBC data types to Java data types
- Using the get*XXX*(), set*XXX*(), and update*XXX*() methods
- Using SQL3 data types such as CLOB and BLOB
- Mapping SQL3 data types to Java classes

In the last three chapters, I discussed how to interact with databases using the Connection, Statement, and ResultSet objects. In this chapter, I'll explain the difference between Java data types and SQL data types, and provide examples illustrating how to work with both.

I'll begin by discussing how Java data types map to JDBC data types. You will need this information when binding values to parameters using the PreparedStatement or CallableStatement object's set*XXX*() or update*XXX*() methods. Next I'll cover how JDBC data types map to Java data types when retrieving information from ResultSet objects using the get*XXX*() method. Finally, I'll cover how to use User−Defined Types (UDTs) and present an example containing a custom Java class that maps to a UDT in the database.

Java, Databases, and Data Types

With its ability to create user−defined classes that reference other classes, Java has rich data type support. Databases, on the other hand, support only a limited number of data types. For instance, SQL2 compliant databases, only support basic character, binary, date, and numeric data types. You cannot define your own data types in SQL2 compliant databases as you can in Java.

The SQL3 standard introduces support for custom data types and increases the size of the data you can store in a column. When using SQL3 compliant databases you can also create your own UDTs. In addition, these databases support large binary or character data (more than 1GB) in a database. Database developers now have significantly more flexibility when choosing data types for their applications.

A Brief History of SQL

SQL, or Structured Query Language, is the standard database−access language. It defines how to manipulate and build database tables as well as how to interact with database data. The first standard was produced in 1986 and provided basic language constructs for defining and manipulating tables with data. In 1989, the language was extended to support data integrity with referential and general constraints. SQL2, or SQL92, was adopted in 1992 and provided new data−definition and manipulation enhancements as well as new data types. Improved schema and database administration were also added. Now the new standard, SQL3, extends SQL to support complex objects defined in business modeling and multimedia applications. New extensions include object identifiers, abstract data types and inheritance mechanisms.

Not all databases support SQL3 standards. Implementing the storage of these new data types is challenging. However, as technology progresses, you will soon see more support for SQL3.

Nonetheless, a large disconnect between Java and database data types still exists. The data types among each environment do not coincide. To get your application data into a database you must convert the Java data types to SQL data types. The reverse is true when you retrieve data. In this case, you must convert from SQL data types to Java data types.

JDBC makes these conversions somewhat easier. You convert the data from one type to the other with the get*XXX*(), set*XXX*(), and update*XXX*() methods. The JDBC driver, which is database−specific, knows how to perform these conversions.

Nonetheless, working with two different data types is challenging. With characters you have to deal with fixed and variable−length formats, while with numbers you run the risk of losing precision or scale. Although the JDBC solution is not perfect, it certainly makes interacting with database data types less cumbersome.

Java−to−JDBC Data−Type Mappings

As I mentioned earlier, Java classes are custom data types you define. In addition, the Java language is composed of class and interface definitions. Most of your interaction with Java will take place through objects instantiated from these classes. These classes and interfaces, or data types, form a library known as the Java API.

However, Java has other data types called primitives that do not need defining. Primitives hold data that Java understands directly. These data type definitions remain constant from one Java application to another and from one JVM to another. This feature aids in making Java portable across multiple platforms.

You cannot instantiate primitives into objects. However, Java does define classes, known as wrappers, that treat primitives as objects. Table 7−1 lists the Java primitive data types, their range of values, and wrapper classes. Most often in your JDBC applications you will be trying to insert Java primitives into a database. Understanding these data types and their corresponding wrapper classes will prove useful.

Table 7−1: Java Primitive Data Types and Wrapper Classes

Primitive	Size/Format	Range	Wrapper Class
byte	8−bit signed integer	−128 to 127	java.lang.Byte
short	16−bit signed integer	-2^{15} to $2^{15}-1$	java.lang.Short
int	32−bit signed integer	-2^{31} to $2^{31}-1$	java.lang.Integer
long	64−bit signed integer	-2^{63} to $2^{63}-1$	java.lang.Long
float		IEEE 754 standard	java.lang.Float

	32−bit single−precision floating point		
double	64−bit double−precision floating point	IEEE 754 standard	java.lang.Double
char	Single−character, 16−bit Unicode 2.1 character	n/a	java.lang.Character
boolean	1 bit	true or false	java.lang.Boolean

When you want to place data into the database you must convert it to the DBMS's correct SQL data type. You convert the data types with the set*XXX*() method used by Statement, PreparedStatement, and CallableStatement objects as well as the ResultSet.update*XXX*() method. The *XXX* represents the Java data type.

Behind the scenes the JDBC driver converts the Java data type to the appropriate JDBC type before sending it to the database. It uses a default mapping for most data types. For example, a Java int is converted to an SQL INTEGER. Default mappings were created to provide consistency between drivers. Table 7−2 summarizes the default JDBC data type that the Java data type is converted to when you call the set*XXX*() method of the PreparedStatement or CallableStatement object or the ResultSet.update*XXX*() method.

Table 7−2: JDBC 2.*x* set*XXX()* and update*XXX()* Data Type Mappings

Method	SQL Data Type
setString	VARCHAR, CHAR,
updateString	LONGVARCHAR[1]
setBoolean	BIT
updateBoolean	
setBigDecimal	NUMERIC
updateBigDecimal	
setByte	TINYINT
updateByte	
setShort	SMALLINT
updateShort	
setInt	INTEGER
updateInt	
setLong	BIGINT
updateLong	
setFloat	REAL
updateFloat	
setDouble	DOUBLE
updateDouble	

setBytes	VARBINARY, BINARY, LONGVARBINARY[2]
updateBytes	
setDate	DATE
updateDate	
setTime	TIME
updateTime	
setTimestamp	TIMESTAMP
updateTimestamp	
setClob	CLOB[3]
updateClob	
setBlob	BLOB[3]
setARRAY	ARRAY[3]
SetRef	REF[3]

[1] Driver will use VARCHAR unless the string's length exceeds its size.

[2] Driver will use VARBINARY unless the byte[] length exceeds its size.

[3]SQL3 advanced data type.

JDBC 3.0 JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The ResultSet object now has updateBLOB(), updateCLOB(), updateArray(), and updateRef() methods that enable you to directly manipulate the respective data on the server.

The set*XXX*() and update*XXX*() methods enable you to convert specific Java types to specific JDBC data types. The methods, setObject() and updateObject(), enable you to map almost any Java type to a JDBC data type. You cannot coerce data types into types that do not make sense, however. For example, it makes little sense to try to convert an Integer to a JDBC CLOB. The two are distinctly different. In addition, the methods only work with object types — not with primitives.

For example, suppose you have a data−entry form in which a human−resources manager updates salary information. On the form the user enters the employee's SSN and new salary. The data entered into the form is accepted as a String, but the database requires DOUBLE values. The user can use a Java primitive wrapper and the setObject() method to supply the data to the database, as shown in the following code snippet:

```
//Assume valid Connection object conn.
String SQL="UPDATE employees SET salary=? WHERE ssn = ?";
PreparedStatement pstmt = conn.prepareStatement(SQL);

//String value strSalary holds the salary submitted on a form
Double salary = new Double(strSalary);
pstmt.setObject(1,strSalary);

//String value strSsn holds the SSN submitted on a form
Integer Ssn = new Integer(strSsn);
pstmt.setObject(2,strSsn);
```

Handling Nulls

An SQL NULL represents unknown or undefined data. For example, if an Employee database holds information about an employee's dependents in a table using an SQL INTEGER column, the value could be greater−than or equal to 0, or be empty (NULL). In the case of a NULL, you do not know if the value means that the person has no children or that the value is not known. This is okay for a database, but it poses a

problem for Java, especially for the numeric primitive data types. The Java int type, for example, can't represent a NULL. When you use the ResultSet.getInt() method, JDBC will translate the NULL to 0. Of course this interpretation may be erroneous. Objects, however, *can* represent a NULL, which poses less of a problem to Java provided you handle NullPointerExecption errors as necessary.

You can call the ResultSet.wasNull() method to determine if the last column retrieved contained a NULL value. The method returns true if the value was an SQL NULL, and enables you to make the correct determination.

This is yet another example of why you need to understand your underlying data source. You should be familiar with the data and data types stored in it. It is practically impossible to call the ResultSet.wasNull() method for every get*XXX*() method that returns a Java numeric type.

JDBC−to−Java Data−Type Mappings

Data returned from SQL queries are formatted as JDBC data types. You must convert them to Java types before assigning them to variables. To do so, you use the ResultSet.get*XXX*() method, which returns a value of type XXX.

Two categories of SQL data types exist: the standard data types (that is, SQL92) and the advanced data types (SQL3). The JDBC API defines support for both. The following sections provide information about the data types and how to access them using Java.

Standard SQL data types

Unlike with the set*XXX*() and update*XXX*() methods, you can specify which Java type you want to cast the JDBC type to with the get*XXX*() method. The following code snippet demonstrates converting an SQL INTEGER to a Java double:

```
//Create a ResultSet that retrieve the SQL INTEGER ssn
String SQL = "SELECT Ssn FROM Employees WHERE Name='ToddT'";
ResultSet rs = stmt.executeQuery(SQL);

//Retrieve as a double
double ssn = rs.getDouble(1);
```

Most JDBC−SQL data types can map to numerous Java data types. However, JDBC provides recommendations with regard to JDBC−to−Java data type conversions. Tables 7−3, 7−4, 7−5, and 7−6 list the JDBC types and shows the ResultSet.get*XXX*() method to use for converting each to the corresponding Java data type. These recommendations help ensure the greatest compatibility when working with unknown database systems. However, if you understand your target database and the data stored within it, feel free to cast the data to another type.

Table 7−3: JDBC−to−Java Mappings for Character Data Types

SQLData Type	Recommended getXXX() Method (JDBC 2.1 Specification)	Comments
CHAR	getString	Holds fixed−length character data. CHAR(5) represents a five−character string: If your string is three characters long its size is still five.
VARCHAR	getString	Holds variable−length character data. VARCHAR(5) will house a character string of up to five characters. Unlike with CHAR, if your string is three characters long, its size is three.
LONGCHAR	getAsciiStream	Used for large variable−length strings. DBMS vendors implement this data type in many different ways making getting data in and out of the database with JDBC difficult. A CLOB may be a better choice.

Table 7−4: JDBC−to−Java Mappings for Binary Data Types

SQL Data Type	Recommended get*XXX*() Method (JDBC 2.1 Specification)	Comments
BINARY	getBytes	Represents fixed−length binary data.
VARBINARY	getBytes	Represents variable−length binary data.
LONGVARBINARY	getBinaryStream	Represents multiple−megabyte, variable−length binary data.

Table 7−5: JDBC−to−Java Mappings for Numeric Data Types

SQL Data Type	Recommended getXXX() Method (JDBC 2.1 Specification)	Comments
BIT	getBoolean	One bit of data.
TINYINT	getByte	8−bit integer with values ranging between 0−255. Use Java short for larger TINYINT values.
SMALLINT	getShort	16−bit signed integer. Widely adopted.
INTEGER	getInt	32−bit signed integer. Precision may vary.
BIGINT	getLong	64−bit integer. Not widely implemented.
REAL	getFloat	Represents a single−precision floating−point number. Moderate adoption among database vendors.
DOUBLE	getDouble	Represents a double−precision floating point number. Wide adoption among database vendors.
FLOAT	getDouble	Like JDBC DOUBLE, represents a double−precision floating−point number. Do not confuse with a Java float, which is only single−precision. Widely adopted.
DECIMAL	getBigDecimal	Represents fixed−precision decimal values.
NUMERIC	getBigDecimal	Represents fixed−precision decimal values.

Table 7−6: JDBC−to−Java Mappings for Date and Time Data Types

SQL Data Type	Recommended get*XXX*() Method (JDBC 2.1 Specification)	Comments
DATE	getDate	Represents a day, month, and year. Not widely adopted.
TIME	getTime	Represents an hour, minutes, and seconds. Not widely adopted.
TIMESTAMP	getTimestamp	Represents a day, month, year, hour, minutes, seconds, and nanoseconds. Not widely adopted.

The following sections provide additional details about the information presented in Tables 7−3 through 7−6.

Character

You will find working with character data relatively straightforward. However, two situations may cause unexpected results. First, if you use the ResultSet.getString() method to retrieve a CHAR(*n*) data type, the driver will likely place padding inside into the String because the underlying data type is a fixed−width CHAR. This is normal and you can use the String.trim() method to remove the padding.

Second, avoid using the ResultSet.getString() method to retrieve large LONGVARCHAR data. The resulting String object may be very large and exhaust memory resources on the client's computer. The combination of large data size and network latency may also result in slow downloads that may make using the data on the client impractical.

Always use the ResultSet.getAsciiStream() method when retrieving very large LONGVARCHAR data. The method returns an InputStream object that enables you to control data flow to your client.

In addition, you may find the SQL CLOB data type easier to work with because you have the choice whether to materialize the data on the client.

JDBC 3.0 JDBC 3.0 introduces two new data types, DATALINK and BOOLEAN. The DATALINK type will enable you to reference an SQL DATALINK type. This type provides access to data source outside the database. The JDBC BOOLEAN type will map to an SQL BOOLEAN. A Java boolean is equivalent to the JDBC BOOLEAN, which logically represents a bit.

Numeric

You can divide the numeric category into two parts, integers and floating−point numbers. Integers present few problems. Just use the appropriate get*XXX*() method for the length of the data type you are using. For example, do not try to stuff a 32−bit integer into a 16−bit short or you will lose precision.

However, floating−point numbers may introduce some confusion. A JDBC FLOAT and a Java float do not share the same decimal point precision. A JDBC FLOAT supports a 15−digit, or double−precision. The Java float only supports a single−precision, or up 7 digits. To avoidconfusion, Sun recommends you use JDBC DOUBLE when working with floating point numbers.

Binary

The BINARY data type is similar to CHARACTER data in that you need to worry only about the SQL LONGVARBINARY data type. You can use the ResultSet.getBytes() method to retrieve the data into a byte[] but you risk creating a very large array.

A better idea is to use the ResultSet.getBinaryStream() method, which returns an InputStream object with which you may control the data flow to your client.

Preferably, you will implement an SQL BLOB data type on the server. Accessing it with a JDBC BLOB does not materialize the data on the client, thereby eliminating memory problems associated with creating large arrays.

Date

It's ironic that one of the most common elements to humans, date and time, is also the most inconsistently implemented data type in programming languages, operating systems, and databases. Despite SQL standards for DATE, TIME, and TIMESTAMP data types, all database vendors implement them differently.

So it should not astound you that the java.util.Date class does not match any SQL date and time–related data types. To compensate for this, JDBC has a set of classes map directly to these data types. All of the JDBC date and time–related data types extend the java.util.Date class. Figure 7−1 shows the UML class diagram for these relationships. In addition, all dates are computed as the total milliseconds from the Java epoch, January 1, 1970.

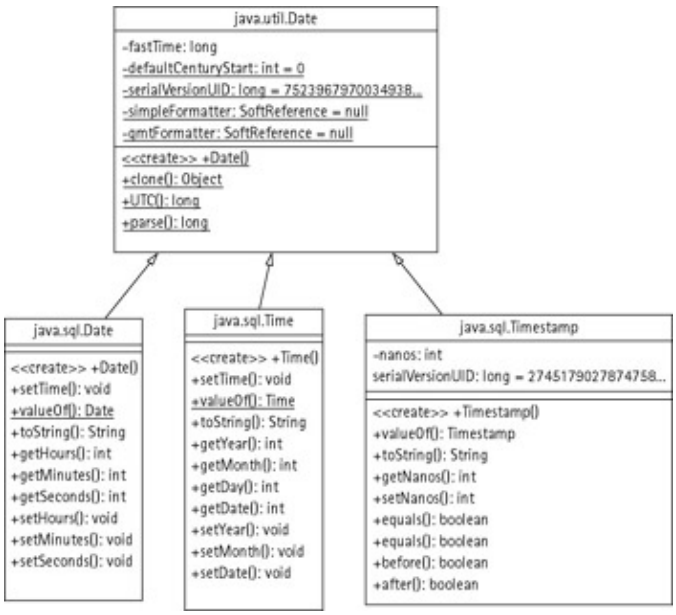

Figure 7−1: UML class diagram for JDBC Date and Time classes

The java.sql.Date class maps to the SQL DATE type, and the java.sql.Time and java.sql.Timestamp classes map to the SQL TIME and SQL TIMESTAMP data types, respectively. Listing 7−1 shows how the Date and Time classes format standard Java date and time values to match the SQL data type requirements.

Listing 7−1: SqlDateTime.java

```
package Chapter7;
```

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.*;

public class SqlDateTime {

  public static void main(String[] args) {
    //Get standard date and time
    java.util.Date javaDate = new java.util.Date();
    long javaTime = javaDate.getTime();
    System.out.println("The Java Date is:
" + javaDate.toString());

    //Get and display SQL DATE
    java.sql.Date sqlDate = new java.sql.Date(javaTime);
    System.out.println("The SQL DATE is:
" + sqlDate.toString());

    //Get and display SQL TIME
    java.sql.Time sqlTime = new java.sql.Time(javaTime);
    System.out.println("The SQL TIME is:
" + sqlTime.toString());

    //Get and display SQL TIMESTAMP
    java.sql.Timestamp sqlTimestamp =
new java.sql.Timestamp(javaTime);
    System.out.println("The SQL TIMESTAMP is:
" + sqlTimestamp.toString());
  }//end main

}//end SqlDateTime
```

The output from Listing 7−1 is as follows:

```
The Java Date is: Sun Mar 11 22:38:55 EST 2001
The SQL DATE is: 2001−03−11
The SQL TIME is: 22:38:55
The SQL TIMESTAMP is: 2001−03−11 22:38:55.163
```

Advanced SQL data types

As computer technology progressed through the 1990s, a need for richer, more advanced data type support in databases arose for two reasons.

First, developers were becoming skilled at modeling complex engineering and business processes using object−oriented programming techniques. However, they had trouble storing data from these object models in relational databases. Only the attributes of the objects could be stored in the database. The developers needed databases to support the user−defined data types so they could mimic the classes in their applications in the database.

Second, multimedia developers began to need databases when their data, sounds, graphics, and videos started appearing on users' computers. By now, thanks to the Internet, most people are accustomed to having rich content provided to them. Suppliers of this content needed somewhere to store these data. File systems don't provide the tools necessary for dynamic distribution of content across the Internet. Databases work better

because they provide advanced searching capabilities and additional data integrity.

SQL3 data types were created to handle the demands of these two groups. The standard defines support for large character and binary data storage as well as for custom UDTs.

JDBC 2.0 introduced support for SQL3 data types. Now it is possible to instantiate Java classes that represent the SQL3 data types then work with the data directly. The java.sql package provides the support for the SQL3 data types BLOB, CLOB, ARRAY, STRUCT, and REF.

The SQL3 data types fall into two categories: predefined and user−defined. The following sections provide an overview of each category along with examples illustrating their use.

Predefined SQL3 data types

Several SQL3 data types are considered predefined. That is, the SQL3 standard defines what the data types represent, much as the Java language predefines primitive data types. The predefined data types include BLOB, CLOB, REF, and ARRAY. This section provides details on these data types.

One interesting feature of the predefined types is that you do not work with the data at the client. You access original data on the server through a logical pointer on the client, called a LOCATOR. As a result, clients do not have to materialize the data on their workstations when using the predefined data types.

Given that some SQL3 data, such as a BLOB, may be quite large, this feature saves you significant download time in addition to minimizing an application's memory footprint. However, you may use the ResultSet.get*XXX*() method to materialize the data on the client when necessary. In all cases, the data remains on the DBMS unless you explicitly materialize it.

Having the ability to materialize the data when you need to gives you a lot of freedom when you're using large SQL3 data types. For example, if a BLOB column stores a file in the database you can look at the header section through a JDBC BLOB to determine if you want the whole file. If you do, you can open an InputStream and place the data into a byte[].

The remainder of this section provides more details on the specific predefined SQL data types.

ARRAY This data type makes it possible to use arrays as data in a column, which enables you to store a group of data within an entity's attribute. For example, you can collect all the grades of type INTEGER for each student in an SQL3 ARRAY. Figure 7−2 illustrates a Student entity table using anSQL3 ARRAY to store grades of SQL type INTEGER.

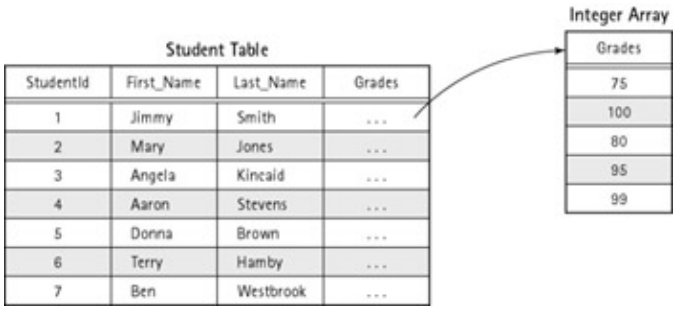

Figure 7−2: Example of SQL ARRAY data type

You can interact with an ARRAY in two different ways. The first is to access the SQL3 ARRAY data on the server by instantiating a java.sql.Array object. This method uses an SQL LOCATOR on the client as a pointer to the data on the server. You may want to use this technique if the dataset is large or if you have no need to materialize the data on the client. Or, if your application is constrained by network bandwidth, as in a wireless application, materializing the data may not be practical.

The second technique is to materialize the data on the client. This is useful if you need to serialize the data to local storage. Be aware that materializing the data means that you retrieve all the data in the SQL ARRAY to the client. Retrieving a dataset that is too large may exhaust the client's memory.

In either instance, you must begin by instantiating an Array object. To do so you use the ResultSet.getArray() method. The method will create an Array object with a logical pointer to the data on the server, as shown in the following code snippet:

```
//Assume a valid Statement object stmt
String SQL = "SELECT Scores FROM Bowlers WHERE Bowler='Benji'";
ResultSet rs = stmt.executeQuery(SQL);


//move to the first record
rs.next();

//Instantiate the Array object
Array bowlingScores = rs.getArray("Scores");
```

Once you have instantiated an Array object, you have eight methods at your disposal for materializing the data on the client — four variations that create a Java array and four that instantiate a JDBC ResultSet object. The variations enable you to specify how much of the data you want. For example, you may only need array elements 10 through 100.

The first Array.getArray() method returns an Object type that holds an array of primitives or an array of objects such as String types or UDTs. As you retrieve each element you must cast it into the underlying Java data type that you want to use. Continuing the bowling example, the following snippet demonstrates how to create an array from anSQL ARRAY:

```
//Create an array to hold the SQL INTEGER values
BigDecimal [] scores = (BigDecimal[])bowlingScores.getArray();

//Loop through the array and print the elements.
for(int i = 0;i<scores.length;i++)
  System.out.println(scores[i].toString());
```

To create a result set, use the Array.getResultSet() method. The ResultSet object you instantiate is forward−only; you cannot create scrollable or updateable result sets with this method. The following is a code snippet that shows you how to use the Array.getResultSet() method to create and use a ResultSet object containing Benji's bowling scores:

```
ResultSet scoreRs = bowlingScores.getResultSet();
while (arrayRs.next())
   System.out.println(arrayRs.getInt(2));
```

XRef Chapter 6, "Working with Result Sets" provides more information on the ResultSet object.

You may have noticed that with both methods, getArray() and getResultSet(), you must know about the underlying data type to properly access the data. To help you with this, the getBaseType() and getBaseTypeName() methods of the JDBC Array object provide you with the underlying JDBC data type for the array elements.

With this information you can build logic into your code to call the correct methods, based on data type, to retrieve the data.

Tip Do not confuse the ResultSet.getArray() method with the java. sql.Array.getArray() method. The first returns a java.sql.Array object, which is a logical pointer to anSQL3 ARRAY data type on the server. It does not contain any data. The former method materializes the data into an array. You must cast the array to the proper data type before using.

CLOB and BLOB data types The JDBC CLOB and BLOB interfaces map to the SQL3 CLOB and BLOB data types, respectively. As with the other predefined types, an SQL LOCATOR is used to point to the data so they are not materialized on the client until you explicitly materialize them.

Since the data may be rather large, both interfaces implement methods that return an InputStream for efficient transfer of data. The CLOB interface provides the getAsciiStream() method, and the BLOB interface the getBinaryStream() method.

Listing 7−2 illustrates how to read and write CLOB and BLOB data. I begin by creating the Connection and Statement objects so I can interact with the database; then I call the createBlobClobTables() method to create the table that holds the BLOB and CLOB data. Next I use an InputStream to read a file from disk and populate the BLOB and CLOB columns. In this example I am using text to represent the binary data so you can verify the output. Once I write the data to the database, I retrieve the same data and materialize it by using an InputStream to populate byte[] and char[] for the BLOB and CLOB data, respectively. Finally, I print the information to the screen, though I could just as easily serialize it to disk.

Listing 7−2: BlobClobEx.java

```
package Chapter7;

import java.sql.*;
import java.io.*;
import java.util.*;

public class BlobClobEx{

  public static void main(String[] args) {

    //Create Connection, Statement, PreparedStatement,
    // and ResultSet objects
    Connection conn = null;
    Statement stmt = null;

    PreparedStatement pstmt = null;
    ResultSet rs = null;

    try{

      //Register driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();
```

```
  //Open database connection
  System.out.println("Connecting to database...");
  String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
  conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

  //Create Statement object to build BLOB and CLOB tables
  stmt = conn.createStatement();

  //Build the tables
  createBlobClobTables(stmt);

  //Create a prepared statement to supply data.
  String SQL = "INSERT INTO BlobClob VALUES(40,?,?)";
  pstmt= conn.prepareStatement(SQL);

  //Load BLOB column
  File file = new File("blob.txt");
  FileInputStream fis = new FileInputStream(file);
  pstmt.setBinaryStream(1,fis,(int)file.length());

  //Load CLOB column
  file = new File("clob.txt");
  fis = new FileInputStream(file);
  pstmt.setAsciiStream(2,fis,(int)file.length());
  fis.close();

  //Execute statement
  pstmt.execute();

  //Retrieve the data
  SQL = "SELECT * FROM BlobClob WHERE id = 40";
  rs = stmt.executeQuery(SQL);
  //Move to the first row
  rs.next();

  //Instantiate blobs and clobs
  java.sql.Blob blob = rs.getBlob(2);
  java.sql.Clob clob = rs.getClob(3);

  //Materialize the BLOB data and print it out.
  byte blobVal [] = new byte[(int)blob.length()];
  InputStream blobIs = blob.getBinaryStream();
  blobIs.read(blobVal);
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  bos.write(blobVal);
  System.out.println(bos.toString());
  blobIs.close();

  //Materialize the CLOB data and print it out.
  char clobVal[] = new char[(int)clob.length()];
  Reader r = clob.getCharacterStream();
  r.read(clobVal);
  StringWriter sw = new StringWriter();
  sw.write(clobVal);
  System.out.println(sw.toString());
  r.close();

//Standard error handling
} catch(SQLException se) {
  //Handle errors for JDBC
  se.printStackTrace();
```

```
    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");

  }//end main

  public static void createBlobClobTables(Statement stmt)
throws SQLException{
    //Create SQL Statments
    String Sql="CREATE TABLE BlobClob(Id NUMBER(3),
b BLOB, c CLOB)";

    try{
      stmt.executeUpdate("DROP TABLE BlobClob");
    }catch(SQLException se){
      //Handle DROP table error. I could just give status message.
      if(se.getErrorCode()==942)
      System.out.println("Error dropping BlobClob table:
" + se.getMessage() );
    }

    //Build the Blob_Ex table
    if(stmt.executeUpdate(Sql)==0)
      System.out.println("BlobClob table created...");

  }// end of createTables method

}//end BlobClobEx
```

The output from Listing 7−2 is as follows:

```
Connecting to database...
BlobClob table created...
BLOB Data:
From Poor Richard’s Almanac:
Necessity never made a good bargain.

CLOB Data:
From Poor Richard’s Almanac:
The worst wheel of the cart makes the most noise.
```

REF An SQL REF is a logical "pointer" to a UDT, usually to a STRUCT (see next section), in a database table. Like a LOCATOR the SQL REF provides indirect access to the object. However, the SQL REF type resides on the server, unlike the LOCATOR, which resides on the client.

JDBC REF data types enable you to interact with the SQL REF type on the database. Like CLOB and BLOB data, the underlying data on the server are not materialized on the client. If necessary you can de−reference

the SQL REF object and materialize it within your application using the ResultSet.getRef() method.

Note You need to fully understand how SQL REF data types work within your DBMS before implementing them. Complete implementation details, however, are beyond the scope of this book.

JAVA_OBJECT The JDBC JAVA_OBJECT type enables you to store Java classes directly in a database. Sun has a vision for Java — relational database systems that can store Java objects natively. Database vendors are not required to provide support for this data type.

The Java database is part of a new breed of database called Object Database Management Systems (ODBMS). These databases are intended to work directly with object−oriented programming languages by combining the elements of object orientation and object−oriented programming within a database system.

The advantage of object−relational databases is that they conceptually eliminate the need to translate between Java data types and SQL data types. You simply store and retrieve your objects from the ODBMS as you need them.

SQL3 user−defined types

SQL3 UDTs enable database developers to create their own type definitions within the database. The types are defined with SQL statements and have nothing to do with Java. However, JDBC provides data types that enable you to access database UDTs from your Java application.

Unlike the predefined data types, these types are materialized on the client as your application accesses them. You work with these values directly and do not use anSQL LOCATOR that references the values in the databases. Be aware that some UDTs may store large amounts of data. This section provides an overview of the different UDTs and how to use them in your programs.

DISTINCT A DISTINCT data type enables you to assign a custom name to a new data type based on another data type. This is analogous to extending a Java class in that the new class, or type, is based on an existing type.

These data types enable you to give data type a name that makes sense to you and other developers. It also ensures that the data type will always have certain attributes, such as character length for character data or precision for numeric data types.

Here are two examples of the SQL syntax used to create DISTINCT data types:

```
CREATE TYPE Salary NUMERIC(9,2)
CREATE TYPE EmployeeName CHAR(20)
```

In the first example, a data type Salary is defined as a NUMERIC type with a precision (total number of digits) of nine and a scale of (number to the right of the decimal point) of two.

In the second example, a data type called EmployeeName is defined as a CHAR type that is always 20 characters long. If you store the name Paige in the type EmployeeName, letters occupy five of the 20 characters and the remaining characters are empty.

As with the other data types, you can use the set*XXX*() and get*XXX*() methods from the ResultSet object to retrieve the data stored in these variables. Refer to Table 7−3, which provides the default data type mappings to use.

Tip String values retrieved from CHAR(*n*) data types will have padding for non−character data. You can use the String.trim() method to remove the padded space after calling the ResultSet.getString() to load the data into the String variable.

STRUCT A structure is a custom data type that has one or more members, called attributes, each of which may have different data types. These data types are used to group associated data together. A Java class without methods is analogous to a STRUCT data type.

Within databases that support SQL3 types you can define STRUCT data types, that have attributes of any SQL data type including other STRUCT types. Once you define the data type, use it just as you would any other. Figure 7−3 illustrates a table that uses the STRUCT data type defined in the following SQL code snippet:

```
CREATE TYPE EMP_DATA(
 SSN Number(9),
 FirstName VARCHAR(20),
 LastName VARCHAR(20),
 Salary NUMBER(9,2)
)
```

STRUCT data types provide a very powerful tool for the database developer, because they enable you to model basic object systems with custom data types. As a Java developer you are probably most concerned with accessing the data stored in SQL STRUCT data types, not creating the data model within the database. However, you should fully understand the model before implementing your solution.

Listing 7−3 shows you how to interact with STRUCT data types. The sample application accesses the EMP_DATA structure defined in the previous SQL code snippet and prints the data contained within it.

The following two lines in Listing 7−3 warrant special attention:

```
emp_struct = (Struct) rs.getObject("Emp_Info");
emp_attributes = emp_struct.getAttributes();
```

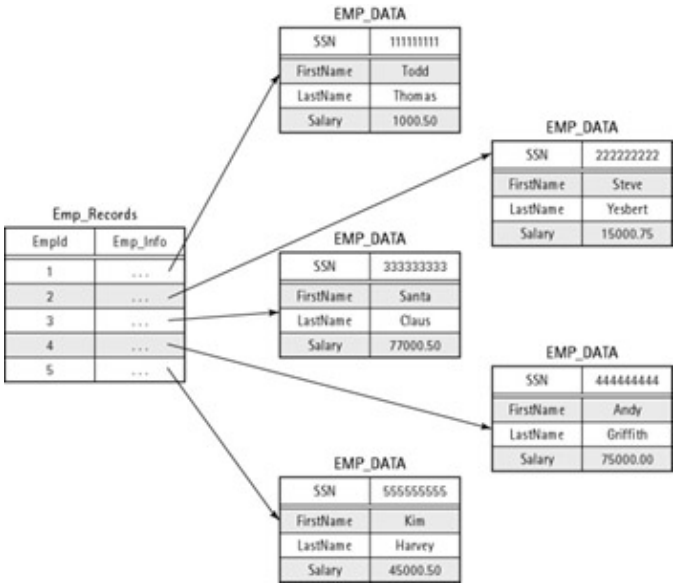

Figure 7−3: Example of an SQL STRUCT data type

The first line casts the Object returned from the ResultSet.getObject() method to JDBC STRUCT. The next line populates an Object[] with the STRUCT attributes. Each element maps to the STRUCT type attributes. After this all I do is retrieve the values from the object array and display them to the screen.

Listing 7−3: StructEx.java

```
package Chapter7;

import java.sql.*;
import java.io.*;
import java.util.*;
import java.math.BigDecimal;

public class StructExample {

  public static void main(String[] args) {
    //Create Connection,Statement, and ResultSet objects
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    try{
      //Load a driver with Class.forName.newInstance()
      Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();

      //Use the getConnection method to obtain a Connection object
      System.out.println("Connecting to database...");
      String jdbcUrl ="jdbc:oracle:thin:@localhost:1521:ORCL";
      String user = "toddt";
      String pwd = "mypwd";
      conn = DriverManager.getConnection(jdbcUrl,user,pwd);

      //Initialize the Statement object
      stmt = conn.createStatement();

      //Build the tables and Types
      createTables(stmt);

      System.out.println("Retrieving data from database....");
      //Retrieve the data
      rs = stmt.executeQuery("Select * from Emp_Records");

      //Declare variables to hold data
      int empId;
      Struct emp_struct;
      Object [] emp_attributes;

      //Loop through ResultSet
      while(rs.next()){
        //Retrieve data from RecordSet
        empId = rs.getInt("EmpId");
        emp_struct = (Struct) rs.getObject("Emp_Info");
        emp_attributes = emp_struct.getAttributes();

        //Populate Java variables with STRUCT data
        BigDecimal empSsn = (BigDecimal)emp_attributes[0];
        String empFirstName = (String)emp_attributes[1];
        String empLastName = (String)emp_attributes[2];
        BigDecimal empSalary = (BigDecimal)emp_attributes[3];
```

```
      //Display results
      System.out.print("Employee Id: " + empId
           + ", SSN: " + empSsn.toString());
      System.out.print(", Name: " + empFirstName
           + " " + empLastName);
      System.out.println(", Salary: $" + empSalary);
    }

  //Standard error handling
  } catch(SQLException se) {
    //Handle errors for JDBC
    se.printStackTrace();

  } catch(Exception e) {
    //Handle errors for Class.forName
    e.printStackTrace();

  } finally {
    try {
      if(conn!=null)
        conn.close();
    } catch(SQLException se) {
      se.printStackTrace();
    }//end finally try
  }//end try
  System.out.println("Goodbye!");

}//end main

public static void createTables(Statement stmt) throws SQLException{

  System.out.println("Starting to create UDT and build table....");
  //Drop the table. Ignore the exception if TYPE does not exist
  try{
    stmt.executeUpdate("DROP TABLE Emp_Records");
  }catch(SQLException se){
    //Ignore the exception
  }

  //Drop the type. Ignore the exception if TYPE does not exist
  try{
    stmt.executeUpdate("DROP TYPE Emp_Data");
  }catch(SQLException se){
    //Ignore the exception
  }

  //Build the String to create the TABLE.
  String createType = "CREATE TYPE EMP_DATA AS OBJECT " +
                  "(SSN Number(9), " +
                  "FirstName VARCHAR(20), lastName VARCHAR(20), " +
                  "Salary NUMBER(9,2))";
  //Submit the update statement
  stmt.executeUpdate(createType);

  //Build the String to create the TABLE.
  String createTable="CREATE TABLE Emp_Records(EmpId number(3),"
                  + "Emp_Info EMP_DATA)";

  //Submit the update statement
  stmt.executeUpdate(createTable);
```

```
    //Insert some data
    stmt.executeUpdate("INSERT INTO Emp_Records VALUES"
                    + "(1,Emp_Data(111111111,'Todd','Thomas',1000.50))");
    stmt.executeUpdate("INSERT INTO Emp_Records VALUES"
                    + "(2,Emp_Data(222222222,'Steve','Yesbert',15000.75))");
    stmt.executeUpdate("INSERT INTO Emp_Records VALUES"
                    + "(3,Emp_Data(333333333,'Andy','Griffith',75000.00))");
    stmt.executeUpdate("INSERT INTO Emp_Records VALUES"
                    + "(4,Emp_Data(444444444,'Santa','Claus',77000.50))");
    stmt.executeUpdate("INSERT INTO Emp_Records VALUES"
                    + "(5,Emp_Data(555555555,'Kim','Harvey',45000.50))");

    System.out.println("Finished creating database structures...");
  } // end of createTable method

}// end StructExample
```

The output from Listing 7−3 is as follows:

```
Connecting to database...
Starting to create UDT and build table....
Finished creating database structures...
Retrieving data from database....
Employee Id: 1, SSN: 111111111, Name: Todd Thomas, Salary: $1000.50
Employee Id: 2, SSN: 222222222, Name: Steve Yesbert, Salary: $15000.75
Employee Id: 3, SSN: 333333333, Name: Andy Griffith, Salary: $75000
Employee Id: 4, SSN: 444444444, Name: Santa Claus, Salary: $77000.50
Employee Id: 5, SSN: 555555555, Name: Kim Harvey, Salary: $45000.50
Goodbye!
```

Custom Data Type Mapping

JDBC enables you to create Java classes to mirror UDTs on the database server. This enables you to call the ResultSet.getObject() method to instantiate a Java class that represents the UDT on the server. The new class contains the attribute values of the UDT on the database and you can use it just as you would any other Java class.

The process of creating and using a Java class for the database UDT is known as type mapping. You may find that creating your own classes to represent a UDT has many advantages. For example, you can control access to the data within the class. If you want to protect data, such as salary information, you can make the data private in the class definition. Another advantage of type mapping is that it enables you to add additional methods or attributes to the class to provide more functionality. You can even define separate classes, which provide different functionality based on the situation, that map to the same UDT.

Note Some companies provide tools to help you create custom mappings and Java classes for database UDTs. For example, Oracle has JPublisher to map UDTs to Java classes.

Building custom data type maps

Building a custom type mapping is a two−step process. The first step is to define a class to represent the UDT.

The class must implement the SQLData interface, which defines methods that allow the driver to retrieve and update the class or UDT attributes. As you will see shortly, implementing these methods is trivial.

The second step is to map the UDT to the Java class by placing an entry into the Connection object's type map. The map is a java.util.Map object that relates a key (the UDT's name) to a value (your custom Java class).

When you call the ResultSet.getObject() method the JDBC driver references the Connection object's map. If an entry is found for the UDT being requested, the class definition is used to instantiate an object to represent the UDT. Otherwise the driver uses a default class definition to handle the UDT.

The rest of this section provides more information about the two steps required to map a custom Java class to an SQL STRUCT. The next section uses the class and map I create in a complete example.

Step 1: Define a custom class

Before creating your Java class to represent the UDT, you must know the exact specification for the database type. For this step I will work with the following STRUCT type:

```
create type EMP_DATA AS OBJECT
     (SSN Number(9),
     FirstName VARCHAR(20),
     LastName VARCHAR(20),
     Salary NUMBER(9,2)
     );
```

The definition for the UDT is Oracle−specific. Each DBMS has its own constructs for creating custom types. This STRUCT defines a data type, EMP_DATA, to hold an employee's Social Security number, first name, last name, and salary.

Based on the EMP_DATA type, I create my Employee class that implements the SQLData interface. Listing 7−4 provides the class definition.

Listing 7−4: Employee.java

```
package Chapter7;

import java.sql.*;
import java.math.BigDecimal;
import java.text.NumberFormat;

public class Employee implements SQLData{

  //UDT's attributes
  public BigDecimal SSN;
  public String FirstName;
  public String LastName;
  public BigDecimal Salary;

  //Needed to identify UDT on database
  private String sqlUdt;

  //Required per SQLData interface
  public void writeSQL (SQLOutput stream) throws SQLException{
    stream.writeBigDecimal(SSN);
```

```
    stream.writeString(FirstName);
    stream.writeString(LastName);
    stream.writeBigDecimal(Salary);
  }

  //Required per SQLData interface
  public String getSQLTypeName() throws SQLException{
    return sqlUdt;
  }

  //Required per SQLData interface
  public void readSQL (SQLInput stream, String typeName) throws SQLException{
    sqlUdt = typeName;
    SSN = stream.readBigDecimal();
    FirstName = stream.readString();
    LastName= stream.readString();
    Salary = stream.readBigDecimal();
  }

  //Custom method to calculate monthly salary rate
  public String calcMonthlySalary(){
      double monthlySalary = Salary.doubleValue()/12;

      //Format as currency to display
      NumberFormat nf = NumberFormat.getCurrencyInstance();
      String str = nf.format(monthlySalary);
      return str;
  }

}//end Employee class
```

Notice the private field sqlUdt in the class definition. This String value holds the SQL type name. In the example the variable equals the name of the UDT, EMP_DATA.

I implemented the getSQLTypeName(), readSQL(), and writeSQL() methods to meet the SQLData interface's definition. The first returns the SQL type defined by sqlUdt. The JDBC driver uses this method to determine the Java class to map the UDT to.

The next method, readSQL(), performs two functions. First, it sets the value of the sqlUDT variable for the type being retrieved. Second, it populates the class instance variables using an SQLInput object with corresponding data from the UDT in the database. The SQLInput object is an InputStream that retrieves the values from the database. The read*XXX*() methods act just like the ResultSet.get*XXX*() methods with respect to data type conversions.

The writeSQL() method populates the UDT on the database with information from your class. This method is implemented much like the readSQL() method except that the SQLOutput object, an OutputStream, supplies the UDT on the database with values from your class variables.

I also implemented the getMonthlySalary() method. This is a "value−add" method that illustrates how you can add functionality to your custom class. This method calculates an employee's monthly pay rate and returns the value as a currency−formatted String.

Step 2: Create the map entry

Once you define your class, you need to inform the JDBC driver to use it whenever the getObject() method retrieves the EMP_DATA UDT from the server. Making the map entry is simple, as the following code snippet demonstrates:

```
//Assume a valid Connection object conn
Map map = conn.getTypeMap();
map.put("EMP_DATA",Class.forName("Employee"));
conn.setTypeMap(map);
```

These operations create a Map object, called map, and inserts an entry into it that links the EMP_DATA type in the database to the custom class Employee on the client. It also sets the default type map for the Connection object to the map object you create. Whenever you use the ResultSet.getObject() method the driver references the map to determine if a custom class exists for the UDT. If it does, the driver instantiates an object, otherwise it creates its own.

Using custom mapping

Now I'll use the custom class and map I created in the previous section to retrieve data from a UDT on the server. The database structures are designed to work with an Oracle 8.1.7 database, but the JDBC concepts remain the same regardless of your database.

Listing 7−5 is the complete application. The first thing I do is create the EMP_DATA type and a table and populate the table with sample data. Next I map the EMP_DATA type to my custom class Employee using the map.put() method. The previous subsection, "Step 2: Make the map entry," provides more details about this operation.

Listing 7−5: SqlDataEx.java

```
package Chapter7;

import java.sql.*;
import java.io.*;
import java.util.*;
import java.math.*;

public class SqlDataEx{

  public static void main(String[] args) {
    //Declare Connection, Statement, and ResultSet objects
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    try{
      //Load a driver with Class.forName.newInstance()
      String driver ="oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Use the getConnection method to obtain a Connection object
      System.out.println("Connecting to database...");
      String jdbcUrl ="jdbc:oracle:thin:@myserver:1521:ORCL";
      String user = "toddt";
      String pwd = "mypwd";
      conn = DriverManager.getConnection(jdbcUrl,user,pwd);
```

```
      //Create Statement object
      stmt = conn.createStatement();

      //Map the EMP_DATA UDT to the Employee class
      Map map = conn.getTypeMap();
      map.put("EMP_DATA",Class.forName("Chapter7.Employee"));
      conn.setTypeMap(map);

      //Rebuild tables with the StructExample.createTables() method
      StructExample.createTables(stmt);

      //Retrieve the data
      System.out.println("Retrieving data from database....");
      rs = stmt.executeQuery("SELECT * from Emp_Records");

      //Custom class to hold EMP_DATA UDT on database
      Employee employee;

      //Loop through ResulSet to and display data
      System.out.println("Displaying data:");
      while(rs.next()){
        int empId = rs.getInt("EmpId");

       //Notice the cast to Employee
       employee = (Employee)rs.getObject("Emp_Info");

       System.out.print("Employee Id: " + empId
          + ", SSN: " + employee.SSN);
       System.out.print(", Name: "
          + employee.FirstName  + " " + employee.LastName);
       System.out.println(", Yearly Salary: $" + employee.Salary
          + " Monthly Salary: " + employee.calcMonthlySalary());
      }

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();
    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try

    System.out.println("Goodbye!");

  }//end main

}//end SqlDataEx
```

The output from Listing 7−5 is as follows:

```
Connecting to database...
Starting to create UDT and build table....
Finished creating database structures...
Retrieving data from database....
Displaying data:
Employee Id: 1, SSN: 111111111, Name: Todd Thomas,
Yearly Salary: $1000.50 Monthly Salary: $83.38
Employee Id: 2, SSN: 222222222, Name: Steve Yesbert,
Yearly Salary: $15000.75 Monthly Salary: $1,250.06
Employee Id: 3, SSN: 333333333, Name: Andy Griffith,
Yearly Salary: $75000 Monthly Salary: $6,250.00
Employee Id: 4, SSN: 444444444, Name: Santa Claus,
Yearly Salary: $77000.50 Monthly Salary: $6,416.71
Employee Id: 5, SSN: 555555555, Name: Kim Harvey,
Yearly Salary: $45000.50 Monthly Salary: $3,750.04
Goodbye!
```

Once the mapping is complete, I query the database to retrieve all the information from the Employee table. Next I retrieve the EmpId and UDT from the result set. I use the ResultSet.getObject() method to get the UDT and cast the object to type Employee, my custom class.

I can now use the Employee class as another Java class. In this example I list all the relevant information stored in the UDT, and the monthly salary rate returned from my Employee.getMonthySalary() method. In the preceding example, the method returns a String that is formatted as $*xxx.xx*.

Summary

This chapter covered handling data type issues in Java database programming. The data types used by Java are fundamentally different from those used in any DBMS. However, JDBC provides the get*XXX*(), set*XXX*(), and update*XXX*() methods with which you can convert between Java types and SQL types. I also presented the recommended mappings between Java data types and JDBC types, as well as how to map a class to a UDT on the database server.

Chapter 8: Mining Database Metadata with JDBC

In This Chapter

- Understanding what metadata is and how it applies to database programming
- Knowing when to use metadata in your applications
- Using Java's DatabaseMetaData interface to collect information about a database
- Using the ResultSetMetaData interface to examine the column properties within a result set

In the simplest terms, metadata is data about data. Applied to databases, metadata is information about managed data, or about the structures and applications that hold managed data. Examples of metadata are descriptions of the tables and column attributes, descriptions of data structures, or descriptions of the actual data (such as its data types).

The JDBC API enables you to uncover metadata about a database and a query's result set using the DatabaseMetaData and ResultSetMetaData interfaces, respectively. The first interface enables you to obtain information about your database's attributes and make runtime decisions based around that information. The interface even provides enough information to enable you to write an entire database−management program.

The second interface enables you to determine the attributes — such as number of columns, names, and data types — for a result set. You can use this information to populate column headers in a report or to determine the correct get*XXX*() method to call.

This chapter details how to obtain metadata for your database and result sets using the two interfaces. I start by introducing the two interfaces and give examples on how you may use them in an application. Next, I describe the ResultSetMetaData interface then move to the DatabaseMetaData interface. Each section also contains examples on how to use the respective interface.

The JDBC Metadata Interfaces

As mentioned, JDBC has two interfaces, ResultSetMetaData and DatabaseMetaData, that supply you with metadata. The first provides information about the columns in a result set, such as the name, data type, and maximum length. The second provides data about a database's structure, such as the table names, primary and foreign keys, and data types.

A ResultSetMetaData object is useful when you want to create a generic method with which to process result sets. By using metadata you can determine the data types of the result set columns and call the correct get*XXX*() method to retrieve the data. A reporting application is a perfect example of an instance in which you might consider a generic result set processing routine. These applications generally require you to submit a query, retrieve a result set, and print the data. Your generic method would enable you to pull the data from any result set for printing. You can extract the column name from a ResultSetMetaData object to create column headings for the report.

You can use DatabaseMetaData when you know a lot about the structure of a database or when you know nothing at all. For example, you may want to create a tool that enables database administrators to inspect or manage databases. Developers may even use the tool to investigate data types or table structures, list user

schemas, or look at a database's supported features. As a result, your administration tool must be able to query the database and determine this information dynamically.

In addition, you can use a DatabaseMetaData object to probe the database to determine its attributes. You'll want to do this when you do not have the luxury of knowing anything about the databases your application will access. Here's an example. Suppose you are writing an installation routine that creates support tables in a database for use by an application. To do this you will need to know about the supported data types within the target database. A DatabaseMetaData object enables you to gather that information so you can build your tables with the correct data types.

I have just scratched the surface with regard to type of applications with which you can use the metadata interfaces. As you become more familiar with the interfaces, I'm sure more ideas will surface. The remainder of the chapter presents the details of the two interfaces and example applications for both.

JDBC 3.0 JDBC 3.0 defines a new metadata interface, ParameterMetaData. This interface describes the number, type, and properties of parameters used in prepared statements.

The ResultSetMetaData Interface

The ResultSetMetaData interface provides descriptive information about the columns in a result set such as the number of columns it contains or each column's data type. The interface does not provide information regarding the database or the number of rows in the result set, only about the result set with which it is associated.

Creating ResultSetMetaData objects

The ResultSetMetaData object is instantiated from a valid ResultSet object. The following code snippet creates a ResultSetMeta object, rsmd, that contains the column metadata for the entire Employees table:

```
//Assume a valid connection conn
Statement stmt = conn.createStatement();

//Create a result set
ResultSet rs = stmt.executeQuery("SELECT * FROM Employees");

//Obtain the result set metadata
ResultSetMetaData rsmd = rs.getMetaData();
```

Using ResultSetMetaData objects

The ResultSetMetaData interface provides you with numerous methods for retrieving information about the result set. You call the various setter and getter methods to retrieve data from the ResultSetMetaData object. Table 8−1 lists the methods I find the most useful.

Table 8−1: Useful ResultSetMetaData Methods

Method Name	Description
getTableName()	Returns the name of the table you queried. Returns a String.
getColumnCount()	Returns the number of columns in the result set. Returns an int.
getColumnName(int *n*)	Returns the name of a column at position *n* in the result set. Returns a String.
getColumnType(int *n*)	Returns the JDBC data type for a column at position *n* of the result set. Returns an int.
getColumnTypeName(int *n*)	Provides the name of the column's data type as defined by the database. The parameter *n* is the column position within the result set. Returns a String.

The getColumnCount() method returns the number of columns returned by the result set. The returned value will equal the number of columns that you requested in your SELECT statement. If you selected all the columns with the following SQL query:

```
SELECT * FROM tableX
```

then the getColumnCount() method will return the number of columns in the table. You will find this method handy when you want to determine the upper−bounds of a counter variable when you loop through all the columns in a result set and look at their metadata. As you might expect, the getColumnName() method returns the column name at a specific position. The order of the columns will be either the order in which you requested them or, if you requested all the columns, the order of the table definition.

Two other useful ResultSetMetaData methods are the getColumnType() and getColumnTypeName() names. The first returns an int that represents the JDBC data type defined in java.sql.Types. The next method provides the SQL data type name as defined in the database. Do not confuse the database data type name with the JDBC or Java data−type name, as they may be different. These methods enable you to write a generic routine to extract result set data by first checking the type or type name of the column and then using a switch block to call the correct getXXX() method.

XRef See Chapter 7, "Understanding JDBC Data Types," for more information on data types and how they relate to JDBC programming.

The methods that require the column position as a parameter use the columns ordinal position in the result set. The positions start with the number 1, not 0 like Java arrays. Using a 0 as a parameter throws an SQLException.

Note ResultSetMetaData.getColumnTypeName() method returns the data−type name used by the database server. You may find that this value differs from JDBC's data−type name for the equivalent database type.

ResultSetMetaData example

I can best illustrate the ResultSetMetaData object with an example. In Listing 8−1, I retrieve and list both the JDBC and SQL data types for the columns in the

Employees and Location tables. In addition, I create a method that enables you to retrieve data from the result set based on the data type of a result set column. The method illustrates how you can create a generic routine to process a result set using metadata. In fact, I use the method to list the contents of the Employees and Location tables.

Listing 8−1: RSMetadata.java

```
package Chapter8;

//Specific imports
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.ResultSetMetaData;
import java.math.BigDecimal;

public class RSMetaData {

  //Global string buffer to buffer Strings before printing
  public static StringBuffer strbuf = new StringBuffer();

  public static void main(String[] args) {

    //Declare Connection, Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Register driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open database connection
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      //Create a Statement object
      stmt = conn.createStatement();
      rs = stmt.executeQuery("SELECT * FROM Employees");
      System.out.println("Meta data for Employees table:");

      //List column details
      System.out.println("Column Information:");
      printColumnInfo(rs);
      System.out.println();

      //List result set data with generic method
      System.out.println("List table data:");
      printColumnNames(rs);
      processRs(rs);
      System.out.println();

      //Create a result set of data from the Location table
      rs = stmt.executeQuery("SELECT * FROM Location");
      System.out.println("Meta data for Location table:");
      System.out.println("Column Information:");
      printColumnInfo(rs);
      System.out.println();

      //List table data with generic method
```

```
      System.out.println("List table data:");
      printColumnNames(rs);
      processRs(rs);
      System.out.println();

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");

  }//end main

  //Generic method to retrieve data from a result set
  public static void processRs(ResultSet rs)
throws SQLException{

    //Create ResultSetMetaData object
    ResultSetMetaData rmd = rs.getMetaData();

    //Loop through result set and retrieve column information.
    while(rs.next()){

      for (int col=1;col<=rmd.getColumnCount();col++)
        getData(rs,rmd.getColumnType(col),col);

      System.out.println(strbuf.toString());
      strbuf = new StringBuffer();

    }//end while
  }//end processRs()

  //Prints column names as headings
  public static void printColumnNames(ResultSet rs)
throws SQLException{

    ResultSetMetaData rmd = rs.getMetaData();
    StringBuffer sb = new StringBuffer();

    for (int col=1;col<=rmd.getColumnCount();col++)
      sb.append(rmd.getColumnName(col) + " ");

    System.out.println(sb.toString());

  }//end printColumnNames()

  //Method to determine the data type of a result set column
  //then call the correct getXXX() method. For brevity, not all
```

```
  //JDBC data types are supported
  public static void getData(ResultSet rs, int type, int colIdx)
                                        throws SQLException{

    //Temporary variable to hold value from result set as a string.
    String s;

    switch (type){

      //Handle character related data types
      case java.sql.Types.CHAR:
      case java.sql.Types.VARCHAR:
        s = rs.getString(colIdx);
        strbuf.append(s + " ");
        break;

      //Handle the INTEGER data type
      case java.sql.Types.INTEGER:
        int i = rs.getInt(colIdx);
        strbuf.append( i + " ");
        break;

      //Handle the NUMERIC data type
      case java.sql.Types.NUMERIC:
        BigDecimal bd = rs.getBigDecimal(colIdx);
        s = bd.toString();
        strbuf.append(s + " ");
        break;

      //Handle date related data types. I just want the date
      //portion so I combine the two data types.
      case java.sql.Types.TIMESTAMP:
      case java.sql.Types.DATE:
        java.sql.Date d = rs.getDate(colIdx);
        s = d.toString();
        strbuf.append(s + " ");
        break;

    }//end select
  }//end getData()

  //Method to print column metadata.
  public static void printColumnInfo(ResultSet rs)
throws SQLException{

    ResultSetMetaData rsmd = rs.getMetaData();

    System.out.println("Column, " + "JDBC_Type,
"+ "Database_Type_Name ");

    int cols = rsmd.getColumnCount();
    for (int colIdx=1;colIdx<=cols;colIdx++){

      String name = rsmd.getColumnName(colIdx);
      int type = rsmd.getColumnType(colIdx);
      String typeName = rsmd.getColumnTypeName(colIdx);
      System.out.println(name + ", " + type + ", " + typeName);

    }//end for
  }// end printColumnInfo
```

```
}//end RSMetaData class
```

The following is the output from Listing 8−1:

```
Connecting to database...

Metadata for Employees table:
Column Information:
Column, JDBC_Type, Database_Type_Name
SSN, 2, NUMBER
NAME, 12, VARCHAR2
SALARY, 2, NUMBER
HIREDATE, 93, DATE
LOC_ID, 2, NUMBER

List table data:
SSN NAME SALARY HIREDATE LOC_ID
111111111 Todd 100000.75 1989−09−16 100
419876541 Larry 1500.75 2001−03−05 200
312654987 Lori 2000.95 1999−01−11 300
123456789 Jimmy 3080.05 1997−09−07 400
987654321 John 4351.27 1996−12−31 500
537642256 Andy 1400.51 2001−04−23 400

Metadata for Location table:
Column Information:
Column, JDBC_Type, Database_Type_Name
LOC_ID, 2, NUMBER
LOCATION, 12, VARCHAR2

List table data:
LOC_ID LOCATION
100 Knoxville
200 Atlanta
300 New York
400 L.A.
500 Tulsa

Goodbye!
```

The DatabaseMetaData Interface

With the DatabaseMetaData interface you can inspect a database’s structure. The interface has more than one hundred methods that return information about a database.

At some point in your career as a Java database programmer you will need to use the DatabaseMetaData interface. If you are writing a database tool, such as a schema documenter, DBA tool, or installation routine, you will spend a lot of time with this interface.

Creating DatabaseMetaData objects

A Connection object represents a database connection and also instantiates a DatabaseMetaData object with the getMetaData() method. The DatabaseMetaData object holds information for the database to which the Connection object is connected. The following code snippet illustrates how to create a DatabaseMetaData object:

```
//Assume a valid Connection conn
DatabaseMetaData dmd = conn.getMetaData();
```

Before creating the object you must log onto the database and have a valid connection.

Using DatabaseMetaData objects

The DatabaseMetaData object has many methods and properties that provide a lot of information about a database. In fact, when getting started with the DatabaseMetaData interface you may find the number of methods overwhelming. However, you can arrange them into two categories to help put them into perspective. The first category deals with database characteristics and the second with database structures.

The remaining sub−sections provide information about each category and examples of how to use the methods to obtain database information.

Database characteristic methods

As mentioned, with a DatabaseMetaData object you can obtain a lot of information about your database characteristics. For example, you can answer questions like:

- Does the database support batch updates?
- What are the SQL keywords?
- What user am I connected as?
- What SQL data types does this database support?

The methods that provide database characteristics can be further broken down into those that provide general database information, those that provide information about database limits, and those that provide information about supported features.

The methods that provide general information about a database return String values. Examples of information you can obtain are schema names, database names, SQL keywords, and database functions. These methods return either single or multiple values. Methods such as getDatabaseName() returns a single String value. Multiple String values are returned in a comma−delimited list. The getSQLFunction() is an example of a method that returns a comma−delimited list of values.

Methods that provide information about a database's limits return an int. For instance, the methods that retrieve the maximum number of connections, maximum column−name length, and maximum row size return the results as an int.

When you need to determine whether or not certain database features are supported, you use methods that return boolean values. For example, you can determine whether your database supports batch updates and transactions with the supportsBatchUpdates() and supportTransactions() methods, respectively.

Structural information

You can also use a DatabaseMetaData object to retrieve information on tables, stored procedures, referential integrity, and data types. These methods return ResultSet objects. Of course, the result sets differ depending upon the methods used to query the database. The number of columns ranges from 1 to 18, all of which may be of different data types.

Most of the method calls are straightforward. However some, such as those that provide column information, enable you to supply string patterns as parameters to limit the number of rows returned. String patterns are essentially wildcard characters that enable you to narrow your search. You can use the underscore (_) character to match any single character or the percent symbol (%) to match zero or more characters.

In general, you should use string patterns to limit the number of rows returned by certain DatabaseMetaData methods. For example, if you call the getColumns() method to retrieve a result set of information on a certain table's columns, you may accidentally retrieve the column names for every column in all the database's tables. In this instance the result set will contain not only the column information for the target table, but the system's and user's tables as well. The result sets in this case will be quite large and probably meaningless.

String Patterns

String patterns enable you to "filter" the number of rows a result set returns. You can only use string patterns in the DatabaseMetaData methods that support them.

Two special characters, % and _, enable you to build string patterns. You use the % to match zero or more characters. For example, EM% will only allow the method to retrieve data that starts with the characters EM. The string pattern EM%S will retrieve data that starts with EM and ends with S. The data can contain any characters in between, and can be any number of characters in length.

You can use the underscore to match a single character. EM_ will only retrieve data that starts with EM and ends with any character. In this case the name of the field you are matching will only be three characters long.

The remainder of this section focuses on how to use the DatabaseMetaData object to retrieve information about tables, columns, data types, and referential integrity constraints. Although you can retrieve other information with the DatabaseMetaData object, I find this information most useful when I need to probe a database.

Table and column metadata You can obtain details on all the tables and table columns for which you have access rights. This may include system and other user schemas as well as your own. As I mentioned before, if you do not use string patterns you may create a very large result set.

The DatabaseMetaData methods getTables() and getColumns() retrieve information about the tables and table columns in a database, respectively. The method signatures are somewhat different from the usual Java methods because they can take string patterns as parameters. The following is the getTable() method definition:

```
public ResultSet getTables(String catalog,
                           String schemaPattern,
                           String tableNamePattern,
                           String[] types);
```

Think of these parameters as result set filters as each one can limit the number of rows returned. In the preceding getTables() method, the first parameter is the database catalog you wish to search for tables. A

catalog is analogous to a namespace and is used to separate data structures.

The next two parameters further filter the result set by enabling you to specify the schema and tables that you want to include in the result set. Notice that you can use string patterns for these parameters in order to control the data the result set returns.

The final parameter is a String array that represents the "types of tables" on which you want information. This is not the data type of the table, but the category. The table types are database−dependent and the getTableTypes() method will provide a result set containing the types available. Examples of different table types are TABLE, VIEW, and SYSTEM.

The following code snippet illustrates the getTables() method:

```
//Assume a valid Connection object conn
DatabaseMetaData dmd = conn.getMetaData();

//Define parameters.
String catalog = null;
String schema = "TODDT";
String tableName = "E%";
String [] types = {"TABLE"};

//Create a ResultSet object to hold the table information
ResultSet rs = dmd.getTables(catalog, schema, tableName, types);
```

Before calling the getTables() method, I first initialize all the parameters. I set the catalog parameter to null, which tells the method to ignore this parameter. In fact, you may inform the method to ignore any parameter by setting it to null. The null is equivalent to the asterisk (*) token in an SQL SELECT statement.

The next parameter is the schema name. In this example, I only want to retrieve the tables from my schema, TODDT. The third parameter is a string pattern that represents the tables to include in the result set. In the previous snippet I want to retrieve all tables that begin with the letter E. Therefore I include the % character to represent any characters after E. The fourth parameter uses a String array to define which category of tables to include in the result set.

The result set that the getTable() method returns has the following five columns: catalogname, schema name, table name, table type, and remarks. In my example, all the columns are Strings, so I can use the ResultSet.getString() method to retrieve the information.

You also use the getColumn() method in a similar manner. The following code demonstrates its use:

```
//Assume a valid Connection object conn
DatabaseMetaData dmd = conn.getMetaData();

//Create a ResultSet object that holds the database table
//information.
String catalog = null;
String schema = "TODDT";
String tableName = "E%";
String columnName = null;

ResultSet rsCols=dmd.getColumns(null, schema, tableName, null);
```

Of the four parameters, the last three accept string patterns. The first parameter is the same as the getTables()

method as it specifies the catalog you wish to work with. The next parameter identifies the schema, which I specify as myself because I only want columns for tables in my schema. With the next parameter I use a string pattern to select only the tables in my schema that start with the letter E. The last parameter enables me to specify a column name. I can also use a string pattern here to limit the number of columns returned, but in this case I want all the columns and so I use a null value.

The getColumns() method returns a result set that has 18 columns, and that unlike the result set returned by the getTables() method has mixed data types. Nonetheless, examples of the information the result set provides are the column name, data type, numeric precision if applicable, and whether the column can store null values. You should refer to the Javadocs for more information on the result set returned by the getColumns() method.

Data type metadata You can determine what data types your target database supports by using the getTypeInfo() method. The result set returned by this method is complex. Like the getColumnMethod() it has 18 columns and provides information such as the data type name, case sensitivity, JDBC data type, and whether the column can contain null values.

You may find this method useful if you want to write a utility program to determine the data types in a database. The following code snippet demonstrates a routine you can use to list all data types in a database. It provides the database's name for the data type and the corresponding short value that represents the JDBC data type.

