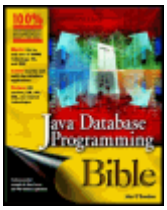


Java Database Programming Bible

by John O'Donahue

ISBN:0764549243

John Wiley & Sons © 2002 (702 pages)

Packed with lucid explanations and lots of real-world examples, this comprehensive guide gives you everything you need to master Java database programming techniques.

Companion Web Site

Table of Contents

Preface

Welcome to *Java Database Programming Bible*. This book is for readers who are already familiar with Java, and who want to know more about working with databases. The JDBC Application Programming Interface has made database programming an important aspect of Java development, particularly where Web applications are concerned.

The ease with which Java enables you to develop database applications is one of the main reasons for Java's success as a server-side development language. Database programming is perhaps the key element in developing server-side applications, as it enables such diverse applications as auction sites, XML-based Web services, shipment-tracking systems, and search engines.

What this Book Aims to Do

The aims of this book are to give you a good understanding of what a relational database is, how to design a relational database, how to create and query a relational database using SQL, and how to write database-centric applications in Java. There are many books that cover individual aspects of the aforementioned topics, such as SQL or JDBC. The intention of this book is to provide a single source of information and application examples covering the entire subject of relational databases.

When I first started to develop database-driven applications in Java, I was working with a database administrator who was responsible for the database side of the project. This is a fairly common approach to managing larger database-driven applications, since it places responsibility for the database in the hands of a database expert and allows the Java programmer to concentrate on his or her own area of expertise. The disadvantages of this approach only became apparent when some of my code proved to be unacceptably slow because of database design considerations that failed to take into account the needs of the business logic.

Working on subsequent smaller projects enabled me to manage my own databases and develop an understanding of how to design databases that work with the business logic. I also learned about the tradeoffs involved in using indexes and the importance of normalization in designing a database. Perhaps the most important thing I learned was that, thanks to the design of the JDBC API and the universality of the SQL language, much of what you learn from working with one database-management system is directly applicable to another.

Although this book aims to give you a good overall understanding of Java database programming and, in particular, to cover the JDBC API thoroughly, it is impossible to cover either all of the different JDBC drivers currently available or all the variations of the

SQL language in a book of this nature. The examples in this book were developed using a number of different JDBC drivers and RDBMS systems; Part II of the book addresses the ease with which you can use the same code with different drivers and different database-management systems.

You will find, as you work with a variety of different Relational Database Management Systems, that the SQL standards are really just guidelines. SQL has as many different dialects as there are relational database management systems. So although the examples in this book should work with only minor changes on virtually any RDBMS, you would be well advised to read a copy of the documentation for your own database-management system.

Who Should Read this Book

This book is aimed at all levels of programmers, including those with no prior database experience. However, you should already have some experience with Java basics and Swing, so no attempt has been made to explain this book's examples at that level. The server-side applications are introduced with a brief discussion of servlets and Java Server Pages, supported by the information in Appendix B on downloading and installing the Apache HTTP server and the Tomcat servlet and JSP eengine. If you are looking for a beginner-level Java book, consider *Java 2 Enterprise Edition Bible* (ISBN 0-7645-0882-2) by Justin Couch and Daniel H. Steinberg. For the beginning- to-intermediate-level programmer, *Java Database Programming Bible* introduces all the various technologies available to you as a J2EE programmer. If you have never used J2EE before, this book will show you where to start and the order in which to approach your learning.

For the more advanced-level programmer, this book serves as a guide to expanding your horizons to include the more concentrated areas of programming. Use this book as a guide to exploring more possibilities within the area that you have already been working on or to find new ways to address a problem. Finally, you can use this book to learn about new areas that you may have not heard of before. Because of the breadth of J2EE, it is always possible that new topics exist that you haven't heard of. Even after six-plus years of Java programming experience, I am constantly finding new items popping up that I want to learn about.

How to Use this Book

This book is divided into a number of parts. Each part covers a different aspect of the technology, while the chapters focus on individual elements. The examples in the various chapters are intended to provide a set of practical applications that you can modify to suit your own needs.

The depth of coverage of each aspect of the technology is sufficient for you to be able to understand and apply Java database programming in most of the situations you will encounter. However, this book assumes that you are comfortable downloading and working with the Javadocs to ferret out the details of an API. Unlike some books, *Java Database Programming Bible* does not reproduce the Javadocs within its covers.

This book's approach is to present the different aspects of the technology in the context of a set of real-world examples, many of which may be useful as they are, although some may form the foundation of your own applications. For example, the book presents JDBC core API in the context of a simple Swing application for the desktop, while the extension API is covered in a series of server-side Web applications.

Since I have never read a programming book from cover to cover, I don't expect you to, either. Individual chapters and even examples within chapters are intended to stand by themselves. This necessarily means that there is a certain amount of repetition of key concepts, with cross-references to other parts of the book that provide more detail.

If you don't have much of an understanding of database technology, I do recommend that you read Part I, which introduces the basic concepts. If you know something about the JDBC core API, but you are not familiar with the extension API, you might want to read just the JDBC chapter in Part I to see how it all fits together.

This book is made up of six parts that can be summarized as follows.

Part I: Introduction to Databases, SQL, and JDBC

The introductory chapters discuss what a relational database is and how to create and work with one. This part is concerned mainly with the big picture, presenting overviews of the technology in such a way that you can see how the parts fit together. This part contains an overview of the SQL language, as well as an explanation of JDBC as a whole.

Part II: Using JDBC and SQL in a Two-Tier Client/Server Application

Part II presents the JDBC core API and SQL in the context of a series of desktop applications. These applications are combined in the final chapter of this part to form a Swing GUI that can be used as a control panel for any database system. A key concept presented in this part of the book is the way that JDBC can be used with any RDBMS system by simply plugging in the appropriate drivers.

Part III: A Three-Tier Web Site with JDBC

One of the most common Java database applications is the creation of dynamic Web sites using servlets, JSPs, and databases. This part discusses the JDBC extension API in the context of developing a Web application. It also talks about using JDBC and SQL to

insert large objects such as images into a database, and retrieving them for display on a Web page.

Part IV: Using Databases, JDBC, and XML

Another big application area for Java and database technologies is the use of XML. This part introduces XML and the Document Object Model, and it presents different ways to work with Java, databases, and XML. This part also discusses the design of a simple JDBC driver and a SQL engine to create and query XML documents.

Part V: EJBs, Databases, and Persistence

Applications using Enterprise Java Beans are another significant area where Java and databases come together. This part introduces EJBs and persistence, and it compares bean-managed persistence with container-managed persistence.

Part VI: Database Administration

The final major topics we discuss are often overlooked in books about database programming: database administration, and tuning. This oversight might be understandable if all databases had a dedicated administrator, but in practice it frequently falls to the Java developer to handle this task, particularly where smaller systems are involved.

Appendixes

The appendixes are a comparison of some major SQL dialects and a guide to installing Apache and Tomcat.

Companion Web Site

Be sure to visit the companion Web site, where you can download all of the code listings and program examples covered in the chapters. The URL for the website is: http://www.wiley.com/extras.

Conventions Used in this Book

This book uses special fonts to highlight code listings and commands and other terms used in code. For example:

```
This is what a code listing looks like.
```

In regular text, monospace font is used to indicate items that would normally appear in code.

This book also uses the following icons to highlight important points:

Note	The Note icon provides extra information to which you need to pay special attention.
Tip	The Tip icon shows a special way of performing a particular task.
Caution	The Caution icon alerts you to take care when performing certain tasks and procedures.
Cross-Reference	The Cross-Reference icon refers you to another part of the book or another source for more information on a topic.

Acknowledgments

Writing a book is both challenging and rewarding. Sometimes, it can also be very frustrating. However, like any other project, it is the people you work with who make it an enjoyable experience. I would like to thank Grace Buechlein for her patience and encouragement, and my co-authors, Kunal Mittal, who also acted as the technical editor, and Andrew Yang, the EJB guru, for their contributions.

Chapter 1: Relational Databases

In This Chapter

The purpose of this chapter is to lay the groundwork for the rest of the book by explaining the underlying concepts of Relational Database Management Systems. Understanding these concepts is the key to successful Java database programming. In my experience, just understanding how to handle the Java side of the problem is not enough. It is important to understand how relational databases work and to have a reasonable command of Structured Query Language (SQL) before you can do any serious Java database programming.