```
//Assume a valid Connection object conn
DatabaseMetaData dmd = conn.getMetaData();

//Create result set
ResultSet rs = dmd.getTypeInfo();

//Loop through result set
while(rs.next()){

  System.out.println(rs.getString("TYPE_NAME") + " "
                   + rs.getShort("DATA_TYPE"));
}
```

In addition, you can use the getUDT() method to obtain a result set containing the information about the UDTs on your database server.

Primary− and foreign−key metadata A DatabaseMetaData object can also provide you with information on the referential integrity constraints for a database or group of tables. The getPrimaryKeys() and getImportedKeys() methods provide the primary− and foreign−key information. You may find these methods useful if you are writing an application to document your database.

Stored procedure metadata If you need to obtain information about the stored procedures in a database, use the getProcedures() and getProcedureColumns() methods of the DatabaseMetaData object.

DatabaseMetaData example

As with most JDBC topics an example is worth a thousand words. Listing 8−2 provides an example of several of the DatabaseMetaData methods described in the previous section. The program starts by listing the name and version of the database to which I am connected. Next it lists all the tables in my schema, TODDT, along with the column data types, and primary and foreign keys. The next items it lists are the supported data types in the database. Notice that the getTypeInfo() method creates a result set that provides both the data type name as defined on the server and the JDBC data type. You can look up the numeric value returned to get a

meaningful type name. Lastly, the program calls getSQLKeywords(), getNumericFunctions(), getStringFunctions(), and getTimeDateFunctions() methods to retrieve a list of the database keywords and helper functions.

Listing 8−2: DBMetaData.java

```
package Chapter8;

//Specific imports
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.DatabaseMetaData;

public class DBMetaData {

  public static void main(String[] args) {

    //Create Connection, Statement, and ResultSet objects
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Load a driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Obtain a Connection object
      System.out.println("Connecting to database...");
      System.out.println();
      String jdbcUrl = "jdbc:oracle:thin:@myserver:1521:ORCL";
      String user = "TODDT";
      String pwd = "mypwd";
      conn = DriverManager.getConnection(jdbcUrl,user,pwd);

      //Initialize a DatabaseMetaData object
      DatabaseMetaData dmd = conn.getMetaData();

      //Retrieve database name and version
      String dbname = dmd.getDatabaseProductName();
      dbname = dbname + "  " + dmd.getDatabaseProductVersion();
      System.out.println("Database information:");
      System.out.println(dbname);
      System.out.println();

      //Retrieve a result set with table information
      String [] types = {"TABLE"};
      rs = dmd.getTables(null,user,null,types);

      while(rs.next()){

        String tableName = rs.getString(3);
        System.out.println("Table Name: " + tableName);
        System.out.println(" Column, Data Type");
        ResultSet rsCols=dmd.getColumns(null,user,tableName,null);
```

```
  while(rsCols.next()){

    System.out.println("  " + rsCols.getString("COLUMN_NAME")
                     + ", " + rsCols.getString("TYPE_NAME"));

  }//end while rsCols

  System.out.println();

  //Get primary keys for tables
  ResultSet rsPkey=dmd.getPrimaryKeys(null,user,tableName);

  if(rsPkey.next()){

    System.out.println(" PK Name, Column Name");

    do{

      System.out.println("  " + rsPkey.getString("PK_NAME") + ", "
                       + rsPkey.getString("COLUMN_NAME"));

    }while(rsPkey.next());

    System.out.println();

  }//end primary key

  //Get foreign keys for tables
  ResultSet rsFkey=dmd.getImportedKeys(null,user,tableName);

  if(rsFkey.next()){

    System.out.println(" FK Name, FK Table, Column Name");

    do{

      System.out.println("  " + rsFkey.getString("FK_NAME") + ", "
                       + rsFkey.getString("PKTABLE_NAME") + ", "
                       + rsFkey.getString("FKCOLUMN_NAME"));

    }while(rsFkey.next());

    System.out.println();

  }//end foreign key

}//end while for table information

//Get supported data types
rs = dmd.getTypeInfo();
System.out.println("Supported data types:");

while(rs.next()){

  System.out.println("Database Type Name:
            " + rs.getString("TYPE_NAME")
            + "  JDBC Type: "
            + rs.getShort("DATA_TYPE"));

}//end while
```

```
      System.out.println();

      //Retrieve SQL Keywords, numeric functions, string, and
      //time and date functions.

      System.out.println("SQL Keywords:");
      String sql = dmd.getSQLKeywords();
      System.out.println(sql);
      System.out.println();

      System.out.println("Numeric Functions:");
      String numeric = dmd.getNumericFunctions();
      System.out.println(numeric);
      System.out.println();

      System.out.println("String Functions:");
      String string = dmd.getStringFunctions();
      System.out.println(string);
      System.out.println();

      System.out.println("Time and Date Functions:");
      String time = dmd.getTimeDateFunctions();
      System.out.println(time);
      System.out.println();

    //Standard error handling
    } catch(SQLException se) {
      //Handle errors for JDBC
      se.printStackTrace();

    } catch(Exception e) {
      //Handle errors for Class.forName
      e.printStackTrace();

    } finally {
      try {
        if(conn!=null)
          conn.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");

  }//end main

}//end DBMetaData.java
```

The output from Listing 8−2 is as follows:

```
Connecting to database...

Database information:
Oracle  Oracle8i Enterprise Edition Release 8.1.7.0.0 − Production
With the Partitioning option
JServer Release 8.1.7.0.0 − Production

Table Name: EMPLOYEES
```

```
 Column, Data Type
  SSN, NUMBER
  NAME, VARCHAR2
  SALARY, NUMBER
  HIREDATE, DATE
  LOC_ID, NUMBER

 PK Name, Column Name
  PK_EMP, SSN

 FK Name, FK Table, Column Name
  FK_LOC, LOCATION, LOC_ID


Table Name: LOCATION
 Column, Data Type
  LOC_ID, NUMBER
  LOCATION, VARCHAR2

 PK Name, Column Name
  PK_LOC, LOC_ID

Supported data types:
Database Type Name: NUMBER  JDBC Type: −7
Database Type Name: NUMBER  JDBC Type: −6
Database Type Name: NUMBER  JDBC Type: −5
Database Type Name: LONG RAW  JDBC Type: −4
Database Type Name: RAW  JDBC Type: −3
Database Type Name: LONG  JDBC Type: −1
Database Type Name: CHAR  JDBC Type: 1
Database Type Name: NUMBER  JDBC Type: 2
Database Type Name: NUMBER  JDBC Type: 4
Database Type Name: NUMBER  JDBC Type: 5
Database Type Name: FLOAT  JDBC Type: 6
Database Type Name: REAL  JDBC Type: 7
Database Type Name: VARCHAR2  JDBC Type: 12
Database Type Name: DATE  JDBC Type: 93
Database Type Name: STRUCT  JDBC Type: 2002
Database Type Name: ARRAY  JDBC Type: 2003
Database Type Name: BLOB  JDBC Type: 2004
Database Type Name: CLOB  JDBC Type: 2005
Database Type Name: REF  JDBC Type: 2006

SQL Keywords:
ACCESS, ADD, ALTER, AUDIT, CLUSTER, COLUMN, COMMENT,
COMPRESS, CONNECT, DATE, DROP, EXCLUSIVE, FILE, IDENTIFIED,
IMMEDIATE, INCREMENT, INDEX, INITIAL, INTERSECT, LEVEL, LOCK,
LONG, MAXEXTENTS, MINUS, MODE, NOAUDIT, NOCOMPRESS,
NOWAIT, NUMBER, OFFLINE, ONLINE, PCTFREE, PRIOR,
all_PL_SQL_reserved_ words

Numeric Functions:
ABS, CEIL, COS, COSH, EXP, FLOOR, LN, LOG, MOD, POWER, ROUND,
SIGN, SIN, SINH, SQRT, TAN, TANH, TRUNC, AVG, COUNT,
 GLB, LUB, MAX, MIN, STDDEV, SUM, VARIANCE

String Functions:
CHR, INITCAP, LOWER, LPAD, LTRIM, NLS,_INITCAP, NLS,_LOWER,
NLS,_UPPER, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR,
 SUBSTRB, TRANSLATE, UPPER, ASCII, INSTR, INSTRB,
LENGTH, LENGTHB, NLSSORT, CHARTOROWID, CONVERT, HEXTORAW,
```

```
RAWTOHEX, ROWIDTOCHAR,
TO_CHAR, TO_DATE, TO_LABEL, TO_MULTI_BYTE, TO_NUMBER, TO_SINGLE_BYTE

Time and Date Functions:
ADD_MONTHS, LAST_DAY, MONTHS_BETWEEN, NEW_TIME, NEXT_DAY,
ROUND, SYSDATE, TRUNC

Goodbye!
```

Summary

This chapter provided an overview of the two metadata interfaces, ResultSetMetaData and DatabaseMetaData. These interfaces enable you to interrogate result sets and databases. Examples of information the ResultSetMetaData interface provides are:

- The number of columns in a result set.
- The data types of the columns in a result set.

Examples of information the DatabaseMetaData interface provides are:

- The supported data types in a database.
- The SQL keywords.
- The names of helper functions for mathematical, character, and date and time calculations.
- The tables, indices, and stored procedures in a database.
- Whether or not the database supports batch−updates.

In addition, this chapter provided two examples showing how to use the interfaces. The ResultSetMetaData interface example includes a generic method that prints the data contained in a result set. The DatabaseMetaData interface example provided the routines used to list the table structure for any schema, the supported data type in the database, and also provided a list of SQL keywords and helper methods.

Part III: Using Java Data Access Design Patterns

Chapter List

Chapter 9: Understanding Design Patterns

In This Chapter

- Understanding design patterns
- Using Java with design patterns
- Relating design patterns to object−oriented programming
- Applying design patterns to JDBC programming

The previous chapters of this book focused specifically on JDBC programming. The chapters in this part deviate slightly to introduce the subject of design patterns.

Design patterns are software architectures used mainly with Object Oriented Programming (OOP) techniques and languages. Building software solutions using design patterns can produce software that has a high level of code reuse and is easy to maintain. The patterns are time−tested and even implemented within the Java API.

Although used long before, design patterns were formally described and cataloged in the book *Design Patterns* (Addison−Wesley, 1995) by Gamma, Helm, Johnson, and Vlissides. After its publication, the subject became popular in the developer community. Programmers formed user groups to study and develop design patterns. Today, a search for “design patterns” on the Internet will provide you more information on the subject than you can absorb.

Design Patterns introduced 23 patterns used in OO software design. The authors drew from their own experience and the experiences of others when they cataloged the patterns. The concepts they present certainly help me design software. Thinking in terms of patterns encourages me to consider objects as components and to take advantage of OO techniques such as encapsulation, inheritance, and polymorphism.

I begin this chapter by introducing the subject of design patterns, and follow this with a short overview of the core patterns. Next, I cover implementation issues you will be faced with when using design patterns with Java. Throughout the chapter, I’ll also point out circumstances in which you can apply design patterns to JDBC programming. However, I save the details for Chapters 10, 11, and 12.

What Are Design Patterns?

Java is an object−oriented language that enables you to rapidly develop applications. It provides a class library, known as the Java API, which gives you reusable components so you do not have to develop them yourself. This enables you to focus on programming, because a lot of the low−level implementation work is handled for you. JDBC is a perfect example of a Java class library as it hides the complexities associated with interacting with a database, and lets you focus on the task at hand.

However, despite Java’s class libraries, designing effective and efficient object−oriented software is hard. It takes experience to familiarize yourself with the best way to assemble objects to perform a particular task. You want a design that takes advantage of object−oriented traits such as code reusability, extensibility, and maintainability. In some cases you may need to sacrifice one trait in favor of another; overall, however, a good object−oriented design will have these attributes.

Developers migrating from procedural languages like FORTRAN and COBOL often have a tough time thinking in objects. They usually apply procedural techniques using object−oriented languages. For example, procedural programmers may overuse the static keyword when declaring methods, thus creating a method library using a class structure.

The difficulty in learning object−oriented programming is that it’s hard to visualize an “object.” Some textbooks teach that an object represents a component of a physical system or business process. But what about low−level objects like a Java Array or InputStream? These objects are hard to visualize and do not map to real−world systems or processes. They provide functionality that is hard to "see."

Standard software design patterns exist to simplify the conceptual problems of designing software using object−oriented programming languages. A design pattern provides you a way to visualize the whole, complete solution before starting to code. It helps you “begin with the end in mind,” as Stephen Covey would say.

With respect to JDBC programming, not all of the 23 patterns apply. The patterns that I find useful have to do with creating objects and abstracting complex functionality. As a result I will only cover in depth those patterns in the following chapters that I feel directly apply to JDBC programming. If you want to gain a deeper understanding I suggest reading *Design Patterns*. It will certainly help you gain a better grasp on object−oriented designs and implementations.

Categories of Design Patterns

The authors of *Design Patterns* grouped the 23 patterns into three categories according to functionality. The categories are *creational*, *structural*, and *behavioral*.

Each category defines the intent of the pattern. Creational patterns concentrate on creating objects, structural patterns concentrate on combining objects and classes, and behavioral patterns concentrate on object and class interactions.

Creational patterns

This category contains design patterns that create objects. The general procedure is to create objects, loosely termed *factories*, that instantiate other objects. Creational patterns abstract the instantiation process from the client. The client retrieves an object from a factory without knowing how the factory instantiated it. Table 9−1 provides an overview of the five creational patterns.

Table 9−1: Creational Design Patterns

Pattern	Description
Singleton	Guarantees only one instance of a class, and defines one point of access to it.
Builder	Separates the creation process of a complicated object from its representation. This pattern enables you to change the creation process to create an alternate

	representation.
Factory Method	Defines the “factory” method in an interface or abstract class. Subclasses usually provide the implementation and therefore decide which object type to create. Clients generally reference the subclass.
Abstract Factory	Defines an interface or abstract class for creating groups, or *families*, of related objects. The client generally references the interface, the Abstract Factory, not the concrete factory.
Prototype	Creates an object based on “prototype” objects. Generally requires cloning, or copying, the prototype object to create a new object.

Creational patterns provide you with many benefits. For example, you can dynamically create objects by defining methods to create different objects based on runtime parameters. This provides you with the flexibility to create objects using input from the command line, a properties file, or user input.

In addition, the creational patterns also enable you to create objects based on an object’s state. For example, if a connection−pooling flag is set in a database−connection factory, users could retrieve a Connection object from an object pool versus opening a new connection every time a user makes a request for a Connection object.

Combining creational patterns and polymorphism enables you to remove factory implementations from the application. You can use abstract classes to define the factory and then provide implementations in the subclasses. Declaring a variable of the abstract factory type in the client allows the client to use any factory subclass to create objects.

Pattern Scope

Another way to categorize design patterns is by *scope* or focus. Some patterns deal with classes while others deal with objects. Class patterns describe a class’s relationships with other classes and with their subclasses. Object patterns focus on object−to−object interactions and relationships. Object patterns are more dynamic than class patterns and that enables you to change them at runtime.

To illustrate the differences between the two, consider the following. Object creational patterns let other objects create objects, while the class creation pattern relies on subclasses. Object patterns in the structural category use composition to gain new functionality. The class patterns in this category use inheritance to create new functionality. Within the behavioral category, object patterns use groups of objects to perform tasks as a unit. The class patterns again rely on inheritance to distribute responsibility.

Most of the 23 design patterns have object scope. Only the Factory Method, Adapter, Interpreter, and Template Method patterns are considered to have class scope.

Earlier I mentioned that the patterns in this category focused on creating other objects. This is true for four out of the five patterns. One pattern, the Singleton, is different from the rest. It does not directly create objects, but ensures that only a single instance of a class is created. This pattern, as you will see, has many uses in JDBC programming.

Generally, when working with JDBC I use the creational patterns by building a factory to supply Connection objects. This approach saves me from having to define JDBC URLs, usernames, and passwords or loading drivers for every connection I need. I prefer to define the connection settings once in a factory object and then

supply Connection objects to clients as needed. In addition, I usually build my factory object as Singleton for more control.

XRef Chapter 10, “Building the Singleton Pattern,” provides more detail on that pattern. I also build a connection manager implemented as a Singleton as an example in Chapter 10.

Structural patterns

The structural category contains patterns that focus on combining objects to create new functionality. These patterns help you to create object hierarchies, share objects, and dynamically add responsibilities to objects. Structural class patterns use inheritance to create new classes, while object patterns use composition. Table 9−2 summarizes the seven structural patterns.

Table 9−2: Structural Design Patterns

Pattern	Description
Proxy	Represents, or is a surrogate for, a complex object. The Proxy interface often provides a simpler interface than the object it represents.
Adapter	Makes one class interface match a different class interface. You generally create the desired interface and use it to wrap another class or interface.
Flyweight	Shares a large number of similar objects. State data about the object is passed to the shared object when needed.
Composite	Defines objects that can hold a collection of other objects. A composite pattern will often arrange the objects in a tree structure.
Decorator	Modifies behavior or adds responsibilities to an object dynamically without you having to create a new object.
Bridge	Separates the implementation of a class from its definition.
Façade	Provides a simplified interface to a complex subsystem of objects.

Of the structural patterns, I find the Façade pattern one of the most useful in JDBC programming. With it you can build an object composed of Connection and Statement objects, which in turn enables you to abstract from the client most of the details of JDBC programming. Clients can submit SQL statements to the Façade object and retrieve either an update count or a result set, without worrying about the connection logic or the semantics of creating the different Statement objects.

XRef I provide more detail about the Façade pattern in Chapter 12, “Creating a Façade Pattern.” This chapter also shows you how to abstract the details of working with Connection and Statement objects from the client.

Behavioral patterns

This category focuses on algorithms, object responsibilities, and object−to−object communications. You can think of these patterns as communication models that control how messages, object behavior, and state data are communicated between objects.

Because of its vast focus, this category is understandably the largest. Table 9−3 provides a list of the 11 behavioral patterns and their descriptions.

Table 9−3: Behavioral Design Patterns

Pattern	Description
Visitor	Creates a class that acts on data within other classes. This pattern enables you to change the implementation of a class without modifying the class itself.
Chain of Responsibility	Handles a request by passing it through a series of objects until one can process it. The pattern reduces object coupling so the objects can act independently.
Template Method	Uses a parent class to define methods and lets the derived classes provide the implementation. Analogous to abstract methods in Java.
Command	Allows an object to encapsulate a command as an object and send it to the Command object. The Command object is responsible for dispatching the command to the appropriate object for execution.
Strategy	Encapsulates a group of related algorithms in an interface or abstract class. The client that references a subclass automatically uses the appropriate algorithm.
Interpreter	Interprets a custom grammar. You generally use this pattern when you define a command language for a client and must interpret and process requests.
State	Changes an object’s behavior when that object’s internal state changes. You usually control the state by manipulating a class’s private properties.
Mediator	Defines a central object that mediates, or controls, the way in which objects in a group communicate. The objects in a group typically do not communicate directly. Although the Model−View−Controller pattern is not covered here, those familiar with it will recognize the Mediator as being the controller in Model−View−Controller.
Observer	Uses objects called observers, which change their state or behavior when another object, known as a subject, changes its state. This is also known as publish−subscribe.
Memento	Collects and saves data about an object so you can restore the object later.
Iterator	Uses an object with a standard interface to provide access to the elements of a list or collection of data, either primitives or objects.

You will find behavioral patterns in the JDBC API as well. For example, the ResultSet interface implements the Iterator, or "cursor," pattern. The interface defines numerous methods for moving around and manipulating data in a ResultSet object. An instantiated ResultSet object hides the implementation details from you. When you call any of the cursor−movement methods, the ResultSet object moves the cursor without letting you know how. If you open an updateable ResultSet and delete a row, the ResultSet object handles the deletion of the data from its internal storage without your intervention or knowledge.

Java and Design Patterns

As an object−oriented programming language, Java supports the implementation of the 23 design patterns presented in the previous section. In fact, the Java API uses many of the patterns I just defined.

For example, the Enumeration interface implements an Iterator pattern. It defines methods for retrieving data from an object that implements its interface without letting you know how it stores or retrieves information. The API even defines an Observer interface that implements the Observer pattern.

With respect to JDBC, I've mentioned several design patterns in the last section that the API implements. Nonetheless, the most prevalent pattern is the Factory. For example, the DriverManager class is a factory. You request a Connection object with certain attributes — such as user, password, or database — and retrieve an object with those properties.

Design patterns focus on object interactions and relationships. As a result, they rely heavily on OOP techniques and concepts. In particular, the patterns use class inheritance and object composition extensively. In the remaining sections of this chapter I provide an overview of inheritance and composition, as well as how to implement the concepts in Java and how to apply them to design patterns.

Inheritance

Inheritance is a technique for creating new classes based on other classes. With Java, inheritance is achieved when one class extends another. The class inherited from is called the *super class* or *base class*. The inheriting, or extending, class is called the *subtype* or *subclass*. The subclass has access to the public and protected methods and attributes of the base class. However, the base class may hide its members by declaring them private. The inheritance relationship between classes is called a *class hierarchy*.

Figure 9−1 illustrates a class hierarchy showing inheritance. Class B, the subclass, extends Class A and has access to the public or protected methods and data in class A. In this example, Class B can access attribute1 and method1() because they are public. It does not have access to private members attribute2 and method2(). You are not restricted to using the base class's methods and attributes; you can also add additional methods and properties to give Class B extra functionality. In addition, you can override or overload the methods in the base class.

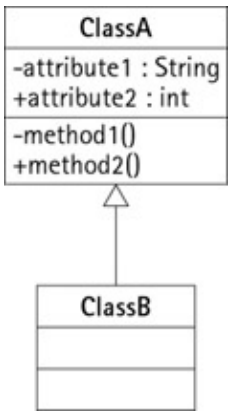

Figure 9−1: Example inheritance class hierarchy

Note Polymorphism, a cornerstone of OOP, is likely the most confusing OO term and concept. The word is derived from the Greek for "many types" or "many forms." Personally, I think of polymorphism in terms of interchangeable objects, because you can assign any subclass to a variable of the base class type.

Inheritance and polymorphism go hand in hand. Inheritance provides a family of objects with identical interfaces because the base class defines the interface for the family. As a result, you can store references to subclasses in variables declared as a base class type. You can pass these variables to methods expecting base types as parameters. Although the class type is different, the interface is the same.

Polymorphism allows a client object to call a method in an object and expect a certain result. The client is not interested in whether the object is instantiated from the base class or subclass, only in whether it can perform the task. The interface definition defines what tasks it can accomplish.

Before continuing, let me provide a concrete example of inheritance and polymorphism. Figure 9−2 shows the class hierarchy for the example. The base class is a Fruit and the two subclasses are Apple and Orange. That is, Apple and Orange are of type Fruit because they extend the Fruit class and thereby share the same interface. The base class has two members, a private String name and a public method, printName(). Each subclass overrides the printName() method to print the value assigned to the name field.

Java OOP Terminology

When working with Java you will often hear many OOP terms. Like most other technologies, unless you use the terms daily you may not be that familiar with them. The following list should help remind you of the most common OOP definitions.

- Class: An abstract data type, a collection of attributes and the methods that operate on those attributes.
- Member: A method or attribute defined in a class.
- Object: An instance of a class created with the new keyword or returned from a method.
- Object instantiation: The act of creating a new object.
- Class variable: A variable defined with the static keyword. Only one copy exists for the class. Multiple objects instantiated from the class share the variable.
- Class method: A method defined with the static keyword. A static method does not operate on a particular class but acts as a utility function. It can also manipulate static (that is, class) variables.
- Inheritance: The derivation of new classes from existing classes. The new class (the subclass) may, or may not, have access to the methods and attributes in the base class.
- Overriding: Keeping the same method signature in the subclass that is defined in the base class, but changing the implementation.
- Overloading: Changing the signature of the method in either the base class or the same class and changing the implementation.
- Polymorphism: The ability of one class to represent another. It enables you to use a variable of a base class type to hold a reference to an object of the same base class type or a subclass type. It also allows you to use a variable of an abstract interface type to hold a reference to any of a number of objects implementing that interface.
- Encapsulation: The ability to provide users with a well−defined interface to a set of functions in a way that hides the internal workings of those functions. In OOP, encapsulation is the technique of keeping together data structures and the methods (procedures) that act on them.

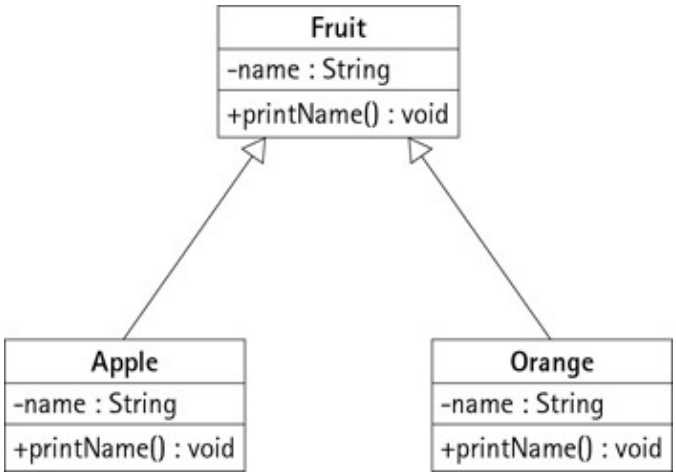

Figure 9−2: FruitBasket inheritance diagram

Listing 9−1 demonstrates the polymorphic behavior with the Fruit class hierarchy. Notice that I create two Fruit objects, Orange and Apple, and assign the variables references to instances of Orange and Apple objects. Although I declare the variables to be of type Fruit, the compiler does not complain about these declarations because Apple and Orange are Fruit by inheritance and share the same interface.

Listing 9−1: FruitBasket.java

```
package Chapter9.Inheritance;

//Base class
class Fruit{

  private String name = "I am a Fruit";

  public void printName(){

    System.out.println(name);

  }
}

//Subclass
class Orange extends Fruit{

  private String name = "I am an Orange";

  //Overridden method
  public void printName(){

    System.out.println(name);

  }
}

class Apple extends Fruit{

  private String name = "I am an Apple";

  //Overridden method
  public void printName(){
```

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

  }
}

public class FruitBasket{

  public static void main(String[] args) {

    //Declare two variables of type Fruit
    Fruit Apple = new Apple();
    Fruit Orange = new Orange();

    //Call methods.
    Apple.printName();
    Orange.printName();

    System.out.println("Goodbye!");
  }
}
```

The output from Listing 9−1 is as follows:

```
I am an Apple
I am an Orange
Goodbye!
```

From the output of Listing 9−1, you can see that polymorphic behavior is demonstrated when I call the method printName(). The objects Apple and Orange automatically call the correct printName() method from the correct class. You may have expected the Fruit object's printName() method to be called because the two objects were defined as Fruit types.

Java handles polymorphism for you automatically. The object type is checked at runtime (this is known as late−binding). Fortunately you get the behavior for free. You need not do anything special to implement polymorphism, except create the correct class hierarchy.

Composition

Composition is a means of creating new functionality in an object by combining other objects within it. This technique is simple and you use it more often than you think. To employ composition all you do is use instance variables, which you can declare the variables as either public or private.

Let me demonstrate by implementing the FruitBasket example in Listing 9−1, using composition instead of inheritance. Notice from the class diagram in Figure 9−3 that no inheritance relationship exists between Apple and Fruit or between Orange and Fruit. However, in the class definition for Apple and Orange I define a variable fruit and assign a reference to a Fruit. This is composition.

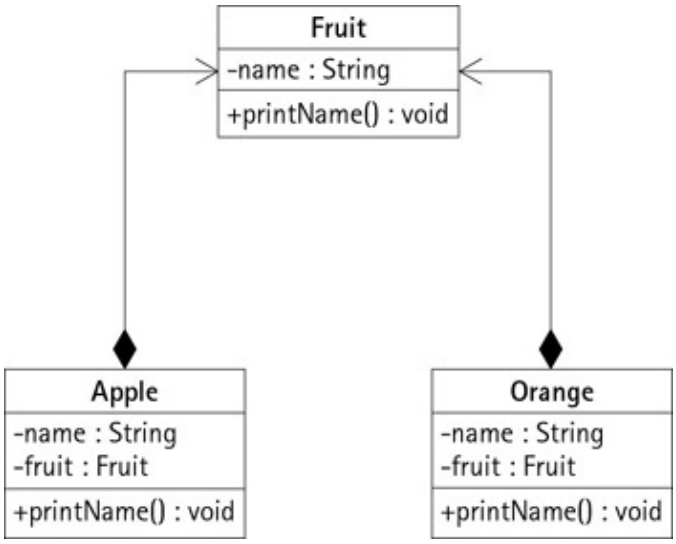

Figure 9−3: FruitBasket composition−class diagram

Listing 9−2 provides the composition example. I must also create an implementation of the printName() method in each class. This method calls the printName() method of the Fruit class, which prints the fruit's name. This is the opposite of polymorphism. To access the methods in the composed class you must define accessor methods in the composing class. A client does not know that you hold references to these objects (this is encapsulation), and so you must provide a way for the client to use the functionality of the composed class.

Listing 9−2: FruitBasket.java

```
package Chapter9.Composition;

//Class the gets "composed"
class Fruit{

  private String name = "I am a Fruit.";
  public void printName(String name){

    System.out.println(name);

  }
}

//Composing class
class Apple{

  private String name = "I am an Apple.";
  private Fruit fruit = new Fruit();

  public void printName(){

    fruit.printName(name);

  }
}

//Composing class
class Orange{

  private String name = "I am an Orange.";
```

```
  private Fruit fruit = new Fruit();

  public void printName(){

    fruit.printName(name);

  }
}

//Test bed class
public class FruitBasket{

  public static void main(String[] args) {

    //Variable for the composing classes
    Apple apple = new Apple();
    Orange orange = new Orange();

    //Print the names of the fruit
    apple.printName();
    orange.printName();

    System.out.println("Goodbye!");

  }
}
```

The output from Listing 9−2 is as follows:

```
I am an Apple.
I am an Orange.
Goodbye!
```

Design−pattern implementation guidelines

Design patterns promote code reuse and object independence, two of the goals of OOP software design. Just because you create an application using the three characteristics of OOP — inheritance, encapsulation, and polymorphism — does not guarantee easy code maintenance or a high degree of code reusability.

As a result, the authors of *Design Patterns* provide the following guidelines for implementing patterns:

1. *Do not program to implementations, only to interfaces.* This guideline suggests that your classes only rely on well−defined interfaces, not class implementations. Relying on implementations creates tightly coupled objects. In Java, this guideline suggests that you define interfaces or abstract classes, and then derive your classes from them and provide method implementations in the subclasses. The client will declare variables of the interface type or the base class, and use references to the classes that implement the methods.
2. *When practical, use composition over inheritance.* Inheritance has its place. It is a very useful technique for promoting code reuse. After all, when a subclass can use methods in the base class, it saves you from having to re−implement those methods. In addition, inheritance enables you to take advantage of polymorphism.

   However, when you use inheritance you must ensure that the interface of the base class remains stable. Changing it can break the inheritance chain. For example, suppose you define a method in the

base class that returns an int. Later, you change the method to return a long. This change will break the base class's interface, which the clients rely upon.

Because of the difficulty of changing the interface of the base class, the *Design Pattern* authors suggest that you use composition. This concept requires you to combine existing objects into an object to create new functionality, instead of using inheritance. The ripple effect produced by changing a class interface used in composition is far smaller than the ripple effect produced by changes in the inheritance model.

Composition also encourages encapsulation. When you assemble objects to create new functionality, each one will have a definite purpose. You only care about the interface and the results the objects provide, not the implementation.

A good question to ask when choosing between inheritance and composition is: Do the classes define an "is−a" relationship? If you answer yes, then you should use inheritance. To illustrate, in previous examples it is obvious that an Apple "is−a" Fruit and therefore warrants using inheritance.

However, you may encounter a questionable "is−a" relationship. For example, at a track meet you may think a sprinter "is−a" Runner. However, what if the sprinter is in a decathlon? Now he or she can become a shot putter, pole−vaulter, and/or javelin thrower, for example. The sprinter has changed roles. In this case you should use composition to create a type, Athlete, that is composed of the different types that match the events.

Deciding whether to use composition or inheritance is hard. Fortunately, following a prescribed design pattern will lead you in the right direction.

Summary

Creating robust and scalable OO applications can be difficult. Design patterns can make it easier, because they describe time−proven standard architectures for implementing software. In addition, the patterns can help you see how a solution fits together and provide a "big picture" view of your application.

In this chapter, I presented the three categories of design patterns: creational, structural, and behavioral. Next, I presented an overview of the patterns in each category. In the last section I covered inheritance and composition, the two OOP principles that greatly influence how you implement a design pattern. I also presented the implementation guidelines recommended by the authors of *Design Patterns*.

Chapter 10: Building the Singleton Pattern

In This Chapter

- Understanding the purpose of the Singleton pattern
- Understanding the structure of the Singleton pattern
- Implementing the Singleton pattern
- Making a Singleton thread−safe

In Chapter 9, I introduced the subject of design patterns and presented the 23 patterns in the book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides. In this chapter, I go into more detail about one of the more useful and frequently used patterns, the Singleton.

The Singleton pattern is a creational pattern that you use to ensure that only one instance of an object exists. This means that other objects share the same instance. Why would you want only one instance of an object? Think of the Singleton as a dispatcher. All communications must go through the same "person." The dispatcher has complete control over what gets communicated. The Singleton pattern is advantageous when you need to emulate a single point of contact as with the dispatcher just described.

I begin the chapter with a discussion of the details of the Singleton pattern. Next, I present its structure and present two implementation examples. The first example shows the basic implementation of the Singleton pattern, and the second shows how to apply it to JDBC programming to create a connection manager.

What Is a Singleton Pattern?

The purpose of most creational patterns is to create other objects. The purpose of the Singleton pattern, however, is to ensure that only one instance of an object exists at any one time. Other client objects, instead of instantiating another Singleton, retrieve a reference to the active object.

A Singleton object also maintains responsibility for controlling its only instance. It hides its constructor to prevent other objects from instantiating it. A Singleton object provides an access method with which clients may retrieve a reference to the instance as opposed to calling the constructor directly.

Using a Singleton pattern is not the only technique you can use to create a single−object instance. For example, you can declare a global variable in a class and instantiate it once within your application. However, if you are creating a toolkit or application framework you may not want other objects, over which you have no control, to create another instance. Singletons provide the solution to this problem.

So when would you want to implement an object as a Singleton? One example is when you need to implement a "manager" object of some sort. Manager objects can do many things — such as controlling other objects, providing objects to clients, or overseeing services such as printing. By making a manager a Singleton you ensure that all requests funnel through the same object.

With respect to JDBC programming, a connection manager is an ideal candidate to implement as a Singleton. The connection manager can provide a single protected point of access into the database. You can pre−configure connection attributes such as username, password, and JDBC URL so they are consistent and also eliminate duplicate connection code throughout your application. You can also use a connection manager

to create a connection pool of *n* number of Connection objects and provide clients with objects from that pool. You can add many features and capabilities to a connection manager to suit your needs. These are only a few examples.

> Note You will find the Singleton pattern very useful and you may want to use it to implement other patterns as well. The Abstract Factory, Builder, and Prototype patterns are prime candidates.

Structure of the Singleton Pattern

The structure of the Singleton pattern is straightforward. It consists of one class that provides the functionality to ensure only a single instance is ever instantiated. The class also defines a method, or global access point, with which other objects can retrieve a reference to the instance.

Figure 10−1 provides a UML class diagram of a Singleton class called MySingleton. Notice that the constructor is private to prevent other objects from instantiating it. The class also uses the private class variable, mySingleton, to store a reference to the single instance. The getInstance() method provides the global access point by returning a reference to the single instance of the object stored in mySingleton. You can also define other methods to give the object additional functionality. In this example I define method1(), method2(), and method3().

The programming flow of the Singleton pattern is depicted in Figure 10−2. When you need a reference to the Singleton object you call the getInstance() method. The getInstance() method checks to see if the private class variable mySingleton equals null. If it does not, the private constructor MySingleton() is called to instantiate the object before returning the reference. Otherwise, the method returns the reference to the instance held in the mySingleton variable.

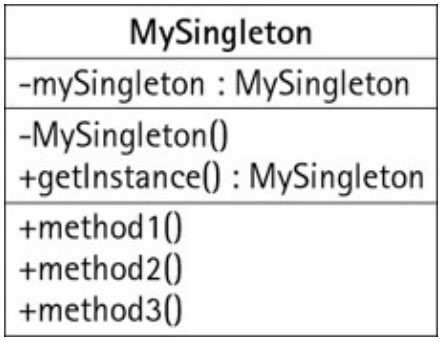

Figure 10−1: UML diagram of the Singleton pattern

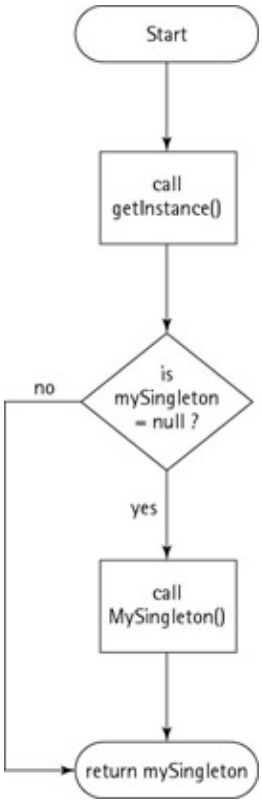

Figure 10−2: Flowchart of the Singleton pattern

Using the Singleton Pattern

To fully implement the Singleton pattern, you must make sure that the following two criteria are met:

1. Only one instance of the object can exist. This is the key feature of the Singleton pattern. To meet this goal you typically define a class, with a private, static class variable of that class's type, to hold a reference to the only instance. In addition you protect the class's constructor by making it private. Clients trying to call the constructor will receive a compilation error.
2. You must define a global access method with which clients can obtain references to the instance. You do this by providing a static method that returns a reference to the only instance of the object.

In the remainder of this section I'll develop two example applications to illustrate the Singleton pattern. In the first example, I show the basic architecture of the pattern. In the second example, I apply the pattern to JDBC programming by creating a connection manager that controls access to the database and divvies out Connection objects.

Basic Singleton example

A simple Singleton pattern is easy to implement. You create a single instance of an object and provide an accessor method that returns its reference to clients as needed.

Figure 10−3 shows the UML class diagram for my example. Notice that the SimpleSingleton class has two variables, s1 and s2, that are references to the Singleton class. When the application executes, these variables hold the same object reference. I created the two variables to show that the same reference is used in both

cases.

Also, notice that I define a private constructor to prevent client objects from trying to instantiate the class directly. Any call to the constructor made from outside the class generates compilation errors. To obtain a reference to the instance a client must call the getInstance() method, which returns a reference to the Singleton object. This method is the global access point, defined by the Singleton pattern, which allows other objects to retrieve a reference.

Listing 10−1 is the source code for my example, which demonstrates how to construct and use a basic Singleton, plus how the Singleton pattern limits the number of instances an object can have.

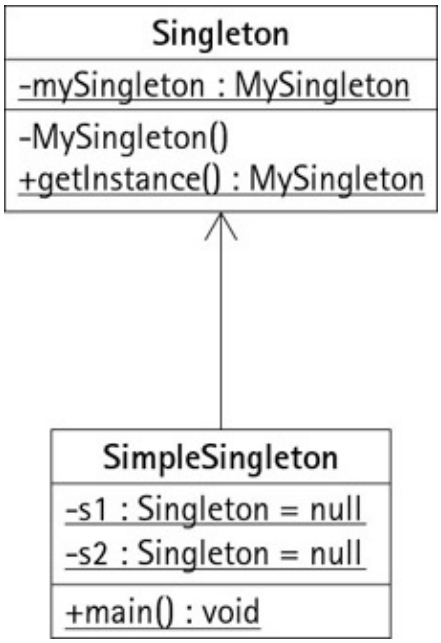

Figure 10−3: UML diagram of basic Singleton example

Listing 10−1: BasicSingleton.java

```
package Chapter10;

//This is the test harness class
public class BasicSingleton {

  //Variables to hold references to the Singleton object
  static Singleton s1 = null, s2 = null;

  public static void main(String[] args) {

    //Request a reference to the Singleton object
    String msg="Beginning to get an instance of a Singleton.";
    System.out.println(msg);
    s1 = Singleton.getInstance();


    //Try to assign a new reference to the second
    //Singleton variable s2.
    msg="Assign a reference to the Singleton object to another "
      + "Singleton variable. The Object Id should be the same.";
    System.out.println(msg);
```

```
    s2 = Singleton.getInstance();

   }//end main()

}//end BasicSingleton class


//Implementation of the Singleton Class
class Singleton{

  //Private reference to the only instance of the Singleton object.
  private static Singleton mySingleton = null;

  //Private constructor to prohibit direct object instantiation.
  private Singleton(){}

  //Method to retrieve a reference to the only instance.
  public static synchronized Singleton getInstance(){

    //If null create the instance
    if(null == mySingleton){

      System.out.println("Singleton object not instantiated yet.");
      mySingleton = new Singleton();
      System.out.print("Singleton object instantiated. Object id:");
      System.out.println(mySingleton.toString());
      System.out.println();

    //If not null return the only instance
    }else{

      System.out.print("Singleton object already instantiated. " +
                       "Object id:");
      System.out.println(mySingleton.toString());
      System.out.println();

    }//end if

    return mySingleton;

  }//end getInstance()

}//end Singleton class
```

The output from Listing 10−1 is as follows:

```
Beginning to get an instance of a Singleton.
Singleton object not instantiated yet.
Singleton object instantiated. Object
id:Chapter10.Singleton@2a340e

Assign a reference to the Singleton object to another
Singleton variable. The Object Id should be the same.
Singleton object already instantiated.
Object id:Chapter10.Singleton@2a340e
```

The output from Listing 10−1 shows the flow of the program. In the example, I always print the Object Id in the getInstance() method so you can verify that the same object is always returned from the getInstance()

method. The first thing I do is obtain a reference to the Singleton object and store it in variable s1. Notice from the output that the first call to the getInstance() method indicates that the Singleton object does not exist. Therefore, the method calls the private constructor of the Singleton class to create the instance and assign it to the private class variable, mySingleton. After the instance is created, the method returns the reference and the Object Id is printed.

Tip You should synchronize the Singleton method that returns the object reference. If you implement a multi−threaded application you may have several threads calling the method simultaneously. You need to synchronize this method to ensure that the constructor is not called multiple times.

Next, I declare another variable, s2, to store another reference to the Singleton object. This time when I call the Singleton.getInstance() method, it returns a reference to the same Singleton object. You can see from the output that the object already exists, and so the identical reference is returned.

This example illustrates the concepts and components necessary to implement the Singleton pattern. The main points to remember are:

- Make the constructor private so clients cannot instantiate the class
- Maintain a private reference to the Singleton object
- Provide a global access method to return a reference to the Singleton object

Connection manager Singleton example

In this section, I apply the Singleton pattern to JDBC programming to create a connection manager. As I mentioned earlier, a connection manager is a perfect object to implement as a Singleton because having one point of access into a database can have many benefits.

For example, you can place another layer of security into your application by allowing certain classes or users to access the object and obtain a database connection. Or you may want to abstract the connection details, such as loading drivers and building JDBC URLs, from the client. In addition, you can build an application with static usernames and passwords to ensure that the same database user connects every time. This can help if you roll your own application security and need everyone to connect as the same user.

Implementing the connection manager as a Singleton is a process much like the one in the previous example. You might expect this because design patterns provide standard robust implementations with consistent implementations.

Figure 10−4 shows the UML class diagram for the example. The ConnectionMgr class is a standard implementation of the Singleton pattern. I declare a private class variable, connMgr, to hold an instance of the object, make the constructor private, and create a method, getInstance(), that provides the global access point. I also synchronize the getInstance() method to make it thread safe.

As in the previous example, I declare two class variables, cm1 and cm2, to hold references to the ConnectionMgr instance. I do this to demonstrate that all calls to the getInstance() method of ConnectionMgr return the same object reference. I do the same with the variables conn1 and conn2, which reference the same Connection object.

Remember that I only want one connection to the database. Therefore I declare within ConnectionMgr a private variable, conn, to hold the database connection. In essence, I am treating this object as a Singleton as well. In the connect() method, I check to see if conn equals null. If it does, I create a database connection and return the object reference. Otherwise I return the existing reference, thus preventing another connection from

occurring.

Listing 10−2 provides the code for the connection manager application.

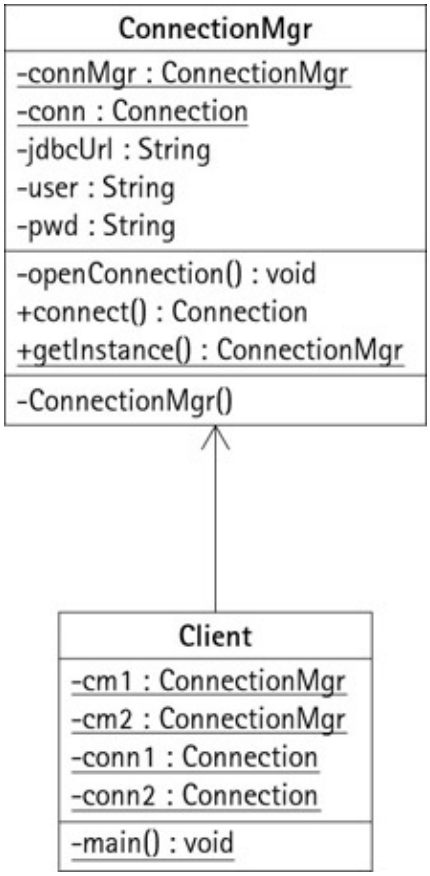

Figure 10−4: UML class diagram for Connection Manager Singleton

Listing 10−2: Client.java

```
package Chapter10;

//Specific imports
import java.sql.*;

public class Client{

  //References to the Singleton object
  public static ConnectionMgr cm1 = null;

  public static ConnectionMgr cm2 = null;

  //Database Connection objects
  public static Connection conn1 = null;
  public static Connection conn2 = null;

  public static void main(String[] args){

    try{

      //Get two instances  of Connection Manager
      cm1 = ConnectionMgr.getInstance();
```

```
  cm2 = ConnectionMgr.getInstance();

  //Print out object ids, they should be the same
  System.out.println("Verify ConnectionMgr object ids are equal:");
  System.out.println("cm1 object id: " + cm1.toString());
  System.out.println("cm2 object id: " + cm2.toString());
  System.out.println();

  //Obtain two references to the Connection object
  System.out.println("Obtain database connections:");
  conn1 = cm1.connect();
  conn2 = cm2.connect();

  //Print object ids for the connection object. They should be the same.
  System.out.println("Verify object ids are the same:");
  System.out.println("conn1 object id: " + conn1.toString());
  System.out.println("conn2 object id: " + conn2.toString());
  System.out.println();

  //Demonstrate the connection works
  Statement stmt = conn1.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT * FROM Employees");

  System.out.println("Listing result set contents:");
  while(rs.next()){

    int ssn= rs.getInt("ssn");
    String name = rs.getString("name");
    double salary = rs.getDouble("salary");

    System.out.print("Row Number=" + rs.getRow());
    System.out.print(", SSN: " + ssn);
    System.out.print(", Name: " + name);
    System.out.println(", Salary: $" + salary);

  }
     conn1.close();
     conn2.close();

//Standard error handling.
}catch(SQLException se){
  //Handle errors for JDBC
  se.printStackTrace();
}catch(Exception e){
  //Handle all other errors
  e.printStackTrace();
}finally{
  //Ensure all database resources are close
  try{
    if(conn1!=null)
     conn1.close();
    if(conn2!=null)
     conn2.close();
  }catch(SQLException se){
     se.printStackTrace();
  }

}//end try

System.out.println("Goodbye!");
```

```
  }

}//end Client class

class ConnectionMgr{

  //Create the Singleton variables
  private static ConnectionMgr connMgr= null;
  private static Connection conn = null;

  //Private attributes used to make connection
  private String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
  private String user = "toddt";
  private String pwd = "mypwd";

  //Ensure no one can instantiate the object except this class

  private ConnectionMgr() {}

  //Private method to create connection. The ConnectionMgr does
  //not let the client do this.
  private void openConnection() throws Exception{
    String msg = "Connection has not been opened. Begin connection phase...";
    System.out.println(msg);

    //Load a driver
    String driver = "oracle.jdbc.driver.OracleDriver";
    Class.forName(driver).newInstance();

    //Obtain a Connection object
    System.out.println("Connecting to database...");
    System.out.println();
    conn = DriverManager.getConnection(jdbcUrl,user,pwd);
    System.out.println("Connection successful..");

  }

  //Access method to return a reference to a Connection object
  public Connection connect() throws Exception{

    //If it doesn't exist open a connection which
    //instantiates a Connection object.
    if(conn==null||conn.isClosed())
      openConnection();

    return conn;

  }

  //Public method to get the only instance of Connection Manager.
  public static ConnectionMgr getInstance(){

    //If it has not been initialized, do it here.
    //Otherwise just return existing object.
    if(connMgr==null)
      connMgr = new ConnectionMgr();

    return connMgr;

  }
```

```
}//end ConnectionMgr class
```

The output from Listing 10−2 is as follows:

```
Verify ConnectionMgr object ids are equal:
cm1 object id: Chapter10.ConnectionMgr@3e86d0
cm2 object id: Chapter10.ConnectionMgr@3e86d0

Obtain database connections:
Connection has not been opened. Begin connection phase...
Connecting to database...

Connection successful..
Verify object ids are the same:
conn1 object id: oracle.jdbc.driver.OracleConnection@4741d6
conn2 object id: oracle.jdbc.driver.OracleConnection@4741d6

Listing result set contents:
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5000.0
Row Number=2, SSN: 419876541, Name: Larry, Salary: $1500.0
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2000.95
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3080.0
Row Number=5, SSN: 987654321, Name: John, Salary: $4351.0
Goodbye!
```

Notice from the output that the object references for the ConnectionMgr variables are identical. The same holds true for the Connection object variables. This is the marquee feature of the Singleton pattern.

I can easily extend Listing 10−2 to provide a lot more functionality. For example, I can provide a reference counter to allow a maximum number of Connection objects to be created. Going one step further, if I store each Connection object in a container such as a Vector, and provide access methods to the container, I can create a connection pool. Now when clients request a connection, the ConnectionMgr object can provide one that has already been created from the pool. This approach saves the time it would take to create new connections on demand.

XRef In Chapter 14, “Using Data Sources and Connection Pooling,” I show you how to create a connection pool and use the JDBC Optional Package connection−pooling features.

I can also define a boolean flag in the ConnectionMgr class to indicate when to open a database connection. Currently, I open the connection only when the getConnection() method is called. This technique is called *lazy initialization* — that is, the connection is opened only when requested. The other option is to open the connection from the getInstance() method. If you are creating a connection pool it may prove beneficial to initialize the Connection objects in the beginning. This can save time because the client does not have to wait for the connection to be opened; it retrieves the reference immediately. However, you incur the overhead of having numerous open database connections that may never be used.

Summary

The Singleton pattern proves useful when you need to ensure that only one instance of an object exists. An object that acts as a manager is typically implemented as a Singleton because you usually want only one

instance to exist so that all activity will funnel through it.

With respect to JDBC programming, a connection manager object is an ideal candidate for a Singleton object. Ensuring that all requests for database connections go through one point can help you implement security schemes and abstract connection details from client object.

As a brief overview, to create a Singleton object you do the following:

- Make the class constructor private so clients cannot instantiate the class.
- Define a private reference to the Singleton object in the class.
- Provide a global access method to return a reference to the Singleton object to the client

Chapter 11: Producing Objects with the Factory Method Pattern

In This Chapter

- Details of the Factory Method pattern
- Components of the Factory Method
- Applying the Factory Method to JDBC programming

This chapter provides details on the creational pattern called the Factory Method. You will find this pattern used frequently in object−oriented programming to create other objects. The general approach of this pattern is to define a factory method in a base class and rely on the subclasses to provide implementations that create specific objects.

Note I use the term Factory Method to refer to the pattern definition. I use the term factory method to refer to the method within a factory that creates a product object. The term factory refers to an object that implements a factory method.

I begin this chapter with a discussion of the details of the Factory Method pattern. Next, I present the structure of this pattern and provide two implementation examples that create JDBC Connection objects. The first is a simple factory that produces connection objects upon request. The second is an enhanced version of the first: It can support multiple databases, accept connection parameters, and has a connection manager to control access.

What Is the Factory Method Pattern?

The purpose of the Factory Method pattern, like that of other creational patterns, is to create other objects. Patterns in this category tend to abstract the object−creation process from the client. In other words, the client knows nothing about how the factory produces the objects, only that it (the client) receives the correct object when it makes a request to the factory method.

The Java and JDBC APIs use factory methods extensively. For example, DriverManager is a factory that creates Connection objects with the getConnection() method. The Connection object is also a factory that instantiates Statement objects with the Connection.createStatement() method. These are just two examples; I’m sure you will discover more if you look in greater detail at the API documentation.

It may help to think of the Factory Method pattern as a brick−and−mortar factory. Instead of producing tangible items like washers and dryers, it produces objects. Just as some manufacturing facilities use a machine to create a product, the Factory Method pattern uses a method. As you might expect, this method is called the factory method.

Producing objects via factories has many benefits. First, you abstract object creation from the client. This enables you to change the way you create the objects in the factory without affecting the client. In addition, using factories provides you with a single consistent method for creating objects. For example, suppose you need to create Connection objects with certain timeout limits. Defining a timeout parameter in a factory enables you to produce Connection objects with the desired timeout value time after time.

Introducing the Factory Method Structure

In general, the Factory Method pattern relies on subclasses to create instances of objects. The base class defines the interface and the factory method to create the product object. Subclasses provide the implementation of the factory method. This architecture yields a lot of flexibility and extensibility. As you need to support different objects, you can create new subclasses and implement a factory method to create the different objects.

To go one step further, you may define interfaces for the factory products and reference them in the client instead of concrete product classes. You may find this method beneficial when your client cannot anticipate which factory product it will use. By referencing the base class in the client, it can use any subclass product. By defining factory interfaces you allow the client to work with any factory, and by defining the product interfaces you allow the client to use any product. By letting the client rely on interfaces you completely remove application−specific code.

You must define four classes to implement the Factory Method in its purest form. However, you can combine the functionality of some classes into one depending upon your needs. For example, if you only have one product you can implement the factory method in a base class and only override it in a subclass when you need to create different products. Defining an abstract class and extending it to build a single product is needlessly complex.

Like most design patterns, this one relies heavily on abstract classes and interfaces. Figure 11−1 shows the relationships among the classes used in this pattern. The following list provides an explanation of the components shown in the figure:

1. FactoryInterface: The base class or interface that defines the factory method but does not implement it. You can create the FactoryInterface as an interface, abstract class, or standard class.
2. Factory: The class that implements the FactoryInterface. When the FactoryInterface is a class or abstract class, the Factory class extends it. When the FactoryInterface is a pure interface, this class implements it.
3. ProductInterface: The base class or interface that defines the Product objects that the Factory object creates.
4. Product: The class that implements the ProductInterface. When the ProductInterface is an abstract class, the Product class extends it. When the ProductInterface is a pure interface, this class implements it.

Abstract Factory Pattern

The book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (Addison−Wesley, 1995) also describes the Abstract Factory, a pattern very similar to the Factory Method. Like the Factory Method pattern, this pattern abstracts the object−creation process from the user. The main difference between the two is that the Abstract Factory pattern creates families, or groups, of related products. The Factory Method, like other creational patterns, only creates one product per subclass.

Implementing this pattern requires you to define an abstract class or interface that defines multiple factory methods to create related objects. The subclasses then provide the implementation to create the objects belonging to that family.

With regard to JDBC programming, you can use the Abstract Factory pattern to create factories that produce families of database−specific objects. You may have an Oracle family, an SQL Server family, and an Informix family of objects. Each factory will be dedicated to a specific database and create objects for that family. For example, an Oracle Abstract Factory object could produce objects, such as Connection and Statement, specifically tailored to Oracle. This approach is particularly helpful when a vendor extends the JDBC API to take advantage of a particular database's functionality. Using the Abstract Factory pattern you can build frameworks that support different database systems and take advantage of the extra functionality.

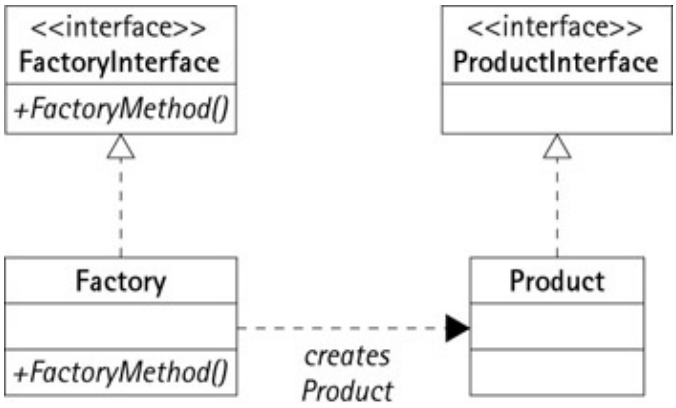

Figure 11−1: UML class diagram for the Factory Method pattern

Using the Factory Method

The defining feature of the factory pattern is that the subclasses decide which Product object to create. As I mentioned earlier, you define the factory method in an interface or abstract class and provide the implementation in the subclasses. This will enable you to use the different subclasses to create the objects you need.

Although the pattern often focuses on factories that create different products of the same type, you can also use it to create only one product. In this case, you can opt to implement the factory method in an abstract class. This will enable you to create subclasses that override the method and create different products if needed.

An advantage of using the Factory Method pattern is that it uses polymorphism. Because the client relies on the base class interface, you can create additional factories without affecting the client object. The client object only cares that the factory is of the base class type.

A disadvantage of using the Factory Method pattern is that you must define a new factory for each additional product you want to support. If you have a lot of product classes, you may find this pattern difficult to manage because the number of subclasses increases to match the number of products. You may want to consider the Prototype pattern (described in *Design Patterns*), which focuses on "cloning" objects versus creating new subclasses.

So far I have provided a lot of theory; now for an example. Listing 11−1 shows you an implementation of the Factory Method pattern applied to JDBC programming. I create a simple connection factory that produces Connection objects for an Oracle database. Figure 11−2 shows the class diagram for the example and Table

11−1 maps the standard Factory Method's components to the classes in the example.

Table 11−1: Mapping of Factory Method Components to BasicFactory Example Components

Factory Method Component	BasicFactory Components
FactoryInterface	ConnectionFactory
Factory	ConnectionFactory
ProductInterface	Connection (interface defined in java.sql package)
Product	Connection
Client	BasicFactory

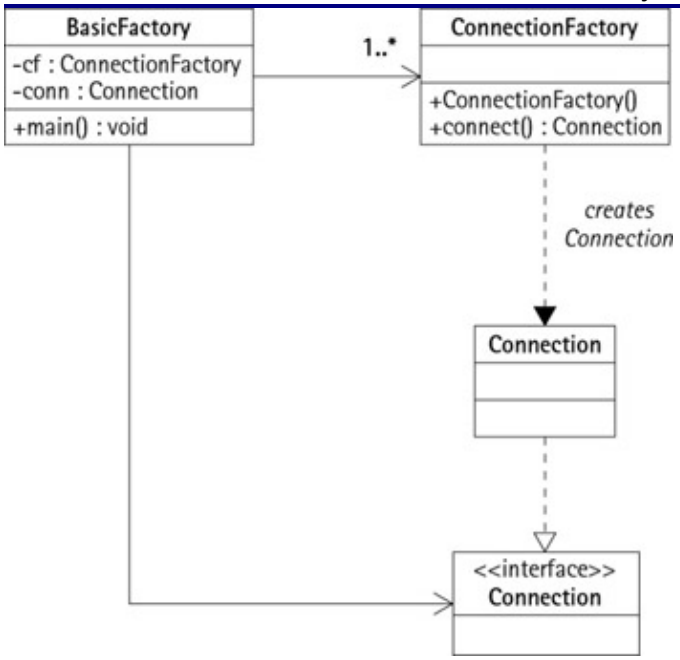

Figure 11−2: UML class diagram of BasicFactory method example

Listing 11−1: BasicFactory.java

```
package Chapter11;

import java.sql.*;

//Factory class
class ConnectionFactory{

  public ConnectionFactory() {}

  //Factory method to return a Connection object
  public Connection connect()throws Exception{

    //Load a driver
    String driver = "oracle.jdbc.driver.OracleDriver";
    Class.forName(driver).newInstance();
```

```
    //Set connection parameters
    System.out.println("Connecting to database...");
    String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
    String user = "toddt";
    String pwd = "mypwd";

    //Create a Connection object
    Connection conn = DriverManager.getConnection(jdbcUrl,user,pwd);
    System.out.println("Connection successful...");

    //Return Connection object
    return conn;

  }

}//end ConnectionFactory


//Class to demonstrate use of Factory Method
public class BasicFactory{

  public static ConnectionFactory cf = null;
  public static Connection conn = null;

  public static void main(String[] args) {

    try{

      //Instantiate a ConnectionFactory object
      cf = new ConnectionFactory();

      //Obtain a connection to the database
      conn = cf.connect();

      //Populate a result set and show the results
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM Employees");

      //Iterate through the result set
      while(rs.next()){

        //Retrieve column values and display values
        int ssn= rs.getInt("ssn");
        String name = rs.getString("name");
        double salary = rs.getDouble("salary");
        System.out.print("Row Number=" + rs.getRow());
        System.out.print(", SSN: " + ssn);
        System.out.print(", Name: " + name);
        System.out.println(", Salary: $" + salary);

      }

    //Standard error handling.
    }catch(SQLException se){

      se.printStackTrace();

   }catch (Exception e){

      e.printStackTrace();
```

```
   //Ensure all database resources are closed.
   }finally{
      try{
        if (conn!=null)
          conn.close();
      }catch(SQLException se){
        se.printStackTrace();
      }
   }//end try

   System.out.println("Goodbye!");
  }

}//end BasicFactory class
```

The output from Listing 11−1 is as follows:

```
Connecting to database...
Connection successful...
Row Number=1, SSN: 111111111, Name: Todd, Salary: $5000.55
Row Number=2, SSN: 419876541, Name: Larry, Salary: $1500.75
Row Number=3, SSN: 312654987, Name: Lori, Salary: $2000.95
Row Number=4, SSN: 123456789, Name: Jimmy, Salary: $3080.05
Row Number=5, SSN: 987654321, Name: John, Salary: $4351.27
Goodbye!
```

In Listing 11−1, I combine the functionality of the FactoryInterface and Factory into one class, ConnectionFactory, because I have only one product, a Connection object for an Oracle database. If I want to support other databases I can create a class that extends ConnectionFactory, and override the factory method to create the appropriate Connection object. In this example, the connect() method is the factory method. It is responsible for creating the product, a Connection object.

The ConnectionFactory abstracts the database connection process from the client. The client knows nothing about how the ConnectionFactory implements this process; it only cares whether or not the method returns a valid Connection object. In the preceding example I hard−code the connection parameters such as username, password, JDBC URL, and driver. Ideally you want to dynamically capture these values, most likely by using XML or .properties file.

The BasicFactory class demonstrates the Factory Method pattern in action. Using this class is straightforward. In the Listing 11−1 I instantiate a ConnectionFactory object and retrieve a Connection object using its connect() method. Next, I create a Statement object and populate a result set by executing a simple SELECT query. Once populated, I loop through the result set to list its contents.

Listing 11−1 represented a very simple implementation of the factory method. Although it hides the details of connecting to a database from a client, it is very limiting. For example, the client directly instantiates the factory, which tightly couples the two objects. Including the concrete factory class name binds the code to using that class, which is akin to hard−coding usernames and passwords. To use a different factory you have to change the class name and recompile the code.

Fortunately, you have several options when implementing the Factory Method pattern to help solve this problem. You can apply any of the following variations:

1. Implement the factory object as the Singleton pattern to guarantee that you receive your objects from

a single source.
2. Parameterize the Factory class to create different product objects. Clients can supply different parameters to the factory to retrieve different products. This option enables you to change the factory method dynamically based on your application's current need.
3. Define an interface for the Factory and let the client reference it in order to change factories as needed.