Understanding Relational Database Management Systems

A *database* is a structured collection of meaningful information stored over a period of time in machine-readable form for subsequent retrieval. This definition is fairly intuitive and says nothing about structure or methodology. By this definition, any file or collection of files can be considered a database. However, to be useful in practical terms, a database must form part of a system that provides for the management of the data it contains. Seen from this perspective, a database must be more than a mere collection of files. It must be a complete system.

A *practical* database management system combines the physical storage of data with the capability to manage and interact with the data. Such a system must support the following tasks:

- Creation and management of a logical data structure
- Data entry and retrieval
- Manipulation of the data in a logical and consistent manner
- Storage of data reliably over a significant period of time

Prior to the development of modern relational databases, a number of different approaches were tried. In many cases, these were simple, proprietary data-storage systems designed around a specific application. However, large corporations, notably IBM, were marketing more general solutions.

The Relational Model

The big step forward in database technology was the development of the *relational database model.* The relational database derives from work done in the late 1960s by E.F. Codd, a mathematician at IBM. His model is based on the mathematics of set theory and predicate logic. In fact, the term *relational* has its roots in the mathematical

terminology of Codd's paper entitled "A relational model of data for large shared data banks," which was published in *Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377-387. In this paper, Codd uses the terms *relation*, *attribute*, and *tuple* where more common programming usage refers to *table*, *column*, and *row*, respectively.

The importance of Codd's ideas is such that the term "database" generally refers to a relational database. Similarly, in common usage, a Database Management System, or DBMS, generally means a Relational Database Management System. The terms are used interchangeably in this chapter, and throughout the book.

Codd's model covers the three primary requirements of a relational database: structure, integrity, and data manipulation. The fundamentals of the relational model are as follows:

- A relational database consists of a number of unordered tables.
- The structure of these tables is independent of the physical storage medium used to store the data.
- The contents of the tables can be manipulated using nonprocedural operations that return tables.

The implementation of Codd's relational model means that a user does not need to understand the physical structure of the data in order to access and manage data in the database. Rather than accessing data by referring to files or using pointers, the user accesses data through a common tabular architecture. The relational model maintains a clear distinction between the logical views of the data presented to the user and the physical structure of the data stored in the system.

Codd based his model on a simple tabular structure, though his term for a table was a *relation*. Each table is made up of one or more rows (or *tuples*). Each row contains a number of fields, corresponding to the columns or *attributes* of the table.

Throughout the rest of this book, the more common programming terms are used: table, column, and row. Generally, only database theorists use Codd's original terminology; in that context, you are most likely to see references to relations, attributes, and tuples.

The tabular structure Codd defines is simple and relatively easy for the user to understand. It is also sufficiently general to be capable of representing most types of data in virtually any kind of structure. An additional advantage of a tabular structure is that tables are amenable to manipulation by a clearly defined set of mathematical operations that generate results that are also in the form of tables. These mathematical operations lend themselves readily to implementation in a high-level language. In fact, Codd's rules require that a high level language be incorporated in the RDBMS for just this purpose. That language has evolved into the Structured Query Language, SQL, discussed in subsequent chapters.

The use of a high-level language to manipulate the data at the logical level is an important feature, providing a level of abstraction which lets the user insert or retrieve data from the tables based on attributes of the data rather than its physical structure. For example, rather than requiring the user to retrieve a number stored in a certain location on disk, the use of a high-level query language allows the user to request the checking balance of a particular customer's account by account number or customer name.

A further advantage of this approach is that, while the user defines his or her requests in logical terms, the database management system (DBMS) can implement them in a highly optimized manner with respect to the physical implementation of the storage system. By decoupling the logical operations from the physical operations, the DBMS can achieve a combination of user friendliness and efficiency that would not otherwise be possible.

Codd's Rules

When Codd initially presented his paper, the meaning of the relational model he described was not widely understood. To clarify his ideas, Codd published his famous Fidelity Rules, which are summarized in Table 1-1. In theory, a RDBMS must conform to these rules. As it turns out, some of these rules are extremely difficult to implement in practice, so no existing RDBMS complies fully.

Table 1-1: Codd's Rules

Rule	Name	Description
0	Foundation Rule	A RDBMS must use its relational facilities exclusively to manage the database.
1	Information Rule	All data in a relational database must be explicitly represented at the logical level as values in tables and in no other way.
2	Guaranteed Access Rule	Every data element must be logically accessible through the use of a combination of its primary key name, primary key value, table name, and column name.
3	Systematic Nulls Rule	The RDBMS is required to support a representation of missing and inapplicable information that is systematic, distinct from all regular values, and independent of data type.
4	Dynamic Catalog Rule	The database description or catalog must also be stored at the logical level as tabular values. The relational language must be able to act on the database design in the same manner in which it acts on data stored in the

Table 1-1: Codd's Rules

Rule	Name	Description
		structure.
5	Sub Language Rule	An RDBMS must support a clearly defined data-manipulation language that comprehensively supports data manipulation and definition, view definition, integrity constraints, transactional boundaries, and authorization.
6	View Update Rule	Data can be presented to the user in different logical combinations called views. All views must support the same range of data-manipulation capabilities as are available for tables.
7	High Level Language Rule	An RDBMS must be able to retrieve relational data sets. It has to be capable of inserting, updating, retrieving, and deleting data as a relational set.
8	Physical Data Independence Rule	Data must be physically independent of application programs.
9	Logical Data Independence Rule	Applications software must be independent of changes made to the base tables.
10	Integrity Independence Rule	Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints when appropriate without unnecessarily affecting existing applications.
11	Distribution Independence Rule	Existing applications should continue to operate successfully when a distributed version of the DBMS is introduced or when existing distributed data is redistributed around the system.
12	Non Subversion Rule	If an RDBMS has a low-level (record-at-a-time) interface, that interface cannot be used to subvert the system or to bypass a relational security or integrity constraint.

Rather than explaining Codd's Rules in the order in which they are tabulated, it is much easier to explain the practical implementation of a RDBMS and to refer to the relevant rules in the course of the explanation. For example, Rule 1, the Information Rule, requires that all data be represented as values in tables; it is important to understand the idea of tables before moving on to discuss Rule 0, which requires that the database be managed in accordance with its own rules for managing data.

Tables, Rows, Columns, and Keys

Codd's Information Rule (Rule 1) states that all data in a relational database must be explicitly represented at the logical level as values in tables and in no other way. In other words, tables are the basis of any RDBMS. *Tables* in the relational model are used to represent collections of objects or events in the real world. A single table should represent a collection of a single type of object, such as customers or inventory items.

All relational databases rely on the following design concepts:

- All data in a relational database is explicitly represented at the logical level as values in tables.
- Each cell of a table contains the value of a single data item.
- Cells in the same column are members of a set of similar items.
- Cells in the same row are members of a group of related items.
- Each table defines a key made up of one or more columns that uniquely identify each row.

The preceding ideas are illustrated in Table 1-2, which shows a typical table of names and addresses from a relational database. Each row in the table contains a set of related data about a specific customer. Each column contains data of the same kind, such as First Names, or Middle Initials, and each cell contains a unique piece of information of a given type about a given customer.

Table 1-2: Customers Table

ID	FIRST_NAME	MI	LAST_NAME	STREET	CITY	ST	ZIP
100	Michael	A	Corleone	123 Pine	New York	NY	10006
101	Fredo	V	Corleone	19 Main	New York	NY	10007
103	Francis	X	Corleone	17 Main	New York	NY	10005
106	Kay	K	Adams	109 Maple	Newark	NJ	12345
107	Francis	F	Coppola	123 Sunset	Hollywood	CA	23456
108	Mario	S	Puzo	124 Vine	Hollywood	CA	23456

The ID column is a little different from the other columns in that, rather than containing information specific to a given customer, it contains a unique, system assigned identifier for the customer. This identifier is called the primary key. The importance of the primary key is discussed in Chapter 2.

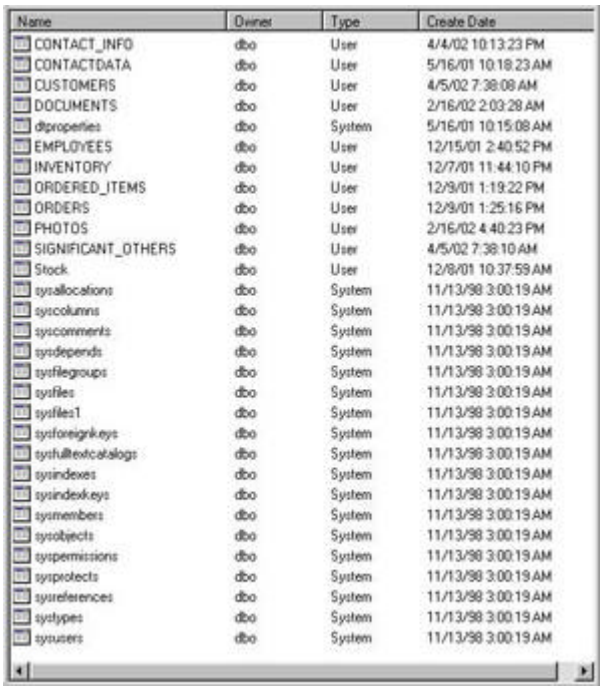
This simple table illustrates two of the most significant requirements of a relational database, which are as follows:

- All data in a relational database is explicitly represented at the logical level as values in tables.
- Every data element is logically accessible through the use of a combination of its primary key name, primary key value, table name, and column name.

It is also apparent from the example that the order of the rows is not significant. Each row contains the same information regardless of whether the rows are ordered alphabetically, ordered by state, or, as in the example, ordered by ID.

Codd's Foundation Rule (Rule 0) states that a RDBMS must use its relational facilities exclusively to manage the database; his Dynamic Catalog Rule (Rule 4) states that the database description or catalog must also be stored at the logical level as tabular values and that the relational language must be able to act on the database design in the same manner in which it acts on data stored in the structure.

These rules are implemented in most RDBMS systems through a set of system tables. These tables can be accessed using the same database management tools used to access a user database. Figure 1-1 shows a SQL Server display of the tables in the Customers database discussed in this book. The system tables are normally displayed in lower case in SQL Server, so I usually use upper case names for my own application specific tables. The table syscolumns, for example, is SQL Server's table of all the columns in all the tables in this database. If you open it, you will find entries for each of the columns specified in the Customers Table shown above, as well as every other column used anywhere in the database.



Name	Owner	Type	Create Date
CONTACT_INFO	dbo	User	4/4/02 10:13:23 PM
CONTACTDATA	dbo	User	5/16/01 10:18:23 AM
CUSTOMERS	dbo	User	4/5/02 7:38:08 AM
DOCUMENTS	dbo	User	2/16/02 2:03:28 AM
dtproperties	dbo	System	5/16/01 10:15:08 AM
EMPLOYEES	dbo	User	12/15/01 2:40:52 PM
INVENTORY	dbo	User	12/7/01 11:44:10 PM
ORDERED_ITEMS	dbo	User	12/9/01 1:19:22 PM
ORDERS	dbo	User	12/9/01 1:25:16 PM
PHOTOS	dbo	User	2/16/02 4:40:23 PM
SIGNIFICANT_OTHERS	dbo	User	4/5/02 7:38:10 AM
Stock	dbo	User	12/8/01 10:37:59 AM
sysallocations	dbo	System	11/13/98 3:00:19 AM
syscolumns	dbo	System	11/13/98 3:00:19 AM
syscomments	dbo	System	11/13/98 3:00:19 AM
sysdepends	dbo	System	11/13/98 3:00:19 AM
sysfilegroups	dbo	System	11/13/98 3:00:19 AM
sysfiles	dbo	System	11/13/98 3:00:19 AM
sysfiles1	dbo	System	11/13/98 3:00:19 AM
sysforeignkeys	dbo	System	11/13/98 3:00:19 AM
sysfulltextcatalogs	dbo	System	11/13/98 3:00:19 AM
sysindexes	dbo	System	11/13/98 3:00:19 AM
sysindexkeys	dbo	System	11/13/98 3:00:19 AM
sysmembers	dbo	System	11/13/98 3:00:19 AM
sysobjects	dbo	System	11/13/98 3:00:19 AM
syspermissions	dbo	System	11/13/98 3:00:19 AM
sysprotects	dbo	System	11/13/98 3:00:19 AM
sysreferences	dbo	System	11/13/98 3:00:19 AM
systypes	dbo	System	11/13/98 3:00:19 AM
sysusers	dbo	System	11/13/98 3:00:19 AM

Figure 1-1: SQL Server creates application tables (uppercase) and system tables (lowercase) to manage databases.

Codd's Physical Data Independence Rule (Rule 8), which states that data must be physically independent of application programs, is also clearly implemented through the tabular structure of an RDBMS. All application programs interface with the tables at a logical level, independent of the structure of both the table and of the underlying storage mechanisms.