In my next example, I build a connection manager that uses a factory that incorporates the enhancement listed above. The following list identifies some of the features:

- The ConnectionManager and Factory objects are implemented as a Singleton.
- You can supply usernames, passwords and database IDs as parameters to control what database you connect to and which database user account you connect with.
- I define an abstract factory class and build specific factory classes that provide connections to either an Oracle or an Access database.

I also remove the client's reliance on concrete classes in the following example. The product and factory class names appear only in the connection manager, not in the client's code base. This architecture creates a framework in which you can add your own database connection factory for your system. Remember, the authors of *Design Patterns* suggest that you program to interfaces rather than to implementations. Listing 11−2 provides the code for an enhanced version of the Factory Method pattern in which I show you how to do this.

Listing 11−2: EnhancedFactory.java

```
package Chapter11;

import java.sql.*;

//Abstract class that defines interface for factory objects
abstract class AbstractConnFactory {

  //Protected variables that hold database specific information
  protected static Connection conn;
  protected String dbType = null;
  protected String user = null;
  protected String password = null;
  protected String driver = null;
  protected String jdbcUrl = null;
  protected String database = null;

  //Close the database connection
  public  void close() throws SQLException {

    //Check if conn is null, if not close it and set to null
    if (conn!=null){

      System.out.println("Closing connection");
      System.out.println();
      conn.close();
      conn = null;

    }

  }
```

```
  //Access method to return a reference to a Connection object
  public Connection connect() throws Exception{

    if(conn!=null){
      System.out.println("Connection exists. Returning instance...");

    }else{

      System.out.println("Connection not created.
Opening connection phase...");
      openConnection();

    }//end if

    return conn;

  }

  //Private method to create connection.
  private void openConnection() throws Exception{

    //Register a driver
    Class.forName(driver).newInstance();

    //Obtain a Connection object
    System.out.println("Connecting to " + dbType + " database...");
    conn = DriverManager.getConnection(jdbcUrl, user, password);
    System.out.println("Connection successful..");

  }

}//end AbstractConnFactory

//Subclass of the AbstractConnFactory for connecting
to an ODBC database.
class OdbcConnFactory extends AbstractConnFactory{

  //Private variables
  private static OdbcConnFactory ocf= null;

  //Private constructor
  private OdbcConnFactory() {

   jdbcUrl ="jdbc:odbc:";
   driver = "sun.jdbc.odbc.JdbcOdbcDriver";

  }

  //Public method used to get the only instance of OdbcConnFactory.
  public static synchronized AbstractConnFactory getInstance(){

    //If not initialized, do it here. Otherwise return existing object.
    if(ocf==null)
      ocf = new OdbcConnFactory();

      return ocf;

  }

  //Overridden method to open a database connection
  public Connection connect() throws Exception{
```

```
    //Configure the JDBC URL
    jdbcUrl = jdbcUrl + database;

    //Call the base class method to provide the connection
    return super.connect();

  }

}//end OdbcConnFactory

// Subclass of the AbstractConnFactory for connecting
to an Oracle database.
class OracleConnFactory extends AbstractConnFactory{

  //Private variables
  private static OracleConnFactory ocf= null;

  //Private constructor
  private OracleConnFactory() {

   jdbcUrl = "jdbc:oracle:thin:@localhost:1521:";
   driver = "oracle.jdbc.driver.OracleDriver";

  }

  //Public method used to get the only instance of OracleConnFactory.
  public static synchronized AbstractConnFactory getInstance(){

    //If not initialized, do it here. Otherwise just return
existing object.
    if(ocf==null)
      ocf = new OracleConnFactory();

      return ocf;

  }

  //Overridden method to open a database connection
  public Connection connect() throws Exception{

    //Configure the JDBC URL
    jdbcUrl = jdbcUrl + database;

    //Call the base class method to provide the connection
    return super.connect();

  }

}//end OracleFactory

//Class to demonstrate the enhanced Factory Method
public class EnhancedFactory {

  //Only reference to ConnectionManager
  static public ConnectionManager cm = null;

  //Main method
  public static void main(String[] args){

    try{
```

```
      //Retrieve the only instance of ConnectionManager
      cm = cm.getInstance();

      //Create and close a connection to the Oracle database to
      //demonstrate that it works.
      Connection conn = cm.connect(cm.ORACLE, "toddt", "mypwd", "ORCL");
      cm.close();

      //Open a connection to an Access database using ODBC
      conn = cm.connect(cm.ODBC, null, null, "employees");
      cm.close();

    //Catch all the relevant errors
    }catch(ConnectionManagerException cme){
     cme.printStackTrace();
    }catch(SQLException se){
      se.printStackTrace();
    }catch(Exception e){
      e.printStackTrace();
    //Use finally block to ensure database resources are closed
    }finally{
      if(cm!=null)
        try{
          cm.close();
        }catch(SQLException se){
          se.printStackTrace();
        }
    }

  }//end main()

}//end EnhancedFactory


//Class that manages database connections
class ConnectionManager {

  //Constants to represent database types
  public static final int ORACLE = 100;
  public static final int ODBC = 200;

  //Variables to hold only instance of ConnectionManager class
  private static ConnectionManager connMgr = null;

  //Holds reference to the specific connection factory
  private static AbstractConnFactory acf =null;;

  //Private constructor
  private ConnectionManager() {}

  //Method that provides connection logic
  public Connection connect (int dbType, String user,
                      String password, String db) throws Exception{

    //Examine the dbType parameter and assign the appropriate
    //factory to the
    //acf, which is the base class type.
    switch(dbType){

      //Specific factories are Singletons so get the only
```

```
      //instance and set the appropriate connection values.
      case ORACLE:
        acf = OracleConnFactory.getInstance();
        acf.dbType = "Oracle";
        break;

      case ODBC:
        acf = OdbcConnFactory.getInstance();
        acf.dbType="ODBC";
        break;

      //Error handling for unsupported database types.
      default:
        throw new SQLException("Type not supported");

    }//end switch

    acf.database=db;
    acf.user=user;
    acf.password=password;

    //Connect to the database and return reference.
    Connection conn = acf.connect();

    return conn;

  }

  //Close the database connection.
  public void close() throws SQLException{

    acf.close();

  }

  //Public method used to get the only instance of ConnectionManager.
  public static synchronized ConnectionManager getInstance(){

    if(connMgr==null)
      connMgr = new ConnectionManager();

      return connMgr;

  }

}//end ConnectionManager


//Used to handle ConnectionManager specific errors
class ConnectionManagerException extends SQLException {

  //default constructor
  public ConnectionManagerException(){

    super();

  }

  //Constructor that allows you to specify your own error messages.
  public ConnectionManagerException(String msg){
```

```
    super(msg);

  }
}// end ConnectionManagerException
```

The output from Listing 11−2 is as follows:

```
Connection not created. Beginning connection phase...
Connecting to Oracle database...
Connection successful..
Closing connection

Connection not created. Beginning connection phase...
Connecting to ODBC database...
Connection successful..
Closing connection
```

Listing 11−2 uses numerous objects. Figure 11−3 provides a UML class diagram showing the relationships of the classes used in the example. To further help understand the classes, Table 11−2 maps the Factory Method components to the classes in the example.

In the code listing, I create the AbstractConnFactory to represent the FactoryInterface. I make the class abstract for two reasons. First, I do not want the class to be directly instantiated. Rather, I want clients to use one of the connection factories, either Oracle or ODBC. Secondly, the two factories share implementations of the openConnection() and the close() methods. If the methods need to be different, I can always override them in a subclass.

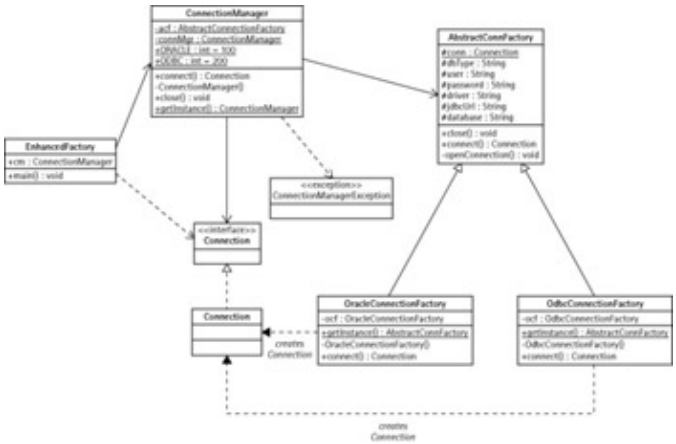

Figure 11−3: UML class diagram of the enhanced Factory Method example

Table 11−2: EnhancedFactory Example Classes

Factory Method Components	EnhancedFactory Components
FactoryInterface	AbstractConnFactory class
Factory	OracleConnFactory class
	OdbcConnFactory class
ProductInterface	java.sql.Connection interface
Product	java.sql.Connection interface

Client ConnectionManager

Interfaces versus Abstract Classes

Sometimes it is not obvious when you should use an interface or abstract class. Both provide similar functionality in that neither can be instantiated and each defines a common interface for subclasses.

An interface is an abstract class without any method implementations. As a result, no storage is associated with an interface. In Java, combining interfaces in a class definition enables you to mimic multiple inheritance. This in turn enables you to upcast the class to any interface type included in the definition.

The advantage of an abstract class is that you can provide common implementation code without requiring subclasses to rewrite the same code.

When deciding which one to use, you should consider if you need multiple inheritance, or if you have numerous methods with identical implementations, that you can share among subclasses.

In Listing 11−2 the two factories, OracleConnFactory and OdbcConnFactory, extend the AbstractConnFactory class. This allows the subclass access to the protected properties and methods of the base class, and ensures that they share the same interface. I override the connect() method of the base class in order to configure the jdbcUrl property. However, I still call the base class's implementation to take advantage of how it manages the Connection object. You can easily add additional functionality in these factory subclasses. For example, Oracle has numerous extensions to the JDBC API that enables you to take advantage of specific Oracle characteristics. You can use those extensions in this class.

You may notice the getInstance() method implemented in each subclass. Instantiated subclass objects behave as Singletons to divvy out a Connection object specific to each database.

The AbstractConnFactory class is the workhorse of the example. As I mentioned earlier, it implements all the database's connection−related methods: connect(), openConnection(), and close(). I let the subclasses handle the chores relevant to the specific database. In this example, they set specific driver information and configure the jdbcUrl variable. This class is also a Singleton, so it can control access to the connection logic.

XRef Chapter 10, "Building the Singleton Pattern," provides details on implementing the Singleton pattern.

The ConnectionManager object, also a Singleton, controls access to the factories. The connect() method returns a Connection object. It also accepts parameters in order to allow a client to specify which type of database to connect to, and the username and password to connect with. The heart of the method is the following code:

```
switch(dbType){

    //Specific factories are Singletons so get the only
    //instance and set the appropriate connection values.
    case ORACLE:
      acf = OracleConnFactory.getInstance();
      acf.dbType = "Oracle";
      break;

    case ODBC:
      acf = OdbcConnFactory.getInstance();
```

```
      acf.dbType="ODBC";
      break;

    //Error handling for unsupported database types.
    default:
      throw new ConnectionManagerException("Type not supported");

}//end switch
```

The switch block contains the logic used to create the correct factory based on the input parameter dbType, which maps to two int constants, ORACLE and ODBC, defined in the class. The variable acf is of type AbstractConnFactory. Depending on the value of the parameter dbType, it is assigned a reference to either the OracleConnFactory or the OdbcConnFactory class. Depending upon the database type, information is propagated up to the base class to populate the variables that hold the connection parameters. In this example, the variable dbType in the base class is only used to identify the factory type.

The example also has some supporting classes. The java.sql.Connection interface provides the ProductInterface component for my example, and the two factories create Connection objects as the Product.

You may have noticed that I create my own SQLException class called ConnectionManagerException to handle ConnectionManager−related issues. In the preceding example I trap the error that occurs if you supply an unsupported database type to the connect() method.

Summary

The Factory Method pattern is likely one that you will often implement. It enables you to create consistent objects while abstracting the details of their creation from the client.

You can also enhance the basic Factory Method pattern to separate specific implementations from your application. Clients can refer to the base class interface for the factory and products to allow them to use the objects they need when they need them.

To summarize, you will find the Factory Method useful if:

- You need to abstract object creation from the client.
- You want subclasses to decide how to create objects and what objects to create.
- You cannot anticipate the type of object you must create at runtime.

Chapter 12: Creating a Façade Pattern

In This Chapter

- Understanding the purpose of the Façade pattern
- Introducing the structure of the Façade pattern
- Applying the Façade pattern to JDBC programming
- Implementing the Façade pattern

The previous chapters focused on creational patterns. In this chapter, I switch gears and cover a structural pattern, the Façade. This category of design patterns focuses on combining objects and classes to create larger, more feature−rich, structures. Class patterns in this category, such as the Adapter, use inheritance, while object patterns, like the Façade, use composition to create the new structures.

The Façade pattern provides a front−end to a complex object subsystem. Clients use the Façade object to interact with the subsystem. This minimizes the number of methods and objects a client needs to know about. It also allows the subsystem to change without affecting the client.

I begin the chapter with a discussion on the details on the Façade pattern. Next I present its structure, and finally provide an implementation example that hides the details of making JDBC database connections and SQL queries. Unlike the previous chapters, this one places more emphasis on the example.

What Is the Façade Pattern?

The word façade means “an artificial or deceptive front” and that is exactly what this pattern is. The Façade pattern puts an “artificial” front, or interface, onto an object subsystem to simplify a client’s use of it. Instead of having to know about methods from all the objects in the subsystem, the client only needs to understand the methods defined by the Façade object.

The only object in the pattern, the Façade object, provides the common interface with an object subsystem. Many different objects can exist in the subsystem, and the interactions among them can be highly complex. However, the Façade pattern wraps the complexity into one object. From the client’s point of view, the Façade object is the subsystem.

Façades occur everywhere in the physical world. Consider your microwave, for example. The control panel, or the Façade for the microwave, has numerous buttons, and each of which controls some complex functionality that in turn requires that many components or subsystems work together. Setting the microwave to run on HIGH power for one minute requires only a few control−panel steps, but it sets off a flurry of activity among subsystems inside the microwave. You know nothing about how the microwave completes its task, only that it runs on HIGH for one minute.

The same scenario occurs in JDBC programming. For example, a lot of background activity occurs to return a Connection object when you call the Connection.getConnection() method. What happens behind the scenes doesn’t matter, because all you need is a database connection, not the details of what the driver does to make the connection.

Nonetheless, the JDBC API is an example of an object subsystem that you can wrap with the Façade pattern. Although using the API is not complex, a lot of redundant method calls occur. For instance, opening a database connection requires the same method calls every time. Executing queries is equally repetitive.

You can create a Façade object to hide these details so a client only needs to know one or two methods to access and interact with a database. In fact, this is what I do in this chapter's example.

Introducing the Structure of the Façade Pattern

The Façade pattern has the simplest pattern structure I have covered yet. It has only one object, which provides the gateway into a subsystem of objects. Figure 12−1 shows the general structure.

The Façade object needs intimate knowledge of the classes in the subsystem. When a client sends a request to the object, the Façade object must know exactly which subsystem object to send the request for execution. The client only needs to know the correct Façade method to call and nothing about the subsystem.

However, the subsystem objects have no knowledge of the Façade object. As indicated in Figure 12−1, there is an unidirectional relationship between the Façade object and the subsystem components. That is, they operate independently of the Façade and treat it as their client.

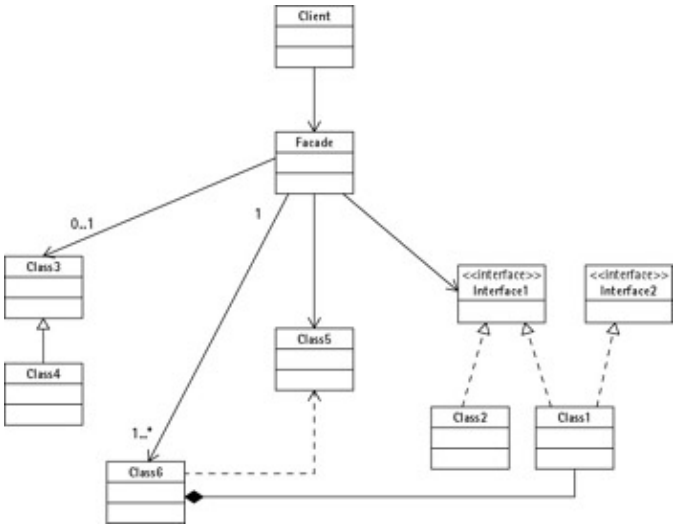

Figure 12−1: Façade pattern structure

Implementing the Façade Pattern

Although you may find the Façade pattern conceptually simple, you will probably find it the most difficult to implement. Reducing a complex subsystem to an interface with a few methods for a client to use is challenging. You need to spend a significant portion of your design time identifying the interface that will best meet the client's needs.

Apart from designing the interface, you have a couple of options available when implementing the pattern. First, you can make the Façade object a Singleton to ensure that only one access point into the subsystem

exists. Why? Because you may want to control the number of clients using the subsystem, and having one controlling object enables you to do this. Or, with respect to JDBC programming, you may want the clients to share one database connection or retrieve a Connection object from a pool.

Another implementation option is to define the Façade in an abstract class or interface and provide the implementation in concrete classes. Referencing the Façade interface in the client can allow it to use any of the concrete Façade classes. This option allows a client to access different subsystems that share the same interface. In this way, you can remove the implementation details of specific Facades from the client's application and allow them to choose one at runtime.

Now for an example of how the Façade pattern can abstract from a client all the JDBC tasks associated with opening database connections, statement preparation, error handling, and database−resource cleanup. Figure 12−2 provides a UML class diagram of the classes used in my DbFacade example.

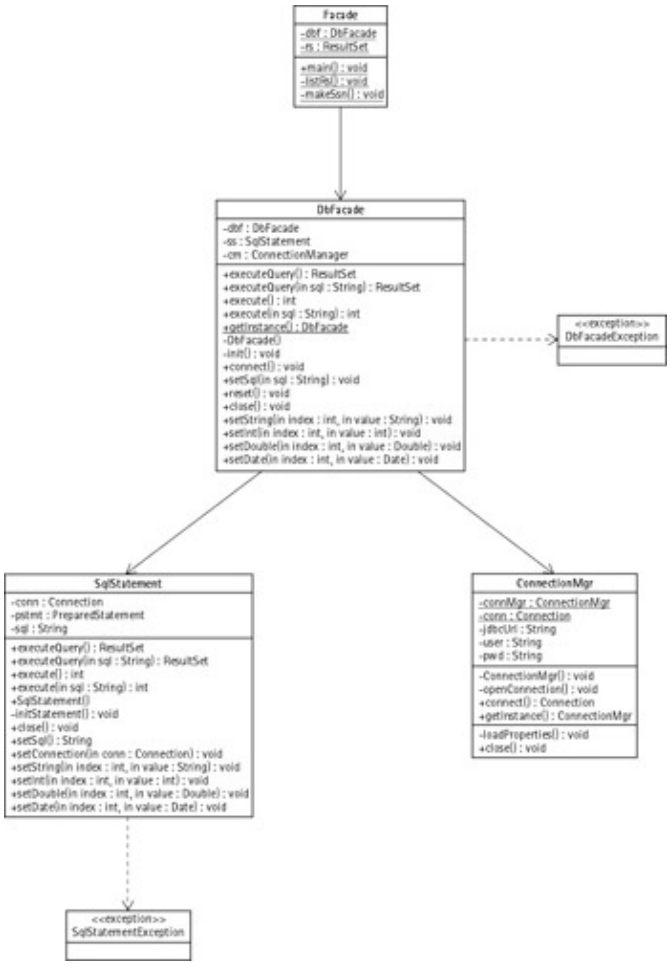

Figure 12−2: UML class diagram for DbFacade example

My Façade object, DbFacade, allows a client to execute either static or parameterized SQL statements with one of two method calls, executeQuery() and execute(). The first method enables you to execute SQL SELECT statements, and the second enables you to execute either INSERT, UPDATE, DELETE, or DDL statements. As you probably expected, the executeQuery() returns a ResultSet and the execute() returns an int representing an update count.

The architecture of the DbFacade wraps two objects, SqlStatement and ConnectionMgr. The SqlStatement object abstracts the functionality of a PreparedStatement object to make the SQL calls, and returns the results.

In order to do so it requires a reference to a Connection object to instantiate the private PreparedStatement object, which executes all the SQL statements, either static or parameterized. Both the executeQuery() or the execute() methods work by either presetting the SQL statement with setSql() or supplying an SQL statement as String as a parameter.

If you want to use a parameterized SQL statement you must use the DbFacade setSql() method to initialize the PreparedStatement object reference within SqlStatement. The method requires a String parameter used when creating the internal reference to a PreparedStatement object. Next, you must bind the parameters to values using my custom set*XXX* methods in the DbFacade object. These methods provide the same functionality as do the standard PreparedStatement and set*XXX*() methods. In fact, the DbFacade object passes these calls through to the SqlStatement object, which uses the native PreparedStatement methods. Remember, if you use an SQL statement with parameters you must bind the values to the parameters or a runtime error occurs.

DbFacade also uses a ConnectionMgr object, which is implemented as a Singleton, to manage connections. The DBFacade object never holds a direct reference to a Connection object. Instead it retrieves the Connection object reference and passes it to the SqlStatement object in the init() method. It also closes the connection when requested via the close() method.

I implement DbFacade as a Singleton to allow only one access point into the subsystem. I use this pattern because I want the DBFacade object to manage the SqlStatement and ConnectionMgr objects. As with the other Singleton examples listed in chapter 10, the client calls the getInstance() method to retrieve a reference to the DBFacade object and work with it.

XRef Chapter 10, “Building the Singleton Pattern,” describes the ins and outs of creating Singleton objects.

The Facade class provides the test bed for the example by acting as the client. In it, I show various examples of how to use the DbFacade object.

The example contains a lot of code and Table 12−1 provides an overview of the objects involved in the application. To help you further understand the DbFacade object, Table 12−2 provides a list of the methods along with their descriptions. Finally, Listing 12−1 shows the complete code listing for this example.

Table 12−1: Classes in DbFacade Example

Class	Description
Facade	Test bed that illustrates the use of the DbFacade object.
DbFacade	Implements the Façade design pattern.
SqlStatement	Executes the SQL statements. Supports basic and parameterized queries using the PreparedStatement object.
ConnectionMgr	Manages database connections. Only shares one connection to the database.
DbFacadeException	Handles DbFacade−specific errors.
SqlStatementException	Handles SqlStatement−specific errors.

Table 12−2: DbFacade Method Summary

Method	Description	Return Type
getInstance()	Global−access method that returns a reference to instance of the DbFacade object.	DbFacade
connect()	Explicitly connects to the database.	void
executeQuery(String *sql*)	Executes a static SQL SELECT statement.	ResultSet
executeQuery()	Executes an SQL SELECT statement after the setSql() method has been called.	ResultSet
execute(String *sql*)	Executes a static INSERT, UPDATE, DELETE, or DDL statement.	ResultSet
execute()	Executes an INSERT, UPDATE, DELETE, or DDL statement after the setSql() method has been called.	ResultSet
setSql(String *sql*)	Preloads an SQL statement for execution.	void
reset()	Flushes entire DbFacade object, causing a re−initialization.	void
close()	Closes all database resources, including Connection and PreparedStatement objects.	void
setString(int *index*, String *value*)	Identical to PreparedStatement. setString().	void
setInt(int *index*, int *value*)	Identical to PreparedStatement.setInt().	void
setDouble(int *index*, double *value*)	Identical to PreparedStatement setDouble().	void
setDate(int *index*, Date *value*)	Identical to PreparedStatement. setDate()	void
DbFacade()	Private constructor.	
init()	Retrieves a reference to the ConnectionMgr object and creates the SqlStatement object. Also causes a database connection to be created.	void

Listing 12−1: Facade.java

```
package Chapter12;

import java.io.*;
import java.util.*;
import java.sql.*;
import java.sql.Date;
import Chapter5.MakeEmpDb;//REMOVE


public class Facade {

    //Set a ResultSet object to hold results.
    public static ResultSet rs = null;
    public static DbFacade dbf = null;

    public static void main(String[] args) {

      try{
```

```
String[] s = new String[0];//REMOVE
MakeEmpDb.main(s); //REMOVE

  //Get an instance reference to the DbFacade object
  dbf = DbFacade.getInstance();

  //Retrieve an employee as a baseline.
  System.out.println("Static SELECT with executeQuery()");
  String sql = "SELECT * FROM employees WHERE name='Todd'";
  rs=dbf.executeQuery(sql);
  listRs();

  //Give myself a BIG raise. Demonstrates setSql.
  System.out.println("Parameterized UPDATE with execute() " +
                       "to update my salary and hire date");
  dbf.setSql("UPDATE employees SET salary = ?" +
            "WHERE name = ?");
  dbf.setDouble(1,100000.75);
  dbf.setString(2,"Todd");
  dbf.execute();

  //Change my hire date. Demonstrates static SQL execution.
  dbf.execute("UPDATE employees SET hiredate = " +
              "{d '1989-09-16'} WHERE name = 'Todd'");

  //List results of changes.
  System.out.println("Verify updates.");
  rs=dbf.executeQuery(sql);
  listRs();

  //Demonstrate INSERT
  System.out.println("Add new employee with INSERT " +
                     "and execute() then verify results.");
  sql = "INSERT INTO Employees VALUES (?,?,?,?,?)";
  dbf.setSql(sql);

  //Bind values into the parameters.
  int ssn = makeSsn();
  dbf.setInt(1,ssn);
  dbf.setString(2,"Andy");
  dbf.setDouble(3,1400.51);
  Date d = new Date(System.currentTimeMillis());
  dbf.setDate(4,d);
  dbf.setInt(5,400);
  dbf.execute();

  //Verify results.
  sql = "SELECT * FROM employees WHERE name = 'Andy'";
  rs = dbf.executeQuery(sql);
  listRs();

  //Demonstrate how to close and open a connection
  System.out.println("Close and open database connection,
  then verify");
  dbf.close();
  dbf.connect();
  rs = dbf.executeQuery(sql);
  listRs();

  System.out.println("Exiting program...");
  dbf.close();
```

```
      //Handle errors
      }catch(DbFacadeException dbfe){

        dbfe.printStackTrace();

      }catch(SQLException se){

        se.printStackTrace();

      }catch(Exception ex){

        ex.printStackTrace();

      }finally{

        if(dbf!=null)
          try{
            dbf.close();
          }catch(SQLException se){
            se.printStackTrace();
          }
      }//end finally

        System.out.println("Goodbye!");

    }//end main()

    private static void  listRs() throws SQLException{


      while(rs.next()){

        //Retrieve column values and display values
        int ssn= rs.getInt("ssn");
        String name = rs.getString("name");
        double salary = rs.getDouble("salary");
        Date hiredate = rs.getDate("hiredate");
        System.out.print("Row Number=" + rs.getRow());
        System.out.print(", SSN: " + ssn);
        System.out.print(", Name: " + name);
        System.out.print(", Hiredate: " + hiredate);
        System.out.println(", Salary: $" + salary);

      }//end while

      System.out.println();
    }

    //Helper utility to generate a random SSN to prevent primary
     key violations
    private static int makeSsn(){

      double r = Math.random();
      double d = (r * 899999999);
      return (int) Math.floor(d +100000000) ;

    }

}//end Facade
```

```
class ConnectionMgr{

  //Create Connection, Statement, and ResultSet objects
  private static ConnectionMgr connMgr= null;
  private static Connection conn = null;

  //Private attributes used to make connection
  private String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
  private String user = "toddt";
  private String pwd = "mypwd";

  //Ensure no can instantiate
  private ConnectionMgr() {}

  //Private method to create connection.
  private synchronized void  openConnection() throws Exception{


    //Load a driver
    String driver = "oracle.jdbc.driver.OracleDriver";
    Class.forName(driver).newInstance();

    //Use the getConnection method to obtain a Connection object
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(jdbcUrl,user,pwd);
    System.out.println("Connection successful..");

  }

  //Global access method to return a Connection object reference
  public Connection connect() throws Exception{

    //If it doesn't exist open a connection
    if(conn==null){
      System.out.println("Connection has not been opened. " +
             "Begin connection phase...");
      loadProperties();
      openConnection();
    }

    return conn;

  }

  //Public method used to get the only instance of Connection Manager.
  public static synchronized ConnectionMgr getInstance(){

    //If not initialized, do it here. Otherwise just return
     existing object.
    if(connMgr==null)
      connMgr = new ConnectionMgr();

      return connMgr;

  }

  //Load username and password from properties file to override
   default values
  private void loadProperties() throws Exception{

    Properties prop = new Properties();
```

```
    File f = new File("database.properties");

    //If property file exists load data

    if (f.exists()) {
     InputStream is =  new FileInputStream(f);
     prop.load(is);
     is.close();
     user = prop.getProperty("user");
     pwd = prop.getProperty("password");
     jdbcUrl = prop.getProperty("JDBC_URL");
    }

  }

  //Close database connection
  public void close() throws SQLException{

    if (conn !=null){
      System.out.println("Closing Database Connection!");
      conn.close();
      conn=null;
    }

  }

}//end ConnectionMgr

class DbFacade {

  //Private singleton object
  private static DbFacade dbf = null;

  //Private variables to hold objects.
  private SqlStatement ss= null;
  private ConnectionMgr cm = null;

  //Provides a single point of access to DbFacade instance
  public static synchronized DbFacade getInstance() throws Exception{

    //If it doesn't exist create it.
    if(dbf==null)
      dbf = new DbFacade();

    return dbf;

  }

  //Private constructor to keep clients from instantiating
  private DbFacade() throws Exception {

    cm = ConnectionMgr.getInstance();
    init();

  }

  //Initialization to open a connection and pass it to the
SqlStatement object.
  private void init() throws Exception{

    Connection conn = cm.connect();
```

```
  ss = new SqlStatement();
  ss.setConnection(conn);

}

//Connect to database
public void connect() throws Exception{

  Connection conn = cm.connect();

  if(ss == null)
    ss = new SqlStatement();

    ss.setConnection(conn);
}

//Method to execute SELECT SQL statements supplied as a parameter
public ResultSet executeQuery(String sql) throws SQLException{

  return ss.executeQuery(sql);

}

//Method to execute SELECT SQL statements
public ResultSet executeQuery() throws SQLException{

  return ss.executeQuery();

}

//Method to execute all SQL statements except SELECT
public int execute() throws SQLException{

  return ss.execute();

}

//Method to execute all SQL statements except SELECT
public int execute(String sql) throws SQLException{

  return ss.execute(sql);

}

//Sets the SQL string in the SqlStatement object
public void setSql(String sql) throws SQLException{

  ss.setSql(sql);

}

//Clears the SqlStatement object
public void reset() throws Exception{

  //Set the reference to the ss to null;
  ss = null;

  //Reinitialize object
  init();

}
```

```
  //Close database connection
  public void close() throws SQLException{

    if(ss!=null)
      ss.close();

    if(cm!=null)
      cm.close();

  }

  //Set a String value in a PreparedStatement
  public void setString(int index, String value) throws SQLException{

    ss.setString(index,value);


  }

  //Set an int value in a PreparedStatement
  public void setInt(int index, int value) throws SQLException{

    ss.setInt(index,value);

  }

  //Set a double value in a PreparedStatement
  public void setDouble(int index, double value) throws SQLException{

    ss.setDouble(index,value);

  }

  //Set a Date value in a PreparedStatement
  public void setDate(int index, Date value) throws SQLException{

    ss.setDate(index,value);

  }

}//end DbFacade()

class DbFacadeException extends SQLException {

  public DbFacadeException() {

    super();

  }

  public DbFacadeException(String msg) {

    super(msg);

  }

}//end DbFacadeExecption class

class SqlStatement {
```

```
//Internal JDBC objects
private Connection conn = null;

private PreparedStatement pstmt = null;

//Holds the SQL statement for execution
private String sql;

//Default constructor
public SqlStatement(){}

//Initialize the Statement object
private void initStatement() throws SQLException{

  //Only initialize PrepareStatement if member sql is not null.
  if(sql==null)
    throw new SqlStatementException("SQL string is null");

    pstmt = conn.prepareStatement(sql);

}

//Close PreparedStatement object
public void close() throws SQLException{

  if(pstmt!=null){
    System.out.println("Closing SqlStatement!");
    pstmt.close();
    pstmt=null;
  }

}

//Used to set SQL statement and reinitialize PreparedStatement object
public void setSql(String sql) throws SQLException{

  this.sql=sql;
  pstmt=null;
  initStatement();

}

//Returns the current SQL statement
public String getSql(){

  return sql;


}

//Executes static SQL statement supplied as a parameter
public ResultSet executeQuery(String sql) throws SQLException{

  setSql(sql);

  return executeQuery();

}

//Executes an SELECT statement
public ResultSet executeQuery() throws SQLException{
```

```
    //Check to see if pstmt statement is null.
    if(pstmt ==null)
      throw new SqlStatementException("PreparedStatement
not initialized");

    return pstmt.executeQuery();

  }

  //Executes static UPDATE, INSERT, or DELETE statement
  //supplied as a parameter
  public int execute(String sql) throws SQLException{

    setSql(sql);

    return execute();

  }

  //Executes a SQL UPDATE, INSERT, or DELETE statement
  public int execute() throws SQLException{

    if(pstmt ==null)
      throw new SqlStatementException("PreparedStatement
not initialized");

      int count = pstmt.executeUpdate();
      return count;

  }

  //Sets the Connection object to valid database connection

  public void setConnection(Connection conn){

    this.conn = conn;

  }


  //The following set methods set the appropriate value
  in a PreparedStatement.
  public void setString(int index, String value) throws SQLException{

    pstmt.setString(index,value);

  }

  public void setInt(int index, int value) throws SQLException{

    pstmt.setInt(index,value);

  }

  public void setDouble(int index, double value) throws SQLException{

    pstmt.setDouble(index,value);

  }
```

```
  public void setDate(int index, Date value) throws SQLException{

    pstmt.setDate(index,value);

  }

  //End set methods

}//end SqlStatement

//Exception class to handle SqlStatement errors
class SqlStatementException extends SQLException {

  public SqlStatementException() {

    super();

  }

  public SqlStatementException(String msg) {

    super(msg);

  }

}//End SqlStatementException}
```

The output from Listing 12−1 is as follows:

```
Connection has not been opened. Begin connection phase...
Connecting to database...
Connection successful..
Static SELECT with executeQuery()
Row Number=1, SSN: 111111111, Name: Todd, Hiredate: 1995−09−16,
Salary: $5000.55

Parameterized UPDATE with execute() to update my salary and
hire date Verify updates.
Row Number=1, SSN: 111111111, Name: Todd, Hiredate: 1989−09−16,
Salary: $100000.75

Add new employee with INSERT and execute() then verify results.
Row Number=1, SSN: 586669377, Name: Andy, Hiredate: 2001−04−23,
Salary: $1400.51

Close and open database connection then verify open connection.
Closing SqlStatement!
Closing Database Connection!
Connection has not been opened. Begin connection phase...
Connecting to database...
Connection successful..
Row Number=1, SSN: 586669377, Name: Andy, Hiredate: 2001−04−23,
Salary: $1400.51

Exiting program...
Closing SqlStatement!
Closing Database Connection!
Goodbye!
```

In Listing 12−1, I first retrieve a reference to the DbFacade object and use this reference throughout the session. Next I demonstrate how to execute a static SQL statement with the executeQuery() method. After I print the result set to verify that the query executed correctly, I show how to process a parameterized query with the execute() method. Before calling the execute() method I must initialize the DbFacade object with the SQL statement using the setSql() method, and then bind the variables with the appropriate set*XXX*() methods. In the rest of the example, I demonstrate how to execute additional SQL statements as well as how to explicitly close and connect to the database.

Despite the amount of code necessary to implement the Façade pattern, using the DbFacade object is fairly straightforward because only two rules apply. The first is that you must call the DbFacade.getInstance() method to retrieve a reference to the object. This is the reference you use to work with the database.

The second rule is that you must call the setSql() method when using a parameterized query before calling the execute() or executeQuery() methods. If you want to execute static SQL statements you have two options. The first is to call the setSql() method to preload the SQL statement before you call execute(). Your second option is to call the executeQuery() method and supply a String as a parameter representing the SQL statement. The second option saves you a step.

One additional comment about the DbFacade pattern design is that the two objects, SqlStatement and ConnectionMgr, are façades themselves. The objects hide the behind−the−scenes process of sending SQL statements and creating database connections. In this case their client is the DbFacade object.

As you can see from the example, it is possible to use the Façade pattern to hide a lot of the complexity and mundane programming tasks associated with JDBC programming. The example in Listing 12−1 allows a client to retrieve results from a database with two method calls, which is certainly easier than opening connections and creating Statement objects every time you need to access a database.

Summary

In this chapter I demonstrated the uses of the Façade pattern. The pattern proves useful when you need to abstract the complexities of one or more object subsystems from the client. Using this pattern has many benefits. One is that you can create a loose coupling between the client and the subsystem: Because the client relies on the Façade for the system’s interface, changes to the underlying subsystem do not directly affect the client. Another benefit is that the Façade pattern reduces the number of objects your client must know about. To the client, the subsystem *is* the Façade object.

You can easily wrap the common functions and programming tasks of JDBC using the Façade pattern. Clients can use the object to minimize the amount of code they need to interact with the database.

Part IV: Taking It to the Enterprise

Chapter List

Chapter 13: Accessing Enterprise Data with JNDI

In This Chapter

- Introducing naming and directory services
- Understanding the Java Naming and Directory Interface (JNDI)
- Using JNDI to access data in naming and directory services
- Working with LDAP
- Using the JNDI LDAP service−provider interface
- Searching an LDAP−enabled directory service

As enterprise applications grow larger and more complex, finding application services and objects becomes more difficult. Creating distributed applications only makes the problem worse, because you must accurately track each software component's location and functionality. This task can quickly become a documentation nightmare.

In addition, this problem is not limited to software components. Finding employee or customer information in an enterprise proves equally challenging. Very rarely will you find it in a centralized location. Most often individual people store this information in spreadsheets or simple workgroup databases on file servers, and very often only these individuals know the location of this information and how to interpret it.

In general, an enterprise, or a single user, may use many different methods to store resources. Some store files in well defined directory hierarchies within file systems or Java objects in a RMI registry. Some companies use LDAP−enabled directory services to store employee information, such as phone numbers and e−mail addresses, or personalization data for Web−site users. Each of these storage techniques represents a different type of naming or directory service. Consequently, the more types of storage techniques that exist, the more difficult the task of locating objects or resources.

The numerous storage methods make your job as a developer tougher because you must use a different programming technique to retrieve data from the sources. Each has its own semantics as well as its own API. For example, you retrieve information differently from a file system, an RMI registry, and an LDAP−enabled directory.

However, JNDI provides a solution for unifying access to the many different technologies used to store data. It defines a common, flexible, interface for accessing common naming and directory services. You can use the JNDI API to access an LDAP−enabled directory, RMI registry, or file system with the same methods.

JNDI is also a cornerstone of the J2EE platform. Clients, and business components like Enterprise Java Beans (EJBs), use it to find and retrieve resources such as JDBC DataSource objects from well−known data stores within an organization.

I begin this chapter by introducing the concepts associated with naming and directory services. Next I provide the details of JNDI's architecture and show you how to use it to access LDAP−enabled directory services.

Naming and Directory Services

Simply put, naming and directory services store information in a centralized location and enable you to enter, manipulate, and retrieve this information. You can use the services for a variety of applications, such as employee directories or object repositories.

This section provides an overview of the concepts associated with naming and directory services. I begin with naming services though the concepts involved apply to directory services as well.

Naming services

Naming services are very popular storage mechanisms and provide a simple, centralized, storage facility for resources. They are often compared to phone books because they store only name−value pairs. The following is a partial list of common naming services you may use on a regular basis:

- **Domain Name System (DNS):** The Internet relies heavily upon DNS, a simple naming service that maps IP addresses to user−friendly names.
- **File systems:** These bind a user−friendly name, such as resume.doc, to a file handle used by the operating system. You use the name to access the file.
- **RMI Registry:** Java's RMI Registry maps names to Java objects.

Figure 13−1 shows a conceptual view of a naming service. As the diagram illustrates, it only maps a name to an object. Nonetheless, it does provide a single location where clients can use common names to look up the locations of resources.

Names and bindings

A *name* is a label that an object in a naming service is known by. You can create user−friendly names for objects, such as "Finance Printer" for a printer object, or use a cryptic 32−bit Object Identifier (OID). You access objects in a naming service by their names, therefore you should try to create user−friendly, descriptive names to make it easy for a client to find the desired object.

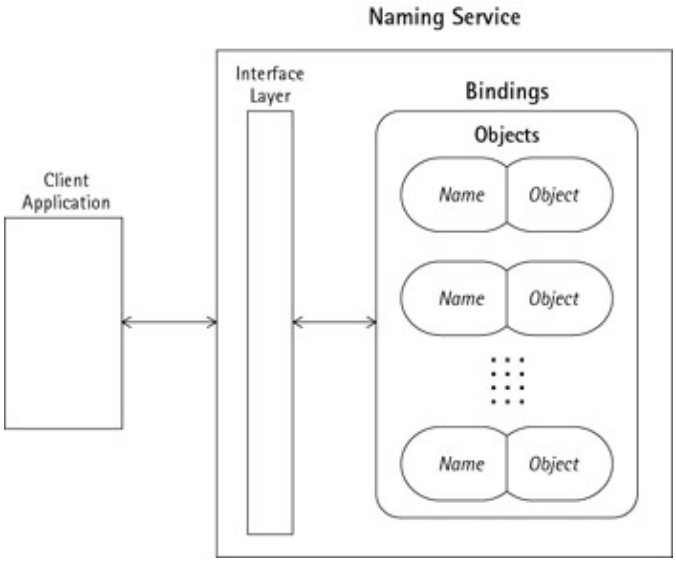

Figure 13−1: Conceptual view of a naming service

However, do not confuse objects in naming services with Java objects. An object in a naming service is an *entry*. You can store strings or binary data as objects in a naming service.

Assigning a name to an object is called *binding*. For example, you can associate the name Employee_DB with a JDBC DataSource object. In this instance the name Employee_DB is bound to the DataSource object. A DNS entry binds a name, http://www.javasoft.com/, with an object, in this case the IP address 204.160.241.83. Without a DNS, you would have to memorize or otherwise store the IP address of the hosts you interact with. DNS enables you to associate a friendly name with an IP address.

Contexts, subcontexts, and initial contexts

A *context* represents a set of objects with names bound to them. On a FAT file system the root drive, c:\, is a context because it is an object that holds bindings. For example, the file autoexec.bat is a name bound to a file handle. The same holds true for config.sys. Directories under c:\, like c:\windows, are called *subcontexts* because they are contexts (that is, objects that hold bindings) that exist within a context.

When you access a directory service with JNDI you must identify a starting context, or *initial context*. Consider the UNIX directory structure /usr/bin and the file readme.txt. The context is /usr because it contains a group of name−object bindings. The directory /usr/bin is a subcontext, and the file readme.txt is directory object. When you first initialize your JNDI program you can use either /usr or /usr/bin as the initial context. However, you cannot start with the file /usr/readme.txt, because it is a directory object and not a context.

Directory services

In Java terminology, directory services extend naming services. That is, they have the features of a naming service, but provide additional functionality. The added benefit of a directory service is that you can attach descriptive attributes to stored objects. Besides the ability to better describe objects, a directory service enables you to group the entries according to functionality, or search them according to their attributes.

Note Objects within a directory service are known as directory objects. You may also see them referred to as directory entries in some literature.

Using attributes to describe objects makes directory services very flexible. For example, you may see a directory service used for:

- **Access control** to provide a single login to a service such as a network or an application.
- **Object repositories** in which to store Java components, such as EJB's or JDBC's DataSource objects.
- **Lookup directories** to store information on people, assets, or software components.

Figure 13−2 provides a conceptual view of a directory service. As you can see, it stores objects bound to user−friendly names, just as a naming service does. However, with each object you can attach attributes. The number and type of attributes vary among directory services. Some directory services may enable you to store only numerical or textual information, while others may accept data such as Java objects.

An attribute consists of two components, an *identifier* and a *value*. The identifier is the name of the attribute and the value is the content. Values are independent of attributes. In other words, the value does not represent an attribute "type," such as an object type. You can store any type of value in an attribute as long as the directory service supports it.

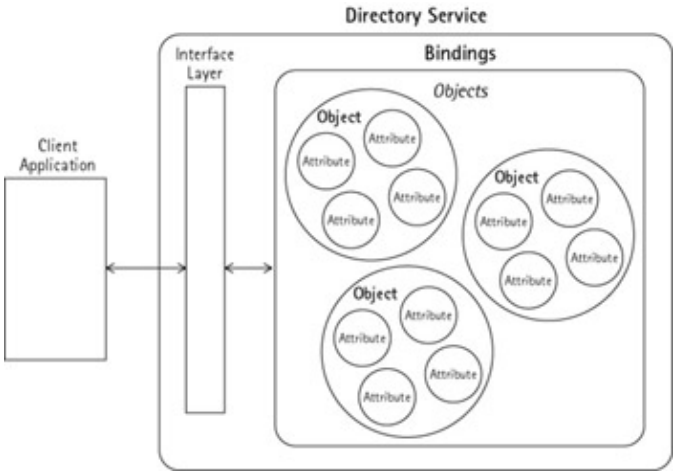

Figure 13−2: Conceptual view of a directory service

Attaching attributes to objects provides several benefits. First, it enables you to describe an object. You can specify information, such as the object's function or location. In addition, you can search for an object within a directory tree based on its attributes. For example, if you want my contact information, you can search the directory for the attribute uid with the value of toddt, and retrieve my entire record. And finally, you can use attributes to group directory objects. If you have an object repository, you can combine all the components associated with processing payroll into a group called "Payroll."

Numerous directory services exist, and you likely use one every day. Some examples are:

- Microsoft's Active Directory Services (ADS)
- Novell's Directory Services (NDS)
- Sun's Network Information Services (NIS/NIS+)

Note Some people consider LDAP a directory service. It is not. LDAP defines a protocol for communication with a directory service. Directory services that use LDAP for the communication layer are known as "LDAP−enabled."

Helpful JNDI Definitions

- *Attribute*: Descriptive information attached to directory objects.
- *Binding*: A name associated with an object in a naming or directory service.
- *Context*: An object that contains name−object bindings that may include other contexts (see *Subcontext*).
- *Directory entry*: See Directory object.
- *Directory object*: Any object or elements stored in a directory service. Directory objects can represent printers, computers, or software objects. For example, when stored in a directory service, a JDBC DataSource object is a directory object.
- *Directory service*: Enables you to attach attributes as well as names to an object. A directory service extends a naming service.
- *Name*: A user−friendly label given to an object for easy retrieval and recognition.
- *Naming service*: A mechanism for assigning "people−friendly" names to objects. For example, a file system maps file names to operating−system file handles, or an e−mail address book maps e−mail addresses to people.
- *Service provider*: A "driver" that gives you access to different naming and directory services.
- *Subcontext*: A context within a context.

Every directory service has a unique API that you use to interact with it. The interface gives you access to the service's methods for creating, modifying, deleting, and adding attributes, as well as to its methods for adding or removing directory objects.

Directory services play a major role in software architecture that use EJBs, either for storing resources natively or for specifying their location. A typical use of a directory service is to store JDBC DataSource objects in a directory service. JDBC clients retrieve the object from the service whenever needing database connectivity. You can also use a directory service to store the location of an RMI Registry. In this scenario, a client retrieves the location from the directory whenever it needs to locate and use an RMI object.

Data Access with JNDI

As I mentioned earlier, storing data in a naming or directory service helps create a central location in which you can find and retrieve resources. During the previous section you probably realized that directory services provide the real power because of its ability to attach descriptive attributes to its entries.

Naming services are useful when you need to store objects based on name only. Directory services, because they support attributes, give you much more flexibility. For this reason, the remainder of this chapter focuses on using JNDI to access directory services.

Traditionally, you use a vendor−specific API to create applications that access directory services. Generally, the API is written as a C/C++ DLL; you are unlikely to find a Java API for accessing most directory services.

Using a vendor−specific API creates several problems. First, when you use Java, a non−Java API forces you to use JNI, which makes your program platform−dependent: Your code will only execute on platforms the vendor supports. Next, you must learn the API's method calls. This may sound trivial, but to speed up development and save your sanity, the fewer programming interfaces you must learn the better. Finally, you generally must use a different API for every directory service your enterprise implements. For example, if your organization uses iPlanet's Directory Server, Novell's NDS, and Microsoft's ADS, you must use three different APIs to access information in the directory service.

JNDI helps because it defines a standard set of methods and properties for interacting in a consistent way with either a naming or directory service. JNDI also provides several other benefits. For example:

- It makes adding support for an additional directory service easy: All you have to do is add the appropriate service provider.
- It abstracts the programmatic interface of the naming and directory services from the client.
- It enables you to store and retrieve Java objects, such as JDBC DataSource objects, within naming and directory services.

I should mention that JNDI does not completely abstract the implementation of a particular directory service from you; you must still know the naming convention used in the target service. For example, DNS uses a dot (.) to separate components, while LDAP−style directory services use a comma (,). You will not find a standard like SQL that defines a common grammar for accessing the directory services.

The bright side is that JNDI provides a single API, which you must learn to use in order to access directory services. Although you still must know the naming convention, you do not need to understand each vendor's

specific API.

JNDI architecture

JNDI's architecture resembles JDBC's. This should not surprise you, because they both abstract the communication protocol and interface details of a data store. Like JDBC, JNDI consists of two components: the client API and a driver. However, the JDBC architecture does not separate the two as cleanly.

Obtaining the JNDI API

If you are using JDK version 1.3 or greater, you have everything you need for JNDI development. However, if you are using JDK version 1.1.*x* or 1.2.*x*, you need to download the JNDI API from Sun at java.sun.com/products/jndi. The distribution contains the API and the LDAP, COS, RMI, and NIS service providers.

Sun also provides, as separate downloads, a File System, Domain Name System (DNS), and Directory Service Markup Language (DMSL) service provider. The JNDI Web site provides more information on obtaining the software.

JNDI clients use the API to access naming and directory services. The API calls are standard, regardless of the data store you are trying to access. Database clients use the JDBC API in a similar fashion.

The other JNDI component is the Service Provider Interface (SPI), which is equivalent to a driver. Service providers are third−party components that you can plug into the JNDI framework when you need to support a specific data source. The service−provider vendor embeds the communication logic within the "driver" to access specific directory services such a LDAP or ADS. The SPI is where the JNDI architecture differs from JDBC. JDBC does not provide a separate package definition for driver writers to use; it keeps the driver interfaces and client API in the same package.

Figure 13−3 illustrates JNDI's architecture. As you can see, the components are layered. JNDI handles the communication between the client and the service provider.

The following two sections provide more details on both the JNDI API and the SPI.

JNDI API

The JNDI API consists of five core packages, listed in Table 13−1. As a developer you will primarily use the javax.naming and javax.naming.directory packages. The following list describes the packages:

- javax.naming: This package provides the methods that enable you to access naming services and defines the Context interface, the primary interface for interacting with a naming service. It also defines the methods for you to look up, retrieve, bind, and unbind objects.
- javax.naming.directory: Just as a directory service extends a naming service, this package extends the javax.naming package. As a result, you can still look up and retrieve objects from the directory service, just as you would with a naming service. This package works specifically with directory services to provide access to create, delete, modify, and search attributes.
- javax.naming.ldap: This package works with features associated with LDAP v3 directory services.

(For more information on LDAP v3, try the Internet Engineering Task Force's (ITEF) Web site at http://www.itef.org/.) Specifically, this package provides support for extended operations and controls provided by RFC 2251. Unless you specifically need LDAP v3 functionality, you will use the javax.naming.directory package for most of your work.

- javax.naming.event: This package monitors a directory service for certain activities, such as the addition of an object or the rebinding of a new name to an existing object. Modeled after the JavaBean architecture, it uses event listeners. The following event listeners are available for your use:
    - NamespaceChangeListener: Handles events generated when a namespace changes.
    - ObjectChangeListener: Handles events generated within a naming or directory service when an object changes.
- javax.naming.spi: Used by service provider vendors to build support for additional directory services.

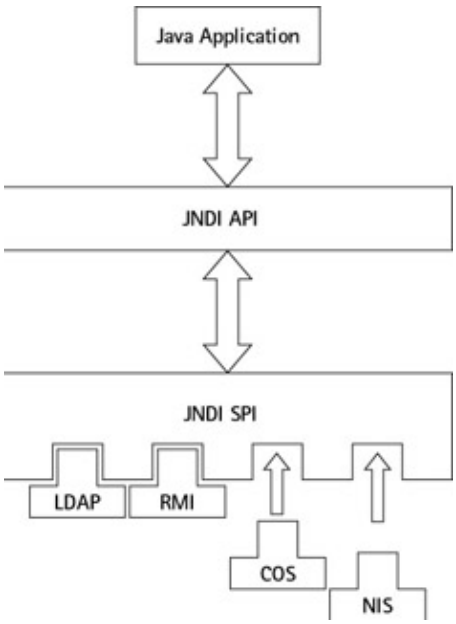

Figure 13−3: JNDI architecture

Table 13−1: JNDI API Packages

Package	Description
javax.naming	Defines classes and interfaces that enable you to interact with naming services.
javax.naming.directory	Defines classes and interfaces that enable you to interact with directory services.
javax.naming.event	Defines classes and interfaces that handle event notifications when you're working with naming and directory services.
javax.naming.ldap	Defines classes and interfaces for working with LDAP v3.0.
javax.naming.spi	Defines the service−provider interface that vendors of naming and directory service providers must implement.

JNDI Service Provider Interface (SPI)

The SPI provides a plug−in architecture with which third parties can use to develop support for additional naming and directory services and integrate that support seamlessly into the JNDI framework. The SPI defines the methods they use to hook into the JNDI API. As a developer, all you need to do in order to use the service provider is load the library during runtime or at startup.

As I mentioned earlier, a service provider is analogous to a JDBC driver in that it encapsulates the "know−how" for communicating with a specific directory service. However, unlike a JDBC driver, the client API and service−provider implementations are separate and are usually supplied as a separate jar file.

Sun provides several service−provider implementations with the JNDI API, and also makes additional ones available as separate downloads. Table 13−2 lists them for you.

Table 13−2: Available JNDI Service Providers

Service Provider	Description
LDAP	Enables you to access directory services that use LDAP as their directory protocol.
COS	Provides access to CORBA's COS (Common Object Services).
NIS	Enables clients to access the Network Information Services (NIS) used in Sun's Solaris operating systems.
RMI Registry	Enables a client to use JNDI to search for objects and retrieve them from an RMI Registry.
File System[1]	Provides standardized access to different file systems regardless of the platform. The File System service provider treats a file system as a naming and directory service.
DNS[1]	Provides access to a Domain Name Service using JNDI.
DSML[1]	Provides support for Directory Service Markup Language (DSML). With this service provider you can read and manipulate DSML documents as well as export the directory data in the DSML format. (See http://www.dsml.org/ for more information on the DSML specification.)
Novell NDS[1]	Provides access to Novell Directory Services (NDS). (Check out http://www.novell.com/ for more information.)

[1]Available as a separate download from Sun.

JNDI programming

Getting started with JNDI programming is relatively straightforward. Not only does the JNDI architecture resemble JDBC, but the programming concepts are very similar as well. For instance, with either API you must load a driver and open a connection to the data source before you can access a data store.

Working with the Context and DirContext interfaces

The Context and DirContext interfaces play a central role in JNDI programming. The first defines methods for interacting with naming services. It defines the bind(), rebind(), and unbind() methods that you use to bind and

unbind names to objects. It also has a lookup() method that enables you to retrieve an object based on its name. Remember that naming services only enable you to store objects using only a name attribute.

The DirContext interface provides the functionality for using directory services. The interface extends the Context interface, enabling you to manipulate objects as you do in a naming service. Most of the extra functionality associated with the DirContext interface pertains to managing and searching object attributes.

For example, the bind() and rebind() methods are overloaded so you can associate attributes with the objects as you insert them into a directory service. The interface also defines new methods, such as getAttributes() and setAttributes(), that enable you to retrieve and assign attributes independently regardless of when you bind objects. The search() method enables you to look up objects based on their attributes.

Most of what I discuss during the remainder of this chapter applies to both Context and DirContext objects. Because I am focusing only on directory services, I will refer only to DirContext objects, and specifically mention when anything different applies to the Context object.

Connecting to JNDI data sources

Just as with JDBC, with JNDI you must connect to a data source before you can work with the data. Instantiating an InitialDirContext object opens a connection to a directory service and defines the starting point within the directory structure, called the *initial context*.

All operations in the directory service occur relative to the initial context. For example, you may wish to start at the root context if you need to search for an object whose location within the directory you do not know. Or you may start in a specific subcontext that contains a certain set of objects, such as employee−related EJBs.

Figure 13−4 shows a flow chart of the steps required to instantiate an InitialDirContext object. Before calling its constructor you must prepare its "environment," which consists of adding entries into a Hashtable, and then supplying it as a constructor parameter. The environment consists of entries such as the service−provider information, directory−service location, and security information.

At a minimum, you must set two environment properties, the service provider and the directory−service location. To set the service provider, you assign a String value representing the fully qualified name for the driver to the property Context.INITIAL_CONTEXT_FACTORY. (Doing this is analogous to using the Class.forName() method to initialize a JDBC driver.) The following code snippet demonstrates how to set the service−provider property to connect to an LDAP−enabled data store:

```
//Create a Hashtable object to place environment settings
Hashtable env = new Hashtable();

//Specify service provider class
String sp = "com.sun.jndi.ldap.LdapCtxFactory";
env.put(Context.INITIAL_CONTEXT_FACTORY, sp);
```

The other property, Context.PROVIDER_URL, identifies the location of the data store. The format of the JNDI URL varies according to the service provider. The following snippet demonstrates how to set the property for the LDAP−enabled directory service using the Hashtable declared in the preceding code sample:

```
//Specify location of LDAP directory service
String jndiUrl = "ldap://localhost:389/o=MyLdapData";
env.put(Context.PROVIDER_URL,jndiUrl);
```

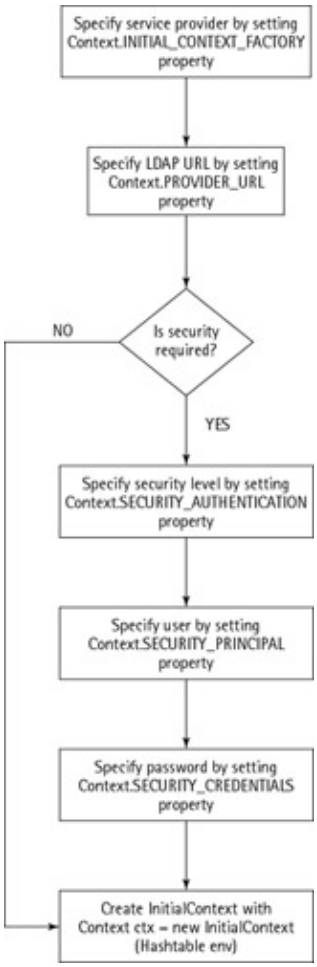

Figure 13−4: Basic JNDI connection steps

Note LDAP−enabled directory services listen on a default port of 389.

You may also need to set security properties, such as username and password, depending upon the security level implemented by the directory service. LDAP, for example, requires three parameters. The first identifies the type of security to use — simple, SSL/TLS, or SASL. (I discuss LDAP in greater detail in the next section.) The next two parameters are username and password. The following code snippet demonstrates how to set the three parameters:

```
//Specify an user and password
String user = "uid=toddt, ou=People, o=toddt.com";
String pwd = "mypwd";

//Set the authentication type
env.put(Context.SECURITY_AUTHENTICATION, "simple");

//Set username and password
env.put(Context.SECURITY_PRINCIPAL, user);
env.put(Context.SECURITY_CREDENTIALS, pwd);
```

Once you have configured the environment, you call the constructor to instantiate an InitialDirContext object. For example:

```
DirContext dctx = new InitialDirContext(env);
```

Because the InitialDirContext object represents a physical connection to the data source, you should close it when you finish working with it — just as you would when working with JDBC. To close the DirContext object, just call the close() method:

```
dctx.close();
```

Now you know the step involved in opening a JNDI connection to a data store. What you actually do once you have instantiated an InitialDirContext object depends upon your naming or directory service. JNDI lacks an SQL−type language that allows a standard grammar to interact with any naming or directory service. As a result, you must tailor your program according to the directory service you need to work with.

JNDI NamingExceptions

Before moving on to the examples, I should mention JNDI exceptions. Just as most JDBC methods throw an SQLException, most JNDI methods throw a NamingException. For this reason you should place a try−catch block dedicated to that exception in your code, to handle any errors that might occur.

Working with the JNDI LDAP SPI

LDAP−enabled directory services are becoming increasingly popular storage vehicles for corporate information. At some point in your Java development career you may find yourself working with them. Enterprises typically use these directory services for:

- User authentication and access control.
- Storage of objects, or object locations, in a distributed computing architecture.
- Corporate address books for e−mail addresses, phone numbers, and other information about employees.
- Personalization data for Web−site users.

LDAP is a book−length subject unto itself. However, in this section I will show you how to work with it using JNDI. For more in−depth coverage, such as information about creating schemas and implementing authentication mechanisms, please consult additional references. The World Wide Web Consortium’s (W3C) Web site (http://www.w3c.org/), is a great place to start your research.

I use iPlanet’s Directory Server 5.0 for the examples in this section. (This product was formerly known as Netscape Directory Server.) You may obtain a 90−day evaluation copy from http://www.iplanet.com/downloads. iPlanet has versions for the Microsoft NT/2000, HP−UX, Solaris, and AIX platforms.

LDAP overview

A lot of misconception surrounds LDAP and what it actually does. Before continuing I will attempt to explain its purpose. To begin, here’s what LDAP is not:

- LDAP is not a directory service.
- LDAP is not a specification for a directory service.
- LDAP is not a schema for a directory service.

LDAP defines a protocol for communicating with a directory service. It provides a “lighter−weight” protocol than the Directory Access Protocol DAP used to access X.500 data stores. Although its creators originally designed LDAP as an alternative for DAP to access X.500 directories, you can use the protocol to access other

directory services as long as they implement the LDAP interface.

Figure 13−5 illustrates the typical architecture of an LDAP application. The client uses the LDAP protocol to communicate with an LDAP server, which may or may not hold the directory service. Figure 13−5 shows the directory service located on the LDAP server. Directory services that are not X.500, which is most of them, and that use LDAP are said to be *LDAP−enabled*. iPlanet's Directory Server and Novell's NDS are examples of LDAP−enabled directory services.

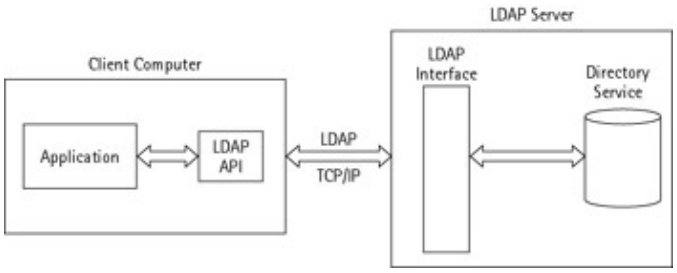

Figure 13−5: LDAP application architecture

LDAP is well tested. RFC 1777, approved in 1995, defines LDAP v2 and provides most of the functionality you will use to interact with a directory service. This version, however, is not ideal for use in an enterprise environment. For example, LDAP v2 only supports three types of authentication: anonymous, clear−text (simple), and Kerberos v4. Given today's emphasis on security, enterprises need more robust authentication mechanisms.

LDAP v3, defined by RFC 2251, fills in the gaps of LDAP v2. In general, it improves security and extensibility. With regard to authentication, it uses the Simple Authentication and Security Layer (SASL) framework defined in RFC 2222. This architectural change enables you to use other authentication mechanisms. In addition, LDAP v3 adds extensions and controls, both of which enable you to add new functionality to a directory service. The javax.naming.ldap package defines the methods you need in order to use the LDAP v3 features.

LDAP naming model

An LDAP−enabled directory service generally uses a hierarchical directory structure. Figure 13−6 shows a representation of a typical LDAP−enabled directory service, which uses a family−tree−like structure of entries, also called *directory objects*, called a Directory Information Tree (DIT). As with a family tree, the first entry is called the *root*.

You identify an entry by its Distinguished Name (DN), which shows its position in the DIT. You read a DN from right to left. For example, my DN is uid=toddt, ou=people, o=toddt.com, as shown in Figure 13−6.