Nulls

In a practical database, situations arise in which you either don't know the value of a data element or don't have an applicable value. For example, in Table 1-2, what if you don't know the value of a particular data item? What if, for example, Francis Xavier Corleone changed his name to just plain Francis Corleone, with no middle initial? Does that blow away the whole table? The answer lies in the concept of *systematic nulls.*

Codd's Systematic Nulls Rule (Rule 3) states that the RDBMS is required to support a representation of missing and inapplicable information that is systematic, distinct from all regular values, and independent of data type. In other words, a relational database must allow the user to insert a NULL when the value for a field is unknown or not applicable. This results in something like the example in Table 1-3.

Table 1-3: Inserting NULLs into a Table

ID	FIRST_NAME	MI	LAST_NAME	STREET	CITY	ST	ZIP
103	Francis	<NULL>	Corleone	17 Main	New York	NY	10005

Clearly, the requirement to support NULLS means that the RDBMS must be able to handle NULL values in the course of normal operations in a systematic way. This is managed through the ability to insert, retrieve, and test for NULLS and to specify NULLS as valid or invalid column values.

Primary Keys

Codd's Guaranteed Access Rule (Rule 2) states that every data element must be logically accessible through the use of a combination of its primary key name, primary key value, table name, and column name. This is guaranteed by designating a primary key that contains a unique value for each row in the table. Each table can have only one primary key, which can be any column or group of columns in the table having a unique value for each row.

It is worth noting that, while most relational database management systems will let you create a table without a primary key, the usability of the table will be compromised if you fail to assign a primary key. The reason for this is that one of the strengths of a relational database is the ability to link tables to each other. These links

between tables rely on using the primary key as a linking mechanism, as discussed in Chapter 2.

Primary keys can be *simple* or *composite*. A simple key is a key made up of one column, whereas a composite key is made up of two or more columns. Although there is no absolute rule as to how you select a column or group of columns for use as a primary key, the decision should usually be based upon common sense. In other words, you should base your choice of a primary key upon the following factors:

- Use the smallest number columns necessary, to make key access efficient.
- Use columns or groups of columns that are unlikely to change, since changes will break links between tables.
- Use columns or groups of columns that are both simple and understandable to users.

In practice, the most common type of key is a column of unique integers specifically created for use as the primary key. The unique integer serves as a row identifier or ID for each row in the table. Oracle, in fact, defines a special ROW_ID pseudo column, and Access has an AutoNumber data type commonly used for this purpose. You can see how this works in Table 1-2.

Another good reason to use a unique integer as a primary key is that integer comparisons are far more efficient than string comparisons. This means that accessing data using a single integer as a key is faster than using a string or, in the case of a multiple column key, several integers or strings.

Note Since primary keys are used as unique row identifiers, they can never have a NULL value. The NOT NULL integrity constraint must be applied to a column designated as a primary key. Many Relational database Management Systems apply the NOT NULL constraint to primary keys automatically.

Foreign Keys

A *foreign key* is a column in a table used to reference a primary key in another table. If your database contains only one table, or a number of unrelated tables, you won't have much use for your primary key. The primary key becomes important when you need to work with multiple tables. For example, in addition to the Customers Table (Table 1-2), your business application would probably include an Inventory Table, an Orders Table, and an Ordered Items Table. The Inventory Table is shown in Table 1-4.

Table 1-4: Inventory Table

Item_Number	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	178	1.95

Table 1-4: Inventory Table

Item_Number	Name	Description	Qty	Cost
1002	Rice Krispies	Cereal	97	1.87
1003	Shredded Wheat	Cereal	103	2.05
1004	Oatmeal	Cereal	15	0.98
1005	Chocolate Chip	Cookies	217	1.26
1006	Fig Bar	Cookies	162	1.57
1007	Sugar Cookies	Cookies	276	1.03
1008	Cola	Soda	144	0.61
1009	Lemon	Soda	96	0.57
1010	Orange	Soda	84	0.71

Notice that the Inventory Table includes an Item_Number column, which is the primary key for this table.

When a customer places an order, two additional tables come into use. The first of these is the Ordered Items Table, which lists the inventory items and quantities in each order. This is shown in Table 1-5.