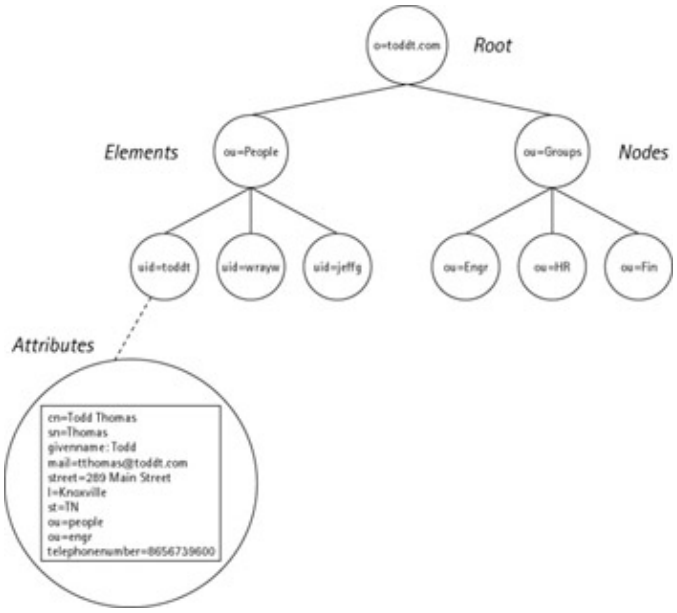

Figure 13−6: Example LDAP directory information tree

SBT Directory Services versus Relational Databases

A directory service provides some of the same functionality of a relational database. As a result, you may wonder why you wouldn't use a database instead. As with most competing technologies, each has its strengths and weaknesses.

The following list identifies situations in which you should consider using a directory service over a relational database:

- Where there is a high read/write ratio. Directory services are optimized for searching and retrieving data, not for writing or updating data.
- Where data remain static, as in a corporate phone book or an object repository.
- Where you do not need transactional control. Directory services do not support transactions, so you should not use them for applications that require that functionality.

Storing Java Objects with LDAP

Not only can you store text and numerical information in an LDAP−enabled directory service, but you can also store and manipulate Java objects. Doing so enables you to build an object repository without having the client worry about other interfaces, such as RMI or CORBA.

JNDI provides three alternatives for storing data in a directory service:

- *Serialized objects*: If an object implements the java.io.Serializable interface, you can store it directly in the directory service. Using this method enables you to save object state or configuration. The disadvantage of storing Java objects natively comes into play when the objects become too large, as storing them natively can negatively affect the performance of the directory service.
- *Referenceable objects*: When you cannot store an object directly, you can store a reference to it. You may want to consider this approach when working with large objects. To store an object reference, the

object must implement the javax.naming.Referenceable interface. The requires that you implement the getReference() method, which returns a Reference object. The service provider uses this method behind the scenes when storing the object.

- *Objects with attributes*: You can store objects in a directory service so that other (non−Java) applications can use them. Although this approach provides the most flexibility, it is also the most difficult to implement. Objects eligible for storage with attributes must implement the DirContext interface. Not all service providers support this method of object storage.

However, a DIT differs from a true tree structure because it can employ *aliases*. An alias is analogous to a pointer: You can use it in situations where you would use a symbolic link in UNIX. For example, if you move a DIT entry you can create an alias that points to its new location. The client will access the alias without knowing the entry was moved.

Working with LDAP data

You use JNDI's javax.naming.directory package and its DirContext interface to work with an LDAP−enabled directory service. The DirContext interface extends the Context interface and defines additional methods that apply to directory services.

As you might expect, the LDAP specification is extensive and defines several core operations that you can perform on objects in a directory service. The following list describes the four actions you will use most often:

- **Add:** Creates a new entry with an optional set of attributes.
- **Delete:** Removes an entry from the directory.
- **Modify:** Changes the attribute value(s) associated with an entry.
- **Search:** Looks up entries based on their attributes.

Most of your work with LDAP−enabled directory services will involve looking up entries. A systems administrator usually has the responsibility of adding and maintaining the entries. Because directory services are built and optimized for searching and finding entries, you probably should minimize the number of objects you add programmatically anyhow.

LDAP−enabled directory services can store various types of data. Besides text and numerical information, you can also store Java objects. This enables you to create an object repository and retrieve them directly from the directory service for use in your application. You may store the object natively or use a *reference*, which is analogous to a pointer, to the object if the object is too large to store in a directory service.

In the rest of this section I cover how to interact with an LDAP−enabled directory service. I begin by covering attributes, and then explain how to search for elements based on them.

Directory−object attributes As I mentioned previously, directory elements can have descriptive attributes. This enables you to search the DIT for objects whose attributes meet certain criteria. For example, a directory object for an employee can have attributes that specify an employee's e−mail address, phone number, and Social Security number. To find an employee, you can perform a search on any attribute defined in the LDAP−enabled directory service.

Most directory services enable you to define your own attributes. However, there exist standard attributes for describing objects, and these standard attributes are used in most directory services. Table 13−3 lists them and provides a short description of each.

Table 13−3: Common Attributes Used in Directory Services

Symbol	Name	Definition	Example
dn	Distinguished Name	Unique name of an entry in a DIT.	Uid=tthomas, ou=people, o=toddt.com
uid	userid	Unique ID for a user.	Uid=tthomas
cn	common name	First and last name of a user.	cn=Todd Thomas
givenname	first name	First name of a user.	givenname=Todd
sn	surname	Last name of a user.	sn=Thomas
l	location	City user lives.	l=Knoxville
o	organization	Typically the directory root.	o=toddt.com
ou	organizational unit	Major directory branches.	ou=People
st	state	State user lives.	TN
c	country	Country user lives.	US
mail	e−mail address	SMTP e−mail address.	tthomas@toddt.com

When working with directory−object attributes you have two interfaces at your disposal: Attribute and Attributes. The first represents a single attribute of an element and the second all the attributes of an element.

In general, you use methods that return Attributes objects containing all the attributes of an object. You then use additional methods to retrieve an Attribute object that holds the value of an object's attribute. The following code snippet demonstrates how to retrieve the mail attribute of a directory object:

```
//Assume ctx is a valid InitialDirContext object
//Retrieve attributes associated with the named directory object.
Attributes attrs =ctx.getAttributes("uid=awhite, ou=People");

//Retrieve a single attribute.
Attribute attr = (Attribute)attrs.get("mail");
```

The BasicAttribute and BasicAttributes classes implement the Attribute and Attributes interfaces. You use objects of this type when you work with methods — typically methods associated with the DirContext interface — that require them as parameters. For example, the modifyAttributes() and search() methods can accept a parameter of type Attributes. The following section provides more details on working with attributes.

Searching a directory service Searching is one of the most useful and powerful features of a directory service. In fact, you will likely do more searching than updating or adding of new objects. Because LDAP−enabled directories are built for searching, you have a lot of control over how you search. For example, you can search the entire DIT, a specific named context, or a named object.

To conduct an LDAP search, you use the IntialDirContext.search() method. The JNDI API has eight overloaded versions of the method that enable you to customize your search. For instance, you can define the following:

- The starting point of the search.
- A search filter (to narrow the results).
- The scope of the search (to limit the contexts evaluated).

You must always specify the starting point of the search. You can specify any context or named object you wish. The search filter helps you focus the query to return only objects whose attributes match certain criteria. You may find this helpful if you have a large employee directory and you need to limit the number of records returned. The last component enables you to define the area of the DIT you want to search. Table 13−4 lists the three options you have when setting the search scope.

Table 13−4: JNDI Search Scopes

Scope	Description
OBJECT_SCOPE	Searches a specific named object; you can use it for simple equality tests.
ONELEVEL_SCOPE	Searches only one level below the specified named context.
SUBTREE_SCOPE	Searches the sub−tree below the specified named context.

Don’t confuse the scope of a search with the starting context. In fact, the two components work together. The starting context influences the scope. Table 13−5 provides several examples of search scopes based on the directory hierarchy in Figure 13−6.

Table 13−5: Example Search Scopes

Starting Context	Search Scope	Result
uid=awhite	OBJECT_SCOPE	Tests equality, or that an object has certain attributes specified by a search filter.

dc=siroe, dc=com	ONELEVEL_SCOPE	Searches the next level down in a tree, in this case ou=People and ou=Groups.
dc=siroe, dc=com	SUBTREE_SCOPE	Searches the entire DIT, including the ou and uid levels.

You can narrow a search using either attribute constraints or a search filter. Searching with attribute constraints is the simplest way to locate an object. With this method you specify the attributes you want an object to have. The results will contain every object whose attributes match your search criteria. The following code demonstrates a search that returns an object whose uid attribute equals awhite:

```
//Create Attributes object
Attributes attrs = new BasicAttributes(true);

//Put search criteria in Attributes collection
attrs.put(new BasicAttribute("uid=awhite, ou=People"));

// Search for objects that match the attributes
NamingEnumeration answer = ctx.search("ou=People", attrs);
```

To use a search filter you need to use the class javax.naming.directory. SearchControls and a String object representing the filter. The SearchControls class enables you to specify the scope, or what contexts to search. The filter enables you to search for elements using logical expressions and wildcard characters. (RFC 2241 defines the String representations of the LDAP search symbols.) The following code snippet illustrates how to perform a search using a filter and the SearchControls class:

```
//Define a starting context to search from.
String base = "ou=People";

//Create a SearchControls object and define a search scope
SearchControls sc = new SearchControls();
sc.setSearchScope(SearchControls.SUBTREE_SCOPE);


//Create a filter. Here I look for anyone with the last name=White
//who works in Sunnyvale. I also ignore the first name.
String filter = "(&(givenname=*)(sn=White)(l=Sunn*))";

// Search subtree for objects using filter
NamingEnumeration ne = ctx.search(base, filter, sc);
```

Table 13−6 lists the most common search symbols and their meanings.

Table 13−6: Common Search Symbols from RFC−2254

Search Symbol	Description
!	Logical not.
|	Logical or.
&	Logical and.
*	Wildcard (any value).

=	Equality.
>=	Greater than.
<=	Less than.

You may notice that the preceding code snippet returns a NamingEnumeration object from the search() method. This object contains results from JNDI methods, which return multiple values. In this case the search may return any number of records. The NamingEnumeration object lets you traverse the results and retrieve the elements. You will see this object in action in the next section.

Searching an LDAP−enabled directory example

As I mentioned earlier, you can do a lot different things with LDAP. However, the most common task is searching for and retrieving objects. Therefore, the most practical data−access example I can provide is an example that shows you how to search and retrieve objects using LDAP.

Listing 13−1 demonstrates how to search for objects in an LDAP−enabled directory service whose attributes meet certain criteria. In the example I want to find all the employees who work in Cupertino and have last names starting with the letter *w*. This is an example of the kind of application you might need to use when accessing data in a corporate directory.

Listing 13−1: LdapSearch.java

```
package Chapter13;

import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;
import java.io.Serializable;

public class LdapSearch {

   public static void main(String[] args) {

      //Create Hashtable and load environment variables
      Hashtable env = new Hashtable();

      String sp="com.sun.jndi.ldap.LdapCtxFactory";
      env.put(Context.INITIAL_CONTEXT_FACTORY, sp);

      String ldapUrl="ldap://localhost:389/dc=siroe, dc=com";
      env.put(Context.PROVIDER_URL,ldapUrl);

      try{
         // Create initial context
         DirContext dctx = new InitialDirContext(env);

         //Set search base
         String base = "ou=People";

         //Set attribute filter and search scope
         SearchControls sc = new SearchControls();
         String[] attributeFilter = {"cn", "mail"};
         sc.setReturningAttributes(attributeFilter);
         sc.setSearchScope(SearchControls.SUBTREE_SCOPE);

         //Define filter
         String filter = "(&(sn=W*)(l=Cup*))";
```

```
         //Perform search
         NamingEnumeration results = dctx.search(base, filter, sc);
         System.out.println("Employees in Cupertino:");

        //Print results
         while (results.hasMore()) {
            SearchResult sr = (SearchResult)results.next();
            Attributes attrs = sr.getAttributes();

            Attribute attr = attrs.get("cn");
            System.out.print(attr.get() + ": ");
            attr = attrs.get("mail");
            System.out.println(attr.get());
         }

         //Close resources an say goodbye
         dctx.close();
         System.out.println("Goodbye!");

      }catch(NamingException ne){
         ne.printStackTrace();
      }catch(Exception e){
        e.printStackTrace();
      }
   }
}
```

The output from Listing 13−1 is as follows:

```
Employees in Cupertino:
John Walker: jwalker@siroe.com
Cecil Wallace: cwallace@siroe.com
Morgan White: mwhite@siroe.com
Alan Worrell: aworrell@siroe.com
Andy Walker: awalker@siroe.com
Eric Walker: ewalker@siroe.com
Goodbye!
```

To begin the application, I create a Hashtable in which to store the environment settings I need in order to instantiate an InitialDirContext object. To do so, I specify the service provider I need to use. In this example I am using Sun's LDAP service provider. The String entry name of the service provider's driver, com.sun.jndi.ldap.LdapCtxFactory is the class name.

Next, I put the location of the LDAP server into env. In this example, I am connecting the root node (dc=siroe, dc=com) on a local LDAP server listening on port 389, the default port for LDAP servers. Now that I have the environment setting prepared I can instantiate a DirContext object with a call to the constructor InitialDirContext and use the Hashtable, env, as a parameter.

The next major step is to set the search criteria and controls. To do this I first define a String variable, base, that specifies the context in which to begin the search. Because I'm searching for people, I specify the context ou=People. Next I instantiate a SearchControls object and make the following settings:

- Return only the values for the cn and email attributes with each object that matches the search criteria.
- Perform the search on the entire sub−tree of the context defined in the variable base.

Now I am ready to define my search filter. As I mentioned earlier, I want to find all the employees who work in Cupertino and have last names starting with the letter *w*. The String variable filter defines this filter.

To execute the search I call the dctx.search() method and supply the search base, filter, and scope as parameters. The method returns a NamingEnumeration object, which contains all the objects that match the search criteria. After retrieving the results I print them out using a SearchResult object and a simple while−loop.

Although this is a straightforward example, it contains all the components you need in order to perform a search of an LDAP−enabled directory service.

Summary

As enterprises gather data and resources into a central location they often use a naming or directory services. JNDI provides a uniform API that enables you to access these data stores.

You can use JNDI to access corporate directories as well as object repositories. In fact, JNDI plays a major role in J2EE technologies. It enables clients to retrieve objects from repositories or look up their locations in other enterprise data stores

Besides presenting the JNDI architecture, the chapter also showed you how to use JNDI with LDAP. Most corporations and directory vendors use LDAP, although JNDI supports other naming and directory services. To that end, Sun provides service providers for RMI, File System, and NIS, to name a few.

The biggest benefit of JNDI is that it provides a single API that can access different data stores. You only need to learn one API, not one for each naming or directory service.

Chapter 14: Using Data Sources and Connection Pooling

In This Chapter

- Defining a Java DataSource object
- Using Java DataSource objects locally and with JNDI
- Understanding connection pooling
- Using the PooledConnection and ConnectionPoolDataSource interfaces

Enterprise database development provides you with many challenges. Not only must you create scalable and robust applications, but you must also make them easy to deploy and maintain. In addition, you need to ensure that your applications are sensitive to client, server, and network resources.

For example, most enterprise applications have many users, which may reside in different locations. As a result, deployment strategies should not only consider the initial client installation, but how to maintain the code base once it is installed. For example, if you add or change a database location you want to avoid having to re−deploy your application. You can do this by making the client's code base independent of any database−specific information such as server location or database driver names.

In addition, as a database developer, you want to ensure that your applications respect server resources such as CPU and memory. Minimizing the number of connections the clients open and close helps. You especially need to consider the impact of connection cycling on the application and database servers when you use entity EJBs in J2EE programming.

To help you address this challenge, Java 1.4 provides an improved javax.sql interface. It defines interfaces for connection pooling and abstracting database−specific information from the client. Specifically, the DataSource and ConnectionPoolDataSource interfaces solve many of the problems associated with enterprise development.

In this chapter, I cover how to work with JDBC 3.0 DataSource objects. I begin with an overview and then demonstrate how to use the objects in a distributed environment using JNDI. Finally, I demonstrate how to use ConnectionPoolDataSource objects to implement connection pooling. At the end of this chapter you should have a good understanding of how to take advantage of both interfaces.

Working with Java DataSource Objects

One theme of object−oriented and Java programming is abstraction. You should always try to hide implementations behind interfaces. This helps create reusable and easily maintainable code. In addition, abstraction promotes code independence. By relying on interfaces instead of on concrete classes, you reduce an object's dependency on specific implementations. JDBC DataSource objects continue this theme by abstracting the database server's location and connection details from a client.

XRef See Chapter 9, "Understanding Design Patterns," and Chapter 10, "Building the Singleton Pattern," for more information on design patterns that help you architect applications that take advantage of

abstraction and polymorphism.

You can use a DataSource object either locally or with JNDI. When you use it locally you do not need to register and load database−driver information. When you use it with JNDI you get all the benefits of local use, and in addition you can abstract the database location and connection information from a client. If you do this, the client won't have to supply usernames, passwords, or a JDBC URL to open a database connection.

Note The DataSource interface and ConnectionPoolDataSource interface are often used interchangeably. Some vendors may implement connection pooling in their DataSource implementations. However, most provide this functionality with the ConnectionPoolDataSource interface.

Using DataSource objects

JDBC DataSource objects offer an alternative to DriverManager for opening database connections_—_in some ways a superior alternative. The main advantage of using a DataSource object is that you avoid having to register the JDBC driver. DataSource objects handle this detail so you never need to hard−code the driver name or set the value in a property file.

However, to take full advantage of a DataSource object you should use it with JNDI. Using a JNDI naming service provides the following benefits:

- You do not need to specify a JDBC URL, username, or password to make a connection. The system administrator configures these parameters when binding a DataSource object into a naming or directory service.
- You avoid having to reference the JDBC driver name, which helps mitigate your dependence on vendor−specific code.
- The client does not need to know the database server's location. If the database changes physical hosts, the change is made to the DataSource object and is transparent to the client.

Figure 14−1 shows a typical configuration using JNDI and DataSource objects. The client uses JNDI to retrieve a DataSource object from a directory service that is pre−configured with the connection information. To open a database connection, the client just calls the DataSource.getConnection(). Once a Connection object is instantiated, the client can communicate with the database as normal.

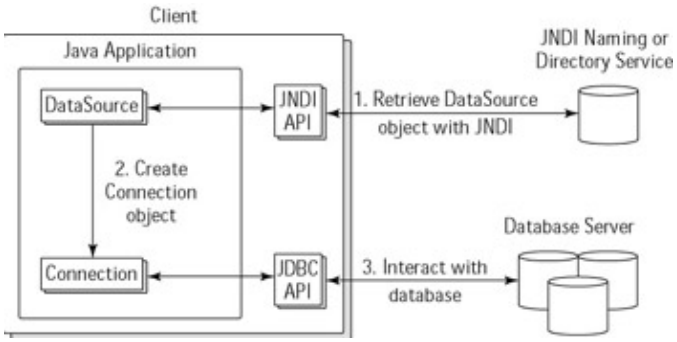

Figure 14−1: DataSource and JNDI configuration

After reading about the advantages the DataSource object provides, you may wonder why you wouldn't use it exclusively. The primary reason is vendor implementations.

Because the DataSource interface is part of the javax.sql package, driver vendors must implement the functionality. Unless you have a driver that provides an implementation, you cannot take advantage of the

DataSource object's functionality. The following section provides further details on typical vendor implementations.

Looking at DataSource implementations

The javax.sql package that Sun distributes consists mainly of interfaces. As a result, the driver vendor must implement the methods defined in the API's interfaces.

Note Prior to JDK1.4 the DataSource interface was part of the JDBC 2.0 Optional package. Sun has included it with the standard distribution. If you are using a prior JDK, go to www.javasoft.com/products/jdbc to obtain the optional package.

Figure 14−2 shows the UML class diagram for the DataSource interface. As you can see, the interface defines the getConnection() method. As I mentioned earlier, the method returns a standard physical connection, represented as a Connection object, to the database, just as DriverManager does.

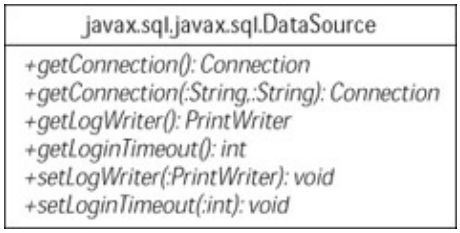

Figure 14−2: UML class diagram of the DataSource interface

A second inspection of the UML class diagram shows that the interface lacks methods for specifying connection parameters. For example, how do you set the username and password, or JDBC URL? The answer: Vendors must provide these setter and getter methods.

The interface does not define these methods because different databases may require different connection parameters. For example, some drivers may have a parameter that specifies a certain network protocol, while others may not. However, for the sake of consistency, Sun has developed standard property names. They are listed in Table 14−1.

Table 14−1 : Recommended DataSource Property Names

Property Name	Java Data Type	Comment
databaseName	String	The name of the database you want to connect to.
serverName	String	The name of the database server you want to connect to.
user	String	The user ID with which you want to connect to the database.
password	String	The password for the user ID specified in the user property.
portNumber	Int	

	The number of the port to which the database server is listening.

When using a DataSource object locally you must use the vendor's methods to set the necessary connection information. This approach ties your code to the specific vendor's class name that implements the DataSource interface. The constraint only applies when you are using the DataSource interface locally.

For example, with Oracle's implementation the OracleDataSource class implements the DataSource interface. To access the setter and getter methods you must declare a variable of type OracleDataSource. However, having this class name in your code makes your code less portable.

If you use a DataSource object retrieved from a JNDI naming service, the connection properties are usually preset. The JNDI system administrator, or whoever deploys the DataSource object, sets these parameters. This is one advantage of using JNDI and DataSource objects together: You do not need to worry about the connection details.

A DataSource example

Now I want to provide an example of using a local DataSource object to open an Oracle database connection. Listing 14−1 provides the code for the example. Because I'm using the object locally, I must set the connection properties of the DataSource object. As a result, I need to declare a variable, ods, of type OracleDataSource, so I can access the setter methods as part of Oracle's implementation. Every vendor will have different methods. However, notice that I never reference Oracle's JDBC driver class name in the example. The OracleDataSource object knows how to communicate with it.

Listing 14−1: DataSource.java

```
package Chapter14;

import javax.sql.*;
import java.sql.*;
import oracle.jdbc.driver.*;
import oracle.jdbc.pool.*;

public class DataSource {

  public static void main(String[] args){
    try{
      //Instantiate a DataSource object
      //and set connection properties.
      OracleDataSource ods = new OracleDataSource();

      ods.setUser("toddt");
      ods.setPassword("mypwd");
      ods.setDriverType("thin");
      ods.setDatabaseName("ORCL");
      ods.setServerName("localhost");
      ods.setPortNumber(1521);

      //Open connection
      Connection conn = ods.getConnection();
      System.out.println("Connection successful!");

    }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
```

```
    }//end try
    System.out.println("Goodbye!");
  }
}
```

The output from Listing 14−1 is as follows:

```
Connection successful!
Goodbye!
```

Using DataSource objects with JNDI

The Java JNDI API provides access to naming and directory services so that you may locate and retrieve a variety of resources. For example, you can use JNDI to retrieve an employee's phone number and e−mail address from an LDAP−enabled directory service. Or you can retrieve a DataSource object from a directory service and use it to interact with a database.

Combined, the DataSource interface and JNDI play a key role in the database−component layer of a J2EE program. With the combination you can remove the need for vendor−specific code in the client. In addition, you can place a DataSource object, pre−configured with the correct information for connecting to a database, into a directory service. When a client retrieves the object, all it needs to do is call DataSource.getConnection() to open a database connection.

XRef Chapter 13, "Accessing Enterprise Data with JNDI," provides more information on how to use JNDI.

Using DataSource objects and JNDI together requires two steps:

1. You must load the DataSource object into a directory service and bind a logical name to it. This requires that you use the Context.bind() method found in the javax.naming package.
2. The client has to retrieve the DataSource object from the JNDI naming system using the Context.lookup() method. After the client retrieves the object, it uses the DataSource.getConnection() method to open a database connection.

Listing 14−2 provides an example of using JNDI and the OracleDataSource object provided with the Oracle 8.1.7 JDBC driver. Remember that the DataSource interface does not define any methods for setting connection information; the vendor must provide this implementation. In this case, the OracleDataSource object implements the JDBC DataSource interface and has methods for setting the connection properties.

Listing 14−2: JndiDataSource.java

```
package Chapter14;

import java.sql.*;
import javax.sql.DataSource;
import oracle.jdbc.pool.OracleDataSource;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class JndiDataSource{
```

```
static Connection conn = null;
static Statement stmt = null;
static ResultSet rs = null;
static Context ctx = null;
static DataSource ds = null;

public static void main (String args []){

  // Initialize the Context
  String sp = "com.sun.jndi.fscontext.RefFSContextFactory";
  String file = "file:/e:/JNDI";
  String dataSourceName = "jdbc/myDatabase";

  try {

    //Create Hashtable to hold environment properties
    //then open InitialContext
    Hashtable env = new Hashtable();
    env.put (Context.INITIAL_CONTEXT_FACTORY, sp);
    env.put (Context.PROVIDER_URL, file);
    ctx = new InitialContext(env);

    //Bind the DataSource object
    bindDataSource(ctx, dataSourceName);

    //Retrieve the DataSource object
    DataSource ds = null;
    ds = (DataSource) ctx.lookup(dataSourceName);

    //Open a connection, submit query, and print results
    Connection conn = ds.getConnection();

    Statement stmt = conn.createStatement();
    String sql = "SELECT Name FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);

    System.out.println("Listing employee's name:");
    while(rs.next())
      System.out.println(rs.getString("name"));

    // Close the connections to the data store resources
    ctx.close();
    rs.close();
    stmt.close();
    conn.close();

  }catch (NamingException ne){
    ne.printStackTrace();

  }catch (SQLException se){
    se.printStackTrace();

  //ensure all resources are closed
  }finally{
    try{
      if(ctx!=null)
        ctx.close();
    }catch (NamingException ne){
      ne.printStackTrace();
    }finally{
      try{
```

```
          if(conn!=null)
            conn.close();
        }catch (SQLException se){
          se.printStackTrace();
        }
      }
    }
    System.out.println("Goodbye!");
  }

  //Method to bind DataSource object
  public static void bindDataSource(Context ctx, String dsn)
        throws SQLException, NamingException{

    //Create an OracleDataSource instance
    OracleDataSource ods = new OracleDataSource();

    //Set the connection parameters
    ods.setUser("toddt");
    ods.setPassword("mypwd");
    ods.setDriverType("thin");
    ods.setDatabaseName("ORCL");
    ods.setServerName("localhost");
    ods.setPortNumber(1521);

    //Bind the DataSource
    ctx.rebind (dsn,ods);

  }
}
```

The output from Listing 14−2 is as follows:

```
Listing employee's name:
Todd
Larry
Lori
Jimmy
John
Andy
Goodbye!
```

In Listing 14−2 I use Sun's File System service provider because it is easy to use and you do not need access to an external directory service in order to use it. However, before running the application you need to ensure that the initial context (here e:\JNDI) exists. You can change this context to reference another directory to suit your needs. In addition, you must ensure you have the File System service provider from Sun.

Note To use the JNDI File System service provider you must download it from Sun. You can find the driver, along with those for other service providers, at http://www.javasoft.com/products/jndi.

I start the application by instantiating an InitialContext object, which opens a connection to the data store and specifies my initial context. I will use this object to load and retrieve the DataSource object. Next, I use the bindDataSource() method to bind the DataSource object into the naming service.

This example actually combines two functions into one. In a real application you will probably not have to bind your own DataSource object into a naming service. A systems administrator usually performs this task.

After binding the DataSource object, I simulate a client retrieving the object from the naming service. To do so, I define a variable, ds, of type DataSource. Polymorphism enables me to assign any class that implements the DataSource interface to a variable of that type. Notice that I must cast the object retrieved from the directory service to a DataSource type, because the lookup() method returns an Object data type.

Once I retrieve the object, I use the getConnection() method to open a database connection. To illustrate that the connection is valid, I perform a simple query and list the results in the main() method.

Listing 14−2 illustrates the real benefit of using JNDI and DataSource objects together. You completely remove the vendor−specific code from your application. You can switch databases or drivers without affecting the client's code, because they rely only on the DataSource interface.

Implementing Connection Pooling

Establishing a database connection is an expensive operation. A lot of activity occurs, and that requires network bandwidth as well as both client and server resources. Significant handshaking, such as user authentication, must occur before you actually open a connection. You can see the impact of handshaking on your application as it will run sluggishly or appear to hang while establishing the connection.

Ideally you want to open only one physical connection and use it throughout the application. Using a global Connection object works fine for simple applications when you need to make only a limited number of requests.

However, suppose you have a multithreaded application in which every thread needs its own physical connection, that is, its own Connection object? Whenever you spawn a new thread you open another database connection, thereby slowing your application and consuming resources on the server.

On the enterprise level, consider a J2EE solution that uses an entity EJB that requires database access. Because clients share this component, every request opens and closes a database connection. However, when you have a lot of traffic or usage, you run the risk of slowing down both the application and the database server.

Connection pooling helps combat this problem. This programming technique allows a client to retrieve pre−connected Connection objects from a cache. In this scenario, you open the database connection once and provide it to clients when they need connections. This enables you to share one or more physical connections for the entire session, thus reducing the overhead associated with opening connections.

If you are ambitious, you can implement connection pooling yourself. After all, a connection pool is nothing but a pool of objects. Plenty of examples exist on the Internet that show you how to create and manage object pools.

> Note Connection pooling is only available if a vendor implements it in the javax.sql package. Some vendors create distributions with only basic DataSource functionality, to enable you to use JNDI.

However, why reinvent the wheel? The ConnectionPoolDataSource interface is meant to supply Connection objects from a pool. Assuming a vendor implements the methods defined by the interface, you can use this interface to provide connection pooling.

Understanding connection−pooling concepts

When connection pooling exists in a JDBC 3.0 driver, the vendor implements the ConnectionPoolDataSource interface. Objects implementing this interface create PooledConnection objects, which represent the physical connection to the database. This object supplies the Connection object you use to interact with the database. The PooledConnection interface does not define methods for creating Statement objects or other objects you normally use to interact with a database; you must use a Connection object to create these.

A Connection object retrieved from a PooledConnection object pool represents a "logical" connection to a database. The vendor hides the physical connection from the client using a PooledConnection. In general you have little control over how many physical connections exist or over the ratio of logical to physical connections.

Logical connections behave nearly the same as physical connections instantiated with DriverManager. For example, they can create Statement, PreparedStatement, or CallableStatement objects and control transaction levels.

However, a logical connection’s close() method operates differently. Calling the close() method on a standard Connection object closes the physical connection. In contrast, calling the close() method on a logical Connection object returns the logical connection to the pool for other clients to use.

> Tip Always call the close() method of a pooled Connection object so it can return to the connection pool, where other clients can use it.

A connection−pooling example

In this section I demonstrate how to take advantage of connection pooling. The example uses a local ConnectionPoolDataSource object, which requires setting the connection parameters. If you use a pre−configured ConnectionPoolDataSource object from a JNDI repository you can skip this step involving the client.

This example uses Oracle’s 8.1.7 JDBC driver, which implements the ConnectionPoolDataSource interface with an interface called OracleConnectionPoolDataSource. I will use objects of this type so I can access the methods that set the connection parameters.

Listing 14−3 illustrates the typical behavior of connection−pooling implementations. I start the application by configuring the OracleConnectionPoolDataSource object with the information needed to open a connection. This requires specifying the JDBC URL, username, and password. The driver uses these parameters when it creates the physical connection for the pool.

Listing 14−3: ConnPool.java

```
package Chapter14;

import java.sql.*;
import javax.sql.*;
import oracle.jdbc.pool.*;

public class ConnPool {

  //Global variables
  static OracleConnectionPoolDataSource ocpds = null;
```

```
static PooledConnection pc_1 = null;
static PooledConnection pc_2 = null;

public static void main(String[] args){
  String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
  String user = "sys";
  String password = "orcl";

  try{
    ocpds = new OracleConnectionPoolDataSource();
    ocpds.setURL(jdbcUrl);
    ocpds.setUser(user);
    ocpds.setPassword(password);

    //Create a pooled connection
    pc_1 = ocpds.getPooledConnection();

    //Open a Connection and a Statement object
    Connection conn_1 = pc_1.getConnection();
    Statement stmt = conn_1.createStatement();

    //Build query string
    String sql = "SELECT count(*) FROM v$session ";
    sql = sql + "WHERE username = 'SYS'";

    //Execute query and print results
    ResultSet rs = stmt.executeQuery(sql);
    rs.next();
    String msg = "Total connections after ";
    System.out.println(msg + "conn_1: " + rs.getString(1));

    ///Open second logical connection and execute query
    Connection conn_2 = pc_1.getConnection();
    stmt = conn_2.createStatement();
    rs = stmt.executeQuery(sql);
    rs.next();
    System.out.println(msg + "conn_2: " + rs.getString(1));

    //Open second physical connection and execute query.
    pc_2 = ocpds.getPooledConnection();
    rs = stmt.executeQuery(sql);
    rs.next();
    System.out.println(msg + "pc_2: " + rs.getString(1));

    //Close resources
    conn_1.close();
    conn_2.close();

  //Standard error handling.
  }catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
  }catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
  }finally{
    //Finally clause used to close resources
    try{
      if(pc_1!=null)
       pc_1.close();
    }catch(SQLException se){
```

```
        se.printStackTrace();
      }finally{
        try{
         if(pc_2!=null)
           pc_2.close();
        }catch(SQLException se){
          se.printStackTrace();
        }
      }
    }//end try
    System.out.println("Goodbye!");
  }
}
```

The output from Listing 14−3 is as follows:

```
Total connections after conn_1: 1
Total connections after conn_2: 1
Total connections after pc_2: 2
Goodbye!
```

In the preceding example I created a PooledConnection object, pc_1, to supply Connection objects. Next, I created two logical connections, conn_1 and conn_2. After their creation I verified that only one physical database connection existed even though I have two logical connections. (I do this by referring to the output.)

However, once I opened a second PooledConnection object, pc_2, the number of physical connections increased by one.

As this example demonstrates, opening additional logical connections has no effect on the number of physical connections to the database. Additional physical connections occur only when you instantiate more PooledConnection objects.

Summary

The JDBC 3.0 javax.sql package provides you with two interfaces, DataSource and ConnectionPoolDataSource, to help you address some of the challenges associated with enterprise−database development.

The DataSource interface, when used with JNDI, enables you to abstract database−specific information from the client. You can eliminate the need for the client to specify usernames, passwords, and JDBC URLs by using objects that implement the interface. This feature enables you to change the physical location of a database without affecting the client’s code base.

A vendor may implement connection pooling with the ConnectionPoolDataSource interface. Connections retrieved from the ConnectionPoolDataSource object are taken from a pool of active database connections. This means that you can share physical connections to the database server, either among multiple clients in a J2EE application or across multiple threads in local applications.

However, before you can take advantage of these features you must ensure that your driver implements the javax.sql package.

Chapter 15: Understanding Distributed Transactions

by Johennie Helton

In This Chapter

- Understanding distributed transactions
- Exploring the components of distributed transactions
- Looking at the relationship between distributed transactions and Java

Database−management systems (DBMSs) and databases are often physically distributed — that is, they reside in different locations throughout the enterprise. This fact has given rise to *transaction management*, which has evolved as the industry matures. This chapter introduces the basic concepts of distributed trans− actions, their importance in the enterprise, and the different standards that have evolved to provide interoperability among different transaction applications.

Transactions are expected to have high availability, good performance, and low response time — and at as low a cost as possible. Distributed transactions are the backbone of the enterprise, and in this chapter I discuss some of the technologies used with Java for distributed transactions, such as Java Messaging Service (JMS), the Java Transaction Service (JTS), and Enterprise JavaBeans (EJBs).

Understanding the Basics

During the last decade, the use of heterogeneous platforms and technology has increased exponentially, and so has the need for effective integration mechanisms to tie the platforms together. One such integration mechanism is transaction processing (TP) which makes distributed computing reliable; transactions are the backbone of all our day−to−day activities. Industries such as banking, manufacturing, health care, and the stock market depend on transaction processing for their everyday business.

Transaction definition and properties

Distributed transactions are transactions that span multiple nodes in a network, and the transaction operations are performed by multiple distributed applications. The concept of transactions is ages old; however, no standard definition of the term exists. However, a *transaction* can be defined as a unit of work that consists of several operations on shared system resources, and that satisfies the ACID criteria: *atomicity*, *consistency*, *isolation* and *durability* criteria.

- **Atomicity** — A transaction is an atomic unit of work; that is, it is performed in its entirety or not performed at all. If the transaction is interrupted by failure, all effects are undone.
- **Consistency** — A transaction takes the system from a consistent state to another consistent state even if an error occurs in the transaction. By *consistent* I mean internally consistent; in database terms, this means the database must satisfy all of its integrity constraints.
- **Isolation** — The effects of a transaction should not be available (or visible) to other transactions until the transaction is committed. The implication of this requirement is that if several transactions are running at once they are treated as if they were being run in sequence.
- **Durability** — Transaction changes are persisted once the transaction is committed; these changes should never be lost because of subsequent system failures.

A transaction has a beginning and an end. A transaction is terminated by either a commit or a rollback. A transaction is *committed* when all the changes made during the transaction are persisted; it is *rolled back* when all changes associated with the transaction are undone.

Two−phase commit

Distributed transactions are transactions that span multiple nodes in a network and typically update data in multiple systems; *resource managers* typically manage the data in those different nodes (which are usually located in different physical locations). In order to preserve the atomicity property, you must synchronize the updates required by distributed transactions. *Two−phase commit* is a protocol that ensures that all changes are synchronized and that synchronization can be undone if required. As the name suggests, the two−phase protocol has two phases.

In the first phase the transaction manager sends a *prepare to commit* message to all resource managers involved in the transaction, which persist the updates to disk for later access and return an acknowledgement to the transaction manager. Once the transaction manager receives the acknowledgment messages from all its resource managers, it continues to the second phase. If the transaction manager receives a *cannot commit* message from one of its resource managers, it sends *abort* messages to all resource managers, instructing them to discard all updates associated with the transaction. If one or more of the resource managers cannot be contacted, the transaction manager logs the messages for later and the abort continues.

In the second phase, the transaction manager sends a *commit* message back to all resource managers. Then the resource managers retrieve the saved information and make it durable. Resource managers return a message to the transaction manager, indicating that their part of the work has been committed. When all these messages are received, the transaction manager commits the transaction and returns the result to the client. If any of the resource managers do not respond because of communication failure, the transaction manager logs it; the resource manager still has locks to the data, and when communication is restored the transaction manager continues with the commit.

> Note There are many more points of failure (such as a resource not being available) here: At any point the transaction manager can send a message to the resource managers to force an abort and undo any of the changes caused by the transaction.

Transaction−processing performance and availability

Performance is critical to distributed systems; clients usually want their requests met in a very short period of time. Therefore, response time is one of the main measurements of performance for a transaction−processing system. Customers (such as banks and airlines) also care about system scalability. Imagine a system with great response time that only works on a few teller machines.

Distributed−transaction applications are a classic example of systems that need to meet response time and throughput at minimum cost. The Transaction Processing Performance (TPC) is a consortium that has defined a set of benchmarks to be used to compare different TP systems. Each benchmark specifies a set of transaction programs, measures the systems response and throughput under a specific workload, and measures the transactions per second (or minute) and the cost (of hardware, software, and the program’s price for five years) of the transaction rate.

The availability is the fraction of time the transaction−processing application is up and running (not down because of hardware or software failures, power failures, or the like). An availability of 99.9 percent means that the system is down for about one hour per month. Because transaction−processing applications are

important — the backbone of the enterprise, in fact — the availability requirements of the system are high. They are affected by software, hardware, environment, and system maintainability and management.

Replication

Replication creates multiple copies of a server data to increase performance and availability. It increases availability because when one server is down its replica is still available and accessible; this is very helpful for mission−critical applications such as financial systems. Replication also increases performance because by creating copies of a server you allow queries to be serviced without disturbing the primary server during updates. Figure 15−1 shows a replication diagram in which the server is replicated but the database resource itself is not; this improves availability but not performance, because queries need to be synchronized against updates.

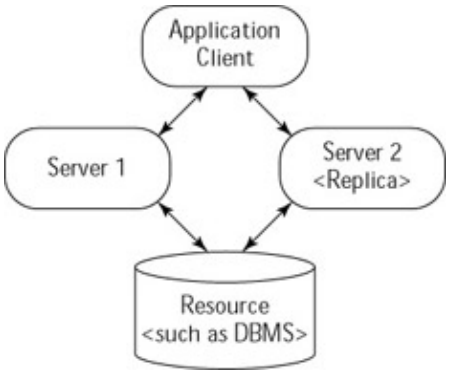

Figure 15−1: Replicating servers

Another choice is to replicate the resource itself so that not only is the server replicated but the database resource is too. This approach improves both performance and availability, but the main challenge is to keep both copies of the data current. Many technologies have been created to address this challenge. One such technology is *synchronous replication*, with which all the replicas are synchronized at the same time. With *asynchronous replication*, one transaction updates its replica and the update is propagated to the other replicas later. Other choices include replication based on timestamps, and read−one/write−all−available replication. A detailed discussion of these techniques is beyond the scope of this book. Figure 15−2 illustrates the replication of resources.

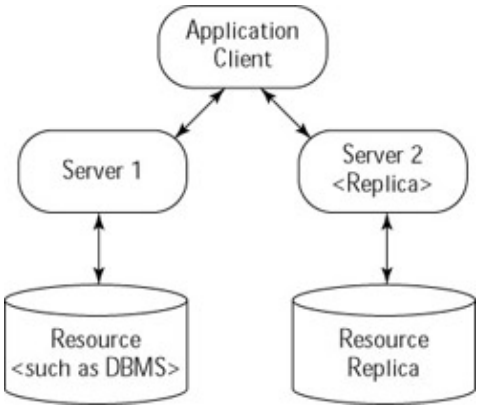

Figure 15−2: Replicating resources

Understanding Distributed Transactions

So far I have described the basics of transaction processing. There are two more concepts that are important in distributed transactions: the Transaction Monitor and the Transaction Service. The Transaction Monitor's main functionality is to provide a framework for the efficient use of system resources and the Transaction Service allows transaction synchronization. The following sections explore these concepts in more detail.

Understanding the Transaction Monitor

Transactional applications are complex and typically service a large number of client requests, and each client requires that its transaction be serviced within a reasonable amount of time. Because of this, the application must make efficient use of system resources, thereby increasing the time for development; the *TP monitor* is the framework that encapsulates use of system resources for the transaction. The TP monitor scales the application for thousands of transactional requests without making heavy demands on system resources. One of the main functions of the TP monitor is to transfer information among the different parts of the system. Another function of a TP monitor is the communication paradigms used such as RPC (remote procedure call), peer−to−peer, and queues. A detailed description of these paradigms is beyond the scope of this book.

Several transaction−processing tools and standards have evolved over the years. The Open Group (X/Open), Object Management Group (OMG), and Open System Interconnection (OSI) have created standards to address transaction−processing and interoperability concerns. The X/Open Distributed Transaction Protocol Reference Model (DTP) is the basis for specifications such as Object Transaction Service (OTS), Java Transaction Service (JTS) and Enterprise JavaBeans (EJB). The DTP Model has three basic components: the application program (AP), the transaction manager (TM), and the resource manager (RM). The X/Open group has added the Communications Resource Manager (CRM) specification to DTP to specify interoperability across different transaction−processing domains. For more information about DTP, visit X/Open at www.xopen.org/.

The OMG (www.omg.org/) created and maintains the Common Request Broker Architecture specification, better known as CORBA. CORBA is an interface for remote−object invocation and the OMG added the OTS specification for transactional services based on CORBA. As I mentioned above, the OTS specification builds on the X/Open DTP model. The following sections give a more detailed explanation of the DTP and OTS models.

The X/Open DTP model

X/Open is a standards organization whose purpose is to create interfaces among components in distributed−transaction systems. A transaction−processing application automates business functionality (such as making airline reservations) and can invoke multiple objects that perform multiple requests. As I mentioned earlier, the Distributed Transaction Processing (DTP) Model, created in 1991, divides a transaction−processing system into application−program (AP), transaction−manager (TM), and resource−manager (RM) modules; in addition, it defines interfaces between the modules. The AP is the client program that interacts with the TM and RM modules. The TM module coordinates and controls the execution of transactions, and the RM module manages the accessing of shared resources. The DTP components and interfaces are as follows:

- TX — This interface enables communication between the AP and TM modules; the Structured Transaction Definition Language (STDL) interface is an alternative to this interface. Some of the operations available through this interface include tx_open, tx_commit, and tx_rollback. Some of the definitions used in this interface are begin, which starts a transaction, commit, which commits a

transaction, rollback, which aborts a transaction, and info, which gets the status of a transaction.

- XA — This is an important interface that enables communication between the RM and TM modules. It supports transactions that can happen across multiple EIS systems. This interface includes the xa_start, xa_commit, xa_rollback, and xa_prepare operations. Some of the definitions used in this interface are starts, which starts a transaction, prepare, which prepares a transaction for the two−phase commit protocol, end, which ends the association on the transaction, commit, which commits the transaction, and reg, which registers the transaction with a TM.
- CRM — This is an API to a communications protocol for remote, transactional communications. The three protocols adopted by the X/Open group are:
    - TxRPC — Based on ACMS and Transarc RPC models, supports remote procedure call communications.
    - XATMI — Based on Bea's Tuxedo, for client/server communications.
    - CPI−C — Based on CICS and LU 6.2, for peer−to−peer communications.
- XA+ — This interface extends the XA interface so that the TM module can communicate with the CRM. Operations with the prefix xa_ are used to request suspension or completion of transactions to the CRM (and also to propagate information to other transaction branches). Operations with the prefix ax_ are used by the CRM for requests to the TM.
- RM — This is the resource manager defined by the system resources and not by the X/Open DTP model.
- XAP−TP — This is an API for CRM and OSI TP communication.
- STDL — This is an optional interface to the CRM protocols.

The OTS standard

OMG's OTS defines CORBA IDL interfaces; these interfaces specify the primitives for distributed transaction−processing systems. OTS does not specify the language to be used or how the application is to be distributed in the system; the application uses the IDL interfaces for transactional work involving distributed objects. The entities in OTS are the *transactional client*, the *transactional server*, the *recoverable server*, and the *transaction context*.

The transactional client begins and ends a transaction; the program that begins a transaction is usually referred to as the *transaction originator*. The transactional server holds *transactional objects* that are not involved in the completion of the transaction but that may force a rollback; a transactional object is an object that takes part in a distributed transaction and may contain persistent data. The recoverable server hosts *recoverable objects* and *recoverable resources;* both of which have the transactional object as their super−class. In addition, recoverable objects and recoverable resources interact directly with recoverable data; they are involved in the completion of a transaction. If the transaction context is set, a transaction is associated with it; the transactional context provides information to all transactional objects and determines if an associated thread is transactional.

Understanding the Transaction Service

The Transaction Service provides transaction synchronization across the different parts of the distributed−transaction application. In a typical scenario, the client begins a transaction via a request to the Transaction Service, which establishes a transaction context and associates it with the client. The client can then continue requests, which are automatically associated with the client and share its context. When the client decides to end the transaction, it makes a request to the Transaction Service. If no errors occur then the transaction is committed; otherwise it is rolled back. Obviously, many different scenarios are possible, including one in which the client controls the propagation of the transaction context. Figure 15−3 shows the major components and interfaces of the Transaction Service; the components are the transaction originator,

the transaction context, the Transaction Service, and the recoverable server (all of which I described at the beginning of this chapter).

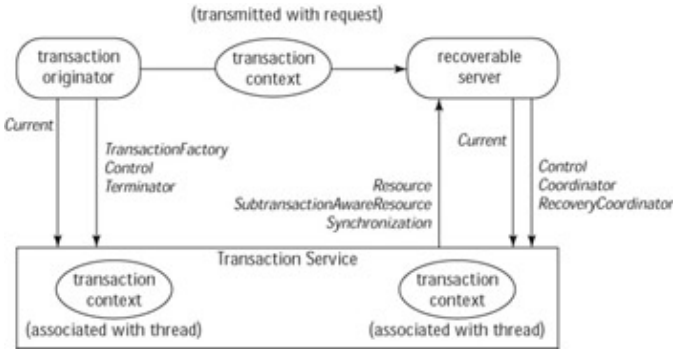

Figure 15−3: Major components and interfaces of the Transaction Service

The Transaction Service functionality

The Transaction Service provides operations to control the scope and duration of a transaction and the coordination of the termination of transactions. It is important to note that the Transaction Service supports two transaction models, *flat transactions* and *nested transactions*. However, an implementation is not required to support nested transactions.

Flat transactions, which are modeled from the X/Open DTP transaction model, are considered to be top−level transactions that do not have child transactions. Nested transactions allow applications to create transactions within existing transactions. An existing transaction is the parent of the *subtransactions*, which are the children of the existing transaction; the existing transaction is committed when all its children are completed.

In addition, the Transaction Service also supports model interoperability (allowing procedural integration), network interoperability (allowing interaction among multiple vendors), flexible transaction−propagation control (allowing the propagation to be managed either by the system or by the application), and TP monitors.

The transaction service interfaces

The interfaces of the Transaction Service enable communication among the different components of the service. Table 15−1 lists the interfaces that the Transaction Service offers and gives a description of each.

Note The OTS specification defines the interfaces in the CosTransactions module.

Table 15−1: Interfaces

Interface	Description
Control	Represents the transaction and provides handles for the terminator and coordinator interfaces, which control the life cycle of the transaction. The two methods in this interface are the get_terminator and the get_coordinator.
Coordinator	Provides operations to be used by recoverable objects to coordinate

	their participation in the transaction. Some of the methods include get_status, is_same_transaction, register_resource, and register_synchronization.
Current	Defines operations with which to begin and end a transaction, and to obtain information about the current transaction. The ORB to all transactional objects propagates the transaction context during the client's invocation. The methods in this interface include begin, rollback, status, and suspend.
RecoveryCoordinator	Used by recoverable objects to drive the recovery process in some circumstances. The replay_ completion methods is the only one offered: It is non−blocking and provides a hint as to whether a commit or rollback has been performed or not.
Resource	Defines operations so that the Transaction Service can coordinate the recoverable object's participation in a two−phase commit protocol. The methods offered are prepare, rollback, commit, commit_one_phase, and forget.
Synchronization	Provided by the Transaction Service to enable objects with transient−state data to be notified before the start of the two−phase commit protocol and after its completion. The two methods offered are before_completion and after_completion.
SubtransactionAwareResource	Used by resources that use the nested transaction model to register a specialized resource object to receive a notification when a subtransaction terminates. Also, the Transaction Service uses this interface on each Resource object registered with a subtransaction to notify a commit or rollback. The two methods offered are commit_ subtransaction and rollback_ subtransaction.
Terminator	Commits or rolls back the transaction; typically used by the transaction originator. The two methods offered are commit and rollback.
TransactionFactory	Allows the transaction originator to begin a transaction. It can be used to create new transactions or to re−create a transaction from an imported context. The only two methods offered in this interface are create and recreate, which return the Control object that represents the newly created transaction. The major difference between the two, is that re−create "reincarnates" an existing transaction in an arbitrary thread.
TransactionalObject	Indicates that an object is transactional. An object that inherits from this interface is marked so that the transaction context (associated with the client thread) is propagated to all the client invocations of the object's methods. This interface is only a marker; no methods are defined in it.

Context management

As I mentioned before, a client application can choose either direct or indirect context management. *Indirect context management* uses the Current object to associate the transaction context with the application thread. With *direct context management* the application manipulates all the objects associated with the transaction, including the Control object. A transaction's operations can be either explicitly or implicitly propagated. An object may request *implicit propagation*, meaning that the Transaction Service implicitly associates the client's transaction context with any of the object's requests. Or it may request *explicit propagation*, meaning that the application propagates the transaction context by passing objects (defined by the Transaction Service)

in the parameters of its requests. The Transaction Service supports all four permutations of context management and propagation. They are described in Table 15−2.

Table 15−2 : Context Management and Propagation Permutations

Permutation	Description
Direct context management with explicit propagation	The application directly accesses the objects that describe the state of the transaction (including the Control object). The appropriate Transaction Service object is passed as a parameter to any request, which is used to control the transaction.
Direct context management the with implicit propagation	Clients that access the Transaction Service can use resume operation of the Current interface, to set the implicit−transaction context associated with its thread; after this, all other requests have implicit propagation of the transaction context.
Indirect context management with explicit propagation	An application using an implicit−propagation model can use explicit propagation. The application then can get access to the Control object using the get_control operation of the Current interface.
Indirect context management with implicit propagation	To create and control transactions the application uses the Current object, and the transaction context is implicitly propagated to the object.

Distributed Transactions and Java

Distributed transactions are complex and building them requires highly experienced architects and developers; just imagine if you needed to design and hand−code a distributed application that guaranteed compliance of all its distributed transactions with the ACID properties! Sun Microsystem’s J2EE architecture takes a divide−and−conquer approach to distributed systems by providing a set of APIs. In this section I give an overview of some of the J2EE components and APIs that offer support for distributed transactions and applications.

J2EE supports transactional−application development; the application can be comprised of servlets and JSPs accessing enterprise beans; each component may acquire one or more connections to shared resources via access to resource managers. J2EE does not require that applets and application clients support transactions; some providers have chosen to support them, which implies that applets and application clients may access the UserTransaction object directly.

Note In multi−tier applications, servlets and JSP pages are typically used for the presentation tier, so accessing transaction resources, data, and other resources is not a good idea. It is best to leave such transactional work to enterprise beans in the EJB tier.

As I discussed earlier in this chapter, transaction managers coordinate distributed−transaction boundaries (beginning, end, and rollbacks), and handle issues such as concurrency and deadlock. In addition, the X/Open requires transaction managers to guarantee the transactional integrity of the different parts of the distributed transaction. *Resource adapters* have evolved to facilitate the interaction of the transaction manager with the heterogeneous selection of resource managers in the industry. For example, JDBC provides a resource adapter to DBMSs that can be used by transaction managers. A transaction is described as a *local transaction* when it is associated with a single physical connection.

XRef Chapter 19, “Accessing Data with Enterprise JavaBeans,” provides a description and introduction to the EJB architecture in general and EJBs in particular.

In the EJB architecture, for instance, a physical connection in the deployment descriptor can be marked as shareable (or not). Local transactions allow application servers to manage resources that are local to the resource adapter; local transactions cannot manage Enterprise Information System (EIS) resources and cannot participate in the two−phase commit protocol. The application server starts a local transaction based on the current transaction context, and does a commit on the local transaction and cleans up the EIS connection after the transaction is completed.

EIS and EAI

The EIS contains the Enterprise Application Integration (EAI) system. It also has the backend data information used for processing information and business logic across the enterprise. The EAI integrates multiple applications: examples of these applications and systems include BEA’s Tuxedo and Vitria’s BusinessWare. Many of these systems provide asynchronous transactions for publish and subscribe messages; messaging systems such as the JMS are used to transfer and manage messages. If the distributed application has an entity bean that needs to commit on the client side, the EAI system synchronizes that commit with the appropriate component on the system using the EIA’s transaction manager and the server’s transaction−manager to ensure transactional integrity. The connector architecture provides a means of simplifying this architecture and establishing the communication. A discussion of the connector architecture is beyond the scope of this book, but it is nice to know that you do not need to create a management system to coordinate your distributed−application connectivity.

In the EIS architecture transactions can be Local or XA, depending on how they are created and deployed. Depending on the transaction type, the system uses either a LocalTransaction interface or the XAResource interface. The application server uses the security manager for method invocation. Also, the application server may use its pool manager as part of the connection manager. In addition, the transaction manager gets the transaction interface (to handle the transactions internally for commits and rollbacks) based on the other managed connections and resources.

Components in a distributed application typically communicate with each other by passing messages. A *messaging service* is software that applications and their components use to pass messages in a reliable, platform−independent, language−independent, and preferably configurable way.

The javax.transaction package provides an API that defines the contract between the transaction manager and either the resource manager, the application itself, or the application server. The application server vendors and the resource manager–driver vendors provide the implementation of the javax.transaction API. The set of interfaces defined in this package are:

- Status — Defines a static variable for transaction−status codes. Some of the values include STATUS_ACTIVE, STATUS_PREPARED, and STATUS_ROLLEDBACK.
- Synchronization — A mechanism for you to get notifications before and after the transaction completes.
- Transaction — Provides operations to be performed against the transaction in the target Transaction object. The Transaction object can be used for resource enlistment, synchronization registration, transaction completion, and status−query operations.
- TransactionManager — Defines the methods with which an application server can manage transaction boundaries.
- UserTransaction — Defines the methods used by an application to explicitly manage transaction boundaries.

JMS

The Java Messaging Service (JMS) is a Java API that allows clients to communicate with underlying message services. The JMS API is part of J2EE, but no requirement exists — at this point — for an underlying service−provider implementation. In other words, the J2EE provide a set of interfaces based on the Java Transaction Architecture (JTA) and the Java Transaction Service (JTS). JMS supports both point−to−point and publish−and−subscribe messaging models. *Point−to−point messaging* refers to the communication between a producer and consumer, while *publish−and−subscribe* refers to the process by which a message producer publishes a message, which any consumer may access, to a particular topic.

You use JNDI to create an initial context to a JMS connection factory, and you use a JMS connection factory to create connections to a JMS−compliant provider. Given a JMS connection, you can retrieve a session context to create message producers and consumers. Message providers can then send messages to the consumers, which can filter the messages at their end.

Message−driven beans can be used for asynchronous messages. This type of bean is a stateless, server−side, transactional−aware component that delivers messages via JMS. *Asynchronous messages* allow applications to communicate so that the sender (producer) does not have to wait for the receiver to receive or process the message.

JTS and JTA

The Java Transaction Service (JTS) uses the CORBA OTS interfaces. JTS (via the standard IDL/Java programming language mapping) uses org.omg.CosTransactions for interoperability, and org.omg.CosTSPortability for portability. These interfaces define a standard mechanism for implementations that use Internet Inter−Orb Protocol (IIOP) to generate and propagate the transaction context among JTS transaction managers. JTS specifies the implementation of a Transaction Manager that supports the Java Transaction API (JTA) at the high level and implements a Java mapping of the OMG OTS specification at the low level. (For more information, visit the JTS Web site at http://java.sun.com/products/jts/.)

In the Enterprise Java middleware are five parties involved in distributed transactions: the *application server*, the *resource manager*, the *transaction manager*, the *stand−alone transactional application*, and the *Communication Resource Manager* (CRM).

- The *application server* (the TP monitor) provides the infrastructure required to support the application runtime environment; an EJB server is an example of an application server.
- The *resource manager* provides the application with access to resources; an example is a relational−database server. The resource manager uses *resource adapters* to access the resource; an example of a resource adapter is a JDBC driver.
- The *transaction manager* provides services and management for transaction demarcation, transactional resource management, synchronization, and transaction−context propagation. The stand−alone *transactional application* is a component−based application that relies on a server for transactional−management support; an example is the EJB architecture.
- The *CRM* supports the transaction context propagation and access to the transaction service for incoming and outgoing requests.

The implementation of the transaction manager must implement the JTA API. The implementation of the Java mapping of the OMG OTS is not exposed to the clients of the transaction manager; the exposed API is that of the JTA that clients can use to gain access to the transaction manager functionality. Some of the functionality of the transaction manager includes giving applications and servers the ability to control the scope and

duration of a transaction, allowing multiple components to perform work that is part of a single atomic transaction, allowing transaction synchronization, and coordinating the completion of global transactions with work performed by transactional resources. The JTA interfaces that give you the control of transaction−boundary demarcation and transactional operation are javax.transaction.TransactionManager, javax.transaction.Transaction and javax.transaction.UserTransaction.

EJBs

Enterprise JavaBeans take a divide−and−conquer approach to transactional−application development because they are ideally suited to be developed by different people with different levels and areas of expertise. You use EJB technology to specify which objects and operations participate in transactions. EJBs can be partitioned to be either client−side or server−side development. The server−side development for distributed applications is usually complicated. However, because much of the functionality is provided by the container, using the EJB architecture simplifies the creation of distributed applications. Functionality including such things as security, management, connectivity, resource management, and allocation is provided by services available to the EJBs either programmatically or declaratively.

You can specify how EJBs handle transactions by setting transaction policies such as TX_REQUIRED, TX_REQUIRES_NEW, TX_NOT_SUPPORTED, TX_BEAN_MANAGED, TX_SUPPORTS, and TX_MANDATORY.

> XRef The transaction−support attributes (the transaction policies) are described in Chapter 19, “Accessing Data with Enterprise JavaBeans,” in Table 19−3.

If the policy is TX_NOT_SUPPORTED, then no transactional support exists. If the policy is any other value, you have to consider the type of transaction support you need. For example, if you have an EJB in a CORBA−based EJB runtime environment, you need to create a mapping with the CORBA OTS as follows: The CORBA client invokes an enterprise bean through stubs from the IDL interfaces created from the bean’s home and remote interfaces, and if the client is invoked within a transaction, the CORBA Object Transaction Service interfaces are used.

The EJB deployer is responsible for taking an enterprise bean and making it available (deploying it) in a certain context. The context determines the policies and behaviors the bean will have during its life cycle. The deployment descriptor will keep information (among other information) about transaction policies, access control, and JNDI names for the EJB. The transaction policies are described by the transaction scope and the isolation levels of the EJB.

Isolation levels

The isolation levels are originally defined in the ANSI SQL specification. The isolation levels were adopted by the ODBC and JDBC standards, and then incorporated into the EJB specification. Concurrent transactions (based on the ACID properties) can be serialized; that is, running concurrent transactions produces the same results as running them one after the other. This “serialization” of results may be violated in a few ways: *dirty reads*, *non−repeatable reads*, and *phantoms*.

Dirty read refers to a transaction (say T2) that reads a value from the database that another transaction (say T1) has written; after T2 reads the value, T1 rolls back. It’s called a *non−repeatable read* when T1 reads a row in the database, T2 updates that same row, and then T1 reads the row again. Finally, *phantoms* refer to the rows that match a query executed by T1 but are not available when T1 performs the query (because T2 has not inserted them yet). Based on these violations, four isolation levels are available to EJBs:

- TRANSACTION_READ_COMMITTED — Does not allow dirty reads.
- TRANSCTION_READ_UNCOMMITTED — Allows all three violations.
- TRANSACTION_REPEATABLE_READ — Allows phantoms, but not the other two violations.
- TRANSACTION_SERIALIZABLE — Does not allow any of the three violations.

These isolation levels define how and when the EJB's data changes are available and/or visible to other applications accessing that data. Therefore, you must be aware of the type of isolation required for your applications; also, it's important to be aware of transaction−demarcation types. Two types of transaction demarcation exist: *bean−managed* and *container−managed*.

Transaction demarcation

Session beans are the only ones that can programmatically define bean−managed transaction demarcation; they do this with the javax.transaction. The container is aware of the type of demarcation selected because the container is able to read the <transaction−type> element of the <session> tag in the EJB deployment descriptor. The session bean can access UserTransaction objects (by calling getUserTransaction from its SessionContext object), which make available the begin, commit, and rollback methods for the transaction.

With session beans, you may set the <transaction−type> element because session beans have the option of using container−managed transaction−management services. However, the <transaction−type> does not exist for entity beans. Container−managed transaction−management services are provided by the container, and the deployment descriptor has a <container−transaction> element within the <assembly−descriptor> element. The transactions of the methods described in these elements are automatically managed by the container.

Transaction commit and message−driven beans

Assume you have an EJB that uses bean−managed persistence. If a client attempts to commit a transaction after accessing the entity instance's operations, the client accesses the commit method in the UserTransaction interface. Then the transaction service invokes the beforeCompletion method on a synchronization object. At this point the bean instance updates its state to the database in preparation for the two−phase commit protocol; the transaction service then performs the two−phase commit protocol and invokes the afterCompletion method on the synchronization object. Finally, the container invokes the ejbPassivate method in the bean's instance.

If you use message−driven beans and bean−managed transactions, the message delivery to the onMessage operation is outside the distributed−transaction context; the transaction begins when UserTransaction.begin is called within the onMessage method, and ends with the UserTRansaction.commit call. Two types of acknowledgments are possible when you use message−driven beans and bean−managed transactions: AUTO_ACKNOWLEDGE, which is the default, and DUPS_OK_ACKNOWLEDGE. The message receipt is not part of the transactions and so the container acknowledges it outside of the transaction.

Note When you are using message−driven beans and bean−managed transactions the UserTransaction.rollback method does not force a message redelivery; however, calling setRollbackOnly does force message redelivery for container−managed transactions.

The first step for message delivery is the same if you have a bean with container−managed persistence. Namely, the client invokes the commit operation of the UserTransaction interface when it attempts to commit the transaction; but the transaction service then calls the beforeCompletion operation. Next, the container invokes the ejbStore method on the bean instance and extracts the values of the CMP elements from the bean's instance. With that information, the container updates the bean's state in the database. The Transaction Service performs the two−phase commit protocol and invokes the afterCompletion method on the

synchronization object. Finally, the container invokes the bean instance ejbPassivate operation.

Note The container has three choices to access the bean's instance state: first, caching the ready instance between transactions and ensuring that the instance has exclusive rights to the state in the database; second, caching the state without ensuring exclusivity; and finally, not caching the ready instance between transactions.

If you are using message−driven beans and container−managed transactions, then you can use the MessageDrivenContext methods. For instance, you can use the setRollbackOnly method for error handling to mark the current transaction so that the transaction's outcome is a rollback. You can also use the getRollbackOnly method to determine whether or not the current transaction is marked for a rollback. Because the container automatically acknowledges the message when it commits the transaction, you do not need to specify an acknowledgement mode when you create a message bean using container−managed transactions.

Note Neither the JMS API nor the EJB specification specifies how JMS API method calls are to be handled outside a transaction. The EJB specification says that the container is responsible for acknowledging messages "successfully" processed by the onMessage method of a message−driven bean which is using bean−managed transactions.

Summary

Distributed transactions allow multiple components to participate in a transaction and allow the transaction to maintain its atomicity, consistency, isolation, and durability. Over the years many proprietary distributed−transaction applications and technologies have evolved, and the major standard for defining a consistent API is X/Open's Distributed Transaction Processing (DTP); it is this standard in which other standards, such as JTS, are built. The two−phase commit protocol allows distributed transactions to consistently terminate, using either a commit or a rollback). Some of the important aspects of a distributed−transaction application are performance and availability; I also described how you can use replications to increase the performance and/or availability of a system.

In the Java Enterprise Architecture, JTS provides an API for the support of transactional applications. Although JTS is based on OTS and DTP, these interfaces are not exposed to the clients; the API exposed to the clients is the JTA. Also, Enterprise JavaBeans are ideal for developing and managing distributed transactional applications. I explained transaction−demarcation types, message−driven beans, transaction attributes, transaction−isolation levels, and the ways in which enterprise beans handle transaction commit.

Chapter 16: Working with JDBC Rowsets

In This Chapter

- Examining the JDBC RowSet interface architecture
- Understanding the advantages of rowsets
- Working with the RowSet interface
- Using the JdbcRowSet class
- Creating serialized rowsets using the CachedRowSet class
- Generating XML documents with the WebRowSet class

Just as DataSource **objects** provide an alternative to DriverManager, RowSet objects provide an alternative to ResultSet objects. The RowSet interface extends the ResultSet interface, giving you the same functionality for viewing and manipulating data plus extra functionality that makes it a flexible and powerful object to use in your applications.

For example, RowSet objects implement the JavaBean component architecture, which means they can generate events and be used with Bean development tools. In addition, certain RowSet objects can operate without a continuous connection to the data source, which enables you to distribute them to disconnected clients. Lastly, rowsets can hold tabular data from any data source, as opposed to result sets, which only store tabular data from databases.

The extra functionality a RowSet object provides make it a very useful tool to keep in your toolbox. In fact, you may decide to use rowsets over result sets.

In this chapter, I provide details on working with classes implementing the RowSet interface. I begin with an overview of the JDBC rowset technology, followed by a presentation of rowset concepts. I conclude by discussing the JdbcRowSet, CachedRowSet, and WebRowSet classes, respectively, in Sun’s reference implementation.

Introducing JDBC Rowsets

In essence, a RowSet interface provides an enhanced version of the ResultSet interface. In fact, it extends the interface giving you access to the same methods and properties. However, rowsets have additional features that you may find advantageous when working in enterprise settings as it is well suited for disconnected, distributive, web−centric environments.

A RowSet object enables you to use data retrieved from a data source in many different ways. For example, using rowsets, you can perform the following tasks:

- Pass populated RowSet objects over networks to thin clients like PDAs
- View and update data in disconnected environments
- Take advantage of JavaBean technology, for example by using standard setter and getter methods for configuring and retrieving properties and generating events
- Mimic scrollable and updateable result sets for drivers that do not support these features
- Generate self−contained XML documents representing RowSet objects

Originally part of the JDBC 2.0 Optional Package, the javax.sql.RowSet interface is included with Java 1.4 JDK and JRE. As of this writing, you can find the formal specification in the Optional Package documentation at www.javasoft.com/jdbc. However, because the RowSet interface is defined in the javax.sql package, you must find a vendor's implementation to take advantage of this interface.

Note Because the RowSet interface is in the javax.sql package, Sun does not provide a standard implementation for it. However, Sun does offer an early−access version, so you can start experimenting with the technology. For the examples and discussions in this chapter, I use the Early Access Release version 4 of the RowSet reference implementation. To obtain the software visit http://www.javasoft.com/ and follow the links to the early−access download area.

Understanding Rowset Concepts

The ResultSet and RowSet interfaces have two primary differences:

- The RowSet interface supports the JavaBean component model. The advantage of this is that a developer can configure a RowSet object's properties using visual Bean−development tools. In addition, the RowSet object can inform registered listeners of events such as row updates.
- Rowsets operate in two modes: connected and disconnected. A connected rowset, like a result set, requires a continuous connection to the database. An error occurs if you use either a ResultSet or RowSet object after explicitly closing the connection or it otherwise breaks unexpectedly. In addition, a connected rowset always requires the presence of a JDBC driver so it can communicate with the database. A disconnected rowset, by contrast, stores row and column data in memory. As a result, the object does not rely on a data source connection to view and manipulate the data; it just accesses it from memory. However, you do need a connection to initially populate a disconnected RowSet object. Once populated, the object disconnects from the data source automatically.

Rowsets also have several convenient features. For example, both connected and disconnected rows contain their own connection logic. You need only to set certain connection parameters, such as username, password, and JDBC URL. The RowSet object will automatically connect when it needs access to the data source. Another useful feature of the RowSet interface is that it implements scrollable and updateable cursors on top of drivers that do not support them. This enables you to provide support for this feature regardless of the driver's capabilities.

I have only presented the tip of the iceberg with respect to the RowSet interface's features; to provide more information I need to discuss the different implementations. The following section introduces the different RowSet implementations available, and provides details about them.

Rowset implementations

The JDBC 2.0 optional package specification does not define specific implementations of the RowSet interface. However, it does present three classes as sample implementations: JdbcRowSet, CachedRowSet, and WebRowSet. The Early Access reference implementation provides working versions of these classes. Table 16−1 lists the classes and their features.

Table 16−1: Features of JdbcRowSet, CachedRowSet, and WebRowSet classes

Feature	JdbcRowSet	CachedRowSet	WebRowSet
Scrollable	x	x	x
Updateable	x	x	x
Connected	x	x	x
Disconnected		x	x
Serializable		x	x
Generate XML			x
Consume XML			x

The JdbcRowSet class provides a basic implementation of the javax.sql.RowSet interface, which turns a ResultSet object into a JavaBean and abstracts the details of working with ResultSet objects. You can use this object to simplify connection steps or to provide scrollable and updateable cursors for drivers that do not support these features. However, JdbcRowSet objects can only operate in connected mode, thus requiring the presence of a JDBC driver.

The CachedRowSet class provides the same functionality as the JdbcRowSet class, with one important difference: a CachedRowSet object can operate in a disconnected environment. As a result, it can function without a JDBC driver present. Once you populate a CachedRowSet object with data you may send it over the network to thin clients, provide it to sales professionals on their laptops, or serialize it for data archiving. The advantage of the CachedRowSet object is that the client to whom you send the object does not need the JDBC driver.

The WebRowSet class extends the CachedRowSet class, which lets you use the same properties and methods as well as operate in disconnected mode. However, the WebRowSet object can generate XML documents representing itself. You can use these documents to create copies of the WebRowSet object, which makes it easy for you to distribute the information across the Web and through firewalls using HTTP. It will likely play a major role as the Web−services architecture continues to develop.

Although the specification only lists three implementations, vendors may choose to create other types. The intent of the RowSet interface is to define a standard container for tabular data. The vendor adds additional functionality in their implementations.

Examining the rowset architecture

As I mentioned in the introduction, the javax.sql package contains the RowSet interface. As with other interfaces in the javax.sql package, you must rely on a vendor to implement the functionality. Some vendors may include this functionality in their driver distributions, or provide separate packages.

Sun chose to create the reference implementation using a different package structure. You will not find the JdbcRowSet, CachedRowSet, or WebRowSet classes in the javax.sql package. Instead, Sun placed them in the sun.jdbc.rowset package. To illustrate the architecture, Figure 16−1 presents the UML class diagram showing the relationships of the classes making up the reference implementation.

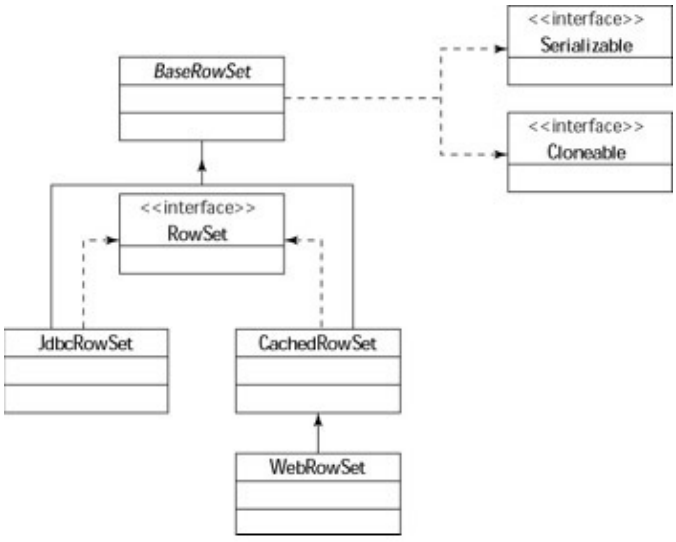

Figure 16−1: UML class diagram of the RowSet reference implementation

XRef Appendix D, “UML Class Diagram Quick Reference,” provides a guide to reading UML class diagrams.

The abstract parent class, BaseRowSet, implements the core functions and properties that all the child classes rely on. It does not implement the javax.sql.RowSet interface, but provides the common methods and properties used by the derived classes.

As you can see from the diagram, the rest of the classes extend the BaseRowSet class and implement the javax.sql.RowSet interface. As you can also see, the WebRowSet extends the CachedRowSet class giving you the ability to operate in disconnected mode.

Working with RowSet Objects

Developing with RowSet objects requires a different approach than developing with the standard JDBC components. Actually, you may find it easier to develop with RowSet objects.

For example, you instantiate fewer objects to accomplish a task using RowSet objects than you do when performing the same task with the standard components. In fact, you only need one: an object that implements the RowSet interface. When using standard JDBC programming techniques, you need a minimum of three objects: Connection, Statement, and ResultSet.

Just as you need fewer objects, you also perform fewer steps to interact with the database. Figure 16−2 shows the five steps required to execute a query using standard JDBC programming. Contrast this with the three steps required in a RowSet query, shown in Figure 16−3. Using RowSet technologies saves you two steps and the associated overhead of dealing with the additional objects. You do not have to explicitly open a connection, nor do you need to instantiate a specific Statement object.

Figure 16−2: The steps required in a standard JDBC query

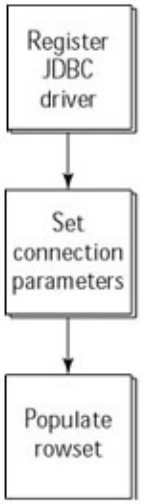

Figure 16−3: The steps required in a RowSet query

As you saw from the architecture diagram in Figure 16−1, the JdbcRowSet, CachedRowSet, and WebRowSet classes all extend the abstract BaseRowSet class. This means that they use the same methods and properties. As a result, you use objects instantiated from the three classes in similar ways because the BaseRowSet class implements a set of utility methods that they can use.

In the following sections I discuss the common items shared among the three different rowset classes. Specifically, I cover how to set properties, configure events, set connection properties, execute commands, retrieve data from a populated rowset, move about a rowset, configure transaction levels, set cursor concurrency, and clean up the application environment. Thanks to the abstract BaseRowSet class, you can do all these things in the same way for each derived class.

Setting rowset properties

As I mentioned earlier, the javax.sql.RowSet supports the JavaBean component model, which prevents you from directly accessing a RowSet object's properties. Therefore, you must use setter and getter methods for configuring the properties in classes implementing the RowSet interface. You can also use a GUI Bean development tool to configure rowset properties.

Most often your choice of property settings affects how the rowset connects to the data source and how the cursor behaves. Some of the properties you can set include:

- url and datasource, which identify the URL or DataSource object, respectively, to use while connecting.
- user and password, which specify the username and associated password, respectively, to use when connecting to the server.
- command, which represents the SQL statement to execute.
- type, which specifies whether the cursor is scrollable and/or sensitive to underlying data changes.
- concurrency, which specifies whether you can update the rowset.

The previous list only provides examples of properties you may find in the RowSet interface. Vendors may choose to include more or fewer properties, depending upon their implementations.

Configuring rowset events

RowSet objects can also generate JavaBean events. This enables you to notify other components implementing the RowSetListener interface of certain events that occur in a rowset object. Figure 16−4 shows a simple UML class diagram for the RowSetListener interface. As you can see from the diagram, the interface provides three methods that correspond to the events a RowSet object can generate. In addition, Table 16−2 lists the events RowSet objects generate, along with simple descriptions.

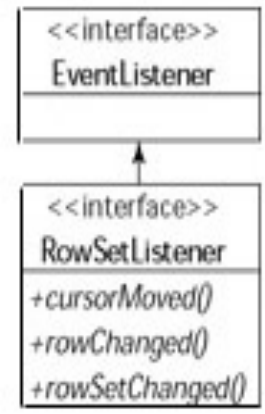

Figure 16−4: UML class diagram of the RowSetListener interface

The first event listed in Table 16−2, row changes, occurs whenever a client modifies the contents of a row. For example, when the client executes an UPDATE SQL statement. The next event, rowset changes, occurs when a client alters the entire rowset. This event occurs when a RowSet object's data changes when you execute a new SELECT statement. The last event, cursor movement, occurs when the RowSet object's cursor moves from one row to another. Calling any cursor−movement method, such as next(), previous(), or relative(), generates this event.

Table 16−2: JDBC RowSet Events

Event type	Description
Row changes	Occurs when the contents of a row change.
Rowset changes	Occurs when the entire rowset changes.
Cursor movements	Occurs when the cursor moves from one row to another.

Setting rowset connection properties

Unlike a ResultSet, a RowSet automatically connects to the data source when it needs to retrieve or update data. You do not need to explicitly call a connection method, as the object handles this task for you. The RowSet object's ability to connect automatically is one of its benefits.

However, you must satisfy two conditions before the RowSet object can connect to the data source. First, you must register a JDBC driver using either the DriverManager.registerDriver() or the Class.forName() static methods. Although you do not explicitly connect to the data source, the RowSet object uses the driver internally for communicating with the data source. Secondly, you must set the connection parameters. These typically include username, password, and the URL or DataSource object name.

> XRef Chapter 4, "Connecting to Databases with JDBC," contains more information about using the DriverManager class and the Class.forName() method.

The following snippet illustrates how to configure the connection properties:

```
//Create JdbcRowSetObject
JdbcRowSet jrs = new JdbcRowSet();

//Load driver and set connection parameters
Class.forName("oracle.jdbc.driver.OracleDriver");
jrs.setUrl("jdbc:oracle:thin:@localhost:1521:ORCL");
jrs.setUsername("toddt");
jrs.setPassword("mypwd");
```

The previous snippet does not make the data source connection. No connection is made until you provide a SQL statement and call the execute() method. In addition, although the snippet uses the JdbcRowSet object, any object implementing the RowSet interface operates in a similar manner.

Earlier I mentioned that a disconnected rowset, such as the CachedRowSet class, does not need a JDBC driver, or even a full JDBC API implementation, to operate. This remains true, but there is a caveat.

Before the object can operate only in disconnected mode you must populate it with data. At that point a CachedRowSet object can work without a driver present so you may send it to a client without an installed driver, and allow the client to view and modify the data. However, if the client manipulates the data, the RowSet object must have access to a JDBC driver before it can apply the changes to the underlying DataSource.

Executing SQL commands using rowsets

You can execute both DDL and DML SQL statements using a RowSet object. However, because the RowSet object acts like a JavaBean, you execute the statements differently from the way you execute them in standard JDBC programming.

One difference is that you do not need to instantiate Statement, PreparedStatement, or CallableStatement objects to execute SQL statements. The RowSet object manages the details of submitting the SQL command and handling the results. It determines whether you are submitting a parameterized query or calling a stored procedure, and acts accordingly.

If you execute a SELECT statement, the RowSet object populates itself with the data that satisfies the query. The object ignores other results, such as update counts, from INSERT, UPDATE, DELETE, or DML statements. However, a SQLException is thrown if an error occurs when the RowSet object executes any type of SQL statement.

Before trying to execute a command you must initialize the RowSet object's command property with the SQL statement. This String value represents the SQL statement you want to process. Once you set the property, calling the RowSet. execute() processes the statement.

You can use both standard and parameterized SQL statements with the RowSet object. You use parameter statements just as you do with PreparedStatement objects, which means that you must supply values for all parameter placeholders before calling the execute() method, or a SQLException will occur. To set the properties use the same set*XXX*() methods used by PreparedStatement objects.

XRef Chapter 5, "Building JDBC Statements," contains more information about creating and using PreparedStatement objects.

The following snippet demonstrates how to execute a parameterized query using a CachedRowSet object:

```
//Instantiate CachedRowSet object
CachedRowSet crs = new CachedRowSet();

//Load driver
Class.forName("oracle.jdbc.driver.OracleDriver");
crs.setUrl("jdbc:oracle:thin:@localhost:1521:ORCL");
crs.setUsername("toddt");
crs.setPassword("mypwd");

//Set and execute the command. Notice the parameter query.
String sql = "SELECT * FROM Produce WHERE Name = ?";
crs.setCommand(sql);
crs.setString(1,'Apple');
crs.execute();
```

Notice that I set the connection parameters in this snippet the same way I set them in the previous snippet. With respect to executing the SQL statement, this first step is to define the statement. I do so by assigning my SQL statement to the String variable sql. Next I set the command property with the setCommand() method, using the sql variable as a parameter.

Finally, I call the execute() method to process the query. This command causes the CachedRowSet object to open a database connection, submit the query, and retrieve the data. Because the CachedRowSet is a disconnected rowset, it closes the connection once it receives the data.

Fetching data from a rowset

Once you have populated a rowset, you need to extract the data from the rowset before it. To do so you rely on an inherited RowSet.get*XXX*() methods; where the *XXX* refers to the Java data type of the variable into which you want to place the value.

Just as when using the methods with a ResultSet object, you need to consider datatype issues. For example, it's possible to lose data precision by using the wrong get*XXX*() method on certain data types. You lose a number's mantissa when trying to retrieve a SQL DOUBLE into a Java int using the getInt() method.

XRef Chapter 7, "Understanding JDBC Data Types," provides more information about working with SQL and Java data types.

Traversing data in a rowset

Because the RowSet interface extends the ResultSet interface, you use its methods for moving around a data set. In addition, you can also control cursor properties such as scrollability and change sensitivity. A rowset object inherits all these behaviors and properties from the ResultSet interface. For your convenience, I present a summary of the cursor−movement methods in Table 16−3.

XRef See Chapter 6, "Working with Result Sets," for more information about moving through the data in a RowSet object using the cursor−movement methods. This chapter also explains how to set transaction−isolation levels and cursor−scrollability characteristics.

Table 16−3: RowSet Cursor−Movement Methods

Method	Description
next()	Moves the cursor to the next row. Returns true if successful.
previous()	Moves the cursor to the previous row. Returns true if successful.
beforeFirst()	Positions the cursor "before−the−first row". Calling the get*XXX*() method immediately after this method will produce a SQLException.
afterLast()	Positions the cursor "after−the−last row". Calling the get*XXX*() method immediately after this method will produce a SQLException.
first()	Moves the cursor to the first row of the result set. Returns true if successful.
last()	Moves the cursor to the last row of the data set. Returns true if successful.
absolute()	Moves the cursor to a specific row relative to the first row of the data set. Returns true if successful.
relative()	Moves the cursor to a specific row relative to the current row. Returns true if successful.

MoveToCurrentRow()	Moves the cursor to the remembered row.
MoveToInsertRow()	Moves the cursor to the insert row.

You are likely quite familiar with the cursor movement methods if you have worked with the ResultSet object. If not, they are simple, intuitively named methods that move the rowset cursor within its data. The most common method is next(), which moves the cursor to the next valid row in the data set. You use this method almost every time you use a rowset.

Like a result set, a rowset has two areas located "before−the−first row" and "after−the−last row" that do not contain data. The next(), previous(), absolute(), and relative() methods return false when moving the cursor into these areas. In addition, calling a get*XXX*() method when the cursor is in either location throws a SQLException.

> Tip Checking the return value from a cursor−movement function helps you track your location and can help prevent a SQLException.

Controlling scrollable and updateable properties

You can set the scrollability properties of the rowset's cursor using the setType() method. This method accepts constants from the ResultSet interface representing the different result set modes. The following lists the constants and their values:

- ResultSet.FORWARD_ONLY creates a rowset in which the cursor can only move forward. Used mainly by connected−only rowsets such as JdbcRowSet objects.
- ResultSet.SCROLL_INSENSITIVE creates a scrollable rowset that does not recognize changes to the underlying data. Most rowsets, especially disconnected ones, use this mode.
- ResultSet.SCROLL_SENSITIVE creates a scrollable rowset that recognizes changes to the underlying data.

You can set a RowSet object's concurrency property using the setConcurrency() method. This property indicates whether or not you can update the rowset data. The default setting does not allow updates. Nonetheless, you can use either of the following ResultSet interface constants as parameters in the setConcurrency() method to control the behavior:

- ResultSet.CONCUR_READ_ONLY configures the rowset so you cannot update the data.
- ResultSet.CONCUR_UPDATABLE configures the rowset so you can make changes to the data set or add new records.

From the previous discussion it should be evident that a RowSet object inherits most all its cursor movement and properties from the ResultSet interface. Although you may use different methods to set the properties, the behavior remains the same.

Setting transaction levels

You can also control the transaction−isolation level of a RowSet object. This setting determines what data a rowset can access during a transaction. Specifically, it effects whether the rowset can view data from other transactions.

Once again the RowSet interface relies on the behavior of another JDBC object to meet its needs. In this case, it uses the transactional settings in the Connection object. Every RowSet object must use a Connection object internally or else it can't access the data source. As a result, a RowSet object lets its internal reference to a

Connection object manage transactional issues.

To set the transaction−isolation level you rely on the setTransactionIsolation() method. It uses a Connection interface constant as a parameter representing the desired transaction level. The following list describes the constants you can use:

- Connection.TRANSACTION_NONE specifies that the connection does not support transactions.
- Connection.TRANSACTION_READ_UNCOMMITTED allows the rowset to read "dirty," or uncommitted, data from other transactions.
- Connection.TRANSACTION_READ_COMMITTED allows the rowset to read only committed data from other transactions.
- Connection.TRANSACTION_REPEATABLE_READ prevents a rowset from reading data with uncommitted data or data that change after a transaction begins.
- Connection.TRANSACTION_SERIALIZABLE does not allow the rowset to read any data, committed or uncommitted, from other transactions.

As you can see, the RowSet interface takes advantage of existing components to provide functionality. This also helps flatten the learning curve for using the objects in your applications.

Cleaning up after a RowSet

Just as you do other objects that occupy database resources, you should close a RowSet object once you finish using it. Calling the RowSet.close() method clears all database resources. Closing the object when you’re finished with it is especially critical when working with the JdbcRowSet object, as this object maintains a connection with the server once you call the execute() command. If you don’t explicitly close it, the connection may persist until either you exit the application or garbage collection occurs.

Objects instantiated from the CachedRowSet and WebRowSet classes connect to the data source as needed. Nonetheless, you should still explicitly close these objects when you’re finished to eliminate the chance of an errant connection occurring before garbage collection.

Using the JdbcRowSet Class

The JdbcRowSet object wraps a ResultSet object as a JavaBean component to simplify its use. Its main advantage is that it abstracts many of the details of connecting and retrieving data from a data source. Once you populate the rowset, you have all the ResultSet methods at your disposal for working with the data.

Of all the rowset implementations, the JdbcRowSet class is the simplest. As a result, it has some limitations. First, a JdbcRowSet object only operates as a connected rowset, and therefore it requires a JDBC driver and a continuous open connection to the database. Remember, the JdbcRowSet class provides a wrapper around a ResultSet object, which needs a connection to operate as well. This behavior carries over to the JdbcRowSet object as well. Another restriction is that you cannot serialize a JdbcRowSet object. This means that you can’t distribute or archive the object.

Despite the limitations, the JdbcRowSet interface provides you with an easy way to extract data from a data source. To use it, all you need to do is set a few properties and call the execute() method. You move around and extract data from the rowset with the same methods used in the ResultSet object. Listing 16−1 provides an

example program using a JdbcRowSet object.

Listing 16−1: JdbcRS.java

```
package Chapter16;

import java.sql.SQLException;
import sun.jdbc.rowset.JdbcRowSet;

public class JdbcRS {

  public static void main(String[] args){

    try {
     //Instantiate a JdbcRowSet object
     JdbcRowSet jrs = new JdbcRowSet();

     //Load driver and set connection parameters
     Class.forName("oracle.jdbc.driver.OracleDriver");
     jrs.setUrl("jdbc:oracle:thin:@localhost:1521:ORCL");
     jrs.setUsername("toddt");
     jrs.setPassword("mypwd");

     //Set and execute the command
     String sql;
     sql = "SELECT SSN, Name, Salary, Hiredate FROM Employees";

     jrs.setCommand(sql);
     jrs.execute();

     //Display values
     while(jrs.next()){
       System.out.print("SSN: " + jrs.getInt("ssn"));
       System.out.print(", Name: " + jrs.getString("name"));
       System.out.print(", Salary: $" + jrs.getDouble("salary"));
       System.out.print(", HireDate: " + jrs.getDate("hiredate"));
       System.out.println();
     }

     //Close the resource
     jrs.close();

    }catch (SQLException se){
      se.printStackTrace();
    }catch (Exception ex) {
      ex.printStackTrace();
    }

    //Say goodbye
    System.out.println("Goodbye!");

  }//end main
}// end JdbcRs
```

The output from Listing 16−1 is as follows:

```
SSN: 111111111, Name: Todd, Salary: $5000.55, HireDate: 1995−09−16
SSN: 419876541, Name: Larry, Salary: $1500.75, HireDate: 2001−03−05
```

```
SSN: 312654987, Name: Lori, Salary: $2000.95, HireDate: 1999-01-11
SSN: 123456789, Name: Jimmy, Salary: $3080.05, HireDate: 1997-09-07
SSN: 987654321, Name: John, Salary: $4351.27, HireDate: 1996-12-31
Goodbye!
```

From Listing 16−1 you can see that populating a JdbcRowSet object only requires a few steps. First I make sure to import a package containing the implementation. In this case I'm using Sun's reference implementation, so I use the statement: import sun.JDBC.rowset.JdbcRowSet. I do not need to import any java.sql classes other than the SQLException class; the JdbcRowSet class encapsulates any additional JDBC classes it requires.

Next, I instantiate a JdbcRowSet object to use throughout the application. Because this is a connected rowset, I must register a JDBC driver. As I mentioned in a previous section, you can use either the DriverManager.registerDriver() or the Class.forName() method to register the JDBC driver. I typically use the latter because of the flexibility the String parameter provides in terms of specifying the driver name.

After loading the driver I set the JdbcRowSet object's connection properties. In general, when working with a RowSet object you need to set only the properties it needs. In this example I must set the JDBC URL, username, and password. Other implementations, databases, or drivers may require different or additional parameters.

Next I initialize the command property with the SQL statement I want to execute. In this case I simply set a String variable, sql, to the SQL statement and pass it into the setCommand() method as a parameter.

Finally, I process the query using the execute() method. Behind the scenes, the JdbcRowSet object builds the appropriate objects for connecting to the database, executes the SQL statement, and populates the RowSet object with data. Next I display the results using a while−loop and the next() method. This is the same technique I use when working with result set data.

As I illustrated in Listing 16−1, using a JdbcRowSet object to retrieve data from a data source requires only a few objects and steps. Although it is simple to retrieve data using this object, it still requires a database session. This limits the ways in which you can use this object.

Using the CachedRowSet Class

As you saw in the previous section, the JdbcRowSet class has many benefits but has some limitations as well. For example, the JdbcRowSet class is a connected rowset that requires a continuous connection with the data source. Furthermore, you cannot serialize a JdbcRowSet object, which limits your ability to distribute or save the object. However, you should not consider any of these limitations serious, unless you want to share the JdbcRowSet object's data in a distributed architecture.

The CachedRowSet class overcomes those limitations. It provides a disconnected and serializable implementation of the RowSet interface. To operate in a disconnected state a CachedRowSet object creates a virtual database by caching the tabular information it receives from the data source. Storing the data internally makes the object self−contained and thereby allows it to operate while disconnected.

Once you have populated the object with data you can serialize it and share the data with other clients. This gives you a lot of flexibility in terms of building your application. For example, Figure 16−5 shows how you might use the object in a J2EE deployment. The figure presents an architecture wherein clients retrieve a

populated CachedRowSet object from a JNDI data source.

This architecture provides a couple of benefits. First, it minimizes the number of queries to the database. You need only query the database once to populate the CachedRowSet object. Every client who needs the same view does not have to query the database. This approach will minimize the impact on database resources if you have numerous clients.

Secondly, it enables you to distribute the data to mobile users such as sales employees. For example, the CachedRowSet object can store inventory levels for a product they sell. By taking the inventory object with them, sales employees can check product levels while disconnected from the network.

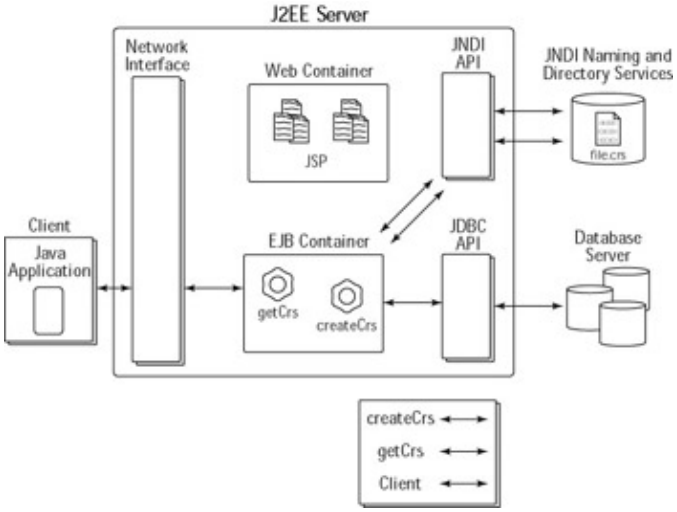

Figure 16−5: An example application architecture using CachedRowSet objects

A CachedRowSet object provides both updateable and scrollable cursor benefits as well. Usually these functions rely on driver implementations. However, because the CachedRowSet object caches its data, it can act as the "driver" to provide this functionality. As a result, you can use the CachedRowSet class to provide both updateable and scrollable functionality to drivers that do not support it.

Caution Do not use the CachedRowSet object with large data sets. Large data sets can easily exhaust available resources, because the objects cache the data in memory.

Although the CachedRowSet class has numerous benefits, it also has some drawbacks and constraints. First, you may not want to use CachedRowSet objects when working with large data sets. In addition, although you can update the data sets stored in a CachedRowSet object, transferring the changes to the underlying data source requires a connection.

Serializing a CachedRowSet object

Listing 16−2 provides an example of how you might use the CachedRowSet class to distribute a disconnected rowset. In the example I create two methods: writeCachedRowSet() and readCachedRowSet(). The writeCachedRowSet() method populates and serializes a CachedRowSet object; the readCachedRowSet() method reads the serialized object from disk and then returns it to the calling method.

The CachedRowSet object returned from the read CachedRowSet() method contains the same data as the original object serialized to disk. In a real application you would be able to store the serialized CachedRowSet object in a JNDI data source, or let a Web client download it.

Listing 16−2: CachedRS.java

```
package Chapter16;

import java.io.*;
import java.sql.SQLException;
import sun.jdbc.rowset.CachedRowSet;

public class CachedRS {

  //Constant to hold file name used to store the CachedRowSet
  private final static String CRS_FILE_LOC ="cachedrs.crs";

  public static void main(String[] args) throws Exception {

    try {

      //Create serialized CachedRowSet
      writeCachedRowSet();

      //Create CachedRowSet from serialized object
      CachedRowSet crs = readCachedRowSet();

      //Display values
      while(crs.next()){
        System.out.print("SSN: " + crs.getInt("ssn"));
        System.out.print(", Name: " + crs.getString("name"));
        System.out.print(", Salary: $" + crs.getDouble("salary"));
        System.out.print(", HireDate: " + crs.getDate("hiredate"));
        System.out.println();
      }


      //Close resource
      crs.close();

      }catch (SQLException se){
        se.printStackTrace();
      }catch (Exception ex) {
        ex.printStackTrace();
     }
  }//end main

  public static void writeCachedRowSet() throws Exception{
    //Instantiate a CachedRowSet object, set connection parameters
    CachedRowSet crs = new CachedRowSet();
    Class.forName("oracle.jdbc.driver.OracleDriver");
    crs.setUrl("jdbc:oracle:thin:@localhost:1521:ORCL");
    crs.setUsername("toddt");
    crs.setPassword("mypwd");

    //Set and execute the command. Notice the parameter query.
    String sql = "SELECT SSN, Name, Salary, Hiredate ";
    sql = sql + "FROM Employees WHERE SSN=?";
    crs.setCommand(sql);
    crs.setInt(1,111111111);
    crs.execute();

    //Serialize CachedRowSet object.
    FileOutputStream fos  = new FileOutputStream(CRS_FILE_LOC);
```

```
    ObjectOutputStream out = new ObjectOutputStream(fos);
    out.writeObject(crs);
    out.close();
    crs.close();

  }//end writeCachedRowSet()

  public static CachedRowSet readCachedRowSet() throws Exception{
    //Read serialized CachedRowSet object from storage
    FileInputStream fis  = new FileInputStream(CRS_FILE_LOC);
    ObjectInputStream in = new ObjectInputStream(fis);
    CachedRowSet crs = (CachedRowSet)in.readObject();
    fis.close();
    in.close();
    return crs;

  }//end readCachedRowSet()

}//end CachedRS
```

The output from Listing 16−2 is as follows:

```
SSN: 111111111, Name: Todd, Salary: $5000.55, HireDate: 1995−09−16
```

Although the CachedRowSet object provides more functionality than a JdbcRowSet object, the configuration remains the same. You perform the same initial steps — importing packages, loading a driver, and setting the connection parameters — as in Listing 16−1. Once again, notice that I need to import only the SQLException and CachedRowSet classes. The CachedRowSet class encapsulates all the required components for connecting, retrieving, and modifying data. However, I also need a java.io.* package to handle the object serialization task.

Stepping through the application shows that it is straightforward. I use the writeCachedRowSet() method to populate a CachedRowSet object and serialize it to a file called cachedrs.crs. (Notice also that I use a parameter query as my SQL command.) In this example I want only my record from the Employees table. As with the PreparedStatement object, I use a set*XXX*() method to supply a value for the parameter. After I set the parameter, I call the execute() method to populate the CachedRowSet. Next, I serialize the CachedRowSet object to disk using an ObjectOutputStream object.

Once I save the CachedRowSet object to disk, I call the readCachedRowSet() method to read the saved file from disk, instantiate a CachedRowSet object, and return the object to the calling main() method. (Notice that I cast the object read from disk to a CachedRowSet type.) After the method returns, I list the output using my standard combination of a while−loop and a next() method. The last task is to close the CachedRowSet with the close() method.

In this example I use the execute() method to populate the CachedRowSet with data. However, I could have used the CachedRowSet.populate() method instead. This method takes a populated ResultSet object as a parameter and uses the object to fill its cache. The disadvantage of the populate() method is that you need to handle all the standard JDBC objects required to execute a simple query. The advantage is that it enables you to leverage existing methods that return a ResultSet object to build rowsets.

As you can see, the CachedRowSet class packs plenty of flexibility. Not only can you operate without a JDBC driver, but you can serialize a CachedRowSet object and store it as a file or stream it over a network to a client. In addition, the CachedRowSet object enables you to update and insert data while disconnected.

Updating and inserting disconnected rowset data

A disconnected CachedRowSet object doesn't limit you to viewing its data; you may update or delete existing rows, or add new rows. Fortunately, you only need to learn one additional method to perform these actions. The RowSet interface inherits all the other required methods from the ResultSet interface.

A populated CachedRowSet object holds its data in memory, meaning that when you modify or insert a row the underlying data source is not immediately affected. Instead, the CachedRowSet object maintains both the new and the original values in memory. To apply the changes to the data source you must call the acceptChanges() method. Calling this method causes the CachedRowSet object to connect to the data source and submit the changes. If the changes fail for any reason, a SQLException occurs. The following snippet demonstrates how to update an existing row in a rowset:

```
//Populate a CachedRowSet object, crs
String sql = "SELECT * FROM Employees WHERE SSN = ?";
crs.setCommand(sql);
crs.setString(1,'Todd');
crs.execute();

//Move to first and only row and give myself a raise
crs.first();
crs.updateDouble(3,10000.22);

//Signal changes are finished
crs.updateRow();

//Write records to database
crs.acceptChanges();
```

To add a new row, you follow the same approach you used for the ResultSet object. First call the moveToInsertRow() method, inherited from the ResultSet interface, to position the cursor in the "insert row" buffer. In this area you build the new row. After populating the row's columns using the update*XXX*() method, call the insertRow() method to place the new row in the rowset's data set. Call the acceptChanges() method to commit the changes in the underlying data source.

To better understand how to insert a row into a CachedRowSet object, examine the following snippet (based on the CachedRowSet object used in the previous example):

```
//Move cursor to the insert row position
crs.moveToInsertRow();

//Add the data for the new row
crs.updateInt(1,997120987);
crs.updateString(2,"Nathan Dunn");
crs.updateDouble(3,2225.77);
crs.updateDate(4,Date.valueOf("1-1-02"));
crs.updateInt(5,100);

//write the rows to the rowset
crs.insertRow();

//Submit the data to the data source
crs.acceptChanges();
```

You can easily undo unwanted changes because both the original and changed data exist in memory. To that end, the CachedRowSet class provides two methods that undo changes. The first method, restoreOriginal(),

returns the entire rowset to its original values before the updates. This method is equivalent to the SQL ROLLBACK statement, which voids uncommitted changes. However, you cannot use the restoreOriginal() method after writing the changes to the data source with acceptChanges().

Caution The acceptChanges() method applies changes to the underlying data source. Once you call the command, you cannot undo its effects. It is equivalent to the SQL COMMIT statement.

The second method, cancelRowUpdates(), undoes changes to the row you are currently updating. The CachedRowSet class inherits this command from the ResultSet interface. Therefore, you use it in the same way. As with the restoreOriginal() method, once you call acceptChanges() the cancelRowUpdates() method has no effect. The bottom line: Be sure of your changes before calling the acceptChanges() method.

Using the WebRowSet Class

Like its parent class, CachedRowSet, objects instantiated from the WebRowSet class can be serialized and the file sent across the network to a client. In addition, the WebRowSet object can also operate as a disconnected rowset. However, the WebRowSet class has the extra benefit of being able to represent itself as an XML document. You can use the XML file just as you would a serialized CachedRowSet or WebRowSet object. For example, you can send it to a network client or store it for archival purposes.

In addition to generating an XML file, the WebRowSet object can also use an XML file to populate itself. Continuing with the example in the previous paragraph, the client can use the XML file you sent to build a duplicate WebRowSet object. The client can then update the data, generate a new XML file containing the changes, and send it to a JSP page or servlet, which in turn could update the data source.

Having the rowset represented in XML also gives you flexibility when you need to display the data on different devices or browsers. You can use XSL to format the XML document to be viewed on thin clients (such as Web browsers) or on mobile devices (such as WAP phones, PDAs, and handheld computers).

The WebRowSet class also works great in HTTP environments. Figure 16−6 shows an example architecture. As you can see, the client and the servers exchange XML documents representing WebRowSet objects. In this architecture, a client requests a data set from the server by accessing a servlet. The servlet uses a WebRowSet object to query the database and generate an XML document, which it passes back to the client. The client uses the XML document to populate a local WebRowSet object. Now the client can view or update the data. If the client updates the data, it uses its WebRowSet object to generate an XML document and return it to the servlet. Now the servlet can recreate the WebRowSet object from the XML document that it received and update the database with the client’s changes.

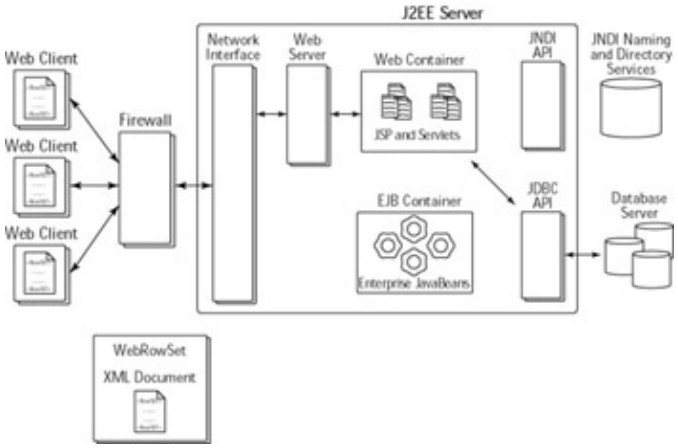

Figure 16−6: An example of a WebRowSet architecture

Relying on HTTP for the transport protocol provides several advantages. First, it eliminates the security issues associated with opening non−standard ports in the firewall for communications. Secondly, Web technologies provide robust and scaleable platforms with which to create applications. Lastly, you avoid issues such as unreliable connections that can come up on the Internet, which hinders your ability to stream objects or use RMI technologies to distribute serialized objects.

Because the WebRowSet class extends the CachedRowSet class, you rely on the same methods and properties for setting connection parameters and populating the rowset. You need only one additional method, writeXML(), to create an XML file representing the Web rowset. Listing 16−3 provides an example of how to generate the XML document using a WebRowSet object.

Listing 16−3: WebRS.java

```
package Chapter16;

import java.io.*;
import java.sql.SQLException;
import sun.jdbc.rowset.WebRowSet;

public class WebRS {

  //Constant representing the XML file
  private static String WRS_FILE_LOC ="wrs.xml";

  public final static void main(String[] args) throws Exception {

    try {
      //Instantiate a WebRowSet object
      WebRowSet wrs = new WebRowSet();

      //Load driver and set connection parameters
      Class.forName("oracle.jdbc.driver.OracleDriver");
      wrs.setUrl("jdbc:oracle:thin:@localhost:1521:ORCL");
      wrs.setUsername("toddt");
      wrs.setPassword("mypwd");

      //Configure command and execute
      System.out.println("Connecting to data source and " +
                         "generating XML document.");
      String sql = "SELECT ssn, name FROM Employees";
      wrs.setCommand(sql);
```

```
      wrs.execute();

      //Write XML out to file
      System.out.println("Writing XML to file: " + WRS_FILE_LOC);
      FileWriter fw = new  FileWriter(WRS_FILE_LOC);
      wrs.writeXml(fw);
      fw.close();
      wrs.close();
      System.out.println("Finished writing XML file.");

    }catch (SQLException se){
      se.printStackTrace();
    }catch (Exception ex) {
      ex.printStackTrace();
    }

    System.out.println("Goodbye!");
  }//end main()
}//end WebRS class
```

The output from Listing 16−3 is as follows:

```
Connecting to data source and generating XML document.
Writing XML to file: wrs.xml
Finished writing XML file.
Goodbye!
```

From the example you can see that the processes of setting the connection and command properties remain the same as in the other rowset example. In reality, the only difference between this example and Listing 16−2 is that I create an XML document instead of serializing the RowSet object into a file. To do so I use the writeXml() method with a FileWriter object as a parameter. This creates the XML document wrs.xml. For your convenience, Listing 16−4 shows the XML document produced by the application.

Listing 16−4: wrs.xml

```
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE RowSet PUBLIC ‘−//Sun Microsystems, Inc.
//DTD RowSet//EN’ ‘http://java.sun.com/j2ee/dtds/RowSet.dtd’>

<RowSet>
  <properties>
    <command>SELECT ssn, name FROM Employees</command>
    <concurrency>1007</concurrency>
    <datasource><null/></datasource>
    <escape−processing>true</escape−processing>
    <fetch−direction>0</fetch−direction>
    <fetch−size>0</fetch−size>
    <isolation−level>2</isolation−level>
    <key−columns>
    </key−columns>
    <map></map>
    <max−field−size>0</max−field−size>
    <max−rows>0</max−rows>
    <query−timeout>0</query−timeout>
    <read−only>true</read−only>
    <rowset−type>1004</rowset−type>
    <show−deleted>false</show−deleted>
```

```
  <table-name><null/></table-name>
  <url>jdbc:oracle:thin:@localhost:1521:ORCL</url>
</properties>
<metadata>
  <column-count>2</column-count>
  <column-definition>
    <column-index>1</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>false</case-sensitive>
    <currency>true</currency>
    <nullable>0</nullable>
    <signed>true</signed>
    <searchable>true</searchable>
    <column-display-size>21</column-display-size>
    <column-label>SSN</column-label>
    <column-name>SSN</column-name>
    <schema-name></schema-name>
    <column-precision>9</column-precision>
    <column-scale>0</column-scale>
    <table-name></table-name>
    <catalog-name></catalog-name>
    <column-type>2</column-type>
    <column-type-name>NUMBER</column-type-name>
  </column-definition>
  <column-definition>
    <column-index>2</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>true</case-sensitive>
    <currency>false</currency>
    <nullable>1</nullable>
    <signed>true</signed>
    <searchable>true</searchable>
    <column-display-size>20</column-display-size>
    <column-label>NAME</column-label>
    <column-name>NAME</column-name>
    <schema-name></schema-name>
    <column-precision>20</column-precision>
    <column-scale>0</column-scale>
    <table-name></table-name>
    <catalog-name></catalog-name>
    <column-type>12</column-type>
    <column-type-name>VARCHAR2</column-type-name>
  </column-definition>
</metadata>
<data>
  <row>

    <col>111111111</col>
    <col>Todd</col>
  </row>
  <row>
    <col>419876541</col>
    <col>Larry</col>
  </row>
  <row>
    <col>312654987</col>
    <col>Lori</col>
  </row>
  <row>
    <col>123456789</col>
    <col>Jimmy</col>
```

```
    </row>
    <row>
      <col>987654321</col>
      <col>John</col>
    </row>
  </data>
</RowSet>
```

Notice that the XML document produced by the WebRowSet class contains not only the query data, but metadata for the WebRowSet object that generated the XML document. This feature allows different WebRowSet objects to clone the original object by using the XML document.

For example, consider the <properties> element, whose child elements provide information about the query used to populate the rowset, the URL, and the concurrency levels. The <metadata> element provides information about the columns contained in the rowset, such as column name and data type. The <data> element contains the row data that match the SELECT statement criteria.

Because the XML document provides a self−contained description of a WebRowSet object, you can use it to populate a new WebRowSet object, as the following snippet demonstrates:

```
//Define the xml file name
String xmlFile = "wrs.xml";

//Instantiate a WebRowSet
WebRowSet wrsIn = new WebRowSet();

//Create a file reader using the xml file name
FileReader fin = new FileReader(xmlFile);

//Populate the WebRowSet object
wrsIn.readXml(fin);
```

As you can see, populating a WebRowSet object using an XML requires little effort. You just pass a FileReader object associated with the target XML file in as a parameter to the readXml() method. Internally, the WebRowSet object validates the XML document against a DTD to ensure conformance. For the reference implementation, the DTD’s filename is rowset.dtd, and is included in the distribution. Once the WebRowSet object populates itself you can traverse or manipulate the data set as usual.

The WebRowSet class provides you with a convenient way to generate XML representing both the data matching a query and a WebRowSet object. You can use the XML document in many different ways, from sending it to a thin client to formatting it for display on a browser.

Summary

This chapter provided an overview of how to work with classes implementing the javax.sql.RowSet interface. In particular, I covered the details of Sun’s reference implementation of the RowSet interface, including the JdbcRowSet, CachedRowSet, and WebRowSet classes.

This chapter also discusses the following benefits and features associated with rowsets:

- CachedRowSet and WebRowSet objects do not require a connection to the data source to operate.
- You can serialize both the CachedRowSet and WebRowSet objects, and then distribute them to thin clients over a network.
- A RowSet object acts as a JavaBean, enabling you to raise events associated with certain activities.
- A RowSet object can open its own connection with the data source, freeing you from having to create the connection logic for it..
- A WebRowSet class defines a method, writeXml(), for generating XML documents describing the WebRowSet object and the data it contains.
- A WebRowSet class defines a method, readXml(), for populating the WebRowSet object using XML documents adhering to its DTD.

As you can see, a RowSet object provides a viable alternative to using the traditional JDBC ResultSet object when you need to simply retrieve and view query data. Furthermore, some Rowset objects also provide the added benefit of being able to function while disconnected from the data source, which gives you a lot of flexibility with regard to how you use the objects. Given the rise in distributed−application development, JDBC rowset technology should play an important role in designing solutions in the near future.

Chapter 17: Building Data−centric Web Applications

by Stephen Nobert

In This Chapter

- A brief overview of enterprise Web applications
- Using JDBC in servlets
- Using JDBC in JavaServer Pages
- Designing and implementing a JDBC JavaBean for use in a JSP application

The World Wide Web has dramatically transformed the Internet from a simple means of sharing information into an essential communication medium. Companies have contributed to this transformation by using the Web as a global marketplace for their wares and services.

The Web experience no longer consists of a small Web site with a few pages of plain static text. Now, a Web surfer uses applications that burst with a dizzying array of rich dynamic content, including multimedia. Additionally, many Web applications offer e−commerce in the form of catalogs and virtual shopping carts. Consequently, databases are now standard in the Web architecture. They are used to store content, multimedia, catalogs, shopping carts, user personalization, and so on.

As more people become Web−savvy and e−commerce evolves into a part of our daily life, it becomes more important that a Web application cater to more people more often. The need for scalable, efficient, and robust solutions to provide 24−7 service and support will be greater than ever.

Out of the pack of available solutions, Java — and J2EE in particular — has emerged as the clear choice. J2EE provides businesses with an answer to all their Web application needs. The three most prevalent components of J2EE that are used in Web applications are JDBC, servlets, and JavaServer Pages (JSP).

This chapter covers three different methods for designing enterprise Web application architectures, two−tier, three−tier and *n*−tier. I briefly discuss each. I also discuss servlets and JavaServer Pages and their roles in the enterprise Web application and J2EE, and how to use JDBC with them. I begin with a discussion of what an enterprise Web application is and of the two underlying architectures that you can use to design and implement it.

XRef All the code, graphics, and databases for all the examples in this chapter can be found in the Chapter 17 section of the book's Web site.

Reviewing Enterprise Web Applications

A Web application is a series of Web pages populated with dynamic and static content, tied together for a single purpose. Although it seems simple at first, its underpinnings are much more complex. To effectively design and implement an enterprise−class Web application, that is truly distributable, requires an understanding of Web architecture.

Regardless of what technology you use to build them, the underlying architecture of Web applications will be one of three kinds: two−tier, three−tier, and *n*−tier. This section provides a brief overview of all. I begin with

two−tier Web architecture since the three−tier and *n*−tier Web architectures evolved from it.

Two−tier Web architecture

The architecture behind the Web is a client−server model. The client−server architecture is a classic example of the two−tier architecture. This is a fairly simple architecture, composed of only two layers or tiers, as shown in Figure 17−1. Let's examine the flow of the request−response model over the HyperText Transfer Protocol (HTTP) using the figure as a map.

The client is a Web browser on a desktop, laptop, or other device such as a PDA, and it resides on the first tier. This tier may be referred to as the *client tier*. To request a Web page, the user enters a URL into the Web browser. The URL specifies the page to retrieve from the desired server, using the specified protocol, typically HTTP. The Web browser opens an HTTP connection over the Internet to the Web server listening on the specified server, which resides on the second tier (which I'll refer to as the *server tier*). Data stores are frequently on the second tier as well. In fact, they usually reside on the same machine as the Web server. The server takes the request, processes it accordingly, generates a response, and sends it back to the client over the same connection. Upon receiving the entire response, the browser closes the connection and renders the response as a page. In the case of multimedia and proprietary file formats, the browser may launch the appropriate plugin.

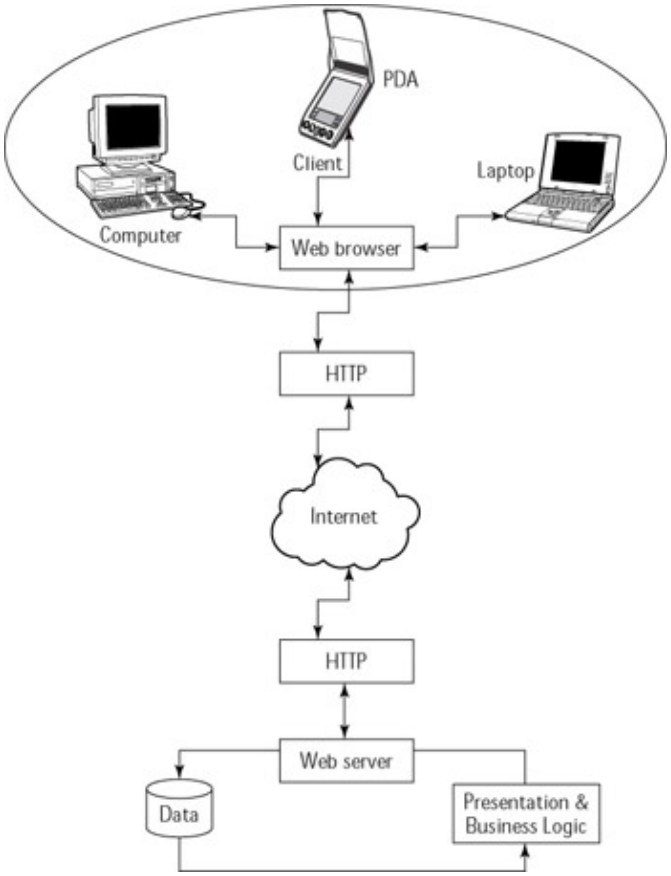

Figure 17−1: Example of the two−tier Web architecture

In many classic client−server applications, the business logic and presentation logic are commingled on the server. In other words, the same code that retrieves data from a database also determines how those data will be displayed on the client. Frequently, all this logic is tightly coupled with the database as well. The two−tier Web architecture is no different. A Web browser is considered a thin client, in that it does a few things very

well and does not tell the server how to process the request. It simply requests a URL from a server and renders the HTML contained in the response as a Web page. The server processes each client request, applying hard−coded business logic, and then sends an HTML page as the response. The format of the HTML page was hard−coded and generated by the same code that contained the business logic.

Commingling presentation and business logic results in maintenance and flexibility problems. If the two are tightly coupled, changing one will break the other. If the schema changes, then so must your presentation. Moreover, you may want to create separate views of the same data by providing a different presentation for each view, and this may prove to be very difficult or even impossible because of the tight coupling between presentation and business logic. Continuing to add functionality and enhancements results in an unmaintainable and inflexible monolith. To solve the commingling problem, you must keep the presentation logic separate from the business logic. This is actually harder than it sounds, as the following discussion shows.

MVC design pattern

When Xerox PARC went public with Smalltalk−80, the world was introduced to the *Model−View−Controller* (*MVC*) design pattern, shown in Figure 17−2. The *Model* represents the business−logic or data component. The *Controller* is the input presentation component, which is the user interface with which the user controls the model. The *View* is the output presentation component, which presents the user with a view of the model.

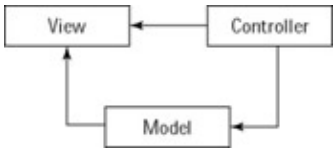

Figure 17−2: The Model−View−Controller (MVC) design pattern

The purpose of the design pattern is to isolate the user interface, or presentation logic, from the business logic. Although this design pattern was intended to build GUIs, its basic framework can be applied to many different programming tasks. The designers of the *Swing* GUI Toolkit Java extension adopted the MVC design pattern for many of its components. The same data can be presented differently with a JTable, JTree, or third−party charting component. These components provide different views of the same model.

This design pattern can effectively be applied to Web architectures as well. Applying MVC to the two−tier Web architecture results in a three−tier Web architecture.

Three−tier Web architecture

To effectively apply the MVC design, you have to separate the presentation logic from the business logic. Separating the components of the server tier into their own tiers results in a three−tier architecture, shown in Figure 17−3. The data stores are moved into a third tier, commonly referred to as the data tier. The server tier is now called the middle tier and it contains all the presentation and business logic. Note that the presentation and business logic are now separated.

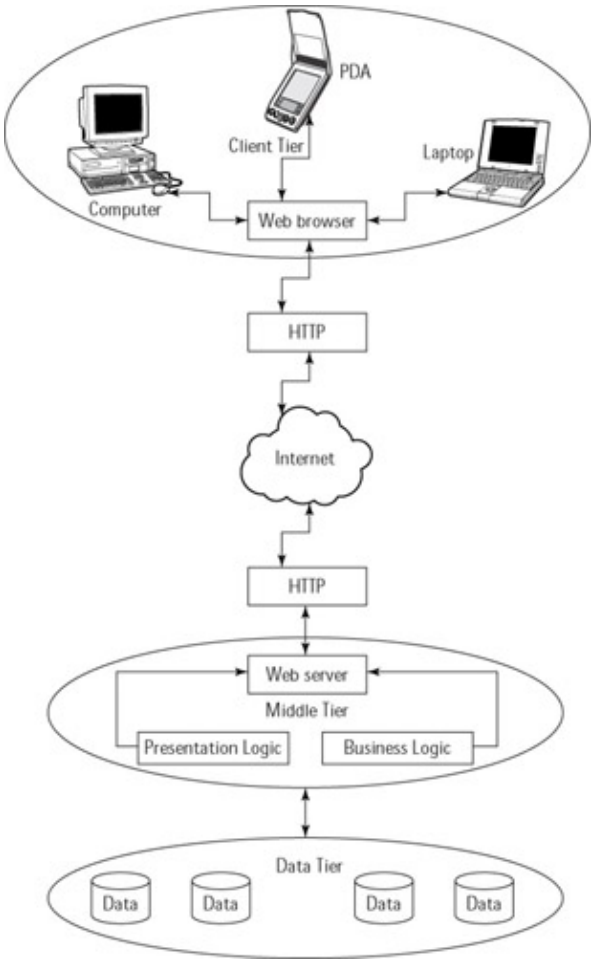

Figure 17−3: Example of the Three−tier Web architecture

The two−tier architecture works for retrieving static files but does not scale well enough for complex, distributed, enterprise−class applications involving dynamic content. The components that make up the server tier are usually tightly coupled — so much so that often you cannot remove or modify even one component without rebuilding the entire server.

Although the client remains thin, the middle tier remains thick. Making a Web application truly distributive involves separating all components into their own tier.

n−tier Web architecture

Separating all the components of the middle tier into their own tier results in a completely thin architecture known as *n*−tier, shown in Figure 17−4. The *n*−tier architecture is analogous to MVC in that each tier represents a separate component. Creating another presentation tier can result in an alternative View. Developing another input user interface results in a different Controller. All the while, the Model remains constant.

The beauty of the *n*−tier architecture is that it can be as divided as necessary. Providing a distributed architecture that effectively handles all traffic and is fault−tolerant may require that each component be a tier in itself.

The advantage of multiple tier Web architectures over two−tier architectures is that they scale well. In

addition, they are more efficient, robust, and maintainable. As I mentioned before, J2EE is the best choice for multiple−tier Web applications.

J2EE enterprise application framework

J2EE is an enterprise Java application for designing and implementing scalable, efficient, and robust Web applications using multiple−tier architectures. In a three−tier architecture, all the J2EE components reside in the middle tier. In the *n*−tier architecture, each component may reside in its own tier.

Figure 17−5 illustrates J2EE in an *n*−tier Web architecture. All the J2EE components relevant to my discussion of Web architectures — JDBC, servlets and JavaServer Pages (JSP) — are an integral part of this architecture. JDBC provides database connectivity for the middle tiers, servlets provide the business logic, and JSPs provide the presentation logic.

Java servlets and JavaServer Pages (JSP) are server−side Sun Java technologies used to develop high−performance and highly scalable enterprise Web applications. Servlets and JSP are Java−based, so they are inherently platform−independent. You can use both; each has its share of strengths and weaknesses.

In the following sections I discuss servlets and JSPs in more detail. I will also discuss how to use JDBC with both technologies.

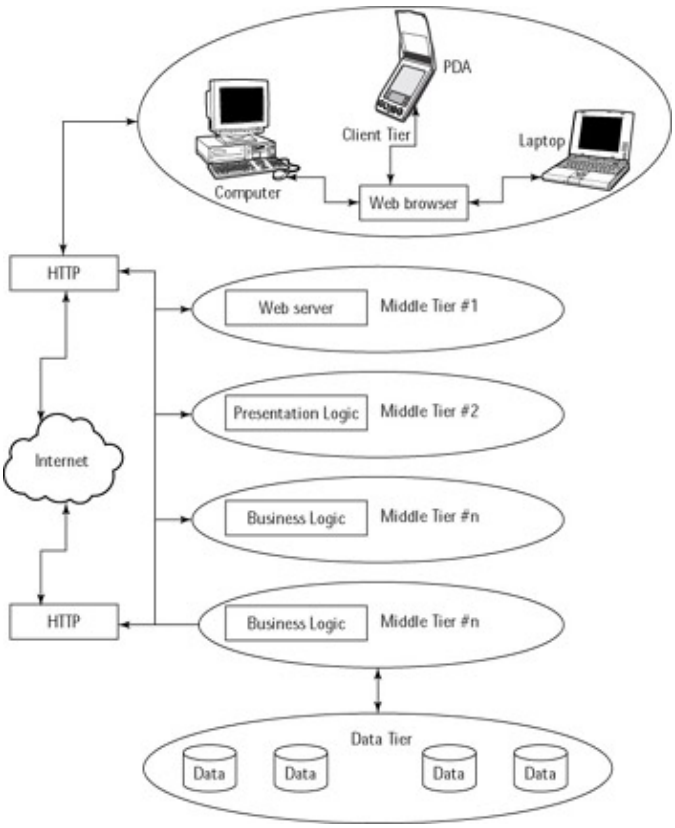

Figure 17−4: Example of the n−tier Web architecture

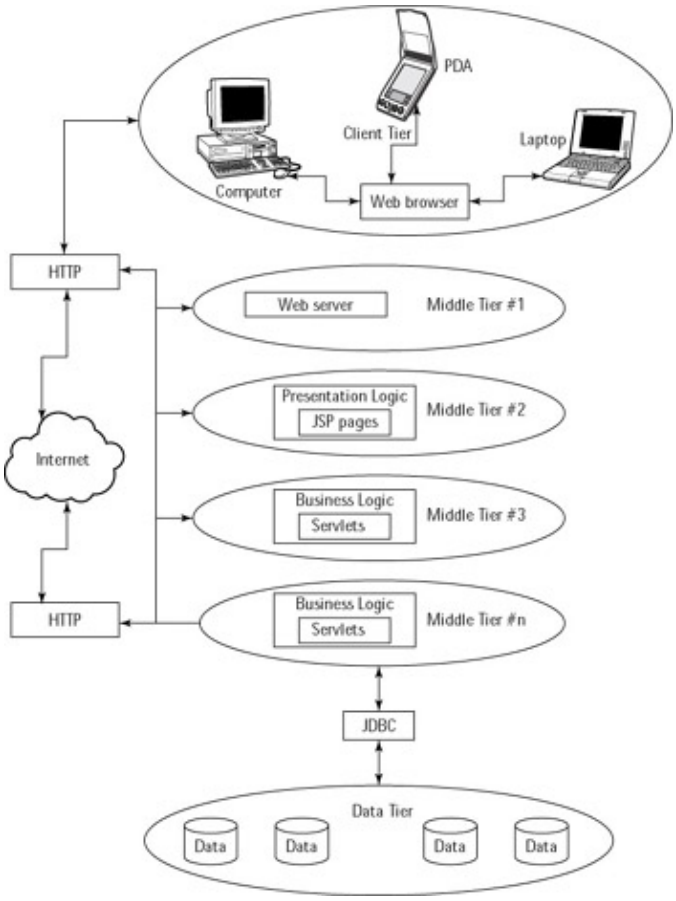

Figure 17−5: The J2EE architecture

Using JDBC with Servlets

Using JDBC with servlets is fairly straightforward. Servlets are a Java technology and have access to the entire Java API, including JDBC. Although there are some performance and design considerations to make, using JDBC with servlets is the same as using it in any other Java application.

In this section I describe what a servlet is and how to write one. I also present an example that uses JDBC.

Servlet overview

A servlet is a Java class that extends a Web server by performing server−side processing. Servlets are much more scalable, efficient, and robust than traditional server−side processing paradigms, such as the Common Gateway Interface (CGI).

Note The official Java Servlet Specification is part of the Java Community Process at Sun. You can find the specifications at http://java.sun.com/products/servlet. My discussion and code examples conform to the version 2.2 specification.

Servlets can be scaled from small workstations to a load−balanced and fault−tolerant server cluster. Since servlets are written in Java, they are inherently platform−independent. Furthermore, servlet containers are Web−server–independent. Consequently, servlets can be deployed on heterogeneous systems running a

myriad of operating systems, Web servers, and servlet containers.

Each servlet is loaded only once and each servlet request is run in a separate thread. This provides an obvious advantage over CGI, which requires the CGI mechanism to spawn a separate process for each request and loading the appropriate CGI. Furthermore, servlets are superior to CGI with respect to database access. Servlets can maintain a pool of database connections to multiplex among these threads providing optimal database−access performance. This eliminates the need to open and close a database connection for each request, which is necessary in CGI.

> XRef Chapter 14, “Using Data Sources and Connection Pooling,” provides more information about using the connection−pool features available in the javax.sql package.

Being written in Java affords servlets additional advantages. Java is a compiled language, and compiled code is generally faster than the interpreted code used in other server−side technologies, such as ASP. Furthermore, a servlet has the entire Java API at its disposal. This alone can make for extremely robust and portable code.

Now that you know what a servlet is, take a look at how one is constructed.

Constructing a JDBC servlet

Building a servlet is simple. You subclass from, or inherit, one of the Servlet classes provided by the Servlet Java Extension class library, and override the appropriate methods. Since serving up Web content via HTTP is so common, the class library features the Servlet class, HttpServlet, created for just this niche.

HttpServlet provides you with a set of overridable methods; one method for each HTTP operation. The default implementation for each method is to return errors or default values. For each HTTP operation you support, you must override the corresponding method with your own implementation to avoid returning these values. Generally speaking, if you override all the methods of a servlet based on HttpServlet, you have effectively created a rudimentary Web server.

Take a look at an example of creating a servlet. Listing 17−1 lists the code for the AllEmployeesServlet servlet. This servlet simply retrieves information for all employees from a database and formats them in an HTML table.

Listing 17−1: AllEmployeesServlet.java

```
// Java Data Access: JDBC, JNDI, and JAXP
// Chapter 17 − Web Applications and Data Access
// AllEmployeesServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class AllEmployeesServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head><title>All Employees</title></head>");
      out.println("<body>");
      out.println("<center><h1>All Employees</h1>");
```

```
Connection conn = null;
Statement stmt = null;
try {
   // Load the JDBC−ODBC Driver.
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   // Establish a connection to the database.
   conn = DriverManager.getConnection("jdbc:odbc:Employees");
   // Create a statement.
   stmt = conn.createStatement();
   // Retrieve the column and direction to sort by.
   // By default, sort by SSN in ascending order.
   String orderBy = request.getParameter("sort");
   if((orderBy == null) || orderBy.equals("")) {
      orderBy = "SSN";
   }
   String orderByDir = request.getParameter("sortdir");
   if((orderByDir == null) || orderByDir.equals("")) {
      orderByDir = "asc";
   }
   // Compose the query statement.
   String query = "SELECT Employees.SSN, Employees.Name, " +
                  "Employees.Salary, " +
                  "Employees.Hiredate, Location.Location " +
                  "FROM Location " +
                  "INNER JOIN Employees " +
                  "ON Location.Loc_Id = Employees.Loc_Id " +
                  "ORDER BY " + orderBy + " " + orderByDir +
                  ";";
   // Execute the query.
   ResultSet rs = stmt.executeQuery(query);
   out.println("<table border=\"1\">");
   out.print("<tr>");
   out.print("<th>SSN</th>");
   out.print("<th>Name</th>");
   out.print("<th>Salary</th>");
   out.print("<th>Hiredate</th>");
   out.print("<th>Location</th>");
   out.println("</tr>");
   // Add one row for each employee returned in the result
   // set.
   while(rs.next()) {
      long employeeSSN = rs.getLong("SSN");
      String employeeName = rs.getString("Name");
      long employeeSalary = rs.getLong("Salary");
      Date employeeHiredate = rs.getDate("Hiredate");
      String employeeLocation = rs.getString("Location");
      out.print("<tr>");
      out.print("<td align=\"right\">" + employeeSSN +
                "</td>");
      out.print("<td>" + employeeName + "</td>");
      out.print("<td align=\"right\">" + employeeSalary +
                "</td>");
      out.print("<td align=\"right\">" + employeeHiredate +
                "</td>");
      out.print("<td>" + employeeLocation + "</td>");
      out.println("</tr>");
   }
   out.println("</table>");
} catch(SQLException e) {

   out.println("<p>An error occured while retrieving " +
```

```
                    "all employees: " +
                    "<b>" + e.toString() + "</b></p>");
      } catch(ClassNotFoundException e) {
         throw(new ServletException(e.toString()));
      } finally {
         try {
            // Close the Statement.
            if(stmt != null) {
               stmt.close();
            }
            // Close the Connection.
            if(conn != null) {
               conn.close();
            }
         } catch(SQLException ex) {
         }
      }
      out.println("</center>");
      out.println("</body>");
      out.println("</html>");
      out.close();
   }
}
```

Building any servlet requires importing the servlet extension classes. The following lines illustrate this:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;
```

The first line imports all the basic classes necessary for developing any servlet. The second imports all classes specific to developing HTTP−based servlets. For my example I also need to import the I/O and JDBC class libraries, which is detailed in the last two lines.

Next you must subclass the appropriate Servlet class. The following line demonstrates how AllEmployeesServlet subclasses HttpServlet:

```
public class AllEmployeesHttpServlet extends HttpServlet {
```

Finally, you override all the methods corresponding to the HTTP operations you want to support in your servlet class body. Table 17−1 lists all the overridable

HttpServlet methods and their corresponding HTTP operations. For my example, I choose to support only the HTTP GET operation, which is the most common type of request made from Web browsers. The method that corresponds to this operation is doGet(). This method is called for each HTTP GET request received for this servlet. Therefore, I only override the corresponding doGet() method.

Table 17−1: Overridable HttpServlet Methods

Method	HTTP Operation	Description
doGet()	HTTP GET	Retrieves the requested resource (file or process result) specified in the URL; idempotent request.
doPost()	HTTP POST	Retrieves the requested resource (file or process result) specified in the request; non−idempotent request.
doPut()	HTTP PUT	Puts a resource on the Web server.
doDelete()	HTTP DELETE	Deletes the specified resource.
doHead()	HTTP HEAD	Same as GET, except that only the message headers and not the body are returned.
doOptions()	HTTP OPTIONS	Queries the Web server's capabilities or information about a resource on the server.
doTrace()	HTTP TRACE	Loopback; returns the request.

The doGet() method has two arguments, request and response. The request argument is an HttpServletRequest instance that provides access to the client's request, such as HTTP−specific information as well as the URL parameters and values. It also contains the name of the content requested by the client. The response argument is an HttpServletResponse instance that provides the necessary means of responding to the corresponding request. In addition to the content returned, it provides the additional information necessary for the client to correctly interpret and process the response.

Ordinarily the doGet() method returns the content specified in the request. This could be an HTML page, a graphic file, or some other piece of content. In this example I am returning an HTML page that I dynamically generate. Before I can start generating the HTML, I must first indicate the type of content I will be sending back. (Content type is also commonly referred to as a MIME type.) Since I am generating straight HTML, the MIME body I will be sending back will be of type text/html, as shown in the following snippet:

```
response.setContentType("text/html");
```

MIME and Binary Content

The Multipart Internet Mail Extension (MIME) was originally developed to provide a standard mechanism to send binary files over text−only mail systems. Later it was incorporated into HTTP for the same purpose, but it has been extended to identify all content. A Web server identifies the content it is sending to the client by specifying the content's MIME type. If it cannot ascertain the content type it is sending, it defaults to text/plain. Most MIME types are based on file extensions. Therefore, a Web server can easily determine the MIME type for a piece of content, simply by looking at the file's extension.

It has become common practice to store the content for an entire Web site in a database. Frequently, this content is binary (images and multimedia, for example). Unfortunately, you don't have the luxury of file extensions when using BLOB data. To easily identify each piece of content you store in a database, I highly recommend instituting a mechanism to tag each piece with its corresponding MIME type. This helps eliminate client−side confusion over the type of a piece of content.

Table 17−2 shows some common binary MIME types.

Table 17−2: Common Binary Mime Types

Content Type	MIME Type
Adobe Acrobat PDF file	application/pdf
AVI	video/avi, video/msvideo, or video/x−msvideo
GIF	image/gif
JPEG	image/jpeg
MP3	audio/mp3
MPEG	video/mpeg or video/x−mpeg
MOV (QuickTime)	video/quicktime
WAV	audio/wav or audio/x−wav
ZIP	application/x−compressed

To send the content back to the client I first obtain a reference to the output mechanism provided for just this purpose. The content type I set in the preceding code determines the output mechanism I need to use. In my case I need to use a PrintWriter. I call getWriter() to obtain the PrintWriter. From this point on, I simply send HTML content by calling either the print() or println() method. The following code snippet shows how I obtain the PrintWriter:

```
PrintWriter out = response.getWriter();
```

> Tip To send a binary file, specify the appropriate content type and use the ServletOutputStream instance returned from response. getOutputStream() to write the content.

Regardless of the number of employees, I have to create the page using static, boilerplate HTML. The following code snippets show how I send the beginning HTML:

```
out.println("<html>");
out.println("<head><title>All Employees</title></head>");
out.println("<body>");
out.println("<center><h1>All Employees</h1>");
```

. . . and the ending HTML:

```
out.println("</center>");
out.println("</body>");
out.println("</html>");
```

Between these two snippets of boilerplate code, I interact with the database and build the HTML table. The table declaration and header row remain static. The table rows are dynamic and are based on the number of rows, and the contents of each row are dependent on the data I receive as a result of the query. (I will discuss how I compose and execute the query in a moment.)

Before querying the database I must perform several JDBC operations. First, I load the JDBC−ODBC driver. Second, I establish a physical connection to the database. Third, I create a JDBC SQL Statement object so I can submit my SQL statement. I am only using static queries, so I do not need a PreparedStatement object. The following code snippet shows the corresponding code for these operations:

```
// Load the JDBC−ODBC Driver.
```

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// Establish a connection to the database.
conn = DriverManager.getConnection("jdbc:odbc:Employees");


// Create a statement.
stmt = conn.createStatement();
```

Now that I have established a connection to the database, I can query it. First I compose a query to retrieve all the employees and their respective attributes. The following code snippet demonstrates how I compose the SELECT statement:

```
// Compose the query statement.
String query = "SELECT Employees.SSN, Employees.Name, " +
               "Employees.Salary, " +
               "Employees.Hiredate, Location.Location " +
               "FROM Location " +
               "INNER JOIN Employees " +
               "ON Location.Loc_Id = Employees.Loc_Id " +
               "ORDER BY " + orderBy + " " + orderByDir +
               ";";
```

Note that I am creating an INNER JOIN between the Employees and Location tables so that I can place the Location name, instead of the Loc_Id, in the resulting HTML table.

Now I query the database by calling the Statement.executeQuery() method. I pass the String variable query which represents the query statement as the method argument. The method returns a ResultSet, which contains all the rows that match my query's criteria:

```
// Execute the query.
ResultSet rs = stmt.executeQuery(query);
```

I build each table row by iterating through the ResultSet. For each row I encounter in the ResultSet, I retrieve the SSN, Name, Salary, Hiredate, and Location column values. I use these values as table−cell data for the corresponding table columns in the table row. The following code snippet illustrates how I do this:

```
// Add one row each employee returned in the result set.
while(rs.next()) {
   long employeeSSN = rs.getLong("SSN");
   String employeeName = rs.getString("Name");
   long employeeSalary = rs.getLong("Salary");
   Date employeeHiredate = rs.getDate("Hiredate");
   String employeeLocation = rs.getString("Location");
   out.print("<tr>");
   out.print("<td align=\"right\">" + employeeSSN +
             "</td>");
   out.print("<td>" + employeeName + "</td>");
   out.print("<td align=\"right\">" + employeeSalary +
             "</td>");

   out.print("<td align=\"right\">" + employeeHiredate +
             "</td>");
   out.print("<td>" + employeeLocation + "</td>");
   out.println("</tr>");
}
```

I finish the table and close the Statement and Connection variables, stmt and conn, before the ending boilerplate HTML. Although the stmt and conn variables will go out of scope and eventually get closed by the VM, it is customary to close them as soon as you are finished using them. Besides, if you rely on the VM to close them, you may very well exhaust all of your allowable simultaneous database connections. Finally, I indicate that I am finished sending content by closing the PrintWriter with the close() method. Closing flushes any content in the PrintWriter's buffer.

Server deployment

A servlet class is like any other Java class; it must be compiled before it can be used. To compile your servlet, your CLASSPATH environment variable must include servlet.jar, the Java Servlet Extension class library. This file is packaged with all servlet containers. The compileservlets.bat file on the book's Web site illustrates how to compile the example servlets under Tomcat and JRun.

Once you successfully compile your servlet, you must deploy it to your servlet container. Place the resulting class file, AllEmployeesServlet.class, in the ROOT or default servlet context of your servlet container. (For Tomcat, the ROOT servlet context is located in *TOMCAT_HOME*/Webapps/ROOT/WEB−INF/classes, where *TOMCAT_HOME* is the root directory of your Tomcat installation. For JRun, the default servlet context is located in *JRUN_HOME*/servers/default/default−app/WEB−INF/classes, where *JRUN_HOME* is the root directory of your JRun installation.) If you have a different servlet container or want more information about setting up servlet contexts and deploying servlets, please consult your servlet container's documentation.

Servlet Containers

A servlet container is the environment necessary to run and support servlets. It can be either an extension or an integral part of a Web−server. All servlet responses and requests are handled and routed by the servlet container. The servlet container is also responsible for managing servlet contexts.

There are several open−source and commercial servlet containers, both stand−alone containers and those that come as part of a Java application server or J2EE platform. (A current list of third−party support for servlet technology is available at http://java.sun.com/products/servlet/industry.html.) Two of the more popular containers are Tomcat and JRun.

Tomcat is open−source and free. Originally started at Sun, Tomcat is now developed and maintained by the Apache Software Foundation as part of the Jakarta Project. Tomcat is considered the reference implementation for both Servlet and JavaServer Pages technologies. Sun integrates and distributes Tomcat as an integral part of its J2EE reference platform. For more information on Tomcat, go to http://jakarta.apache.org/.

JRun is a commercial J2EE application server from Macromedia. It provides Web−based administration as well as companion tools such as JRun Studio. For more information on JRun, go to http://www.macromedia.com/.

To run the servlet, enter the URL http://localhost:8080/servlet/AllEmployeesServlet in your Web browser. Figure 17−6 shows a screen capture of the resulting Web page rendered by a browser. The :8080 portion of the URL tells the Web browser the TCP/IP port on which to connect to the servlet container; replace the 8080 with the appropriate port if necessary for your servlet container's configuration. In this example, I am using

Tomcat and, by default, it services requests on port 8080. For JRun, the default port for the Default Server is 8100.

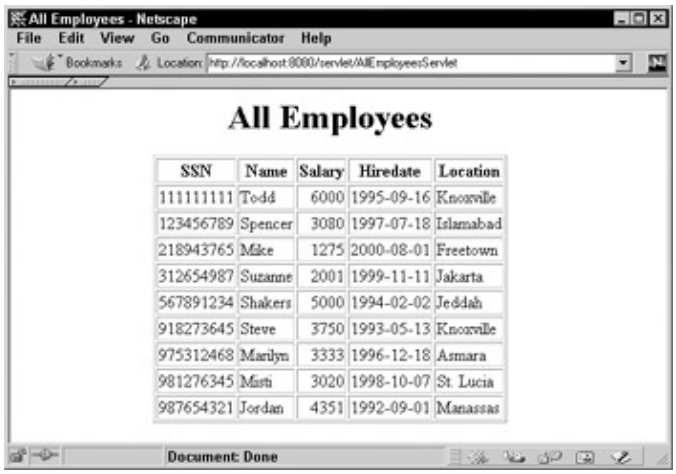

Figure 17−6: The servlet−based All Employees Web page

Design considerations

Although AllEmployeesServlet produces the desired Web page, it does present a performance problem while doing so: The database connection is opened and closed for each HTTP GET request. As you are well aware, they are very expensive operations in terms of latency and network bandwidth, and while it may be acceptable for a site that receives a handful of hits per day, it is totally unacceptable for one that receives thousands or millions. The entire site and its internal network would slow to a crawl as a result of the increased, and unnecessary, amount of handshaking, bandwidth usage, and CPU usage.

I slightly modify the servlet to overcome this inefficiency. The resulting servlet, AllEmployeesServlet2, loads the driver and opens a database connection only once when the servlet instance is first created, and reuses the Connection for each query. I do this by overriding two standard servlet methods: init() and destroy(). When the servlet is first instantiated, its init() method is called only once. All servlet− specific setup code, such as opening files or connections, is placed in this method.

I move the code to load the driver and open the connection to the init() method. I also promote the local variable, conn, to an instance variable so that all methods have access to the established Connection instance. The following code snippet reflects the new servlet:

```
public class AllEmployeesServlet2 extends HttpServlet {
   private static final String jdbcDriverClass =
      "sun.jdbc.odbc.JdbcOdbcDriver";
   private static final String jdbcDatabaseURL =
      "jdbc:odbc:Employees";
   private Connection conn;
   public void init() throws ServletException {
      try {
         // Load the JDBC−ODBC Driver.
         Class.forName(jdbcDriverClass);
         // Establish a connection to the database.
         conn = DriverManager.getConnection(jdbcDatabaseURL);
      } catch(Exception e) {
         throw(new ServletException(e.getMessage()));
      }
   }
```

Servlet Contexts

A single Web application can be composed of many components, including servlets, JSP pages, Web pages (HTML, XHTML, XML), images (JPEG, GIF, PNG), multimedia files (WAV, MP3, MOV), configuration files, and so on. All of these files can be grouped into a single servlet context to represent this Web application. References to other files within the context are relative to the context itself. Therefore, if the path to the context changes, the file references within the context do not. This context can then be packaged and easily moved from one servlet container to the next. A single servlet container can run multiple applications, maintaining each context's sovereignty and state.

Note that I do not create the Statement instance in the init() method. Although this would make sense, the Statement interface has a limitation that prevents it from effectively being used with multi−threaded servlets: At any given moment, only one ResultSet instance can be opened per Statement instance. A call to executeQuery() or executeUpdate() closes the previous ResultSet (if it was open) and returns a new ResultSet.

Note The JDBC 3.0 Specification allows multiple opened ResultSets instances per Statement instance.

To protect a ResultSet in one thread from being closed by another thread calling one of these methods, I would have to wrap the code sequence that uses the Statement in a synchronized block, using the Statement object as the monitor. One effect of doing this is that simultaneous database actions are prevented from using the same Connection instance. Although this arrangement is still a vast improvement over the original, it would be better if I could perform simultaneous database actions using the same Connection instance. Therefore, I continue to create a Statement in each query.

The destroy() method is the opposite of the init() method. All servlet−specific housekeeping tasks, such as closing files and connections, are placed in this method. This method is also called only once, when the servlet container unloads the servlet. A servlet is unloaded when the servlet container is shutting down or is updated. I close the database connection I opened in the init() method, in this method. The following code snippet shows the overridden destroy() method:

```
public void destroy(){
   try {
      // Close the Connection.
      if(conn != null) {
         conn.close();
      }
   } catch(SQLException ex) {
   }
}
```

The code for AllEmployeesServlet2 is available in its entirety from the book's Web site.

This example is somewhat trivial. It is comprised of a single page and does not provide any user interaction. In the JSP section, I take this example one step further and present a small application that comprises multiple pages and provides user interaction.

After examining the servlet code you might come to the conclusion that a more complicated page (such as the home pages of some of the more popular news and sports Web sites) can be very frustrating and tedious to generate using only servlets. You would be correct. Servlets have other disadvantages as well.

The major disadvantage of servlets is that they are not Web author−friendly. A Web author must know Java programming to build a servlet. Servlets provide no conveniences to make authoring easier. Another disadvantage is that servlets provide no easy mechanism for separating presentation and business logic. The need to commingle presentation and business logic results in code that is neither easy to read nor easy to debug. This commingling violates the necessary separation I described earlier. Finally, servlets must be compiled prior to deployment.

Sun listened to the Web authors' cries and used their suggestions as the premise for the more author−friendly JavaServer Pages (JSP pages).

Using JDBC with JavaServer Pages

Using JDBC with JSP is as straightforward as using it with servlets. JSP pages are built on servlet technology, so using JDBC is very similar. And as with using JDBC with servlets, using JDBC with JSP has advantages and disadvantages.

In these sections I describe what JSP pages are and how to write one. I also present a multiple−page example application using JDBC.

JSP overview

JavaServer Pages (JSP) is a specification for a platform−independent mechanism with which you can build Web pages using both dynamically generated and static content. This server−side Java technology is an extension of the Java Servlet technology I described in the previous section. Like servlets, JSP is an integral part of J2EE. Combining JSP and servlets provides the ideal platform for building highly scalable enterprise Web applications.

Note The official JavaServer Pages Specification is part of the Java Community Process at Sun. You can find the specification at http://java.sun.com/products/jsp. My discussion and code examples conform to the version 1.1 specification.

While servlets appeal more to programmers, JSP is intended for Web authors. The beauty of JSP is that Web authors and programmers can develop a Web application together. While programmers are busy developing business logic and encapsulating it into JavaBeans and custom tag libraries for use in JSP, Web authors can concentrate on presentation. (This recalls my earlier discussion of the separation of business logic and presentation.)

JSP is dependent on servlet technology, because each JSP page is converted to and compiled as a servlet. So when you request a JSP page, you are indirectly requesting that a servlet be run. The JSP container is responsible for this servlet and compilation. It is also responsible for keeping the servlet up to date with the corresponding JSP page. If a JSP page has been updated since the last request, the new JSP page is converted to and compiled into a servlet. Additionally, the JSP container is responsible for processing the request and response. The JSP container also provides an API into the underlying servlet called the JSP implementation class, but an in−depth discussion on the subject is beyond the scope of this book.

Constructing a JSP page

JSP pages are composed of both dynamic and static content. The dynamic portion is inserts content based on each request. JSP provides authors with an arsenal of elements and objects with which to handle each request and build each page based on that request.

Table 17−3 lists JSP page components. The JSP *directives*, *actions*, and *scripting* elements can dynamically generate content based on the request. Implicit objects are available for use in *scriptlets*. The static content is called *template text*, and it never changes, regardless of the request.

Table 17−3: JSP Page Components

Component	Description
directive	Specifies page behavior and attributes.
action	Actions performed for each request.
scripting	Server−side scripting, such as Java or JavaScript.
implicit objects	Object−reference conveniences to aid the author.
template text	Static content that does not change regardless of request; frequently HTML.

In the simplest of terms, building a JSP page is simply creating a page in HTML and strategically placing JSP tags and scripts to dynamically generate and insert content.

First, I discuss some of the JSP elements and then briefly touch on template text. Next, I briefly present the implicit objects commonly used in scripting elements. Finally, I present some design considerations for you to keep in mind when constructing your own JSP pages.

Directives

JSP directives are elements that send hints about the page to the JSP container. It is easy to differentiate JSP directive elements from other JSP elements, because they have the format <%@...%>. These elements only apply to the page they are defined in. Table 17−4 lists and briefly describes all the JSP directive elements.

Table 17−4: JSP Directive Elements

Element	Description
<%@ page ... %>	Specifies attributes for the page.
<%@ include ...%>	One of two distinct mechanisms for including another file; this one is intended for static files.
<%@ taglib ... %>	Specifies the custom tag library to use.

Actions

JSP action elements define actions that are performed at runtime for each request. These elements have the format <jsp:.../>. Table 17−5 lists and briefly describes all the JSP standard action elements.

Table 17−5: JSP Standard Action Elements

Element	Description
<jsp:usebean ... />	Associates an identifier to a JavaBean instance.
<jsp:getProperty ... />	Retrieves a value for a bean property.
<jsp:setProperty ... />	Sets the value for a bean property.
<jsp:include ... />	Dynamically includes a JSP page. The second include mechanism.
<jsp:forward ... />	Forwards the request to another page in the same context.
<jsp:plugin ... />	Inserts client−browser–specific HTML (OBJECT and EMBED) that downloads the Java plugin software to execute the specified Java Applet or JavaBean component.
<jsp:param ... />	Adds parameters to the jsp:include, jsp:forward, and jsp:plugin action elements.

Scripting elements

JSP scripting elements are elements wherein blocks of scripting, normally Java, code are inserted into the page. These elements have the format <%...%> and can be differentiated from directive elements because they don’t have the leading @. Table 17−6 lists and briefly describes all the JSP scripting elements.

Table 17−6: JSP Scripting Elements

Element	Name	Description
<%! %>	Declaration	Declares instance variables and methods for the page.
<%= %>	Expression	Inserts the textual representation of the expression result.
<% %>	Scriptlet	Embeds scripting code.

I am going to digress a little more about scripting elements, and scriptlets in particular. HTML authors may use client−side scripting languages such as JavaScript and JScript. Scriptlets differ from these languages in many ways. They are strictly server−side; only the server executes them. This makes them client−independent, so it doesn’t matter what browser you use to access pages containing scriptlets. Unlike interpreted client−side scripting languages, scriptlets are translated and compiled. In fact, each JSP page is compiled into a corresponding servlet upon its first request or when the page changes. This results in much faster execution. Though most scriptlets are written in Java, you are not limited to using Java.

Of course, which languages are supported is dependent on the underlying servlet container you use. Most servlet containers support only Java as a scriptlet language. In fact, if you don’t explicitly specify the

language, it defaults to Java. Some servlet containers, such as JRun, allow JavaScript as a scriptlet language as well. In this case, JavaScript is compiled rather than interpreted. For this discussion I use Java as my scriptlet language.

JSP is truly the best of both client and server−side worlds. It provides the speed and simplicity of server−side execution yet enables you to easily include client−side functionality and customization. If you want to include some JavaScript in your page, you have two choices. The first is to include or dynamically generate the JavaScript in the scriptlets. The second, and more common, is to place the majority of the client−side code in the template text. Choosing between the two is dependent on your requirements.

Implicit objects

The JSP container provides you with several objects to simplify authoring. The objects are automatically made available to all scriptlets. Table 17−7 lists and briefly describes all the JSP intrinsic objects. (In the examples to follow, I frequently use the request and response objects.)

Table 17−7: JSP Implicit Objects

Element	Description
request	Encapsulates the attributes of a client's request.
response	Encapsulates the response back to the client.
out	Buffers the response and writes it back to the client. The size of the buffer is specified in the buffer attribute of the page directive.
application	Identifies the ServletContext in which the JSP/servlet is running.
page	Refers to the JSP class itself. Access it using this.
session	Identifies the HTTPSession associated with this request.
config	Identifies to the servlet configuration.
exception	Identifies the exception caught in a JSP error page.

Template text

Anything the JSP engine does not recognize as a JSP element is deemed template text, and is passed through to the client. It is called a *template* because it normally wraps the scripting elements in text, much like a template or boilerplate in publishing. If you are using well−formed HTML for your template text, you will have an opening html tag (<html>) at the top of the page and a closing html tag (</html>) at the bottom. You may put one or more scripting elements in between these two tags.

JSP is independent of any language you choose for your template text. For all intents and purposes, it doesn't matter what you put in the template text. Most JSP authors use HTML, but you can use XHTML, XML, and WML just as easily. (I use HTML for my template text in the following examples.)

Caution Unless your browser is very robust, remember to use the same language throughout the entire page that you use in your template text. Mixing languages can confuse your browser and result in unrenderable pages.

Most HTML authors are not programmers, and vice versa. HTML authors specialize in presentation while programmers specialize in implementing business logic. A well−designed JSP page requires the synergy of both.

Using JDBC in JSP pages

The example application I am presenting consists of seven JSP pages. For brevity's sake, I didn't spend a lot of time on aesthetics, but I do cover most of the basics you need to get started creating your own JSP applications using JDBC.

> Note You must have a servlet container such as Tomcat or JRun installed. Place all the JSP pages and graphics files in the default or root context of the servlet container; all class files must go into the appropriate class directory for their context. Consult your servlet container's documentation for the proper server and context configuration and location for these files.

This application is a prime example of an intranet application that might be used by human resources and management personnel. All the basics of using JDBC with JSP pages are covered. Naturally, you can expand the functionality to include features such as user authentication. Adding that feature would require more pages, such as a login page, and would affect the views or presentation of data based on authorization. Just as in academia texts, some things are left as an exercise for the reader.

Each page in the application performs a distinct action. These actions include viewing all employees and their respective attributes, editing a specific employee's attributes, deleting an existing employee, and adding a new employee. The other three pages provide confirmation of the latter three actions.

Without further ado, let's delve right into the application. The All Employees sample JSP page is the entry point into the application.

Example: All Employees

Using a Web browser, enter the URL http://localhost:8080/AllEmployees.jsp. Figure 17−7 shows the page generated when you request AllEmployees.jsp. It displays in a table all employees and each one's SSN, Name, Salary, Hiredate, and Location. By clicking on one of the up or down arrows adjacent to a column heading, you can sort the table by that column in either ascending or descending order. In addition, this page enables you to edit an existing employee's data, or delete an employee's data altogether. Also, you can add a new employee to the database by clicking the Add Employee link.

Figure 17−7: A JSP−based All Employees Web page

The JSP and HTML template code to generate this page is fairly straightforward. Listing 17−2 is the entire code listing for this page. In addition to the basic HTML template code, it includes several scriptlet sections that perform the majority of the work. To cater to a broader audience, my template code complies to HTML v3.2. Therefore, you will not see any reference to CSS or HTML v3.2 or earlier tags that are deprecated in HTML v4.*x*, such as the tag.

Listing 17−2: AllEmployees.jsp (All Employees JSP page)

```
<%--Java Data Access: JDBC, JNDI, and JAXP--%>
<%--Chapter 17 – Web Applications and Data Access--%>
<%--AllEmployees.jsp--%>
<html>
<head><title>All Employees</title>
</head>
<%@ page language="java" import="java.sql.*" %>
<body>
<center>
<h1>All Employees</h1>

<%--Build a table of all employees to choose from--%>
<%
Statement stmt = null;
Connection conn = null;
try {
   // Load the Driver.
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   // Establish a connection to the database.
   conn = DriverManager.getConnection("jdbc:odbc:Employees");
   // Create a query statement and execute it.
   stmt = conn.createStatement();
   // Retrieve the column and direction to sort by.
   // By default, sort by SSN in ascending order.
   String orderBy = request.getParameter("sort");
   if((orderBy == null) || orderBy.equals("")) {
      orderBy = "SSN";
   }
   String orderByDir = request.getParameter("sortdir");
   if((orderByDir == null) || orderByDir.equals("")) {
      orderByDir = "asc";
   }
```

```
String query = "SELECT Employees.SSN, Employees.Name, " +
               "Employees.Salary, " +
               "Employees.Hiredate, Location.Location " +
               "FROM Location " +
               "INNER JOIN Employees " +
               "ON Location.Loc_Id = Employees.Loc_Id " +
               "ORDER BY " + orderBy + " " + orderByDir +
               ";";
ResultSet rs = stmt.executeQuery(query);
%>
<table border="1">
<tr>
<th>
   <%--Workaround for Navigator so that the arrow aligns
    with the header--%>
    
   <a href="AllEmployees.jsp?sort=SSN&sortdir=asc">
      <img border=0 align="absmiddle" src="up12.gif"
       alt="Sort by SSN in ascending order">
   </a>
   SSN
   <a href="AllEmployees.jsp?sort=SSN&sortdir=desc">
      <img border=0 align="absmiddle" src="down12.gif"

       alt="Sort by SSN in descending order">
   </a>
</th>
<th>
   <%--Workaround for Navigator so that the arrow aligns
    with the header--%>
    
   <a href="AllEmployees.jsp?sort=Name&sortdir=asc">
      <img border=0 align="absmiddle" src="up12.gif"
       alt="Sort by Name in ascending order">
   </a>
   Name
   <a href="AllEmployees.jsp?sort=Name&sortdir=desc">
      <img border=0 align="absmiddle" src="down12.gif"
       alt="Sort by Name in descending order">
   </a>
</th>
<th>
   <%--Workaround for Navigator so that the arrow aligns
    with the header--%>
    
   <a href="AllEmployees.jsp?sort=Salary&sortdir=asc">
      <img border=0 align="absmiddle" src="up12.gif"
       alt="Sort by Salary in ascending order">
   </a>
   Salary
   <a href="AllEmployees.jsp?sort=Salary&sortdir=desc">
      <img border=0 align="absmiddle" src="down12.gif"
       alt="Sort by Salary in descending order">
   </a>
</th>
<th>
   <%--Workaround for Navigator so that the arrow aligns
    with the header--%>
    
   <a href="AllEmployees.jsp?sort=Hiredate&sortdir=asc">
      <img border=0 align="absmiddle" src="up12.gif"
```

```
            alt="Sort by Hiredate in ascending order">
         </a>
         Hiredate
         <a href="AllEmployees.jsp?sort=Hiredate&sortdir=desc">
            <img border=0 align="absmiddle" src="down12.gif"
             alt="Sort by Hiredat in descending order">
         </a>

      </th>
      <th>
         <%--Workaround for Navigator so that the arrow aligns
          with the header--%>
          
         <a href="AllEmployees.jsp?sort=Location&sortdir=asc">
            <img border=0 align="absmiddle" src="up12.gif"
             alt="Sort by Location in ascending order">
         </a>
         Location
         <a href="AllEmployees.jsp?sort=Location&sortdir=desc">
            <img border=0 align="absmiddle" src="down12.gif"
             alt="Sort by Location in descending order">
         </a>
      </th>
      <th>
         Action
      </th>
      </tr>
      <%
      // Add one select option for each employee name returned in
      // the result set.
      while(rs.next()) {
         long employeeSSN = rs.getLong("SSN");
         String employeeName = rs.getString("Name");
         long employeeSalary = rs.getLong("Salary");
         Date employeeHireDate = rs.getDate("Hiredate");
         String employeeLocation = rs.getString("Location");
         %>
         <tr>
         <td align="right"><%= employeeSSN%></td>
         <td><%= employeeName%></td>
         <td align="right"><%= employeeSalary%>.00</td>
         <td align="right"><%= employeeHireDate%></td>
         <td><%= employeeLocation%></td>
         <td>
            <a href="EditEmployee.jsp?SSN=<%= employeeSSN %>">Edit</a>
            /
            <a href="ConfirmDeleteEmployee.jsp?SSN=<%= employeeSSN%>">
             Delete</a>
         </td>
         </tr>
         <%
      }
      %>
      </table>
      <br>
      <p><a href="AddEmployee.jsp">Add Employee</a></p>
      <%
   } catch(SQLException e) {
      %>
      <p>An error occured while retrieving all employees:
         <b><%= e.toString() %></b></p>
```

```
   <%
} finally {
   if(stmt != null) {
      stmt.close();
   }
   if(conn != null) {
      conn.close();
   }
}
%>
</center>
</body>
</html>
```

This JSP page connects to the Employees database via the JDBC−ODBC driver, queries all employees and their respective location names, and formats the rows received in a table. To use database connectivity in my page I must first import the JDBC package. I do this in the <%@ page...%> JSP directive element. The following code snippet shows the full tag that I use:

```
<%@ page language="java" import="java.sql.*" %>
```

This tag also specifies that the language I will be using in the scriptlets is Java. Inside the first scriptlet, contained in the <%...%> scripting element, I perform the standard actions of loading the driver, opening a database connection, creating a statement, and finally querying the database for all the employees and their location names.

> XRef Chapters 4 through 6 provide detailed information about effectively using JDBC to connect to a database, JDBC Statements, and JDBC Result Sets.

Each employee is returned as a row in the ResultSet I receive as a result of the query. To place the ResultSet data in a table, I start an HTML table in the template

code and create a header row containing the column names and the sort−arrow graphics. The sort links are dynamically generated based on the column, so this is the ideal time to build them. These links are actually links to the page itself, with URL parameters that specify the column to sort on and the direction of the sort (ascending or descending). Clicking one of these links requests the page itself, which re−queries the database, using the SORT BY clause to properly order it.

I iterate through the ResultSet and create a table row for each database row. Each cell in the row corresponds to the column data. The last column is not populated with data; it is reserved for the two action links, which I also dynamically build. Edit, which is the first action, is a link to the EditEmployee.jsp page. The Edit Employee page, when invoked, needs some way of determining which employee needs to be edited. Since the SSN is a primary key, I put that in the link as a URL parameter for the Edit Employee's page to extract. I use the JSP expression scripting element, <%=...%>, to insert the value employeeSSN into the URL. The following code snippet displays the entire anchor:

```
<a href="EditEmployee.jsp?SSN=<%= employeeSSN %>">Edit</a>
```

The second action, Delete, is a link to another page, DeleteEmployee.jsp. Like Edit, this page also needs to know which employee to edit. Therefore, I pass the SSN in the same way.

To finish the page I close the table, add a link to the AddEmployee.jsp page, and clean up the JDBC

environment by closing the Statement and Connection objects. The remainder of the code closes the HTML tags I opened in the beginning of the document, to ensure that the HTML is well formed.

The All Employees Web page contains three links. Selecting one will request one of three other JSP pages: Edit Employee, Update Employee, or Confirm Delete Employee. First I'll discuss the Edit Employee JSP page.

Example: Edit Employee

Figure 17−8 shows a screen capture of the Edit Employees Web page you will receive as a result of traversing the link. Note that the browser URL contains the parameter ?SSN=981276345. Also note that I have already made the salary change in the form.

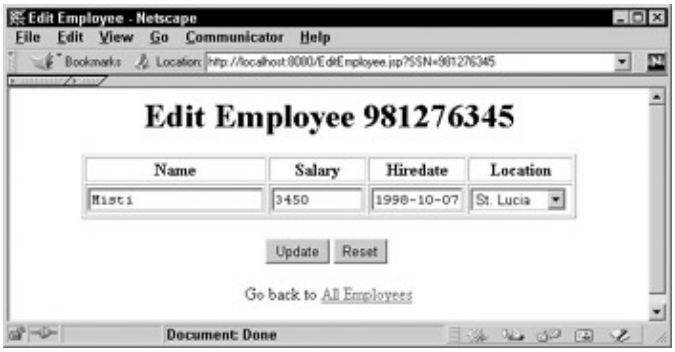

Figure 17−8: The Edit Employee Web page

This page enables the user to modify any attribute, except the SSN, of any employee. If this were a fully featured application, some employee attributes, such as SSN and Salary, would be selectively shown or editable for those with the proper authorization.

Like most JSP pages in this example application, EditEmployee.jsp is similar to AllEmpoyees.jsp. Both establish a database connection, perform a database action such as a query, retrieve the resulting data, format and display the data, and perform some housekeeping. The outstanding difference between the two is that the Edit Employees JSP page uses an HTML form to collect input and perform an action solicited by the user. The form is the only means by which HTML enables you to solicit user interaction. I will discuss the use of the form shortly.

For illustrative purposes, I designed the Edit Employee page so that I can edit only one employee at a time. To determine which employee to edit, I retrieve the SSN from the SSN URL parameter from the page's URL. The following code snippet demonstrates how I use the request.getParameter(*String*) method to retrieve the SSN of the employee:

```
String employeeSSN = request.getParameter("SSN");
```

To make it clear which employee is being edited, I prominently display the SSN attribute in the header at the top of the page. Moreover, I use the SSN value to query the database and retrieve the remainder of the employee's information. I place the returned information in a table with column headers within the form. Each cell in the second row of the table contains the editable information, using the appropriate form input elements to both display the values and solicit user interaction. A *single line text* form input element is sufficient for *Name*, *Salary*, and *Hiredate*, but not for *Location*. A *select* form input element is necessary for Location, so that I can present all the possible locations but still limit the user to a single choice. For usability purposes, I want the user to choose an employee's Location by its name, not the *Loc_Id*. Recall that I also displayed the

Location versus Loc_Id in the All Employees page. To keep the subsequent UPDATE statement simple, I need to use the corresponding Loc_Id. Using Loc_Id instead of Location, defends against any possible conflicts with URL special characters and delimiters I may encounter if I encoded the Location in the URL. This means I don't want any characters in Location to conflict with any of the special characters and delimiters specific to URLs.

Regardless of whether I populate the select input form element with Location or Loc_Id, I still need to dynamically populate the select, which means that I need a separate database query to retrieve all possible values. The query retrieves both columns from the Location table and sorts them by Location name. I populate the select by iterating through the ResultSet and adding an option element for each row I encounter.

The text displayed in each option is the Location, and the value is the corresponding Loc_Id. Therefore, when the user initiates the form action, the Loc_Id for the corresponding Location the user chooses will be encoded as a Loc_Id name−value pair in the URL. In keeping with the rest of the form input elements, I display the current Location by determining which option is associated with the current value, and specifying the selected attribute for the option. The following code snippet exhibits how I dynamically construct the select form input element:

```
<select name="Loc_Id">
<%
// Provide a choice of all valid locations and select the current
// one.
rs = stmt.executeQuery("SELECT Location, Loc_Id " +
                       "FROM Location " +
                       "ORDER BY Location;");
while(rs.next()) {
   String newEmployeeLocation = rs.getString("Location");
   long newEmployeeLocationID = rs.getLong("Loc_Id");
   %>
   <option value="<%= newEmployeeLocationID %>"
      <%
      // If this is the employee's current location that select it.
      if(newEmployeeLocationID == employeeLocationID) {
         %>selected<%} %>><%= newEmployeeLocation
      %>
   </option>
   <%
}
%>
</select>
```

To complete the form, I add two form−input button elements. The first is a *submit* form−input element that initiates the form action. I specify the button text as Update. When the user finishes making modifications and wants to submit them, he or she simply presses the Update button. The action associated with this form requests the UpdateEmployee.jsp JSP page. Prior to the user making the request, the form−input element values are encoded as name−value pairs and appended as parameters to the action URL.

The second button, Reset, is a reset form−input element. I provide this button as a handy mechanism with which users can restart their data entry. Pressing it resets all form−input elements to their default values.

For each form−input element I create, I also specify a default value. This value corresponds to the database column retrieved during the first query. Keep in mind that each time the user presses the Reset button, the browser resets all form input elements to the default values I specified. No database query is performed to obtain these default values.

If users suspect that the data were changed while they were making modifications, they can simply retrieve the new data by refreshing or reloading their browsers. When they do the current Web page is replaced and any modifications made are lost.

Providing a reset mechanism is good design practice. So is providing the user with a way out, home, or back in every page in your application. With this in mind, I finish the page by including a link to the All Employees page.

Now that I've analyzed the creation of the page, let me take you through a very short example of how to use the Edit Employee page. Suppose Misti receives a salary increase from $3,020 to $3,450. To update her employee record to reflect this salary change, you click on the Edit link in the row corresponding to her name on the All Employees page. Recall that each of the Edit links in the All Employees page includes the SSN of the corresponding employee encoded as a URL parameter. When you traverse this link, the EditEmployees.jsp JSP page is requested and is passed Misti's SSN, 981276345, as the SSN URL parameter.

Keep in mind that the Edit Employee Web page is strictly a data−entry page. It does not directly update the database. Rather, when the user presses the Update form action button, the Update Employee JSP page is requested, which performs the UPDATE statement.

Example: Update Employee

This page's sole purpose is to directly update the database and let you know whether the operation succeeded or not. Finishing up the short example regarding Misti's raise, Figure 17−9 shows the Update Employee confirmation Web page received following a successful update of Misti's employee record. The first component rendered on the page is a header indicating the action about to be performed. In this case, this page is updating information regarding Misti, whose SSN is 981276345. Even though the entire URL is not visible in the figure, all the URL parameters are. Note that the first URL parameter, SSN, is present.

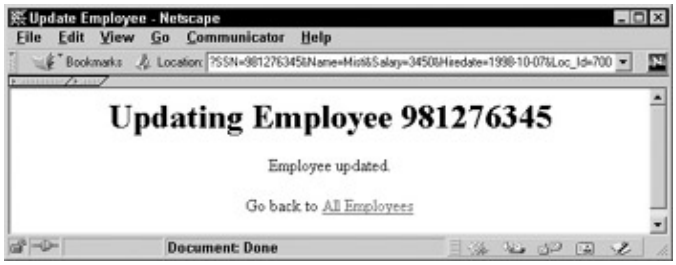

Figure 17−9: The Update Employee Web page

If you recall, the employee's SSN is read−only in the Edit Employees page. In fact, I didn't even provide a form input element from which to obtain it. Or did I?

Actually, one hidden form−input element is designed to provide hard−coded name−value pairs as URL parameters. Although this element can be configured to be visible but not editable, I wanted it to remain invisible. The following code snippet demonstrates how I used the hidden form−input element to pass the SSN as a URL parameter:

```
<input type="hidden" name="SSN" value="<%= employeeSSN %>">
```

The remainder of the URL consists of the name−value pair equivalents for all the visible form−input elements. I can individually retrieve each of these by using the request.getParameter(*String*) method. The following code snippet illustrates how I do just that:

```
String employeeSSN = request.getParameter("SSN");
String employeeName = request.getParameter("Name");
String employeeSalary = request.getParameter("Salary");
String employeeHireDate = request.getParameter("Hiredate");
String employeeLocationID = request.getParameter("Loc_Id");
```

Note that the hidden SSN name−value pair is retrieved in the same manner as the rest of the name−value pairs, by means of the request.getParameter() method.

I then take these values to construct and execute an UPDATE statement. Since only one row is being updated, the executeUpdate(String) method should return an update count of one. I perform a sanity check to ensure that this is the case. If so, I output a success message. Otherwise, I display an appropriate error message.

In my design I made the decision not to automatically redirect to the All Employees page after some predefined time. Rather, I continued with the look and feel of the rest of the application. I finish up the page by providing a link so the user can migrate to that page at any time. Figure 17−10 shows a partial view of the All Employees page I see when I click this link. Note that Misti's current salary reflects her salary increase while her other information remains the same.

Error Pages

In the event that an exception is not handled in the current page, JSP provides a facility for loading another URL. You specify the error−page URL with the errorPage attribute of the <page> directive element. In the following code snippet, SQLError.jsp is specified as the error page:

```
<%@ page language="java" errorPage="SQLError.jsp"%>
```

If you create your own JSP error page, you must set the isErrorPage property. The following code snippet illustrates how to do that:

```
<%@ page language="java" isErrorPage="true"%>
```

Providing an error page for your application can help you maintain your application's look and feel, and the impression that everything is under control.

Figure 17−10: The All Employee Web page — take 2

This completes my discussion of the first of three actions provided by this example application. The other two actions and their associated JSP pages are Delete Employee and Insert Employee, and they work in much the same way. Like Edit Employee, each has an intermediate page that solicits user input. Each also has a confirmation page that performs the designated action. You will find both of these on the book's Web site.

Although the previous JSP application works, it's far from optimized. For example, the same code to interact with the database is duplicated across pages. Simply put, redundant code creates a maintenance nightmare. If you find a bug in one page then you must not only fix it there, but also propagate the change across all the other pages that use the same code. A simple solution to this problem is to put all the redundant code in one place and reference it across all pages. You then only have to maintain one codebase and reuse it. When you do encounter a bug, one change may be sufficient to fix it for all pages that reference this code.

Another example of non−optimized code is that each page request opens a database connection, performs one or more database actions, and then destroys the connection. (This is the same problem I described with regard to the servlet in the previous section.) You can remedy the situation by using the jspInit() and jspDestroy() methods provided to you by the JSP implementation class when the JSP container converts the JSP page into a servlet. There is a better alternative.

The ideal solution to both of these problems is to encapsulate the common functionality into one class. This class will automatically handle the tasks of establishing and maintaining the necessary physical database connections, and of reusing them to attain the highest possible performance. Additionally, this class will provide you with convenient methods for performing common database functions. Furthermore, this class can be instantiated once per application to eliminate the overhead of creating multiple instantiations. A *JavaBeans* component is the type of class that can accommodate all of these requirements and is directly supported by JSP.

Using JSP with JDBC JavaBeans

Using JavaBeans with your JSP pages is an ideal solution to the problem of redundant code and logic. In this section, I describe what JavaBeans are, how to develop them, and how to effectively use them in your JSP pages. I also present the bean I used to overcome the redundancy and performance problems I encountered developing my example application.

What is a JavaBean?

A JavaBeans component, commonly referred to simply as a bean, is a generic class that encapsulates information referred to as properties. JavaBeans are predominantly used to wrap GUI components, such as in AWT, in a generic adaptor class. Java IDEs use the bean wrapper to seamlessly integrate these components. You can easily construct the components using a default constructor or a constructor with no arguments. Furthermore, the bean wrapper provides you with access to the standard GUI properties so you can retrieve and set the values of component attributes such as width and background color.

Note More information regarding JavaBeans can be found at http://java.sun.com/products/javabeans.

However, JavaBeans can also wrap non−GUI components. For example, the javax.sql.RowSet interface acts like a ResultSet that has a bean wrapper. In this case, the properties you set affect connection parameters, cursor types, and transaction levels, not GUI properties.

XRef

> See Chapter 16, “Working with JDBC RowSets,” for more details about how to use the RowSet interface.

I’ll show you how to use a bean in JSP pages, and present a small bean I developed to help overcome the shortcomings of my example application. An in−depth discussion of JavaBeans development is beyond the scope of this book.

How to use JavaBeans within JSP

Using a bean within a JSP page is fairly straightforward. First, the bean must be *used*, meaning that the bean class is loaded, if it hasn’t been already, and is instantiated.

The JSP action <jsp:useBean id=.../> specifies which bean to use and how to use it. The following code snippet shows how I use the DatabaseBean in AllEmployeesBean.jsp:

```
<jsp:useBean id="dbBean" class="Chapter17.DatabaseBean"
             scope="application"/>
```

The id attribute specifies the name by which you will reference the bean instance in your JSP page(s). The class attribute specifies the fully qualified class name. In this case, DatabaseBean is part of the Chapter17 package, so I specify it as Chapter17.DatabaseBean. The last attribute, scope, specifies the scope this bean instance will have. I want one instance to be shared across the entire application, so I specify the scope to be application.

If I were to specify the scope of the bean as page, rather than as application, a separate bean instance would be created each time the page was requested. Since the instance would only be valid for the scope of the page, it would be marked for garbage collection as soon as the request completed, and this would totally defeat the purpose of maintaining physical database connections for the entire application.

> Caution If you don’t specify the same id name in all pages of your application, multiple instances of the bean will be created as well.

Once the bean is used, you can reference it in your scriptlet code just as you would any Java object; you can access all public−class or instance variables and methods. If the bean has any properties, you can also use the <jsp:setProperty> JSP action to set the value of that property. You can also use the complementary JSP action, <jsp:getProperty>, to get the current value of the specified property.

That’s basically all there is to using a bean. Now I’ll discuss the DatabaseBean bean and how I use it.

Example: DatabaseBean

In the previous section I detailed the two shortcomings of the example application. Then I developed the DatabaseBean bean to overcome these two shortcomings. First, I encapsulated the common code for easy reuse. This eliminated the code redundancy problem while dramatically reducing code maintenance. Second, I minimized the amount of database handshaking by establishing the connection only once per bean instance. Moreover, I reuse the connection for multiple database actions, which increases performance without sacrificing maintenance.

Listing 17−3 reveals the bean’s code in its entirety. Observe that I satisfy the JavaBean requirement that my class contain a constructor with no arguments. Any exception thrown during creation of a bean instance is caught, some housekeeping is performed, and the exception is rethrown.

This constructor first loads the necessary JDBC driver. (The driver is only loaded once because the JVM class loader caches classes.) It may have been loaded already by another application or bean, or during the creation of the first instance of this bean. Next, I attempt to establish a database connection.

Listing 17−3: DatabaseBean.java

```
// Java Data Access: JDBC, JNDI, and JAXP
// Chapter 17 − Web Applications and Data Access
// DatabaseBean.java
package Chapter18;

import java.io.*;
import java.sql.*;

public class DatabaseBean {
   private static final String jdbcDriverClass =
      "sun.jdbc.odbc.JdbcOdbcDriver";
   private static final String jdbcDatabaseURL =
      "jdbc:odbc:Employees";
   private Connection conn;

   public DatabaseBean() throws Exception {
      try {
         // Load the JDBC−ODBC driver.
         Class.forName(jdbcDriverClass);
         // Open a database connection.
         connect();
      } catch(SQLException ex) {
         // Perform any cleanup.
         cleanup();
         // Rethrow exception.
         throw(ex);
      }
   }

   public void connect() throws SQLException {
      if(conn != null) {
         cleanup();
         conn = null;
      }
      // Open a database connection.
      conn = DriverManager.getConnection(jdbcDatabaseURL);
   }

   public void cleanup() throws SQLException {
      // Close the Connection.
      if(conn != null) {
         conn.close();
      }
   }

   // The onus is on the user to close the Statement that created
   // the returned ResultSet.
   public ResultSet query(String queryStatement)
      throws SQLException {
      Statement stmt = null;
      ResultSet rs = null;

      try {
         stmt = conn.createStatement();
```

```
            rs = stmt.executeQuery(queryStatement);
         } catch(SQLException ex) {
            try {
               if(stmt != null) {
                  stmt.close();
               }
            } catch(SQLException ex2) {
            }
            finally {
               throw(ex);
            }
         }

         return(rs);
      }

      public int insert(String insertStatement) throws SQLException {
         return(update(insertStatement));
      }

      public int delete(String deleteStatement) throws SQLException {
         return(update(deleteStatement));
      }

      public int update(String updateStatement) throws SQLException {
         Statement stmt = null;
         int numRows = −1;

         try {
            stmt = conn.createStatement();
            numRows = stmt.executeUpdate(updateStatement);
         } catch(SQLException ex) {
            try {
               if(stmt != null) {
                  stmt.close();
               }
            } catch(SQLException ex2) {
            } finally {
               throw(ex);
            }
         } finally {
            try {
               if(stmt != null) {
                  stmt.close();
               }
            } catch(SQLException ex) {
            }
         }

         return(numRows);
      }

      public Connection getConnection() {
         return(conn);
      }
   }
```

If I am successful opening a connection to the database, I store the Connection object in a private instance variable that also serves as a read−only bean property that is only externally accessible via the getter method,

getConnection(). I did not provide a setter method for this property because I didn't want it externally set with a possibly invalid value. Besides, it is populated with a valid value in the constructor. You can retrieve this property in your JSP pages and use it as you would if you manually created it. I also created four convenience methods that handle the majority of the operations you might use these properties for.

The two main methods are query(String) and update(String): They wrap the executeQuery(String) and executeUpdate(String) Statement methods, respectively. The other two methods, insert(String) and delete(String), simply call update(String), because they basically use the executeUpdate(String) as well.

The bean does not generate any template text such as HTML. This is fairly common among beans intended for use by JSP pages, and is a prime example of how to effectively use the MVC design pattern to separate your data source from your presentation.

Although the DatabaseBean bean serves its purpose, it is not a commercial−grade bean. It suffers from an obvious shortcoming: It does not take advantage of connection pooling, which dramatically increases performance. Unfortunately, the JDBC−ODBC driver does not support connection pooling, so I can't do anything about that.

> XRef See Chapter 14, "Using Data Sources and Connection Pooling," for detailed information about using the JDBC connection pooling facilities.

To use the DatabaseBean bean in my example application, I cloned the entire application and modified each page to use the bean. The word *bean* is appended to the core part of each of the filenames of the bean application. Let me show you how I modified the All Employees JSP page to use the DatabaseBean bean. Listing 17−4 shows the AllEmployeesBean.jsp JSP page in its entirety. (Note the word bean in the filename.)

Listing 17−4: AllEmployeesBean.jsp

```
<%--Java Data Access: JDBC, JNDI, and JAXP--%>
<%--Chapter 17 − Web Applications and Data Access--%>
<%--AllEmployeesBean.jsp--%>
<html>
<head><title>All Employees</title>
</head>
<%@ page language="java" import="java.sql.*" %>
<body>
<center>
<h1>All Employees</h1>
<%--Build a table of all employees to choose from--%>
<jsp:useBean id="dbBean" class="Chapter18.DatabaseBean"
             scope="application"/>
<%
try {
   // Retrieve the user agent and determine if it is Netscape
   // Navigator for later use.
   String userAgent = request.getHeader("User−Agent");
   boolean clientIsNetscapeNavigator =
      (userAgent.indexOf("(compatible; MSIE") == −1);
   // Retrieve the column and direction to sort by.
   // By default, sort by SSN in ascending order.
   String orderBy = request.getParameter("sort");
   if((orderBy == null) || orderBy.equals("")) {
      orderBy = "SSN";
   }
   String orderByDir = request.getParameter("sortdir");
```

```
if((orderByDir == null) || orderByDir.equals("")) {
   orderByDir = "asc";
}
// Compose the query statement.
String query = "SELECT Employees.SSN, Employees.Name, "
               "Employees.Salary, " +
               "Employees.Hiredate, Location.Location " +
               "FROM Location " +
               "INNER JOIN Employees " +
               "ON Location.Loc_Id = Employees.Loc_Id " +
               "ORDER BY " + orderBy + " " + orderByDir + ";";
// Execute the query.
ResultSet rs = dbBean.query(query);
%>
<table border="1">
<tr>
<%
String fields[] =
   {"SSN", "Name", "Salary", "Hiredate", "Location"};
for(int i = 0; i < fields.length; i++) {
   String field = fields[i];
   %>
   <th>
   <%--Workaround for Netscape Navigator so that the up arrow
       graphic aligns with the header--%>
   <% if(clientIsNetscapeNavigator){%> <%}
      %>
   <a href="AllEmployeesBean.jsp?sort=<%= field %>&sortdir=asc">
      <img border=0 align="absmiddle" src="up12.gif"
       alt="Sort by <%= field %> in ascending order">
   </a>
   <%= field %>
   <a href="AllEmployeesBean.jsp?sort=<%= field %>&sortdir=desc">
      <img border=0 align="absmiddle" src="down12.gif"
       alt="Sort by <%= field %> in descending order">
   </a>
   </th>
   <%
}
%>
<th>Action</th>
</tr>
<%
// Add one select option for each employee name returned in the
// result set.
while(rs.next()) {
   long employeeSSN = rs.getLong("SSN");
   String employeeName = rs.getString("Name");
   long employeeSalary = rs.getLong("Salary");
   Date employeeHiredate = rs.getDate("Hiredate");
   String employeeLocation = rs.getString("Location");
   %>
   <tr>
   <td align="right"><%= employeeSSN%></td>
   <td><%= employeeName%></td>
   <td align="right"><%= employeeSalary%>.00</td>
   <td align="right"><%= employeeHiredate%></td>
   <td><%= employeeLocation%></td>
   <td>
      <a href="EditEmployeeBean.jsp?SSN=
         <%= employeeSSN %>">Edit</a>
```

```
         /
         <a href="ConfirmDeleteEmployeeBean.jsp?SSN=
            <%= employeeSSN%>">Delete</a></td>
      </tr>
      <%
   }
   // Close the Statement that created this ResultSet.
   rs.getStatement().close();
   %>
   </table>
   <br>
   <p><a href="AddEmployeeBean.jsp">Add Employee</a></p>
   <%
} catch(SQLException e) {
   %>
   <p>An error occured while retrieving all employees: <b>
      <%= e.toString() %></b></p>
   <%
}
%>
</center>
</body>
</html>
```

The remainder of the application using the bean is available on the book's Web site. In those files, all the changes I made are almost identical. As a direct result of my integrating the bean, the application is leaner, faster, more robust, and easier to debug. It takes less code to generate the same pages, which enables me to concentrate more on presentation.

Because all the JSP pages in my application use the same bean instance, only one physical database connection is established and used across my entire application. Furthermore, I don't have to worry about when to actually instantiate the bean. In addition to using the bean, I perform cursory browser detection and make a small adjustment if it is Netscape Navigator, to properly align the sorting arrow graphics with the column headings.

Using a bean solves my problem, but it is still not the optimum solution. A separate bean instance is created for each newly created session. Each bean instance establishes a separate, physical database connection. Although this connection stays open for the life of the bean, having many separate connections can consume a lot of resources, such as memory, CPU, and network bandwidth. To top it all off, the connections will be idle the majority of the time. The ideal scenario would be to pool the physical connections and share this pool across all instances of the bean. A connection pool would ensure that there were enough connections to handle the load, and it would close them when it determined that they had been idle for an extended period of time.

Why didn't I go further and let the bean generate the HTML that would make my JSP pages smaller? Although nothing prevents you from having your beans generate HTML, a more robust approach is to keep your controller bean as is and simply develop another bean whose sole purpose is to wrap the first bean and then generate HTML. This way, other JSP pages or software can use the first bean as is, and you can simply provide a wrapper for each presentation you need. (Recall the discussion of MVC). A bean takes on the role of the Model in an MVC implementation.

I wanted a very generic and easily extensible bean, so I didn't include any business logic either. Although I recommend you follow the MVC design pattern, there are times when it is just not practical. Use your best judgment when including presentation and business logic in a bean.

Design considerations

In keeping with the theme of separating presentation from business logic, template text should contain only the presentation logic. The beauty of JSP is that it provides you with the mechanisms for this natural separation of tasks. Theoretically, HTML authors can happily create their presentation in the template text with no knowledge of how the programmers are implementing the business logic. The reality of it is that neither side can operate in a vacuum without sacrificing the end goal. The HTML authors may either be provided with stubs or they can simply insert placeholders where the scripting elements will eventually reside. Programmers do not have to worry about presentation. They can simply concentrate on implementing the business logic and exposing an interface to this logic.

No matter how well you design your application, at some point you will most likely need to have one or more scripting elements generate some amount of presentation code in order to satisfy a requirement. JSP doesn't impose a policy on the amount of presentation code scripting elements can generate. Obviously, you can easily abuse or violate this virtual separation, so the only advice is to use your best judgment.

It is self−evident that JSP pages can be ugly. They can also suffer from feature bloat and get quite monolithic. Packing a bunch of code into a single JSP page that does everything may work initially. Eventually, you may be tasked to take an existing application, make it distributable and scalable across multiple machines, and, of course, to fix bugs and infuse enhancements. If your page is bloated and monolithic, this may prove to be very difficult, if not impossible.

Tip Although you should always design your pages so that business logic and presentation are as separate and distributable as possible, you don't have to go overboard and separate out every sliver of functionality and place it in its own page or object. Doing so can result in an overly complex system that is just as much a nightmare to grasp and maintain as a monolithic application. Again, use your best judgment.

Let me point out that JSP pages do not need to contain both JSP elements and template text; they can contain one or the other. Many applications have JSP pages that generate no presentation. These pages are designed for inclusion by or in other pages.

Summary

In this chapter I discussed the components that make up enterprise Web applications, and the underlying architectures of those applications. I also covered the role of J2EE in enterprise Web applications and discussed how to use JDBC with servlets and JSPs to effectively create scalable, efficient, and robust enterprise Web applications.

Although JSPs provide you with an easy and fairly natural mechanism with which to create dynamically generated Web pages, Java servlets are beneficial too. They are the foundation of JavaServer Pages. Without servlets, JSP would not function.

My recommendation, and I'm sure you'll agree, is to develop all your dynamically generated pages in JSP. In particular, use it to develop the presentation portion of your application. Servlets should be used only to implement business logic or play the role of the Controller.

As is evident in the above examples, using JDBC with servlets and JSP pages is fairly painless. It is not unlike using JDBC with other Java applications and applets.

Chapter 18: Using XML with JAXP

by Johennie Helton

In This Chapter

- Introducing XML
- Understanding namespaces, DTDs, and XML Schemas
- Exploring XML databases
- Working with SAX, DOM, and XSLT
- Using the JAXP API

The main objective of this chapter is to explore the JAXP API; to accomplish that I need to present a few important concepts as a basis, and for reference. The Java API for XML Parsing (JAXP) is a powerful, easy−to−use API that makes XML easier for Java developers to use. This chapter introduces XML and important XML concepts like namespaces and DTDs, to help you understand technologies such as SAX, DOM, and XSLT, which enable you to process XML documents. Finally, I discuss the JAXP specification that helps Java developers read, manipulate, and generate XML documents by supporting XML standards.

XML is an open standard; it allows the storing and organization of data in any form that best satisfies your needs; you can easily combine XML with style sheets to create formatted documents in any style you want. XML technologies offer simple ways to process XML documents and data. I'll start with an introduction that describes XML in more detail, and then move on to the different ways of processing XML documents. Finally, I'll describe the JAXP API in terms of processing XML data and documents.

Introducing XML

XML, the eXtensible Markup Language, was designed for document markup, but you can use it for many other things as well, such as configuration files and file format definition. XML has even become the default standard for data transfer. Data transfer is important because the ability to exchange and transfer information from one business to another is critical.

What is XML?

XML is a robust open standard based on SGML. Like all markup languages, it uses tags to define what each element means: For example, you can use the tag <title> to describe that the element is the title of a book. In addition, markup languages have grammar rules that define the correct use of the tags. XML is a metalanguage because it does not specify the set of tags to be used but instead defines a framework for creating grammars. The grammar of a language defines the rules to which the language has to conform. The W3C XML specifies the core XML syntax, which can be found at http://www.w3.org/XML/.

The names of the tags and their grammar are up to the designer—these things are completely made up. This makes XML very powerful, because it enables the designer to specify the data via tags and grammar that best correspond to the application, as long as they conform to the general structure that XML requires.

It is best if the tags in an XML document are named after what they represent. Take a look at Listing 18−1. It is a simple XML document, and generic tag names such as <tag1> and <tag2> could have been used, but in

that case it would not have been clear what the elements represent. In this simple example I am describing a book: I specify the author's first and last name, the publication date, and an internal identifier. I could have added other information such as the price or availability. For the moment just ignore the second line; it will be described later in this chapter.

Listing 18−1: bookDesc.xml

```
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE myspace:book SYSTEM "doc.dtd">
<myspace:book xmlns:myspace="http://www.ournamespace.com">
 <title>Alice's Adventures in Wonderland and Through the Looking Glass</title>
 <author>
  <first−name>Lewis</first−name>
  <last−name>Carroll</last−name>
 </author>
 <date> May 2000 </date>
 <internal identifier="12345"/>
</myspace:book>
```

Every XML document has one root element: In the preceding example the root element is <book>. Any XML element may contain character data or other XML elements: This example contains the elements <title> (the book title), <author> (the author's first and last names) and <date> (the publishing date). Finally, each XML element may have an associated list of attributes: In this example the element named internal has an attribute called identifier with a value of 12345. *Attributes* in XML are name−value pairs that allow additional data in start and empty tags. *Empty tags* are those that do not have values associated with them, such as the pair <emptyTag></emptyTag>. The next section discusses the concepts of name collisions and namespaces, which are important to an understanding of XML.

Namespaces

The XML namespace recommendation defines a mechanism for the mapping between an element prefix and an URI, which qualify elements to handle *name collisions*. As long as only one document exists, there can be no name collisions; in effect you are using a single namespace. Once you modularize your XML code and start accessing or referencing one document in another, collisions may arise. For instance, in the preceding example I have a date element. If I were to reference another XML document with a date element on it (say one representing the date the book order was placed), then there would be a collision with the date element. Therefore, I need to distinguish the meaning of both dates, and I do this by associating each date element with a namespace. The namespace recommendation can be found at http://www.w3.org/TR/1999/REC−xml−names−19990114/.

For example, I can replace the <book> element with the line <myspace:book xmlns:myspace="http://www.ournamespace.com">, and replace </book> with </myspace:book>. This creates a namespace called myspace for all the XML code between these two tags.

I must mention couple of things. First, namespaces are not part of the validation process—that is, the site specified in the URI does not have to contain a list of all the names in the namespace. In addition, namespaces are inherited, meaning that if the processor finds an element with a prefix, but without the xmlns, it will look to the element's ancestors (parent, grandparents, and so on) until the namespace declaration is found. If it is not found, then a namespace constraint is violated. Since the grammar and tags of XML documents are not specified, you need to constrain the XML documents to adhere to the grammar and syntax that you want others to follow. Document Type Definitions (DTDs) and XML Schemas do just that.

Document Type Definitions and XML Schemas

Document Type Definitions (DTDs) and XML Schemas enable you to constrain XML documents. That is, they allow the XML document to have its own syntax and both, the schema and the DTD, can be used by many instances of XML documents that satisfy the syntax. Visit the http://www.w3c.org/, http://www.xml.org/ and http://www.xml.com/ sites for more information.

A DTD is a collection of parameters that describe the document type. It has a declaration of the set of allowed elements. Listing 18−2 shows a DTD for the XML example given in Listing 18−1; the allowed elements are book, title, author, date, internal, first−name, and last−name. The DTD also declares the attributes: their types and names, if necessary, and their default values, if necessary. In this example the element internal has an attribute, identifier, which is required but that does not have a default value. Finally, the DTD defines the grammar: what is allowed in the elements and their attributes, the attribute's type, whether the attribute is required, and the required order. For instance, author is composed of first−name and last−name elements, in that order.

Listing 18−2: bookDesc.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT author (first-name, last-name)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT internal EMPTY>
<!ATTLIST internal
       identifier CDATA #REQUIRED
>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT myspace:book (title, author, date, internal)>
<!ATTLIST myspace:book
       xmlns:myspace CDATA #REQUIRED
>
<!ELEMENT title (#PCDATA)>
```

Schemas provide an alternative to DTDs. Like DTDs, schemas describe the syntax of XML documents. However, an XML Schema file is itself a well−formed XML document, and provides greater control over data types and patterns. Listing 18−3 gives an example of a schema; it describes the same elements as the previous example, but notice that book, author, and internal are now complex types composed of their corresponding elements or attributes, which is very different from the DTD style. In addition, this example contains some predefined datatypes, such as xsd:string. You can do a lot more with schemas, such as having enumerated types, and inheriting properties of other declarations and more. For more about schemas, visit http://www.w3.org/XML/Schema.

Listing 18−3: bookDesc.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
elementFormDefault="qualified">
 <xsd:element name="book">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="author">
    <xsd:complexType>
     <xsd:sequence>
```

```
      <xsd:element name="first-name" type="xsd:string"/>
      <xsd:element name="last-name" type="xsd:string"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="date" type="xsd:string"/>
   <xsd:element name="internal">
    <xsd:complexType>
     <xsd:attribute name="identifier" type="xsd:string" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
</xsd:schema>
```

XML Databases

XML technologies are well suited to transmitting and displaying structured data, and because you can define your own syntax and grammar, they can also be well suited to modeling relational databases. Relational databases are very popular for data storage, and XML, as I mentioned earlier, offers great flexibility as a data-exchange technology. Therefore, it is not surprising that these technologies have come together. Do not confuse relational databases with XML databases. RDBMS are designed to store and retrieve data whereas XML databases only represent the data in a vendor-neutral format. XML data access tools use either SAX or DOM to manipulate and retrieve data. The performance of these tools pale in comparison to RDBMS SQL engines.

You have many options when modeling relational databases. For example, you can represent your table columns as XML elements or as attributes of XML elements. You must decide how to model the table's keys, whether as primary or foreign keys. Also, you need to decide how to model the table relationships, whether as one-to-one or as many-to-many.

> XRef Chapter 2, "A Relational Database Primer," provides more information on working with relational databases and SQL.

Let's assume you have a company called "We Ship Books," which keeps a record of invoices of books to customers. Perhaps you have a series of tables such as Invoice, Customer, and Item; you can model them in an XML document as follows:

The Customer table, defined by the following code

```
CREATE TABLE Customer (
     CustomerId integer,
     CustomerName varchar(100),
     Street varchar(50),
     City varchar(50),
     State char(2),
     Zip varchar(10),
     PRIMARY KEY (CustomerId)
);
```

may be modeled with the following elements:

```
<!ELEMENT Customer (id, name, email, street, city, state, zip)>
```

```
<!ELEMENT id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
```

The Invoice table, defined by the following code:

```
CREATE TABLE Invoice (
     InvoiceId integer,
     CustomerId integer,
     InvoiceName varchar(100),
     InvoiceTitle varchar(100),
     CONSTRAINT FKInvoiceCustomer FOREIGN KEY (CustomerId)
        REFERENCES Customer (CustomerId),
     PRIMARY KEY (InvoiceId)
);
```

may be modeled with the following elements:

```
<!ELEMENT invoice (Name, title, invoiceid, Customer, item+)>
<!ELEMENT invoiceid (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
```

Finally, the Item table, defined by the following code:

```
CREATE TABLE Item (
     Itemid integer,
     InvoiceId integer,
     Qty integer,
     Title varchar(100),
     Price float,
     CONSTRAINT FKItemInvoice FOREIGN KEY (InvoiceId)
        REFERENCES Invoice (InvoiceId),
     PRIMARY KEY (Itemid)
);
```

may be modeled with the following elements:

```
<!ELEMENT item (itemid, qty, title, price)>
<!ELEMENT itemid (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT qty (#PCDATA)>
```

There are many ways to model a relational database with XML documents. The preceding example is intended only to give you an idea of how it can be done. (By the way, you can derive relational−database schemas from XML too—a detailed explanation, however, is beyond the scope of this book.) Now that you understand of what an XML document consists, the next sections will show you how to process XML documents.

Working with XML—The Basics

When you have an XML document, the first thing you want to know is whether it is well formed and valid. An XML document is *well formed* if it follows the syntax rules for XML. These constraints are very simple. For example, elements (containing text or other elements) must have a start tag and an end tag, and element names must start with letters or underscores only (refer to the W3C site for a discussion of the rules). An XML document is *valid* if there is a document type (such as a DTD or a schema) and the XML document complies with it. Once you have established that the XML document is well formed and valid, you want to process the document and its data. Processing an XML document means to either parse it, transform it or traverse it; these actions are not exclusive of each other and in fact they may be used in conjunction with each other. *Parsing* is the process of reading the document and breaking it into pieces; *transform* means to modify the document so other applications may process the information; and *traverse* means to navigate the document one piece at a time. During parsing or traversing of an XML document, the document is read and an internal representation of its contents created.

When it is necessary to rearrange the document into a different form so that another application may use it, you *transform* it. For instance, you may want to transform your XML document describing a book title to HTML so its contents may be displayed in a browser. You use parsing or traversing a document when you need to access each document piece. One of the first things a program does with an XML document is parse it. You do not need to write a parser by yourself; several ongoing efforts provide XML parsers. After selecting a parser, you want to make sure that your Java code can interface with it. The Simple API for XML is one of the tools available.

Parsing XML: The Simple API for XML (SAX)

SAX, the Simple API for XML, is just that, an API. It provides an event−driven framework with which to parse XML data. At each step of the parsing SAX defines the events that may occur. SAX is not a parser in itself; it provides interfaces and classes to be implemented by the application or parser. The term *SAX parser* refers to a parser that is compliant with the SAX API. For documentation and other information go to www.megginson.com/SAX/Java/. Note that SAX is public−domain software developed through the XML−dev mailing−list discussions.

Some of the parsers available include Sun Microsystems's Project X, IBM's XML4J, and the Apache Software Foundation's Xerces. Once you have selected a parser and installed it, you need to make sure that the XML parser classes are available in your environment. Usually, the SAX classes are included with your parser.

The term *event−driven model* refers to a model in which, when the application encounters a tag, an event is generated and the application captures that event and processes it as necessary. SAX uses an event−driven model that is, the application has to keep track of the element names and data at every event generated. Table 18−1 lists the interfaces included with the SAX API.

Table 18−1: The SAX API

SAX API	Description
Package org.xml.sax	

AttributeList	Manipulates attribute specifications for an element. Its use is deprecated for SAX 2.0.
Attributes	Represents a list of XML attributes. It is namespace−aware. Methods include getLength, getType, and getValue.
ContentHandler	Interface for callback methods that deal with the contents of the XML document to be parsed. Methods include startDocument, startElement, and skippedEntity.
DTDHandler	Manipulates all DTD−related events. Methods are notationDecl and unparsedEntityDecl.
DocumentHandler	Receives notifications of general document events. Methods include startDocument, endElements, and characters. Its use is now deprecated.
EntityResolver	Basic interface for resolving entities with the method resolveEntity.
ErrorHandler	Error−handling interface that allows a means for exception handling. These exceptions are those that could stop parsing altogether. Methods are warning, error, and fatalError.
Locator	Associates a SAX event with a document location. Methods are getPublicId, getSystemId, getLineNumber, and getColumnNumber.
Parser	All SAX 1.0 parsers implement this basic interface. Methods include setLocale, setEntityResolver, and parse. Its use is now deprecated.
HandlerBase	The default base class for handlers. Methods include notationDecl, setDocumentLocator, and endElement. Its use is now deprecated.
InputSource	Allows a single object to encapsulate input−source information. Methods include setPublicId, getSystemId, and getEncoding.
SAXException	Encapsulates general SAX errors and warnings. Methods include getMessage and getException.
SAXNotReconizedException	Extends SAXException which is used by XMLReader implementation to throw an error for an unrecognized identifier.
SAXNotSupportedException	Extends SAXException which is used by XMLReader implementation to throw an error for an unsupported identifier.
SAXParseException	Encapsulates parsing SAX errors and warnings and extends SAXException. Methods include getPublicId and getColumnNumber.
XMLFilter	Extends XMLReader and obtains its events from another XMLReader rather than a from a document. Methods include setParent and getParent.
XMLReader	Interface for all parsing behavior; it added namespace support. Methods include parse, getErrorHanlde, and getDTDHandler.
Package org.xml.sax.ext	
DecHandler	Implementation for DTD−specific DTD declarations. Methods include elementDecl and attributeDecl.
LexicalHandler	Defines callbacks at the document level at processing time. Methods include startCDATA and comment.
Package org.xml.sax.helpers	
AttributeListImpl	Java−specific implementation for AttributeList; additional methods include setAttributeList and removeAttribute.
AttributeImpl	Default implementation for Attributes. Adds the methods of addAttribute, setAttribute, and clear.

DefaultHandler	Empty implementation for all SAX 2.0 core−handler interfaces.
LocatorImpl	Java−specific implementation for Locator. Adds four methods: setPublicId, setSystemId, setLineNumber, and setColumnNumber.
NamespaceSupport	Provides the namespace behavior. Methods include reset, pushContext, and getPrefixes.
ParserAdapter	Wraps the SAX 1.0 Parser implementation to behave like XMLReader.
ParserFactory	Java−specific class for dynamically loading a SAX parser. Adds the makeParser methods. Implements the XMLReader and DocumentHandler interfaces.
XMLFilterImpl	Provides the implementation for XMLFilter. Implements the XMLFilter, EntityResolver, DTDHandler, ContentHandler, and ErrorHandler interfaces.
XMLReaderAdapter	Wraps the SAX 2.0 XMLReader implementation so that it behaves like the SAX 1.0 parser implementation. Implements the Parser and ContentHandler interfaces.
XMLReaderFactory	Creates an instance of an XMLReader implementation, either from a specified class name or from the org.xml.sax.driver property.

To work with these interfaces you can follow these steps:

1. Make sure that the SAX classes are in the class path.
2. Import the necessary interfaces from the SAX API.
3. Import your parser implementation.
4. Create the parser and set the SAX XMLReader implementation to recognize the parser.
5. Register the callbacks that you want to handle. Notice that these callbacks happen as the document is being parsed, because SAX is sequential.
6. Write the code that will handle the callbacks, such as the code that will handle the start and end of the document and each element. You may need to add empty callback handlers so that your code compiles.
7. Use the parser to parse the document.

Listing 18−4 contains the bookDescParser.java file, which uses the preceding steps to parse an XML document. I have decided to use the AElfred parser version 6.4.3 from Microstar, which comes bundled with Saxon (you can get Saxon from http://saxon.sourceforge.net/). I create an instance of Saxon and then set the XMLReader to it. The SAXErr class is a class that I created to implement org.xml.sax.ErrorHandler; it prints error messages as necessary.

Listing 18−4: bookDescParser.java

```
// the needed java classes
import java.io.*;

// Step 2. Import the interfaces needed from the SAX API.
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;

// Step 3. Import the parser implementation.
import com.icl.saxon.*;
```

```
public class bookDescParser extends DefaultHandler {

// Step 6. Write the code that will handle the callbacks
// note that we extend DefaultHandler in this class but we could
// have used a separate class for this or we could have
// implemented the ContentHandler
public void startDocument() {
  System.out.println("Start of document");
}

public void endDocument() {
  System.out.println("End of document");
}

public void startElement(String namespaceURI, String localName,
      String qName, Attributes atts) throws SAXException {
  System.out.print("start element: namespace=" + namespaceURI
        + ", localName=" + localName + ", qName=" + qName);
  int n = atts.getLength();
  for (int i = 0; i < n; i++) {
    System.out.print(" att["+i+"] " + atts.getQName(i) + "='"
        + atts.getValue(i) + "'");
  }
  System.out.println("");
}

public void endElement(String namespaceURI, String localName,
         String qName) throws SAXException {
  System.out.println("endElement: namespace=" + namespaceURI + ",
      localName=" + localName + ", qName=" + qName);
}

public void characters(char ch[], int start, int length)
         throws SAXException {
  System.out.print("Characters:  \"");
  for (int i = start; i < start + length; i++){
   switch (ch[i]) {
    case '\\':
      System.out.print("\\\\");
      break;
    case '"':
      System.out.print("\\\"");
      break;
    case '\n':
      System.out.print("\\n");
      break;
    case '\r':
      System.out.print("\\r");
      break;
    case '\t':
      System.out.print("\\t");
      break;
    default:
      System.out.print(ch[i]);
      break;
    }
  }
  System.out.println("\\n");
}
```

```
// usage: java bookDescParser bookDesc.xml
public static void main (String args[]) throws SAXException {
 // Step 4. Create the parser and set the XMLReader
 XMLReader parser = new com.icl.saxon.aelfred.SAXDriver();

 // Step 5. Register the callbacks that we are interested to
 // handle.
 bookDescParser handler = new bookDescParser();
 SAXErr ehandler = new SAXErr();
 parser.setContentHandler(handler);
 parser.setErrorHandler(ehandler);

 // Step 7. Use the parser to parse the document
 try {
   FileReader fr = new FileReader(args[0]);
   parser.parse(new InputSource(fr));
 } catch (SAXException se) {
   System.err.println("Error: parsing file " + se);
 } catch (FileNotFoundException fnfe) {
   System.err.println("Error: file " + args[0] +" : "+ fnfe +"
       not found.");
 }catch (IOException ioe) {
   System.err.println("Error: reading file "+ args[0] + " : " +
       ioe);
 }
 }
}
```

The following output is the result of running the bookDescParser program with the bookDesc.xml file (see Listing 18−1) as the input file. Each of the elements is printed as the XML document is parsed.

```
Start of document
start element: namespace=http://www.ournamespace.com, localName=book,
qName=myspace:book
start element: namespace=, localName=title, qName=title
Characters:  "Alice's Adventures in Wonderland and
Through the Looking Glass

endElement: namespace=, localName=title, qName=title
start element: namespace=, localName=author, qName=author

start element: namespace=, localName=first−name, qName=first−name
Characters:  "Lewis
endElement: namespace=, localName=first−name, qName=first−name
start element: namespace=, localName=last−name, qName=last−name
Characters:  "Carroll
endElement: namespace=, localName=last−name, qName=last−name
endElement: namespace=, localName=author, qName=author
start element: namespace=, localName=date, qName=date
Characters:  " May 2000
endElement: namespace=, localName=date, qName=date
start element: namespace=, localName=internal,
qName=internal att[0] identifier=
'12345'
endElement: namespace=, localName=internal, qName=internal
endElement: namespace=http://www.ournamespace.com,
localName=book, qName=myspace:book
End of document
```

Parsing enables you to work with XML documents in two ways. First, you can use SAX to capture the events generated as the document is parsed. Second, you can use the DOM model to parse the XML document and traverse an internal representation of it during processing.

Traversing XML: The Document Object Model (DOM)

DOM, the Document Object Model, is a standard with origins in the W3C and is not specific to Java. It describes the structure and APIs for manipulating the document tree, not how the tree itself is to be created; this is vendor−specific. Binding DOM to Java provides the API that you can use to manipulate documents in Java, as described by the DOM specification. DOM does not have versions, but levels. (For DOM information and specifications go to http://www.w3.org/DOM/, http://www.w3.org/TR/REC−DOM−Level−1/, http://www.w3.org/TR/DOM−Level−2−Core/, and http://www.w3.org/TR/2001/WD−DOM−Level−3−XPath−20010830/.)

DOM enables you to manipulate the whole document, because the DOM model reads the whole document into memory and represents it as a tree that you can then manipulate and traverse. Because the whole document is read into memory, the DOM model puts a strain on the system. Therefore, performance degradation may occur as the document grows large and complex.

Several DOM implementations are available: Apache's Xalan is one of them. The standard API focuses on the manipulation of the document tree, and not on the way you obtain that tree after a parse. The tree is in a usable state only after the complete document has been read and parsed, when it is available to be used as an org.w3c.dom.Document object. Each implementation is slightly different: Sometimes you call a getDocument() method to get the document tree, other times the parse() method returns it. DOM nodes form the document tree; each node implements other more specialized interfaces. You can recursively iterate through a node at a time for processing using the interfaces and classes provided by each level of DOM. Table 18−2 describes some of the interfaces; again, visit the W3C for a complete set.

Table 18−2: The DOM Core Interface

Interface	Description
DOMException	Captures the exceptions generated by DOM, such as when the implementation has become unstable; during processing methods usually return error values, such as out of bounds.
DOMImplementation	Interface for performing operations independent of any particular instance of the document object model. Each DOM level provides different support.
DocumentFragment	A "lightweight" or "minimal" Document object used to hold fragments of the document tree.
Document	Represents the entire HTML or XML document and provides the primary access to the document's data. Also contains the factory methods with which to create elements, text nodes, comments, processing instructions, and so on.
Node	The primary datatype for the entire DOM. It represents a single node in the document tree.

NodeList	Provides the abstraction of an ordered collection of nodes, without defining or constraining the way in which this collection is implemented.
NamedNodeMap	Objects implementing this interface represent collections of nodes that can be accessed by name; DOM does not specify or require an order for these nodes.
CharacterData	Extends Node with a set of attributes and methods for accessing character data in the DOM.
Attr	Represents an attribute in an Element object. The allowable values for the attribute are typically defined in a document type definition. Attr objects inherit the Node interface
Text	Represents the textual content (termed *character data* in XML) of an Element or Attr.
Comment	Represents the content of a comment.
Element	Element nodes are by far the vast majority of objects (apart from text) that authors encounter when traversing a document. They represent the elements of the XML document.

The steps necessary to use DOM can be generalized as follows:

1. Make sure that the DOM classes are in the class path.
2. Import the necessary DOM classes.
3. Import and instantiate the DOM parser explicitly.
4. Acquire the document tree (via a getDocument() call, for example).
5. To process the document tree, traverse it a node at a time using interfaces and classes provided by DOM.

Listing 18−5 shows the bookDescDOM class, which uses the preceding steps to parse an XML document example. I decided to use the Xerces parser version 1.4.3 from Apache (http://xml.apache.org/xerces−j/index.html); you will need this parser to run the example. The only processing I do is to print the node as I traverse the tree. I do not implement all the possible types of nodes; depending on your needs you may want to implement more types and also you may traverse the tree either preorder, postorder or inorder.

Listing 18−5: bookDescDOM.java

```
// the needed java classes
import java.io.*;

// Step 1. Import the necessary DOM classes.
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

// Step 2. Import the vendors DOM parser explicitly.
import org.apache.xerces.parsers.DOMParser;

/**
 * A simple example to show how to use the DOM API
*/
public class bookDescDOM {
```

```
public void processNode(Node node, String spacer)
    throws IOException {
  if(node == null) return;
    switch (node.getNodeType()) {
    //ELEMENT_NODE                  = 1;
    case Node.ELEMENT_NODE:
      String name = node.getNodeName();
      System.out.print(spacer + "<" + name);
      // NamedNodeMap is an array of name-value pairs
      NamedNodeMap nnm = node.getAttributes();
      for (int i=0; i<nnm.getLength(); i++) {
        Node current = nnm.item(i);
        System.out.print(" " + current.getNodeName()+"=
         "+current.getNodeValue());
      }
      System.out.print(">");
      // recurse on each child
      NodeList nl = node.getChildNodes();
      if (nl != null) {
        for (int i=0; i<nl.getLength(); i++) {
          processNode(nl.item(i), "");
        }
      }
      System.out.println(spacer + "</" + name + ">");
      break;
    //TEXT_NODE                     = 3;
    case Node.TEXT_NODE:
     System.out.print(node.getNodeValue());
     break;
    //CDATA_SECTION_NODE            = 4;
    case Node.CDATA_SECTION_NODE:
     System.out.print("<CDATA [["+ node.getNodeValue() +
         "]]>");
     break;
    //ENTITY_REFERENCE_NODE         = 5;
    case Node.ENTITY_REFERENCE_NODE:
      System.out.print("&" + node.getNodeName() + ";");
      break;
    //ENTITY_NODE                   = 6;
    case Node.ENTITY_NODE:
      System.out.print("<ENTITY: "+ node.getNodeName()+">
        </"+node.getNodeName()+"/>");
      break;
    //DOCUMENT_NODE                 = 9;
    case Node.DOCUMENT_NODE:
      // recurse on each child
      NodeList nodes = node.getChildNodes();
      if (nodes != null) {
        for (int i=0; i<nodes.getLength(); i++) {
          processNode(nodes.item(i), "");
        }
       }
      break;
    //DOCUMENT_TYPE_NODE            = 10;
    case Node.DOCUMENT_TYPE_NODE:
      DocumentType docType = (DocumentType)node;
      System.out.print("<!DOCTYPE " + docType.getName());
      if (docType.getPublicId() != null)  {
        System.out.print(" PUBLIC " + docType.getPublicId()
          + " ");
      } else {
```

```
        System.out.print(" SYSTEM ");
       }
       System.out.println(" " + docType.getSystemId() + ">");
       break;
     //ATTRIBUTE_NODE                 = 2;
     //PROCESSING_INSTRUCTION_NODE    = 7;
     //COMMENT_NODE                   = 8;
     //DOCUMENT_FRAGMENT_NODE         = 11;
     //NOTATION_NODE                  = 12;
     default:
       break;
     }
   }

// usage: java bookDescDOM bookDesc.xml
 public static void main(String[] args) {
   if (args.length != 1) {
     System.out.println("Usage: java bookDescDOM inputXMLfile");
     System.exit(0);
   }
   String uri = args[0];
   try {
     bookDescDOM bd = new bookDescDOM();
     System.out.println("Parsing XML File: " + uri + "\n\n");
     // Step 2: instantiate the vendors DOM parser
     // explicitly.
     DOMParser parser = new DOMParser();
     parser.setFeature ("http://xml.org/sax/features/validation", true);
     parser.setFeature ("http://xml.org/sax/features/namespaces", false);
     parser.parse(uri);
     // Step 3. Acquire the document tree; for example via a getDocument() call.
     Document doc = parser.getDocument();
     // Step 4. To process the document tree all we need to do is traverse
    // the tree a node at a time; using interfaces and classes provided by DOM.
    bd.processNode(doc, "");
   } catch (Exception e) {
     e.printStackTrace();
     System.out.println("Error: " + e.getMessage());
   }
  }
}
```

The following code is the output of this example; remember that the elements are printed as the DOM tree is traversed:

```
C:\>java bookDescDOM bookDesc.xml

Parsing XML File: d:\bookDesc.xml

<!DOCTYPE myspace:book SYSTEM  doc.dtd>
<myspace:book xmlns:myspace= http://www.ournamespace.com>
 <title>Alice’s Adventures in Wonderland and Through the Looking Glass</title>
 <author>
  <first−name>Lewis</first−name>
  <last−name>Carroll</last−name>
 </author>
 <date> May 2000 </date>
 <internal identifier= 12345></internal>
</myspace:book>
```

Usually you'll want to style the XML document in a certain form; this is the reason you use a style sheet that describes how to display a document of a given type. XSL is a language for expressing style sheets and consists of three parts: XSLT, a language for transforming XML documents; Xpath, an expression language used by XSLT to refer to parts of an XML document; and the XSL Formatting Objects (XSL−FO), which specifies formatting semantics. You can find more information about XSL at http://www.w3.org/Style/XSL/ I will only discuss XSLT in this chapter, as the other technologies are beyond the scope of this book.

Transforming XML: XSLT

The eXtensible Stylesheet Language for Transformations (XSLT), is a large and powerful language for transforming XML documents. It was originally intended for complex styling operations (such as generating an index of a document), but it has grown to be used as a general XML processing language. An XSLT style sheet is an XML document that takes a source tree and associates its elements. It may also reorganize or filter them to form a separate result tree.

XML documents can be transformed into HTML or any other kind of text, and can be sent to another application. For example, an XML document can be transformed to postscript to be read by a postscript reader. XML parsers work with XSLT processors to transform XML documents.

To make my transformation example more interesting I have created an invoice XML document that is described in Listing 18−6. The listing should look familiar: this DTD is the DTD that I derived at the beginning of the chapter in my discussion of XML modeling relational tables.

Listing 18−6: inv.dtd

```
<!ELEMENT Customer (id, name, email, street, city, state, zip)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT invoice (Name, title, invoiceid, Customer, item+)>
<!ELEMENT invoiceid (#PCDATA)>
<!ELEMENT item (itemid, qty, title, price)>
<!ELEMENT itemid (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT qty (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
```

I have created an XSLT document based on the preceding DTD, as shown in Listing 18−7, which transforms an XML document (constrained by the DTD) to an HTML document that can be displayed in a browser. I used Xalan−Java 2 from Apache, which you can find at http://xml.apache.org/. Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. As usual, you first have to set the classpath to use Xalan; then you can use the command line to run the example with a command like the following: java org.apache.xalan.xslt.Process −IN invoice.xml −XSL inv.xsl −OUT inv.html.

Listing 18−7: inv.xsl

```
<?xml version="1.0" encoding="UTF−8"?>
```

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="xml" version="1.0" encoding="UTF-8" omit-xml-declaration="no"
indent="no" media-type= "text/html"/>
   <xsl:template match="invoice">
      <html>
         <head><title><xsl:value-of select= "Name"/></title></head>
         <h1><center><xsl:value-of select= "Name"/></center></h1>
         <h2><center>
            <xsl:value-of select="title"/>
            <xsl:value-of select="invoiceid"/>
         </center></h2>
         <xsl:apply-templates select="Customer"/>
            <table>
               <tr>
                  <td>
                     <h4>Qty</h4>
                  </td>
                  <td>
                     <h4>Price</h4>
                  </td>
                  <td>
                     <h4>ID</h4>
                  </td>
                  <td>
                     <h4>Description</h4>
                  </td>
               </tr>
               <xsl:apply-templates select="item"/>
            </table>
      </html>
   </xsl:template>
   <xsl:template match = "Customer">
      <html>
         <h3><p>Customer: <xsl:value-of select= "name"/></p></h3>
         <p>e-mail: <xsl:value-of select="email"/></p>
         <tr>
            <td valign="top">
               <h4>Billing Address:</h4>
               <ul>
               <li><xsl:value-of select="street"/></li>
               <li><xsl:value-of select="city"/></li>
               <li><xsl:value-of select="state"/></li>
               <li><xsl:value-of select="zip"/></li>

               </ul>
            </td>
         </tr>
      </html>
   </xsl:template>
   <xsl:template match="item">
      <tr>
         <td>
            <xsl:value-of select="qty"/>
         </td>
         <td>
            <xsl:value-of select="price"/>
         </td>
         <td>
            <b>
               <a>
               <xsl:value-of select="itemid"/>
```

```
            </a>
         </b>
      </td>
      <td>
         <xsl:value-of select="title"/>
      </td>
      <td/>
   </tr>
</xsl:template>
</xsl:stylesheet>
```

<xsl:template><xsl:apply−templates/> <xsl:output>htmlxmltextxml<html>xhtmlxml

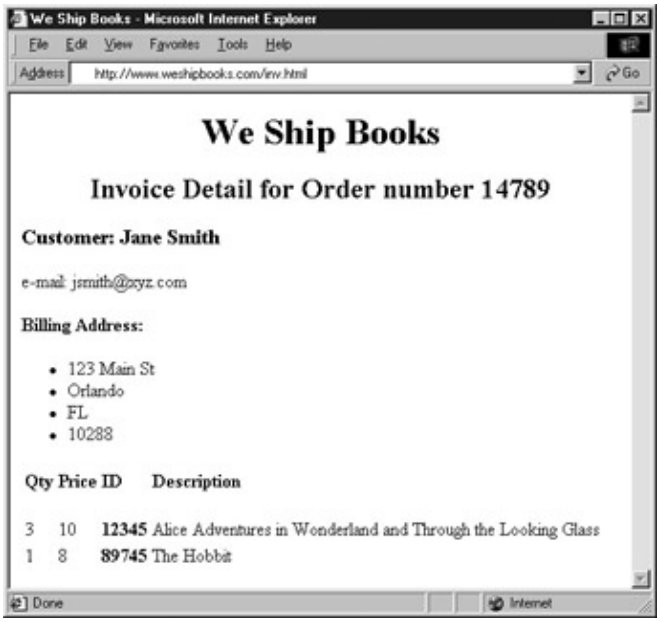

Figure 18−1: This is the page that results from the transformation in Listing 18−6.

Using the JAXP API

Sun’s Java API for XML Parsing (JAXP) provides a standardized set of Java Platform APIs with which to integrate any XML−compliant parser with Java applications. Crimson is the default parser provided by JAXP 1.1. The JAXP API supports processing of XML documents using the SAX (level 2), DOM (level 2), and XSLT APIs, but it does not replace them.

Where to get the JAXP API

JDK 1.1.8 and above have the JAXP 1.1 as an optional package. The JAXP 1.1 will be included in the Java 2 Platform Standard Edition (J2SE) 1.4 and in the Java 2 Platform Enterprise Edition (J2EE) 1.3. In addition, the JAXP 1.1 specification and reference implementation can be downloaded from Sun at http://java.sun.com/xml/download.html.

Using JAXP

The JAXP specification provides a plugability layer for SAX, DOM, and XSLT. The SAX plugability classes enable you to parse XML documents and implement the org.xml.sax.DefaultHandler API. The DOM plugability classes enable you to parse XML documents and obtain an org.w3c.dom.Document object. Finally, the XSLT plugability classes enable you to create an object, based on your XSLT style sheet, with which to transform an XML document. This enables Java developers to use a single API for loading, parsing, and applying style sheets to XML documents.

Table 18−3 describes the packages provided by the JAXP API.

Table 18−3: The JAXP API

JAXP API	Description
Package javax.xml.parsers	
DocumentBuilder	Provides the API with which to create a DOM Document instance from an XML document. The document can then be parsed from input sources such as input stream, files, URLs, and SAX. Methods include getDOMImplementation, newDocument, and parse.
DocumentBuilderFactory	A factory with which to obtain a parser that produces DOM object trees from XML documents. It is not guaranteed to be thread−safe. Methods include getAttribute, isValidating, and isNamespaceAware.
FactoryConfigurationError	Error thrown when the class−parser factory in the system properties cannot be found or instantiated. Methods include getException and getMessage.
ParserConfigurationException	Error thrown when serious configuration error occurs; implements java.io.Serializable.
SAXParser	Defines the API that wraps an instance of org.xml.sax.XMLReader. Use the newSAXParser method to obtain an instance; and then parse from input sources such as input stream, files, URLs, and SAX−input sources. Other methods include getXMLReader, parse, and setProperty.
SAXParserFactory	Obtains and configures a SAX parser for XML documents. Methods include getFeature, setValidating, and setNamespaceAware.
Package javax.xml.transform	
ErrorListener	Provides customized error handling. The application registers an instance of the implementation with the Transformer; if an application does not register an ErrorListener then the errors are reported to System.err. Methods are error, fatalError, and warning.

OutputKeys	Provides string constants with which to set and retrieve output properties for a Transformer. Constants include METHOD, DOCTYPE_PUBLIC, and VERSION.
Result	An object that implements this interface has the information it needs to build a transformation−result tree. Methods are getSystemId and setSystemId.
Source	An object that implements this interface has the information it needs to act as an XML source or transformation instructions. Methods are getSystemId and setSystemId.
SourceLocator	Reports that errors occurred in XML source or transformation instructions. Methods are getColumnNumber, getLineNumber, getPublicId, and getSystemId.
Templates	An object that implements this interface is the runtime representation of processed transformation instructions. Methods are getOutputProperties and newTransformer.
Transformer	An abstract class that transforms a source tree into a result tree; you can obtain an instance of this class by calling the newTransformer method that can be used to process XML. Methods include clearParameter, getOutputProperty, and setURIResolver.
TransformerConfigurationException	Indicates a serious configuration error, one preventing creation of a Transformer. This class extends java.io.Serializable.
TransformerException	Indicates an error in the transformation process. Methods include getCause, printStackTrace, and getLocationAsString.
TransformerFactory	Creates Transformer and Templates objects. Methods include getAssociatedStylesheet, getFeature, and newTemplates.
TransformerFactoryConfigurationError	This exception is thrown with an error in the Transformer Factories configuration—for example, when the transformation factory class in the javax.xml.transform.TransformerFactory system property cannot be found or instantiated. Methods are getException and getMessage.
URIResolver	Runs a URI specified in document(), xsl:import, or xsl:include into a Source object. It provides the member method resolve.
Package javax.xml.transform.dom	
DOMLocator	Extends SourceLocator and is used for error reporting. It indicates the position of a node in a DOM. The SourceLocator methods also contain the getOriginatingNode method.
DOMResult	A holder for a DOM result tree, this class allows an org.w3c.dom.Node to be specified. It implements the Result interface and provides the getNode, getSystemId,

	setNode, and setSystemId methods.
DOMSource	A holder for a DOM source tree; it allows the specification of a DOM org.w3c.dom.Node as the source of the input tree. It implements the Source interface and provides the getNode, getSystemId, setNode, and setSystemId methods.
Package javax.xml.transform.sax	
SAXResult	Allows the setting of a org.xml. sax.ContentHandler to be a recipient of SXA2 events from the transformation. It implements the Result interface. Methods include getHandler, getSystemId, setHandler, and setLexicalHandler.
SAXSource	Allows the setting of an org.xml. sax.XMLReader to be used for pulling parse events; it is a holder for a SAX−style Source and implements the Source interface. Methods include getInputSource, getSystemId, getXMLReader, and sourceToInputSource.
SAXTransformerFactory	Extends TransformerFactory to provide SAX−specific methods for creating TemplateHandler, TransformerHandler, and org.xml.sax.XMLReader instances.
TemplatesHandler	Extends org.xml.sax. ContentHadler to allow the creation of Templates objects from SAX2 parse events. Methods are getSystemId, getTemplates, and setSystemId.
TransformerHandler	Implements ContentHandler, DTDHandler, and LexicalHandler interfaces and listens for SAX ContentHandler parse events and transforms them to a Result. Also includes getSystemId, getTransformer, setResult, and setSystemId.
package javax.xml.transform.stream	
StreamResult	A holder for a transformation result, this class provides methods for specifying java.io.OutputStream and java.io.Writer or an output−system ID as the output of the transformation result. Methods include getOutputStream and getWriter.
StreamSource	A holder for transformation Source, this class provides methods for specifying inputs from java.io.InputStream, java.io.Reader, and URL input from strings. Methods include setPublicId, setReader, and setInputStream.

The following sections show the use of the JAXP API for parsing, traversing, and transforming XML documents.

Parsing XML with JAXP

You have various options when parsing an XML file using the JAXP API. The following steps describe using the JAXP utility classes and the SAXP API:

1. Create a SAXParserFactory object.

2. Configure the factory object.
3. Get the XMLReader; for this you need to create a SAX parser using the factory object.
4. Write the content handlermethods. For example, write the methods that will handle the beginning of an element and the end of the document by extending the DefaultHandler class from the org.xml.sax.helpers package.
5. Set the necessary handlers for your document. This includes the error handlers as well as the content handlers.
6. Use XMLReader to parse the document.

Listing 18−8 uses the preceding steps to create a parser using the SAX−plugability of the JAXP API. The most important similarity between this example and my pure SAX example is that all the handlers (startdocument, endDocument, startElement, endElement, and so on.) remain the same. This is because of the underlying use of the SAX API. In the following example I just list the main method because all the other methods stay the same. The obvious change caused by the JAXP API is the creation of the XMLReader via a SAX parser−factory instance. You can modify the main method from Listing 18−4 and run it via the command line: java jaxpBookParser bookDesc.xml.

Listing 18−8: jaxpBookParser.java

```
static public void main(String[] args) {
  boolean validation = false;
  if (args.length != 1) {
    System.out.println("Usage: java jaxpBookParser inputXMLfile");
    System.exit(0);
  }

 // Step 1. Create a SAXParserFactory object. SAXParserFactory spf =
SAXParserFactory.newInstance();
 // Step 2. Configure the factory object.
 spf.setValidating(true);
 XMLReader xmlReader = null;
 try {
   // Step 3. Get the XMLReader; for this you need to create a SAX parser
   // using the factory object.
   // Create a JAXP SAXParser
   SAXParser saxParser = spf.newSAXParser();
   // Get the SAX XMLReader
   xmlReader = saxParser.getXMLReader();
 } catch (Exception e) {
   System.err.println(e);
   System.exit(1);
}
 // Step 5. Set the necessary handlers for you document. This includes the error
handlers
 // as well as the content handlers.
 xmlReader.setContentHandler(new jaxpBookParser());
 xmlReader.setErrorHandler(new SAXErr(System.err));
 try {
   File name = new File(args[0]);
   String pt = name.toURL().toString();
   // Step 6. Use the XMLReader to parse the document.
   xmlReader.parse(pt);
 } catch (SAXException se) {
   System.err.println(se.getMessage());
   System.exit(1);
 } catch (IOException ioe) {
   System.err.println(ioe);
```

```
   System.exit(1);
  }
 }
```

The output is the same, as it should be:

```
Start of document
start element: namespace=http://www.ournamespace.com, localName=book,
qName=myspace:book
start element: namespace=, localName=title, qName=title
Characters:  "Alice's Adventures in Wonderland and Through the Looking Glass

endElement: namespace=, localName=title, qName=title
start element: namespace=, localName=author, qName=author
start element: namespace=, localName=first-name, qName=first-name
Characters:  "Lewis\
endElement: namespace=, localName=first-name, qName=first-name
start element: namespace=, localName=last-name, qName=last-name
Characters:  "Carroll\
endElement: namespace=, localName=last-name, qName=last-name
endElement: namespace=, localName=author, qName=author
start element: namespace=, localName=date, qName=date
Characters:  " May 2000 \
endElement: namespace=, localName=date, qName=date
start element: namespace=, localName=internal, qName=internal att[0] identifier=
'12345'
endElement: namespace=, localName=internal, qName=internal
endElement: namespace=http://www.ournamespace.com, localName=book, qName=myspace:book
End of document
```

Traversing XML with JAXP

You can follow these steps in order to read and traverse an XML file using the DOM API:

1. Create a DocumentBuilderFactory object.
2. Configure the factory object.
3. Create a DocumentBuilder using this factory.
4. Set the error handlers you need.
5. Parse the input file to get a Document object.

Now you can manipulate the Document object.

Listing 18-9 is an example of using the JAXP DOM implementation. (It is very similar to the pure DOM example.) The following listing shows the main method in which all the changes have taken place. Notice that I used a factory instance to obtain a document (via the document builder), which is then printed by my processNode method (the same method as in the previous DOM example). You can modify the main method from Listing 18-5 and run it via the command line: java jaxpBookDOM bookDesc.xml.

Listing 18-9: jaxpBookDOM.java

```
public static void main(String[] args) {
 if (args.length != 1) {
   System.out.println("Usage: java jaxpBookDOM inputXMLfile");
   System.exit(0);
 }
```

```
 String filename = args[0];
 // Step 1. Create a DocumentBuilderFactory object
 DocumentBuilderFactory dfactory= DocumentBuilderFactory.newInstance();

 // Step 2. now we can set configuration options and then we could save
 // the factory instance to be reused later w/ the same configuration.
 dfactory.setValidating(true);
 dfactory.setIgnoringComments(false);
 dfactory.setIgnoringElementContentWhitespace(false);
 dfactory.setCoalescing(false);
 dfactory.setExpandEntityReferences(true);
// Step 3. Create a DocumentBuilder using the above factory
 DocumentBuilder dbuilder = null;
 try {
   dbuilder = dfactory.newDocumentBuilder();
   // Step 4. Set the error handlers needed.
   OutputStreamWriter errorWriter =
    new OutputStreamWriter(System.err, outputEncoding);
   dbuilder.setErrorHandler(
    new DOMErr(new PrintWriter(errorWriter, true)));
 } catch (ParserConfigurationException pce) {
   System.err.println(pce);
   System.exit(1);
 } catch (IOException ioe) {
   System.err.println(ioe);
   System.exit(1);
 }
 Document doc = null;
 try {
   // Step 5. Parse the input file to get a Document object.
   System.out.println("Parsing XML File: " + filename + "\n\n");
   doc = dbuilder.parse(new File(filename));
   // Step 6. Now we can manipulate the Document object.
   jaxpBookDOM jbd = new jaxpBookDOM();
   jbd.processNode(doc, "");
 } catch (SAXException se) {
   System.err.println(se.getMessage());
   System.exit(1);
 } catch (IOException ioe) {
   System.err.println(ioe);
   System.exit(1);
 } catch (Exception e) {
   e.printStackTrace();
   System.out.println("Error: " + e.getMessage());
 }
}
```

Not surprisingly, running this example with bookDesc.xml as the input generates the same output as our previous DOM example:

```
Parsing XML File: d:\bookDesc.xml

<!DOCTYPE myspace:book SYSTEM  file:d:/doc.dtd>
<myspace:book xmlns:myspace= http://www.ournamespace.com>
 <title>Alice's Adventures in Wonderland and Through the Looking Glass</title>
<author>
   <first-name>Lewis</first-name>
   <last-name>Carroll</last-name>
 </author>
```

```
 <date> May 2000 </date>
 <internal identifier= 12345></internal>
</myspace:book>
```

Transforming XML with JAXP

You can transform an XML file using the JAXP API in many ways. The following steps describe how to use the JAXP Transformer classes with a style sheet:

1. Create a TransformerFactory object.
2. Create a transformation Source using StreamSource.
3. Create a transformation Result using StreamResult.
4. Instantiate a Transformer using TransformerFactory.
5. Use the Transformer specifying transformation Source and transformation Result.

My example of transforming an XML document using the JAXP API follows. In Listing 18−10 I used the preceding steps to create a transformer from the transformer factory. The arguments are the XML document, the XSL document, and the output file you want to create or override. The output generated from running jaxpTransform with the same parameters I used in the XSLT example (java jaxpTransform invoice.xml inv.sxl inv2.html) result in the same output, namely Figure 18−1.

Listing 18−10: jaxpTransform.java

```
// Import java classes
import java.*;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

// Import jaxp classes
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
/**
 * A simple example to show how to use the transform.TransformerFactory
 *  plugability in the JAXP API
 * @author Johennie Helton
 */
public class jaxpTransform {
  public static void main(String[] args)
    throws TransformerException, TransformerConfigurationException,
           FileNotFoundException, IOException
  {
   if (args.length != 3) {
     System.out.println("Usage: java jaxpTransform inXML inXSLT outFile");
     System.exit(0);
   }
    FileOutputStream outstream = new FileOutputStream(args[2]);

    // Step 1. Create a TransformerFactory object.
    TransformerFactory tfactory = TransformerFactory.newInstance();

    // Step 2. Create a transformation Sources using StreamSource.
    StreamSource insource = new StreamSource(args[0]);
```

```
    StreamSource inxsl = new StreamSource(args[1]);

    // Step 3. Create a transformation Result using a StreamResult.
    StreamResult sresult = new StreamResult(outstream);

    // Step 4. Instantiate a Transformer using the TransformerFactory.
    Transformer transformer = tfactory.newTransformer(inxsl);

    // Step 5. Use the Transformer specifying transformation Source
    // and transformation Result.
    transformer.transform(insource, sresult);
  }
}
```

Summary

In this chapter I have covered briefly topics that have whole books to themselves: XML, SAX, DOM, and JAXP. XML is a metalanguage defined by the W3C, a set of rules, guidelines and conventions for describing structured data. The namespace recommendation, DTDs, and XML Schemas provide means by which to constrain and specify the syntax and grammars of XML documents. One interesting application for XML technologies is to model, store, and transmit relational−database structures.

In addition, SAX, DOM, and XSLT provide you with ways to parse, traverse, and transform XML documents. SAX provides sequential, hierarchical access to XML documents. DOM is a clean and easy−to−use interface to XML documents, but it is also very memory−intensive. I provided a series of steps to help you develop with SAX, DOM, XSLT, and the JAXP API. Finally, the JAXP API provides Java developers and architects with a consistent means of manipulating XML with the technologies I mentioned in this chapter.

Chapter 19: Accessing Data with Enterprise JavaBeans

by Johennie Helton

In This Chapter

- Working with EJBs
- Introducing EJB classes and interfaces
- Understanding the EJB lifecycle
- Using the persistence options available for EJBs
- Exploring the benefits of data access and value objects
- Understanding transactions and the transaction attributes in EJBs
- Guidelines for working with EJBs

J2EE Enterprise JavaBeans (EJBs) are components that provide a model for developing server−side enterprise applications. They encapsulate a part of the application's business logic and make building portable and distributed enterprise applications easier than it would otherwise be. EJBs are intended to provide a data−access platform for your enterprise application.

I start with an introduction of the EJB tier and EJB in general, and then show you, with the aid of examples, the different EJB types and their parts, as well as the different options regarding data persistence. Finally I provide a set of guidelines that can help you avoid some common mistakes that developers make.

Working with the EJB Tier

As part of the J2EE standard architecture the Enterprise JavaBean tier provides the environment for the development, deployment, and life−cycle management of enterprise beans. It is in this tier that you find application−specific business objects and system services (such as security and transaction management).

Enterprise beans are components of enterprise applications that encapsulate a part of the business logic and typically communicate with resource managers such as database systems; application clients, servlets, and other client types also access the enterprise bean. An application may include one or more entity beans deployed on EJB containers. When the application includes multiple enterprise beans, deployment may occur either in a single container or in multiple containers located on the enterprise's network. It is this ability to deploy multiple entity beans on an enterprise's network that form the backbone of component−based distributed business applications.

The EJB container not only provides the deployment environment for enterprise beans, but also provides valuable services including security, concurrency, and transaction management. The EJB container is the place in which enterprise beans reside at runtime.

Enterprise JavaBeans are designed for data access and therefore almost everything that has to do with EJBs has to do with data access at one level or another. In this chapter I will discuss the more significant areas of the EJB tier, which make data access possible. For the purpose of this discussion suppose that you are developing a simplified online order application. Now assume that you have identified the need to model a

customer, an order, line items, and products. A simple class diagram may look like the one shown in Figure 19−1; a customer has an order, which may have zero or more line items that represent products.

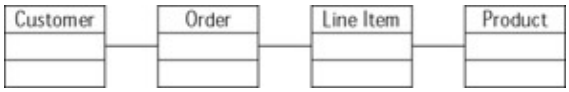

Figure 19−1: Class diagram for a simplified online ordering application

Enterprise bean types

Enterprise beans are essentially units of code that provide services to clients and encapsulate business logic and data. Enterprise beans come in three flavors: entity, session and message−driven beans. A *Message−Driven bean* is an asynchronous message consumer. *Entity beans* represent persistent records, such as those in a database. Another characteristic of entity beans is the modeling of real−world objects, such as customers. *Session beans*, on the contrary, do not represent persisted records; the information they contain is more transient. Note that session beans may access database information via a corresponding entity bean, or even access shared data directly. The differences between the two types of beans can be confusing but the defining characteristic from the Enterprise JavaBeans perspective is the persistent state — an entity bean has it, a session bean does not.

Session beans

Session beans are server−side components that model processes, services, and client sessions. Session beans can be either *stateless* or *stateful*. As their names suggest, the main difference between them is that the stateful session bean keeps its conversional state — or session data — between operations such as managing an order. A stateless session bean does not; stateless session beans model services that do not maintain session data for their clients. In this example, you may implement the Product component as a stateless session bean.

Stateless session beans Stateless session beans do have internal data but the values of those data are not kept for the client and do not represent a conversational state. Another important characteristic of this type of bean is that it does not survive EJB server crashes.

If the data values are required, and since the state is not kept, the client must provide the data values as parameters to the business−logic method of the bean. Instead of providing the data values as parameter, the bean may access a data repository, but then this access must be done every time the method is called.

The EJB container can pool a stateless session bean instance to dynamically assign instances of the bean to different clients. That is, when a stateless session method is called, the EJB container grabs an instance of that bean from a pool, the instance performs the service, and at the end of the method the bean is returned to the pool ready for another client. This design increases scalability by allowing the container to reassign the same bean to different clients without creating new instances. Of course, the specific way in which the reassignment is implemented is EJB container–specific. Figure 19−2 depicts two clients sharing a stateless session bean.

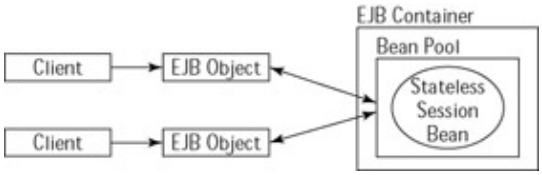

Figure 19−2: Clients can share the same stateless session bean from a pool.

Stateful session beans Stateful session beans keep the conversional state within the bean; so, the state is

maintained across multiple invocations of the same method as well as across different methods of the bean. Note that the state is specific to a given client. Like stateless session beans, stateful session beans do not survive EJB server crashes.

So far I have talked about conversional states, but what exactly is the conversional state anyway? In one sentence, the way Java does object serialization characterizes the conversional state. In general, this means that a member variable is part of the conversational state if it is a non−transient primitive type or non−transient Java object.

Like stateless session beans, stateful session beans are pooled; however, the process is more complex. When a method is called from a client, the client is establishing or continuing a conversation with the bean and the bean must remember the state that conversation is in. When the EJB container wants to return the stateful session bean instance to a pool it must *passivate* it. Passivation is the process in which the conversational state is saved to persistent storage, such as the hard disk, so that it can be retrieved the next time the same client requests a service. *Activation* is the opposite, the process in which a stateful session bean retrieves the conversational state to render the services requested by a client. Notice that the stateful session bean instance may not be the same instance to service the client's requests; however, from the client's perspective the instance behaves as if it were the previous instance because the state is retrieved from persistent storage. Figure 19−3 illustrates this process.

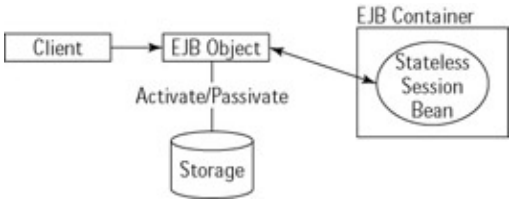

Figure 19−3: Stateful session beans activate and passivate their conversational states.

Entity beans

Following is a definition of an entity bean; keep in mind that from its creation to its destruction the entity bean lives in a container. The EJB container is transparent to the client but provides the security, concurrency, transactions, and other services to entity beans that the client perceives as being part of the entity bean.

Entity beans are server−side components that provide an object view of persisted data, so the state is synchronized to the data store. Entity beans are distributed, shared, transactional, multi−user, and long−lived persistent objects. Since they are representations of persistent storage they can survive EJB server crashes. Note that the entity bean contains a copy of the persistent data and so multiple clients can use multiple EJBs that represent the same underlying data.

The data can be persisted in two significant ways. The first is via *bean−managed persistence* (BMP), wherein the bean manages its own persistent state and its own relationships. The other type of persistence is known as *container−managed persistence* (CMP), wherein the EJB container manages the bean's persistent state and relationships. The section "Dealing with Data Persistence," later in this chapter, provides more details about BMP and CMP.

The EJB specification outlines an architecture in which the EJB container keeps a free pool of entity bean instances. The container moves these instances in and out of the free pool in much the same way that it swaps session beans to disk. The major difference between the two scenarios is that the container swaps session beans to disk to persist their state, whereas when it swaps entity beans to and from the free pool they do not represent a state. Figure 19−4 shows the connection between entity beans and persistent storage.

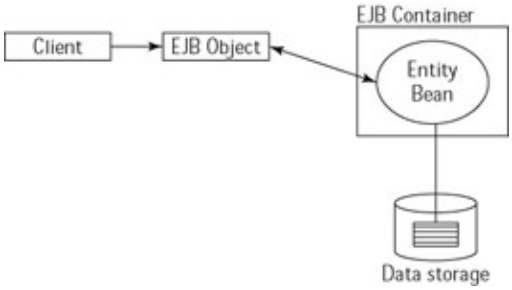

Figure 19−4: Entity beans are representations of persisted data.

Message−driven beans

Message−driven beans (MDBs) were introduced in the 2.0 specification to allow clients to asynchronously invoke server−side business logic. As of this writing they only handle JMS messages but they may be used to process other kinds of messages in the future. Message−driven beans work with queues and both durable and nondurable topic subscriptions. A queue uses point−to−point messaging with at most one consumer. A topic uses publish−and−subscribe messaging and may have zero or more consumers.

MDBs are generic JMS message consumers, to which clients may send asynchronous messages — by sending messages to the destination (of the message−driven bean). The destination represents the target of outgoing messages and the source of incoming messages. MDBs are stateless components, so they maintain no information about the client between calls.

The EJB container provides MDBs with services such as transactions, security, and concurrency, just as it does the other enterprise bean types. The container, when appropriate, handles the message acknowledgement. The container may pool MDB instances to improve server performance; all instances of an MDB are equivalent, so the container may assign a message to any MDB instance. An MDB instance does not retain the conversational state of its client and it can process messages from multiple clients.

The Parts of an EJB

To abstract the system−level details from the bean developer is one of the primary purposes of the EJB architecture: for this reason an Enterprise JavaBean has three parts. First is the enterprise−bean class, second is the enterprise−bean client−view, and third is the deployment descriptor, all of which are described in the next sections.

Enterprise−bean class

The enterprise−bean class specifies a contract or interface among the different parts involved in the deployment and use of the EJB. For session beans, the enterprise−bean class includes the business−related logic that the bean will perform: An example is checking the availability of an item. For entity beans, the enterprise−bean class has the data−related logic such as changing the address of a customer. The EJB container intercepts the calls from a client to an EJB — that is, the client does not talk directly to an EJB instance. The EJB container acts as a layer between the client and the bean−class instance; using vendor−specific code hooks and/or tools, the EJB container automatically generates an *EJB object* for your bean and uses it to delegate the method invocations to your bean.

The EJB specification defines the standard interfaces that beans must implement. The most basic interface is the javax.ejb.EnterpriseBean interface, which identifies your bean as an EJB. Your bean does not extend the javax.ejb. EnterpriseBean directly; your bean extends either the javax.ejb.SessionBean or the

javax.ejb.EntityBean interface, depending on its type. The javax.ejb. EnterpriseBean interface extends java.io.Serializable, and because the beans indirectly extend java.io.Serializable activation and pasivation are possible.

Enterprise−bean client−view API

The enterprise−bean client−view API defines the home interface and the remote interface. Unlike session and entity beans, message−driven beans do not have the remote or local interfaces.

Remember that the client talks to an EJB object rather than to a bean interface: This is possible because the methods exposed by your bean are also exposed in a *remote interface*. Remote interfaces must derive from javax.ejb.EJBObject, which contains several methods to be implemented by the EJB object; note that these methods are automatically generated by the EJB container via the tools/code hooks that the vendor provides. The methods exposed by your bean are also in the remote interface, but they simply delegate to their corresponding beans method. In addition, the parameters for the methods must comply with Java RMI conventions, because they are to be passed through the network making location transparency possible. The client doesn't know whether the call originated in the local machine or across the network.

However, the client cannot acquire a reference to an EJB object directly. The client uses the *home object* to ask for an EJB object. The home object creates, destroys, and finds EJB objects. Home objects are vendor−specific and are automatically generated by the EJB−container vendor via tools. The EJB container−vendor generates home objects based on a *home interface*, which you define as part of your bean development. Home interfaces define how to create, destroy, and find EJB objects. The container implements the home interface in the home object. The specification defines the javax.ejb.EJBHome interface from which all home interfaces must derive. Figure 19−5 describes these relationships.

Most bean developers adhere to a few naming conventions, and in the following examples I will too. You are not bound by these conventions; still, it makes your code easier to follow, understand, and maintain if you adhere to them.

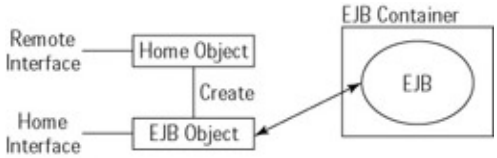

Figure 19−5: The container uses the home interface and object, and the remote interface and object.

One convention specifies that the name of the class should be a description of the function of the bean followed by the word *Bean*. Similarly, the name of the home interface should be the same description followed by the word *Home*. The name of the remote interface is just the description, without any suffix, and finally, the name of the whole enterprise bean is the description followed by *EJB*.

Deployment descriptor

The deployment descriptor is an XML document that describes the enterprise bean and contains the necessary entries that customize the environment in which the bean will reside. These parameters specify the services that your bean will obtain, without embedding them in the code. Specifying these services with parameters, instead of embedding them in code, give great flexibility with regard to the development and deployment of J2EE applications, because you can make configuration changes and dependency changes as needed.

The EJB container uses the deployment−descriptor information (its name, its type, runtime optimizations, and

so on) to know how to manage your bean and to know your bean's transaction requirements and security requirements (who is allowed to access the bean or even a method in the bean). In addition, for entity beans, the deployment−descriptor includes the type of persistence (container− managed or bean−managed) to be used.

Introducing EJB Classes and Interfaces

EJBs have their own classes and interfaces; some are required, others are optional or do not require logic. The following sections describe the session beans, and entity beans classes and interfaces. We also discuss the message−driven beans class; MDBs are not accessed via interfaces.

Session beans

As I mentioned earlier, session beans extend the javax.ejb.SessionBean interface; sometimes you can simply have empty implementations of the methods required by this interface, and sometimes you must provide some logic in the methods. The methods of this interface are described in Table 19−1.

Table 19−1: Session Bean Interface

Method	Description
setSessionContext (SessionContext ctx)	Called when the container associates the bean with a session context. The session context is usually used to request current information about your bean from the container, such as the current security state of the bean, and current transaction information. The context interface is javax.ejb.SessionContext.
ejbCreate(args)	Initializes your bean. Clients may use several methods arguments to initialize the bean in different ways. (Required.)
ejbPassivate()	Called by the container right before your bean is passivated. It should release with different the resources used by your bean in order to make them available for other instances.
ejbActivate()	As you have seen, after passivation an instance may be activated. The EJB container calls ejbActivate right after activation so you can acquire any necessary resources (such as all the ones you released with passivation).
ejbRemove()	Called right before the EJB container removes your bean instance. Every bean must have this method, and only one (with no parameters to it) should exist.
Business methods	Comprised of all the business methods you need to expose in your bean. Zero or more business methods may exist in your bean.

Using the on−line order application example, suppose that you have decided to implement the application with EJBs and to represent line items as stateful session beans. The LineItemEJB will therefore contain the

following:

- The product information, such as the UPC, description, price, and so on.
- The quantity of the product that the customer wants to purchase.
- Any discount that the customer may get for the product.

Given all this, the source code should look like the following LineItem.java (Listing 19−2), LineItemBean.java (Listing 19−1), and LineItemHome.java source−code examples.

The enterprise−bean class

The LineItemBean.java file contains the LineItemBean class that extends SessionBean (see Listing 19−1). It represents an individual order for a line item, which a customer is placing but has not committed to yet. This session bean has getter and setter methods as well as the required EJB methods. The conversational state consists of the quantity to be ordered, the discount the user gets, the order (described by the unique order ID) to which this line item belongs, and the product information that this line item represents.

Listing 19−1: LineItemBean.java

```
import java.rmi.*;
import javax.ejb.*;
import javax.naming.*;
import java.util.*;
import java.rmi.RemoteException;

public class LineItemBean implements SessionBean {
 protected SessionContext ctx;
 // conversational state fields
 // the current quantity of the order
 private int currentQty = 0;

 // the order this product belongs to
 private String orderId = null;

 // any discount the customer gets
 private double discount = 0;

 // all the info regarding this product such as basePrice, UPC and so on.
 private Product productInfo;

public LineItemBean() {
  System.out.println("LineItemBean is created by EJB container.");
 }

 //required EJB methods
public void ejbActivate() throws RemoteException {
  System.out.println("LineItem:ejbActivate()");
 }

 public void ejbCreate(String orderId, Product productInfo)
   throws CreateException, RemoteException {
  System.out.println("LineItemBean:ejbCreate(..,..)");
  this.orderId  = orderId;
  this.productInfo = productInfo;
 }

public void ejbCreate(
```

```
   String orderId, Product productInfo, double discount)
    throws CreateException, RemoteException {
  System.out.println("LineItemBean:ejbCreate(..,..,..)");
  this.orderId  = orderId;
  this.productInfo = productInfo;
  this.discount = discount;
 }

 public void ejbPassivate() throws RemoteException {
  System.out.println("LineItemBean:ejbPassivate()");
 }

 public void ejbRemove() throws RemoteException {
  System.out.println("LineItemBean:ejbRemove()");
 }

// associate the instance with a context
 public void setSessionContext(SessionContext ctx) throws RemoteException {
  System.out.println("LineItemBean:setSessionContex()");
  this.ctx = ctx;
 }

// public business methods
public Product getProduct() throws RemoteException {
  System.out.println("LineItemBean:getProduct()");
  return productInfo;
 }

 public double getPrice() throws RemoteException {
  System.out.println("LineItemBean:getPrice()");
  return (currentQty * productInfo.basePrice()) − discount;
 }

 public double getDiscount() throws RemoteException {
  System.out.println("LineItemBean:getDiscount()");
  return discount;
 }

 public int getQty() throws RemoteException {
  System.out.println("LineItemBean:getQty()");
  return currentQty;
 }

public void setDiscount(double discount) throws RemoteException {
  System.out.println("LineItemBean:setDiscount()");
  this.discount = discount;
 }

 public void setQty(int qty) throws RemoteException {
  System.out.println("LineItemBean:setQty()");
  this.currentQty = qty;
 }
}
```

The enterprise−bean client−view API

As I mentioned before, the enterprise−bean client−view API consists of the remote and home interfaces. According to convention, these interfaces are contained in the LineItem.java and LineItemHome.java files, respectively.

The remote interface The LineItem.java file (see Listing 19−2) contains the LineItem remote interface, which extends the Remote and EJBObject interfaces. This is the interface used by the EJB container to generate the EJB object that describes the LineItem session bean. It is a simple remote interface that exposes the getter and setter methods of the bean. Note that no setProduct exists; the ejbCreate method accepts a Product as a parameter, and so there is no need for a setProduct.

Listing 19−2: LineItem.java

```
import java.rmi.*;
import javax.ejb.*;

public interface LineItem extends Remote, EJBObject {
 Product getProduct() throws RemoteException;
 double getPrice() throws RemoteException;
 double getDiscount() throws RemoteException;
 int getQty() throws RemoteException;
 void setDiscount(double discount) throws RemoteException;
 void setQty(int qty) throws RemoteException;
}
```

The home interface The LineItemHome.java file (see Listing 19−3) contains the line item interface that extends the EJBHome interface. A client calls the home interface to create and destroy LineItemEJB objects. Note the two create methods: They represent the two ejbCreate methods in the LineItemBean class.

Listing 19−3: LineItemHome.java

```
import java.rmi.*;
import javax.ejb.*;

public interface LineItemHome extends EJBHome {
public LineItem create(String orderId, Product productInfo)
  throws CreateException, RemoteException;
public LineItem create(
  String orderId, Product productInfo, double discount)
   throws CreateException, RemoteException;
}
```

The deployment descriptor

The deployment descriptor for the LineItem bean should describe the attributes you need. For example, for the LineItemEJB you have the names for the bean home, home interface, and remote interface in the following ejb−jar.xml file. Note that some parameters (such as transaction−isolation level, transaction attributes, session timeouts and so on) may be entered via deployment tools provided by the vendor.

```
<?xml version="1.0"?>
<!DOCTYPE ejb−jar PUBLIC ‘−//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN’ ‘http://java.sun.com/j2ee/dtds/ejb−jar_1_1.dtd’>
<ejb−jar>
 <enterprise−beans>
  <session>
   <ejb−name>statefulSession</ejb−name>
   <home>jda.ch19.statefulSession.LineItemHome</home>
   <remote>jda.ch19.statefulSession.LineItem</remote>
   <ejb−class>jda.ch19.statefulSession.LineItemBean</ejb−class>
```

```
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>
 <assembly-descriptor>
  <container-transaction>
   <method>
    <ejb-name>statefulSession</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>*</method-name>
   </method>
   <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

Entity beans

Entity beans are Enterprise JavaBeans and therefore have an enterprise−bean class, a remote and home interface, and a deployment descriptor. Also, recall that this type of bean must extend javax.ejb.EntityBean. For entity beans there is a primary key and a corresponding primary−key class.

The primary key, which can be a single attribute or a composite of different attributes, differentiates entity beans. When the primary key is a single primitive type, the implementation of a primary−key class is not necessary. Otherwise you must have a primary−key class that implements java.lang.Object methods, such as equals() and hashCode().

The entity bean interface is similar to that of the session bean. However, some differences do exist. The methods are described in Table 19−2.

Table 19−2: Entity Bean Interface Methods

Method	Description
SetEntityContext (EntityContext ctx)	Called when the container associates the bean with a session context. The session context is used to request current information about your bean from the container, such as the current security state of the bean and the primary key. The context interface is javax.ejb.EntityContext.
unsetEntityContext()	Removes the association to a context.
ejbFind(..)	Finder methods are used to find data; they do not create new data. You must follow a couple of rules when using these methods: their names start with ejbFind, and there must be at least one called ejbFindByPrimaryKey. You implement these methods only with bean−managed persistence.
ejbCreate(...)	These methods initialize your bean; they may be overloaded. They also create data in the underlying storage. (Optional.)
ejbRemove()	The opposite of ejbCreate(...) removes data from storage and gets the primary key to uniquely identify which data to remove

	from storage. This method is unique and has no parameters. EJB container calls this method before data are removed from the database — the in−memory instance is not deleted; it is this instance that the container may add to the free pool. (Required.)
ejbPassivate()	Called by the container right before your bean is passivated. It should release the resources used by your bean in order to make them available for other instances.
ejbActivate()	As you have seen, after passivation an instance may be activated. The EJB container calls ejbActivate right after activation so you can acquire any necessary resources (such as all the ones you released with passivation).
ejbLoad()	Called when the container calls ejbLoad and the bean uses the primary key to identify what data to read from storage.
ejbStore()	Called when the container calls ejbStore and the bean uses the current state to save the data to storage.
Business methods	Comprised of all the business methods that you need to expose in your bean. Zero or more business methods may exist in your bean.

Continuing with the entity bean example, suppose that you decide to model the customer component as an entity bean. Your customer has the following characteristics:

- Customer name
- Customer address
- Customer ID
- Customer password

The enterprise−bean class

The CustomerBean.java file contains the CustomerBean class that describes your entity bean. For simplicity I use container−managed persistence in this example, and so some of the EJB−required methods are not implemented because the EJB container performs the persistent operations as well as the finder methods (see Listing 19−4).

Listing 19−4: CustomerBean.java

```
import java.io.Serializable;
import java.util.Enumeration;
import java.util.Vector;
import javax.ejb.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.rmi.*;
import java.util.*;

public class CustomerBean implements EntityBean {
protected EntityContext context;

 // flag to determine whether or not the bean needs to be written to storage
 private transient boolean isDirty;
```

```
 // container managed field the customer identification.  It is the primary key.
 public String CustomerId;

 // container managed field part of the customer information
 public String name;

 // container managed field part of the customer information
 public String city;

 // container managed field part of the customer information
 public String state;

 // container managed field part of the customer information
 public String zip;

 // container managed field part of the customer information
 public String area;

 // container managed field part of the customer information
 public String phone;

 // container managed field part of the customer information
 public String address;

 // container managed field part of the customer information
 // let's assume you don't care to keep the password secret
 public String password;

 // constructor
public CustomerBean() {
  System.out.println("CustomerBean is created by EJB container.");
 }

 // EJB required methods
 // called after activation by the EJB container
 public void ejbActivate() throws RemoteException {
  System.out.println("CustomerBean:ejbActivate()");
 }

/**
 * When the Home Object's create() is called, the Home Object calls
 * this method.  Populate the attributes so that the container can
 * create the database rows needed.
 */
 public String ejbCreate(
   String name, String addr, String password, String id)
    throws RemoteException {
  System.out.println("CustomerBean: ejbCreate ()");
  this.name = name;
  setAddress(addr);
  this.password   = password;
  this.CustomerId = id;
  return this.CustomerId;
 }

 /**
 * updates the object with data from the database, but you are using CMP
 * so you do not need it. Just do the necessary post−processing.
 */
 public void ejbLoad() throws RemoteException {
  System.out.println("CustomerBean:ejbLoad()");
```

```
 }

 /**
 * called before passivation by the EJB container
 */
 public void ejbPassivate() throws RemoteException {
  System.out.println("CustomerBean:ejbPassivate()");
 }

 /**
 * called after ejbCreate. The instance has been associated with
 * an EJB object and you can get a reference via the context if you
 * need to.
 */
 public void ejbPostCreate(
   String name, String address, String password, String id)
    throws RemoteException {
  System.out.println("CustomerBean:ejbPostCreate()");
 }

 /**
 * The container calls this method before removing
* entity information from the database.
 */
 public void ejbRemove() throws RemoteException {
  System.out.println("CustomerBean:ejbRemove()");
 }

 /**
 * updates the database, but you are using CMP so you do not need it.
 * Just do preprocessing needed.
 */
 public void ejbStore() throws RemoteException {
  System.out.println("CustomerBean:ejbStore( " + CustomerId + ")");
  setModified(false);
 }

 /**
 * keep the context so you can access it later if necessary
 */
 public void setEntityContext(EntityContext ecxt) throws RemoteException {
  System.out.println("CustomerBean:setEntityContext()");
  context = ecxt;
 }

 /**
 * disassociate the context
 */
 public void unsetEntityContext() throws RemoteException {
  System.out.println("CustomerBean:unsetEntityContext()");
  context = null;
 }

// business methods
public String getName() throws RemoteException {
  System.out.println("CustomerBean:getName()");
  return name;
 }

public String getPassword() throws RemoteException {
  System.out.println("CustomerBean:getPassword()");
```

```
  return password;
 }

 public String getAddress() throws RemoteException {
  System.out.println("CustomerBean:getAddress()");
  return address;
 }

 public void setName(String name) throws RemoteException {
  System.out.println("CustomerBean:setName()");
  setModified(true);
  this.name = name;
 }

 public void setPassword(String password) throws RemoteException {
  System.out.println("CustomerBean:setPassword()");
  setModified(true);
  this.password = password;
 }

 public void setAddress(String addr) throws RemoteException {
  System.out.println("CustomerBean:setAddress()");
  setModified(true);
  // address = Street, City, State, zip, area code = xxx, phone = xxxxxxx
  int first = addr.indexOf(',');
  int scnd  = addr.indexOf(',', first + 1);
  int thrd  = addr.indexOf(',', scnd + 1);
  int frth  = addr.indexOf(',', thrd + 1);
  int fth   = addr.indexOf(',', frth + 1);
  System.out.println("address: index of , at " + first + " " + scnd
         + " " + thrd + " " + frth + " " + fth);
  this.address = addr.substring(0, first - 1);
  this.city = addr.substring(first + 1, first + 3);
  this.state   = addr.substring(scnd + 1, scnd + 3);
  this.zip  = addr.substring(thrd + 1, thrd + 6);
  this.area = addr.substring(frth + 1, frth + 4);
  this.phone   = addr.substring(fth + 1, addr.length());
  System.out.println("addr->" + this.address + " city->" + this.city
         + " state->" + this.state + " zip->" + this.zip
         + " area->" + this.area + " phone->" + this.phone);
 }

public boolean isModified() {
  System.out.println("isModified(): isDirty = " + (isDirty
              ? "true"
              : "false"));
  return isDirty;
}

 public void setModified(boolean flag) {
  isDirty = flag;
  System.out.println("setModified(): " + CustomerId + (String) (flag
                 ? ": requires saving" : ": saving not required"));
 }
}
```

The enterprise−bean client−view API

As with any other type of enterprise beans, the remote interface and the home interface are defined so that the EJB container can generate the EJB Object and the home Object. According to the convention, the next two sections should describe the Customer.java and CustomerHome.java files.

The remote interface The container uses the remote interface to generate the EJB object that the client uses. The Customer.java file (see Listing 19−5) contains the getter and setter methods that you want to expose to the clients of the EJB.

Listing 19−5: Customer.java

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface Customer extends EJBObject {
 // business methods
public String getName() throws RemoteException;
public String getPassword() throws RemoteException;
public String getAddress() throws RemoteException;
public void setName(String name) throws RemoteException;
public void setPassword(String password) throws RemoteException;
public void setAddress(String addr) throws RemoteException;
}
```

The home interface The home interface defines the home object used to create and destroy your EJB Objects. The EJB container implements the finder methods; you can customize them using tools provided by the vendor; the CustomerHome.java file (see Listing 19−6) illustrates a home interface.

Listing 19−6: CustomerHome.java

```
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Enumeration;

public interface CustomerHome extends EJBHome {
 public Customer create(
  String name, String addr, String password, String id)
   throws CreateException, RemoteException;

 public Customer findByPrimaryKey(String id)
  throws RemoteException, FinderException;
}
```

The primary−key class

The primary key can be a single attribute or a composite of different attributes when the primary key is a single primitive type the implementation of a primary−key class is not necessary. It is important to realize that the primary key should be a subset of the fields managed by the CMP. In this case you should select from name, ID, address, and password; for the purpose of this example ID is enough, and it’s the same as the ID defined in CustomerBean; since String is a primitive class a primary key class is not needed.

The deployment descriptor

The ejb−jar.xml file is an example of a customer's deployment descriptor. You define the EJB name, home, remote−interface names, the primary−key class, the persistence type, and the attributes that form the set of container−managed fields. You use the vendor's tools to specify the JNDI name and other attributes necessary for deploying the bean.

```
<?xml version="1.0"?>
<!DOCTYPE ejb−jar PUBLIC '−//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb−jar_1_1.dtd'>
<ejb−jar>
  <enterprise−beans>
    <entity>
      <ejb−name>containerManaged</ejb−name>
      <home>jda.ch19.containerManaged.CustomerHome</home>
      <remote>jda.ch19.containerManaged.Customer</remote>
      <ejb−class>jda.ch19.containerManaged.CustomerBean</ejb−class>
      <persistence−type>Container</persistence−type>
      <prim−key−class>java.lang.String</prim−key−class>
      <reentrant>False</reentrant>
      <cmp−field>
        <field−name>CustomerId</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>name</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>address</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>city</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>state</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>zip</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>area</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>phone</field−name>
      </cmp−field>
      <cmp−field>
        <field−name>password</field−name>
      </cmp−field>
      <primkey−field>CustomerId</primkey−field>
    </entity>
  </enterprise−beans>
  <assembly−descriptor>
    <container−transaction>
      <method>
        <ejb−name>containerManaged</ejb−name>
        <method−intf>Remote</method−intf>
        <method−name>*</method−name>
      </method>
      <trans−attribute>Required</trans−attribute>
    </container−transaction>
  </assembly−descriptor>
```

```
</ejb-jar>
```

Message−Driven beans

Unlike session and entity beans message−driven beans are accessed through a bean class; not via interfaces. Although message−driven beans do not have business methods, they may contain helper methods that are invoked internally by the onMessage method. Here are the requirements to an MDB implementation:

- The MDB class is defined as public and is not abstract or final.
- The MDB class has a public constructor with no arguments.
- The MDB class implements the MessageDrivenBean and the MessageListener interfaces.
- The MDB class implements one onMessage, one ejbCreate, and one ejbRemove.
- The MDB class must not define the finalize method.

The container invokes the onMessage method when the queue receives a message; the onMessage method performs any logic needed (like parsing the message and displaying its contents). The ejbCreate and ejbRemove methods must be public, do not return a value (they are void), they cannot be static nor final, and do not have arguments.

Deploying MDBs you need to specify its type and the type of transaction management you want (either BMP or CMP). Also you specify the MDB characteristics such as: topic or queue, the JNDI name of the destination (of the MDB), and if you specified BMP you need to specify either the auto−acknowledge or the duplicates−ok option.

Understanding the EJB Life Cycle

The EJB specification defines the life cycle for session beans and entity beans. I am going to skip many of the details; refer to the specification for a more complete life−cycle description.

Session beans

A stateless session bean can be in either of two possible states:

- It does not exist.
- It is in a pooled state.

Figure 19−6 shows these two states. The stateless−bean life cycle starts when the container creates the instance via a call to the newInstance method, and the container then calls the setEntityContext method. The instance enters its own pool of available instances, in which it is ready to accept calls to the exposed business methods. When the container no longer needs the instance, it calls the ejbRemove and the life cycle ends.

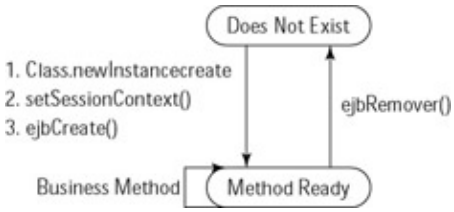

Figure 19−6: The stateless session bean life cycle

A stateful session bean can be in one of four possible states:

- It does not exist.
- It is in a non−transaction method–ready state.
- It is in a transaction method ready state.
- It is in a passive state.

Figure 19−7 describes these states. The transaction method ready state is depicted as “Tx Method Ready.” As with stateless session beans, when a bean instance is in the “does not exist” state, the stateful bean has not been instantiated. Once the client calls create on the home interface, the container calls the newInstance, setSession, and ejbCreate to create the instance.

Then the bean instance is free to receive method invocations from the client. At the time of deployment you identify the bean methods as being part of a transaction or not. If the method invoked is part of a transaction, the bean is in the transaction method−ready state and the method of transition out of this state depends on the transaction (such as completion or rollback). Also, a bean cannot be passivated in this state.

During its lifetime, a stateful session bean can be passivated (zero or more times), it is at this point that the bean instance enters the passive state. The instance returns to the “does not exist” state if the client calls the remove method or the container chooses to remove the bean.

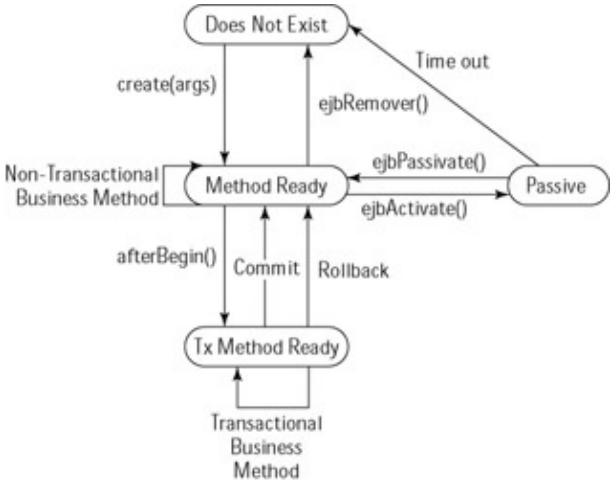

Figure 19−7: Stateful session bean life cycle

Entity beans

An entity bean instance can be in one of three states:

- It does not exist.
- It is in a pooled state.
- It is in a ready state.

These states are shown in Figure 19−8. As with session beans, the entity bean life cycle starts when the container creates the instance via a call to the newInstance method. The container then calls the setEntityContext method. The instance enters its own pool of available instances, and it is considered to be in the pooled state. All instances in the pooled state are considered equal; hence, the container may ask any instance to service a call of finder methods. Then, when the container calls either the ejbCreate or the ejbPostCreate method, the instance goes from the pooled state to the ready state. If no beans are in the ready

state the container may activate an instance from the pooled state to service a request from a client. Passivation and activation of entity beans move the instance from and to the pooled state, respectively. When the bean is in the ready state, it services its business methods; finally, the container may invoke the ejbLoad and ejbStore methods when the instance is in the ready state and then it must synchronize the instance state. Recall that when the instance is in the pooled state it is not associated with an entity object.

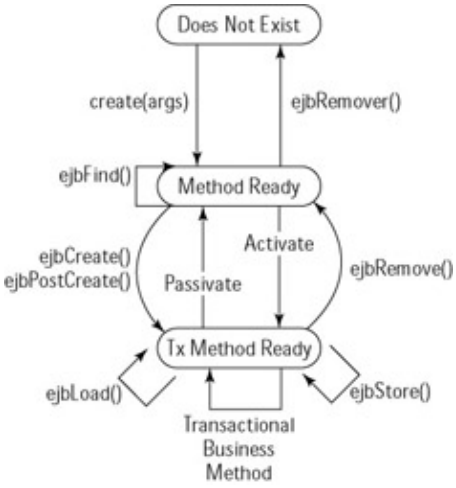

Figure 19−8: Entity bean life cycle

Message−Driven beans

An MDB can be in either of two possible states:

- It does not exist.
- It is in a ready state.

To create a new instance, the container instantiates the bean and calls the setMessageDrivenContext method to set the context object and then calls the ejbCreate method. The instance is in the ready state to process messages. The bean goes back to the does not exist state after the ejbRemove method is invoked. Finally, invocations of the onMessage method are serviced in the ready state.

Dealing with Data Persistence

An enterprise solution has one or more components. Each of the components provides specific behavior that is represented by an object and its attributes. The internal representation of these attributes forms the state of the object, which needs to be persisted. Data persistence is achieved via object serialization, container− managed persistence, or bean−managed persistence.

Object serialization

Java object−serialization mechanisms are designed to make it easy for simple beans to implement persistence. The java.io package defines java.io.ObjectOutput

and java.io.ObjectInput to save and retrieve objects and primitive data types, using any of input and output. The most common method is using a file stream by using an instance of java.io.FileOutputStream to save data

to a file and retrieving the data via an instance of java.io.FileInputStream. Streams are responsible for storing and restoring objects and primitive data types such as int.

Objects that implement the java.io.Serializable interface can have their states saved and restored. Implementing Serializable represents a promise that the class can be successfully saved and restored. Note that static data members are not serialized automatically; you need to provide call−specific serialization. Also note that final static members do not have to be saved during serialization and that the standard object−serialization mechanism will not save and restore transient members. If you need to save a transient member, you can use class−specific serialization by implementing writeObject and readObject to control the reconstruction of the object.

Objects can also implement java.io.Externalizable to save and restore their data. Externalizable objects control their own serialization; to save the data the object provides the writeExternal method and to restore the data the object provides the readExternal method.

Managed persistence

As I have discussed, you can persist data using entity beans in two ways: by using container−managed persistence (CMP) or by using bean−managed persistence (BMP).

Container−managed persistence

A container−managed persistence entity object does not contain the database calls. Instead, the EJB container generates the database−access calls at deployment time. This makes the entity bean independent of the data−store type and schema. The Customer example code (see Listing 19−4 through 6) used CMP and therefore no data−access code is embedded in it. The deployment descriptor identifies the variables to be persisted by the EJB container.

Also, specification 2.0 allows multiple entity beans to have container−managed relationships. The EJB container automatically manages these relationships and their referential integrity. These relationships may be one−to−one or one−to−many, using Java collections to represent the “many” part. At deployment time, variables are identified as either container−managed persistent fields or container−managed relationship fields. The developer defines getter and setter methods to access the bean instances.

Bean−managed persistence

When an entity bean uses BMP, the bean writes all the database access calls (using JDBC, for example). The ejbCreate, ejbRemove, ejbLoad and ejbStore methods, along with the finder methods, contain the data−access calls. This makes the bean dependent on the storage type and schema.

Continuing with the example, if you change the CustomerBean to use BMP, then the file that will need the most changes is the CustomerBean.java file, as shown in Listing 19−7.

Listing 19−7: CustomerBMPBean.java

```
import javax.ejb.*;
import java.io.*;
import java.rmi.*;
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
```

```
public class CustomerBMPBean implements EntityBean {
  final static boolean  VERBOSE = true;
  protected EntityContext context;

  // the customer identification.  It is the primary key.
  public int id;

  // the customer name
  public String name;

  // the  customer address
  public String address;

  // assume you do not care to keep the password secret
  public String password;

  public void ejbActivate() {
    System.out.println("CustomerBean:ejbActivate()");
  }

  public CustomerPK ejbCreate(
    String name, String address, String password, int id)
      throws CreateException {
    System.out.println("CustomerBMPBean: ejbCreate ()");
    this.name   = name;
    this.address  = address;
    this.password = password;
    this.id   = id;
    Connection    con = null;
    PreparedStatement ps  = null;
    try {
    con = getConnection();
    ps  = con.prepareStatement(
      "insert into Customer (name, address, password, id) values (?, ?, ?, ?)");
    ps.setString(1, name);
    ps.setString(2, address);
    ps.setString(3, password);
    ps.setInt(4, id);
    if (ps.executeUpdate() != 1) {
      throw new CreateException("JDBC did not create the row");
    }
    CustomerBMPPK primaryKey = new CustomerBMPPK(id);
    return primaryKey;
    } catch (CreateException ce) {
    throw ce;
    } catch (SQLException sqe) {
    throw new CreateException(sqe.getMessage());
    } finally {
    try {
      ps.close();
      con.close();
    } catch (Exception ignore) {}
    }
  }

  public BidBMPPK ejbFindByPrimaryKey(CustomerBMPPK pk)
    throws FinderException {
    if (pk == null) {
    throw new FinderException("primary key cannot be null");
    }
    refresh(pk);
```

```
  return pk;
}

public void ejbLoad() throws EJBException {
  System.out.println("CustomerBMPBean:ejbLoad()");
  try {
  refresh((CustomerBMPPK) context.getPrimaryKey());
  } catch (FinderException fe) {
  throw new EJBException(fe.getMessage());
  }
}

public void ejbPassivate() {
  System.out.println("CustomerBMPBean:ejbPassivate()");
}

public void ejbPostCreate(String name, String address, String password,
        int id) {
  System.out.println("CustomerBMPBean:ejbPostCreate()");
}

public void ejbRemove() throws RemoveException {
  System.out.println("CustomerBMPBean:ejbRemove()");
  // you need to get the primary key from the context because
  // it is possible to do a remove right after a find, and
  // ejbLoad may not have been called.
  Connection    con = null;
  PreparedStatement ps  = null;
  try {
  con = getConnection();
  CustomerBMPPK pk = (CustomerBMPPK) context.getPrimaryKey();
  ps = con.prepareStatement("delete from Customer where ID = ?");
  ps.setInt(1, pk.id);
  int i = ps.executeUpdate();
  if (i == 0) {
    throw new RemoveException("CustomerBMPBean (" + pk.id
            + " not found");
  }
  } catch (SQLException sqe) {
  throw new RemoveException(sqe.getMessage());
  } finally {
  try {
    ps.close();
    con.close();
  } catch (Exception ignore) {}
  }
}

public void ejbStore() {
  System.out.println("BidBMPBean:ejbStore()");
  Connection    con = null;
  PreparedStatement ps  = null;
  try {
  con = getConnection();
  ps  = con.prepareStatement(
    "update Customer set NAME=?, ADDRESS=?, PASSWORD=? where ID = ?");
  ps.setString(1, this.name);
  ps.setString(2, this.address);
  ps.setString(3, this.password);
  ps.setInt(4, this.id);
  int i = ps.executeUpdate();
```

```
  if (i == 0) {
    throw new EJBException(
      "CustomerBMPBean::ejbStore: CustoemrBMP " + id
      + " not updated");
  }
  } catch (SQLException sqe) {
  throw new EJBException(sqe.getMessage());
  } finally {
  try {
    ps.close();
    con.close();
  } catch (Exception ignore) {}
  }
}

public double getName() {
  System.out.println("CustomerBean:getName()");
  return name;
}

public String getPassword() {
  System.out.println("CustomerBean:getPassword()");
  return password;
}

private String log(String s) {
  if (VERBOSE) {
  System.out.println(s);
  }
  return s;
}

private Connection getConnection() throws SQLException {
  InitialContext initCtx = null;
  try {
  initCtx = new InitialContext();
  DataSource ds =
    (javax.sql
      .DataSource) initCtx
      .lookup("java:comp/env/jdbc/CustomerPool");
  return ds.getConnection();
  } catch (NamingException ne) {
  String msg;
  msg = log("UNABLE to get a connection from CustomerPool! ");
  msg = msg
    + log("Please make sure that you have setup the connection
   pool properly. ");
  throw new EJBException(msg + ne);
  } finally {
  try {
    if (initCtx != null) {
      initCtx.close();
    }
  } catch (NamingException ne) {
    String msg = log("Error closing context: " + ne);
    throw new EJBException(msg);
  }
  }
}

public Date getAddress() {
```

```
    return address;
  }

  private void refresh(CustomerBMPPK pk)
    throws FinderException, EJBException {
    if (pk == null) {
    throw new EJBException(
      "Customer Bean primary key cannot be null");
    }
    Connection    con = null;
    PreparedStatement ps  = null;
    try {
    con = getConnection();
    ps  = con.prepareStatement("select * from Customer where ID = ?");
    ps.setInt(1, pk.id);
    ps.executeQuery();
    ResultSet rs = ps.getResultSet();
    if (rs.next()) {
      this.id   = pk.id;
      this.name   = rs.getString("NAME");
      this.address  = rs.getString("ADDRESS");
      this.password = rs.getString("PASSWORD");
    } else {
      throw new FinderException("Refresh: CustomerBMPBean ("
              + pk.id + ") not found");
    }
    } catch (SQLException sqe) {
    throw new EJBException(sqe.getMessage());
    } finally {
    try {
      ps.close();
      con.close();
    } catch (Exception ignore) {}
    }
  }

  public void setEntityContext(EntityContext ecxt) {
    System.out.println("CustomerBMPBean:setEntityContext()");
    context = ecxt;
  }

  public void unsetEntityContext() {
    System.out.println("CustomerBMPBean:unsetEntityContext()");
    context = null;
 }
}
```

Using Data Access Objects

Data access objects (DAO) enable you to separate business and data−access logic inside Enterprise JavaBeans. DAOs provide a simple API with which to access the database or other persistent storage, execute queries, and retrieve the results of those queries. That is, DAOs abstract and encapsulate all access to the data source, managing the connection in order to obtain and store data.

Transparency is one of the advantages of DAOs, because objects remove complexity and non−portable code from the EJB by using data sources without specific implementation details. Also, DAOs enable easier migration because data access is encapsulated and changing schemas or even different storage types is

simplified. Because CMP automatically manages the persistence services, DAOs ability to encapsulate data and data access is not useful to EJBs that use container−managed persistence. However, DAOs present an easy path from BMP to CMP, because all you need to do is discard the corresponding data−access object and its references in the bean. In addition, DAOs are useful when you need a combination of CMP (for entity beans) and BMP (for servlets and session beans). Figure 19−9 illustrates DAOs relationship with data and enterprise beans.

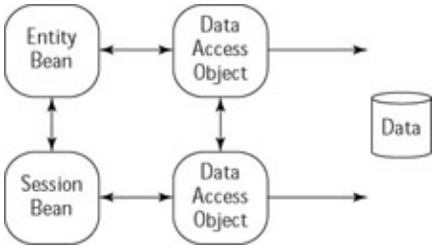

Figure 19−9: Using data access objects

Using Value Objects

Enterprise JavaBeans use value objects to return data to their clients. The value object is any serializable Java object; it may contain all the data in the bean. Usually EJBs expose to their clients getter and setter methods that return or set a piece of data at a time. In the current example you used getName, getAddress, setName, and setAddress to get and set the customer's name and address information; by doing this, you are requiring the client to make the getter and setter method calls; creating network overhead. The application uses the network layer regardless of bean proximity. Some vendors can tell if the calls are local and can provide direct access, thus reducing network overhead; however, this is vendor−specific.

Value objects provide you with a way to return all necessary information at once; that is, the bean exposes a method that, when invoked, creates a new instance of the value object from its member variables and returns it to the client. The client then has access to all the attributes it needs after just one method invocation. You can also use value objects to model one−to−many relationships between the bean and its clients, by exposing different methods that return different value−object instances with subsets of the member variables.

You can also use value objects to update the bean member variables. The bean exposes a setter method (or methods), and the client then generates an instance of the value object with the necessary data values and submits it to the bean. When the setter method is invoked the bean receives the value object and the bean sets its internal attributes accordingly, maybe even updating persistent storage if necessary. Figure 19−10 shows a value object being used by an entity bean and its client.

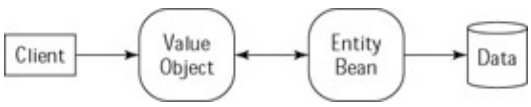

Figure 19−10: A value object is used by a Client and an Entity Bean

Transactions and EJBs

One of the advantages of EJBs is that they provide semi−automated transaction management. The EJB

container is responsible for generating and managing transactions defined between the bean and the client.

The client or bean declares a new transaction by creating a UserTransaction object; the transaction starts by calling begin() on the transaction object. The client, the EJB container, or the bean itself can establish the end of the transaction by calling the commit() or rollback() method. When the bean is deployed, a bean can choose the type of transaction support needed.

A session bean can (optionally) demarcate its own transactions by specifying the beginning and ending of the transaction. This is known as *bean−managed transaction demarcation*.

Transaction attributes are values associated with one or more methods of the bean. Table 19−3 explains the type of transaction support available to an enterprise bean that uses *container−managed transaction demarcation*, in which the EJB container is responsible for managing transaction boundaries.

Table 19−3: Transaction Support Attributes

Attribute	Description
TX_NOT_SUPPORTED	Specifies that the bean does not support transactions; if the client has initiated a transaction, the container suspends it and restarts the transaction after the method is done.
TX_SUPPORTS	Specifies that the bean needs to support transactions if the client requests it. That is, if the client is within a transaction when it invokes a method, the EJBContext passes the transaction context to the bean.
TX_REQUIRED	Specifies that the bean requires the method invocation is part of a transaction. If the client is within a transaction, the transaction context is available through the EJBContext. Otherwise the container creates a new transaction before calling the method, and the container tries to commit the transaction before returning the results to the client.
TX_REQUIRES_NEW	Specifies that the container always creates a new transaction before invoking the method, and commits it before the results are returned to the client. If the client is associated with a transaction at the time, the association between the transaction context and the current thread is suspended and later resumed when the new transaction is committed.
TX_MANDATORY	Specifies that the container invokes the bean method in a client's transaction context. If the client is not associated with a transaction the container throws a TransactionRequiredException.
TX_BEAN_MANAGED	Specifies that the container does not automatically generate a transaction for each method call. The bean manages all its own transaction boundaries and the bean's method runs within the transaction until it is ended (by the bean).

Guidelines for Working with EJBs

Here are a few guidelines for working with Enterprise JavaBeans:

1. Avoid mapping relational models directly to the entity bean model, as doing this translates the inner−table relationships into entity−to−entity bean relationships, thus affecting scalability.
2. Avoid mapping use−cases directly to session beans, as this creates a lot of session beans that may be performing similar services. Instead, create session beans based on the services they must provide.
3. Use value objects instead of exposing multiple getter/setter methods.
4. Use data−access objects instead of creating read−only beans that waste resources and make unnecessary updates to the database.
5. Select transaction demarcation very carefully. Experienced users, who require more control than the EJB container provides, should select bean−managed transaction demarcation.
6. Use a session bean to access data that span concepts, and an entity bean to provide an interface to a set of shared data that define a concept.
7. Finder methods are not suitable for caching results; instead use a DAO to execute the query and create a collection of value objects for the results.
8. Value objects may introduce stale objects because they allow the client to modify the local copy and the value object must therefore call the bean in order to update the attributes.

Summary

Enterprise JavaBeans are server−side components that enable you to develop enterprise applications. EJBs operate inside EJB containers that provide valuable services such as security and transaction management.

The main types of EJBs are session beans (either stateless or stateful), entity beans, and message−driven beans. According to your needs, you can persist your data in different ways: You can use container−managed persistence, bean−managed persistence, and even object serialization. By following the guidelines and using J2EE standards to develop EJB components and their XML−deployment descriptors, you will be able to use the same code and deployment information across different J2EE−compliant EJB application servers, and have a uniform data−access platform.

Appendix A: JDBC 3.0 New Features Summary

The JDBC 3.0 API is the latest release of Java's database−connectivity technology. Overall the API has not changed radically since version 2.x. In fact, it is more of an update than a new version release. However, version 3.0 does include more features supporting enterprise development, the J2EE platform, and the ANSI SQL standards.

One of the biggest changes is how Sun distributes the JDBC packages. Sun now ships both the java.sql and javax.sql packages in the core 1.4 JDK and JRE distribution. Prior distributions only included the java.sql package and you needed to download the JDBC Optional Package to obtain the javax.sql package. Starting with the 1.4 release, you no longer need to download the Optional Package to have access to the javax.sql package.

Besides the distribution changes, the new release features enhancements in the following areas:

- Transactions
- Metadata
- Connection pooling
- Data type additions and enhancements

The following sections provide an overview of the changes in each area. For more detailed information, refer to the JDBC 3.0 specification. You can download it at www.javasoft.com/products/jdbc.

Transactions

JDBC 3.0 gives you greater control over SQL transactions than you could get from prior versions. Previously, rolling back a transaction removed any changes to the database since the last commit. This behavior had advantages and disadvantages. One advantage was that you could remove any changes that might corrupt data. One disadvantage was that a rollback removed all changes, even valid ones. This proved inconvenient for programmers working with transactional blocks that include several UPDATE or INSERT statements. You could work around the problem by issuing commits after the valid operation(s). However, this approach hinders performance because of the server−side overhead associated with commits.

To give you better transactional control, JDBC now supports transactional savepoints. This enables you to define logical commits within a transaction block to which you can rollback changes. Creating savepoints after valid operations helps you avoid the overhead of commits and provides a location to terminate a rollback. Using this approach keeps a rollback from removing all the changes — including the valid ones.

XRef Chapter 5, "Building JDBC Statements," provides more detail on this new Savepoint interface.

To create and remove savepoints you need to use a Connection object's setSavepoint() or releaseSavepoint() method, respectively. The java.sql package contains the Savepoint interface that defines methods to return information on savepoints you have set.

Metadata

JDBC 3.0 enhances the metadata interfaces to provide more information about the data source and JDBC objects.

In particular, the interface defines new methods to enable you to obtain more information on user−defined data types (UDTs) in the database. For example, you can now retrieve the hierarchy and attributes associated with UDTs defined on the server.

In addition, a new interface, ParameterMetaData, provides information on the parameters passed to PreparedStatement objects. You can determine the type, number, and other properties for parameters using methods the interface defines. Table A−1 lists some of the more useful ParameterMetaData interface methods.

Table A−1: Selected ParameterMetaData Interface Methods

Method	Return Type	Description
getParameterCount()	int	Returns the number of parameters in the PreparedStatement object that created the metadata object.
getParameterMode	int	Tells you whether the parameter is (int parameter) an IN, OUT, or IN/OUT.
getParameterType	int	Returns the SQL type for the (int parameter) parameter.
getParameterTypeName	String	Returns the SQL type name used by (int parameter) the database for the parameter.

Connection Pooling

Changes to the ConnectionPoolDataSource interface give you more control than before over how JDBC connection pools operate. Prior to JDBC 3.0 you had very little influence on their behavior. For example, you could not specify the maximum number of physical connections to create, or an idle time for inactive connections.

Now you can control connection pools by setting new properties in a ConnectionPoolDataSource object. As with all javax.sql packages, the vendor must provide an implementation. In this case, the vendor must provide the setter and getter methods that give you access to the properties. Table A−2 lists the properties used to control connection pools.

Table A−2: ConnectionPoolDataSource Properties

Property	Description
int maxPoolSize	The maximum number of PooledConnection objects to create. A value of zero (0) means to create as many physical connections as needed.
int minPoolSize	The minimum number of PooledConnection objects to keep available for use. A value of zero (0) means to create physical connections only when needed.
int maxIdleTime	The time (in seconds) for which a physical connection should remain inactive before it is closed. A value of zero (0) indicates infinity.
int maxStatements	The maximum number of PreparedStatement objects that should remain open. A value of zero (0) disables statement pooling.
int initialPoolSize	The number of physical connections to open when creating the pool.
int propertyCycle	The time (in seconds) that should elapse before the pool begins applying restrictions defined by the previous properties.

The addition of statement pooling is another change to the connection−pooling architecture. Statement pooling allows a PooledConnection object to maintain a cache of PreparedStatement objects. The result is increased performance, because the statements do not have to be re−parsed before execution, and you can share them between logical connections.

Of course, the driver vendor must implement statement pooling. The DatabaseMetaData.supportsStatementPooling() method returns true if the driver supports statement pooling.

Data Type Enhancements

JDBC 3.0 provides two new data types and extends support for existing types. Also, the changes further enhance JDBC's support for the SQL data type standards. The following list outlines the changes:

- JDBC now has a new data type, java.SQL.types.BOOLEAN, which maps to the SQL99 BOOLEAN data type. You can use the getBoolean() method to retrieve a value into the Java boolean data type.
- You can now update columns containing BLOB, CLOB, ARRAY, and REF data types using the updateBlob(), updateClob(), updateArray(), and updateRef() methods of the ResultSet interface.
- New methods in the Blob and Clob interfaces enable you to update the respective objects.
- A new data type, java.SQL.Types.DATALINK, enables you to access external data. The data type maps to the Java java.net.URL type.

As you can see, JDBC 3.0 eases your work with respect to handling the issues associated with Java and SQL data types.

Appendix B: Java Database Programming on Linux

by Stephen Norbert

Linux is an open−source UNIX derivative, started as a hobby in 1991 by Finnish college student Linus Torvalds. Soon thereafter, a worldwide community, linked via the ever−growing Internet, took on the challenge of creating a first−class, open−source operating system. This operating system would continue to be maintained and enhanced by developers who wished to share in the open−source development experience. The popularity of Linux continues to increase: Today, it remains a free operating system and continues to be updated by countless international developers in the open−source community. Each new Linux version brings vast improvements in performance, reliability, usability, compatibility, and documentation. In addition, new and innovative features continue to appear.

As Linux's popularity increases, so will the amount of third−party software available for it. Several Java Virtual Machines (JVMs) are already available for the platform, with more to come. A large number of databases are available as well.

JVMs for Linux

As Java's popularity has increased, the number of JVMs available for Linux has also increased. Some of these are licensed ports of Sun's Java Virtual Machine, while others have been written to support Sun's Java VM specification.

The minimal JVM package you can use to run your Java applications is known as the Java Runtime Environment (JRE). You can run compiled applications with a JRE, but you cannot develop them; to develop applications you will need the Java Development Kit (JDK), also known as the Software Development Kit (SDK). This package includes the JRE and some development tools, such as a compiler and a debugger. To develop applications on Linux, you need a Linux−specific JDK. Table B−1 describes the latest version of four popular JVMs for Linux.

Table B−1: JVMs for Linux

JDK	Vendor	Version	Information and Download
Java 2 SDK, Standard Edition	Sun Microsystems	1.3.1	http://java.sun.com/linux
IBM Developer Kit for Linux, Java 2 Technology Edition	IBM	1.3.0	www.ibm.com/developerworks/linux
Blackdown Java 2SDK, Standard Edition	Blackdown	1.3.1 FCS	Open Source. http://www.blackdown.org/
Kaffe OpenVM	Transvirtual	1.0.6	Open source. http://www.kaffe.org/

Current JVM implementations are smarter, faster, and higher−performing than earlier versions. For example, the latest JVMs take advantage of the native threading capabilities of the underlying operating system. In addition, byte code is dynamically compiled to native code, which results in the best possible performance for Java applications. Other improvements include more efficient garbage collection and memory management.

Java Integrated Development Environments (IDE) aid in Java application development. Several free IDEs are available. Some are limited versions of commercial products that still provide enough functionality for serious development. There are also powerful open−source versions available as well. Java IDEs provide a rich graphical development environment and commonly include an editor, compiler, debugger, and project manager. Many IDEs are written in Java, which means you can seamlessly use them across platforms.

Databases for Linux

Numerous databases exist for the Linux platform. Many are open−source, but several are commercial. Most commercial databases are free for development use, while others are limited versions of their more powerful, full−featured cousins.

You can choose from among relational and object databases, as well as some databases that are hybrids of both types. The following sections describe the different types of databases available for Linux, including a corresponding list of some of each type. Be advised, this list continues to grow. An Internet search engine query will yield more than we describe here.

Regardless of which database you choose, there will be at least one Java data−access mechanism, such as a JDBC or ODBC driver, available for it. If you need JDBC drivers, check Sun's JDBC driver page (http://industry.java.sun.com/products/jdbc/drivers) for a list of those that are available for your database.

Relational Database Management Systems

The Relational Database Management System (RDBMS) is the most popular of the different types of databases. RDBMSes provide the necessary functionality for a myriad of applications with respect to transaction processing, e−commerce, and so on. Table B−2 lists the latest versions of several RDBMSes available for Linux. Given the large number available, you can surely find one to meet your needs.

Table B−2: Linux RDBMSes

Database	Vendor/Author(s)	Version	Comments/Reference
Adabas D	Software AG http://www.adabas.com/	12.0	Linux and other editions available.
D3 Linux	Raining Data http://www.rainingdata.com/	7.2.0	Uses a three−dimensional data model, whereby one table can store all the information that formerly required three tables.
DB2 Universal	IBM http://www.ibm.com/	7.2	DB2 Version 7.1 for Linux HOWTO: www.linuxdoc.org/HOWTO/DB2−HOWTO/index.html

Database			
Informix	Informix http://www.ibm.com/	Informix Internet Foundation 2000	The future of Informix products is uncertain since Informix w acquired by IBM.
Ingres II	Computer Associates http://www.ca.com/	2.5	A commercial version of the first open−source RDBMS, developed at the University of California, Berkeley. A free version is included in the free SDK. Ingres II HOWTO: www.linuxdoc.org/HOWTO/IngresII−HOWTO/ndex.html
InterBase	Borland http://www.borland.com/	6.01	Open−source.
Mini SQL	Hughes Technologies Pty Ltd. http://www.hughes.com.au/	3.0	A mSQL and perl Web Server Mini (mSQL)HOWTO: www.linuxdoc.org/HOWTO/WWW−mSQL−HOWTO.html
MySQL	MySQL AB http://www.mysql.com/	3.23	Open−source
Oracle	Oracle http://technet.oracle.com/	9i	Oracle for Linux Installation HOWTO (installing Oracle 8i Enterprise Edition for Linux): www.linuxdoc.org/HOWTO/Oracle−8−HOWTO.html
SAP DB	SAP www.sapdb.org/	7.3.00	Open−source
Sybase Adaptive Server Enterprise	Sybase http://www.sybase.com/	12.5	Sybase Adaptive Server Anywhere for Linux HOWTO: www.linuxdoc.org/HOWTO/Sybase−ASA−HOWTO/index.h
Texis	Thunderstone–EPI http://www.thunderstone.com/	3.0	Natural−language and advanced query mechanism.

Object Database Management Systems

Although an RDBMS satisfies most database requirements, it cannot handle Java object persistence. If your application must persistently store Java objects, then an Object Database Management System (ODBMS) should be your database of choice. Table B−3 lists some ODBMSes available for Linux. The list is short in comparison to the number of RDBMSes, but as the need for ODBMSes grows, it will get longer.

Table B−3: Linux ODBMSes

Database	Vendor/Author(s)	Version	Comments/Reference
Orient ODBMS	Orient Technologies http://www.orientechnologies.com/	2.0	Uses a JDO layer.
Ozone	Open Source Community http://www.ozone−db.org/	1.0	Open Source. Java−based, somewhat ODMG−compliant.

Shore	The Shore Project Group Computer Sciences Department University of Wisconsin−Madison www.cs.wisc.edu/shore/	2.0	Formal support ended in 1997, but software is still available.

Object−Relational Database Management Systems

The latest trend in database technology is the hybridization of relational and object database concepts. These hybrids are called Object−Relational Database Management Systems. Some ORDBMSes have pluggable modules that enable them to persistently store objects. Table B−4 lists the latest versions of some ORDBMSes available for Linux. Given that they are becoming more popular, the number of available ODBMSes will definitely increase.

Table B−4: Linux ORDBMSes

Database	Vendor/Author(s)	Version	Comments/Reference
KE Texpress	KE Software http://www.kesoftware.com/	5.0.93	Purported by vendor to be “TheWorld’s Fastest Database.”
Polyhedra	Polyhedra PLC http://www.polyhedra.com/	4.0	Real−time, embedded.
PostgreSQL	Open Source Community http://www.postgresql.org/	7.1.3	Open−source.
PREDATOR	Praveen Seshadri and the Predator group at the Computer Science Department of Cornell University www.cs.cornell.edu/predator	2.0	Free under GNU Public License (GPL). May no longer be supported.

Appendix C: JDBC Error Handling

Proper exception handling is required in any application you develop. You must catch and handle errors to prevent your application from crashing or to give the user information regarding the application's "state."

As you work with JDBC you will encounter exceptions. This is guaranteed. Interacting with a database is a complex process and a lot of components must work together for the system to operate correctly. Here is a short list of areas that can cause you problems:

- Network connectivity
- User authentication and access rights
- Invalid SQL statements
- Class−loading errors with drivers
- Invalid data type conversions

Because so many issues can cause exceptions, almost all JDBC methods throw an SQLException if an error occurs. In other words, you must place your JDBC statements in try−catch blocks or propagate the error back to the calling method.

When working with JDBC you will encounter four exception classes: SQLException, SQLWarning, BatchUpdateException, and DataTruncation. The SQLException class extends the Exception class so it has all the functionality used to handle Java exceptions. This is the class you will deal with most often. The SQLWarning and BatchUpdateException classes extend the SQLException class. The DataTruncation class extends SQLWarning and handles a very specific warning. Figure C−1 highlights the relationships between the JDBC exception classes.

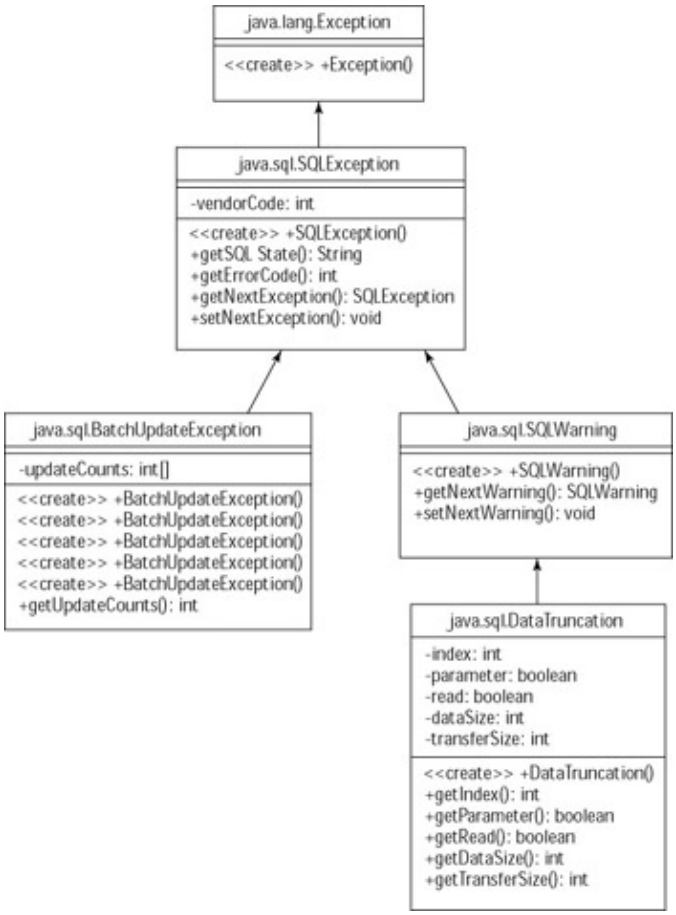

Figure C−1: UML class diagram of JDBC Exception classes

SQLException

The SQLException occurs whenever you encounter a "show stopping" event in JDBC programming. For example, improper data type conversions or problems making a database connection throws the error.

In general, you must use try−catch blocks around all JDBC code to catch this exception as shown in the following code snippet:

```
try{
    //Process some JDBC stuff here....

}catch(SQLException se){
    //Handle SQL or JDBC exceptions
    se.printStackTrace();

}catch(Exception e){
    //Handle any other exceptions
    e.printStackTrace();
}
```

An SQLException contains more information than an Exception class. Table C−1 summarizes the most useful information. You will probably use the error message and error code the most, because they contain the specific cause for your database error. Vendors supply this content, so it will be directly applicable to your database. SQLState, a String object, uses error codes from the X/Open SQL specification, which defines a set

of standard error codes. You may not be able to decrypt these esoteric codes. So again, your best bet is to stick with the vendor−specific information.

Table C−1: SQLException Information

Information	Explanation
Error message	String that contains a message explaining the error.
Error code	Vendor−specific error code.
SQLState	SQLState identifier based on the X/Open SQL specification. Your DBMS should have a listing of these industry−standard error codes.

One nice feature of the SQLException class is that it gives you the ability to chain exception events together using the setNextException() method and retrieve the exceptions from the chain using the getNextException() method. These methods enable you to add custom−defined exceptions and messages to an SQLException that occurs. You might find this helpful if you want to present one message to the user and log the original message.

Listing C−1 shows an example of an SQL query on a table that does not exist. I chain my own exception, mySqlEx, to seRs and re−throw it. This time the main() method's catch block handles the error and loops through the chained exceptions, displaying the appropriate information.

Listing C−1: SqlException.java

```
package AppendixC;

import java.sql.*;

public class SqlException {

  public static void main(String[] args) {

    //Declare Connection,Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Register a driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open a connection
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      stmt = conn.createStatement();

      try{
```

```
        rs = stmt.executeQuery("Select * from no_table_exisits");

      }catch(SQLException seRs){

        String exMsg = "Message from mySqlEx";
        String exSqlState = "Not set by me";
        SQLException mySqlEx = new SQLException(exMsg,exSqlState);
        seRs.setNextException(mySqlEx);
        throw seRs;

      }

    }catch(SQLException se){

      int count = 1;

      //Loop through exception and retrieve messages
      while (se != null){

        System.out.println("SQLException " + count);
        System.out.println("Code: " + se.getErrorCode());
        System.out.println("SqlState: " + se.getSQLState());
        System.out.println("Error Message: " + se.getMessage());

        //Load next SQLException
        se = se.getNextException();
        count++;

      }

     }catch(Exception e){
      e.printStackTrace();

    }//end try

  }//end main
}//end SqlException
```

The output from Listing C−1 is as follows:

```
Connecting to database...
SQLException 1
Code: 942
SqlState: 42000
Error Message: ORA−00942: table or view does not exist

SQLException 2
Code: 0
SqlState: Not set by me
Error Message: Message from mySqlEx
Connecting to database...
```

SQLWarning

In JDBC programming you will sometimes receive an SQLWarning after some event. An SQLWarning is less

critical than an SQLException and does not stop execution. The Connection, Statement, and ResultSet objects are the primary JDBC objects that throw an SQLWarning.

An SQLWarning may occur, for example, if you try to create an updateable result set and the driver will not support one. When you call the Statement object's execute() or executeQuery() method to create a ResultSet object you will not receive an error; you will receive an SQLWarning indicating that the action could not be performed. The method still instantiates a valid ResultSet object, but you will receive an SQLException if you try to update a column in the result set. An SQLWarning occurs silently, so you do not know it is occurring unless you explicitly check for the warning.

Because SQLWarning extends the SQLException class, it contains the same properties and methods. It is also chainable, meaning that you can add your own errors to an exception event. Refer to Table C−1 for more on the information provided by these classes.

Listing C−2 provides an example of the scenario I just described. It demonstrates catching an SQLWarning and processing it. Notice that the SQLWarning occurs but allows the ResultSet object to be created. I also receive an SQLException when I try to update a column.

Listing C−2: SqlWarning.java

```
package AppendixC;

import java.sql.*;

public class SqlWarning{

  public static void main(String[] args) {

    //Declare Connection,Statement, and ResultSet variables
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;

    //Begin standard error handling
    try{

      //Register a driver
      String driver = "oracle.jdbc.driver.OracleDriver";
      Class.forName(driver).newInstance();

      //Open a connection
      System.out.println("Connecting to database...");
      String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:ORCL";
      conn = DriverManager.getConnection(jdbcUrl,"toddt","mypwd");

      stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                  ResultSet.CONCUR_UPDATABLE);
      SQLWarning sw = null;

      rs = stmt.executeQuery("Select * from employees");
      sw = stmt.getWarnings();
      System.out.println(sw.getMessage());

      while(rs.next()){
        System.out.println("Employee name: " + rs.getString(2));
      }
```

```
      //Try to use an updateable result set method
      //This will cause an error
      rs.previous();
      rs.updateString("name","Jon");

    }catch(SQLException se){
      System.out.println("SQLException occurred: "
      + se.getMessage());

    }catch(Exception e){
      e.printStackTrace();

    }//end try

  }//end main
}// end SqlWarning class
```

The output from Listing C−2 is as follows:

```
Connecting to database...
Warning: Unable to create resultset at the requested type and/or concurrency
level: ORA−00936: missing expression

Employee name: Todd
Employee name: Larry
Employee name: Lori
Employee name: Jimmy
Employee name: John
Employee name: Rebecca
SQLException occurred: Invalid operation for read only resultset: updateString
```

BatchUpdateException

The Statement, PreparedStatement, or CallableStatement object throws a BatchUpdateException if an error occurs with an SQL statement during a batch processing operation.

The class is derived from the SQLException class, and therefore contains the same information. However, it also contains one additional property that helps you track the number of successful SQL statements that occurred during batch processing. The property is an array that contains the update counts for the SQL statements that complete successfully. With this information you can determine which SQL statement failed. For an extended example that uses this class, refer to Chapter 5, "Building JDBC Statements."

DataTruncation

A DataTruncation exception occurs when you attempt to write or retrieve data from the database and for some reason the length of the data is truncated. For example, this error can occur if you set the Statement.setMaxFieldSize() to low.

Truncation errors on read operations often cause a silent SQLWarning to be thrown. If an error occurs during a write operation then an SQLException generally occurs. However, all drivers will likely handle these errors differently. Check your driver documentation for details.

Appendix D: UML Class Diagram Quick Reference

In a nutshell, the Unified Modeling Language (UML) is used to document systems. Although it is associated with software design, you can also use it to model business and other non−software systems.

UML is a standardized language maintained by the Object Management Group (OMG). Grady Booch, Jim Rumbaugh, and Ivar Jacobson created a “draft” version of the language in the mid−1990’s without the OMG. As the OMG became involved, their work became the basis for the first anointed version. The latest version of UML that the OMG has blessed is 1.3.

The full language contains many elements beyond the scope of this appendix. However, the UML class diagram is one of the more useful elements. Class diagrams provide a visual representation of individual classes and interfaces and how they relate to each other.

Although there is a standard for UML class diagrams, you will notice that all diagrams differ slightly. For example, some use stereotypes while others do not. Some use navigation arrows for associations while others do not. These differences are minor, and you will still be able to interpret the diagrams.

The information I use in this appendix represents the notation I employ throughout the book. Although I try to follow the 1.3 specifications I may make some deviations for clarity. For example, I provide a representation for object instantiation where the specification does not readily define one.

For more information on UML and class diagrams, check the OMG’s website at http://www.omg.org/.

Class Diagrams

Class diagrams visually describe class definitions and class relationships in a software system. You can represent both methods and attributes in these diagrams. This section provides examples of class, interface, and abstract class diagrams.

Class

Figure D−1 shows a class diagram. It is a rectangle divided into three sections. The top contains the class name, the middle section the attributes, and the last the methods. A class diagram does not need to represent all the methods and attributes of the class in the class diagram. You can show only the information needed to convey your message.

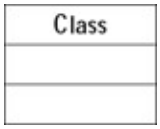

Figure D−1: Simple class diagram

Figure D−2 shows a ConnectionMgr class diagram populated with attributes and methods. Notice that some attributes and methods have different symbols that indicate visibility. Table D−1 shows the meaning of the symbols.

In addition, class, or static, methods are underlined. If you want to specify a method’s return type, place a colon (:) between the method name and the return type. For example, the getName() method returns a String value. Parameters are represented in a similar fashion. The connect() method in the figure provides an

example of how the methods, parameters, and return types are formatted.

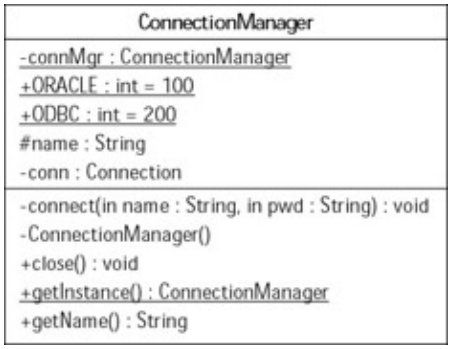

Figure D−2: ConnectionManager UML classdiagram

Table D−1: Method and Attribute Visibility Symbols

Symbol	Meaning
+	Public
#	Protected
−	Private

Interface

Interfaces are similar to class diagrams except that the methods are in italics to indicate that they are methods without implementations. Figure D−3 shows an example of an interface diagram.

In some cases you may see the entity's "stereotype" enclosed in double angle brackets (<<>>) to define a special kind of class. Stereotypes are a means of extending UML by creating your own custom labels. However, some standard stereotypes exist. For example, the stereotype <<interface>> is frequently used to identify a Java interface.

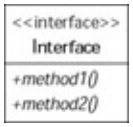

Figure D−3: Interface diagram

Abstract class

An abstract class diagram is a combination of an interface diagram and a class diagram. Figure D−4 provides an example. Notice that concreteMethod() is not in italics. Only abstract methods are placed in italics.

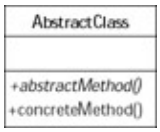

Figure D−4: Abstract class diagram

Class Relationships

Not only do class diagrams represent class structures, but they are used to represent class relationships as well. This section provides you with on overview of the major relationships presented in class diagrams and in this book.

Association

Relationships between classes are known as *associations* and are represented by lines drawn between classes. Associations have two attributes, navigability and multiplicity.

Navigability indicates the direction of the association and is represented by an arrow on one or both ends of an association line. Figure D−5 shows a unidirectional association between Class1 and Class2. The figure indicates that Class2 has no knowledge of Class1. Arrows on both ends indicate a bi−directional association, which means that both classes know about each other.

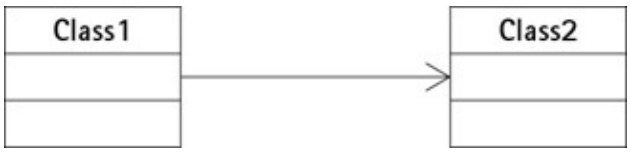

Figure D−5: Unidirectional−association example

Multiplicity describes the number of objects participating in a relationship. Figure D−6 shows a one−to−one relationship between objects, and Figure D−7 provides an example of a one−to−many relationship. Table D−2 provides examples of other multiplicity values. Multiplicity values can exist on either end of the association line.

Figure D−6: One−to−one–multiplicity diagram

Figure D−7: One−to−many–multiplicity diagram

Table D−2: Examples of Multiplicity Values

Value	Meaning
1	One and only one
*	Any number
0...1	Zero to one
0...*n*	Zero to number *n*
0...*	Zero to an infinite number

Generalization

Generalization, or *inheritance*, is represented by an open arrow and a solid line. The arrow points to the base class. Figure D−8 shows an example of a generalization. This figure indicates that Class1 is a subclass of Class2.

Figure D−8: Generalization and inheritance example

Realization

A *realization* is what happens when a class implements an interface or extends an abstract class. Figure D−9 provides an example. A realization is represented by an open arrow and a dashed line. The arrow points to the interface or abstract class.

Figure D−9: Realization example

Dependency

Dependencies are weak relationships between classes. For example, usually no member variables are involved; the relationship might be limited to using another class as a member method argument. A dependency is represented by a dashed line with a solid arrow pointing from the dependent class to the class depended upon. Figure D−10 provides an example.

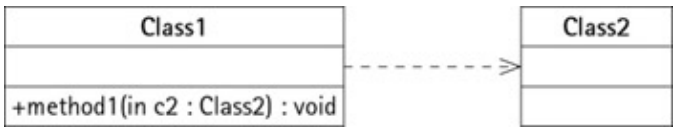

Figure D−10: Dependency example

Aggregation

Aggregation is one of the more difficult class−diagram concepts to understand. It is considered a “part−of” relationship. It is functionally the same as composition, and as a result is not used very often. An aggregation is represented by a solid line and an open diamond that touches the aggregated class. Figure D−11 shows that Class1 aggregates Class2, which means that Class2 is part of Class1.

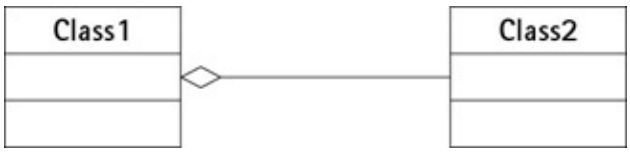

Figure D−11: Aggregation example

Composition

A stronger, more concrete form of aggregation is *composition*, as it represents a whole/part relationship. It differs from aggregation in that it suggests that the composing object controls the lifetime of the composed object. Figure D−12 shows a composition relationship, represented by a solid line and a solid diamond that touches the composing class. You interpret the figure to mean that Class1 contains Class2 and controls the lifetime of an object of type Class2.

Composition is very important to design patterns, especially patterns with object scope. Because of its importance I have provided a code snippet associated with Figure D−12.

```
public class Class2{}

public class Class1{

   Class2 class2;
   ...
   ...
   ...
}
```

Figure D−12: Composition example

Instantiation

Although *instantiation* is not formally defined in the UML standard, I use the notation shown in Figure D−13 to indicate it. The dashed line and solid arrow indicates that Class1 instantiates an object of type Class2.

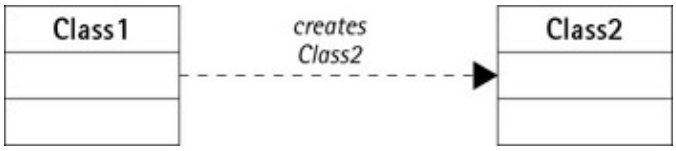

Figure D−13: Instantiation example