Table 1-5: Ordered Items Table

ID	Order_Number	Item_Number	Qty
5000	2	1001	2
5001	2	1004	1
5002	2	1005	4
5003	2	1010	6
5004	3	1006	4
5005	3	1009	2
5006	4	1002	5
5007	4	1003	2
5008	5	1006	3
5009	5	1007	1
5010	5	1008	2

In addition to its primary key, the Ordered Items Table contains two foreign keys. In this case, they are the Item_Number, from the Inventory Table, and the Order_Number, from the Orders Table. The Orders Table is shown in Table 1-6.

Table 1-6: Orders Table

Order_Number	Customer_ID	Order_Date	Ship_Date
2	101	12/8/01	12/10/01
3	103	12/9/01	12/11/01
4	104	12/9/01	12/11/01
6	120	12/12/01	12/14/10
5	106	12/10/01	12/12/01
7	121	12/14/01	12/16/01

The Orders Table contains all the information defining the customer's order. Its primary key is the Order_Number column, and it contains the foreign key column Customer_ID, pointing to the Customers table, to identify the customer placing the order.

Notice that the way these tables have been designed eliminates redundancy. No item of information is saved in more than one place, and each piece of information is saved as a single row in the appropriate table.

Cross-Reference Eliminating redundancy is an important aspect of database design. By ensuring that information is stored in only one place, the problems resulting from discrepancies between different copies of the same data item are eliminated.

It is easy to understand how the keys are used if you analyze one of the orders. For example, you can find out all about the customer who placed order 4 by looking up customer 104 in the Customers Table. Similarly, by referring to the Ordered_Items Table, you can see that the items ordered on order 4 were 5 of inventory item 1002 and 2 of inventory item 1003. Looking these numbers up in the Inventory Table tells you that inventory item number 1002 refers to Rice Krispies, while inventory item number 1003 refers to Shredded Wheat.

By combining the information in these tables, you can see that order 4 was placed by customer 104, Vito Corleone, on 12/9/01, and that he ordered 5 boxes of Rice Krispies and 2 boxes of Shredded Wheat, inventory numbers 1002 and 1003, respectively, for shipment on 12/11/01. This information is obtained by matching up the various keys, using a SQL statement such as the following:

```
SELECT c.First_Name, c.Last_Name, i.Name, oi.Qty
FROM CUSTOMERS c, ORDERS o, ORDERED_ITEMS oi,
     INVENTORY i
WHERE o.Order_Number = 4 AND
    c.Customer_Id = o.Customer_Id AND
    i.Item_Number = oi.Item_Number AND
     o.Order_Number = oi.Order_Number;
```

SQL commands such as the SELECT command shown above are reviewed briefly later in this chapter and are discussed in considerable detail in subsequent chapters.

Relationships

As illustrated in the preceding discussions of primary and foreign keys, they are defined to model the relationships among the different tables in a database. These tables can be related in one of three ways:

- One-to-one
- One-to-many
- Many-to-many

One-to-one relationships

In a *one-to-one* relationship, every row in the first table has a corresponding row in the second table. This type of relationship is often created to separate different types of data for security reasons. For example, you might want to keep confidential information such as credit-card data separate from less restricted information.

Another common reason for creating tables with a one-to-one relationship is to simplify implementation. For example, if you are creating a Web application involving several forms, you might want to use a separate table for each form.

Other reasons for breaking a table into smaller parts with one-to-one relationships between them are to improve performance or to overcome inherent restrictions such as the maximum column count that a database system supports.

Tables related in a one-to-one relationship should always have the same primary key. This is used to perform joins when the related tables are queried together.

One-to-many relationships

In a *one-to-many* relationship, every row in the first table can have zero, one, or many corresponding rows in the second table. But for every row in the second table, there is exactly one row in the first table. For example, there is a one-to-many relationship between the Orders Table and the Ordered_Items Table reviewed previously.

One-to-many relationships are also sometimes called *parent-child* or *master-detail* relationships because they are commonly used for lookup tables. The relationship between the Orders Table and the Ordered_Items Table is an example of a one-to-many relationship, where a single order corresponds to multiple ordered items.

Many-to-many relationships

In a *many-to-many* relationship, every row in the first table can have many corresponding rows in the second table, and every row in the second table can have many corresponding rows in the first table. Many-to-many relationships can't be directly modeled in a relational database. They must be broken into multiple one-to-many relationships.

The Ordered_Items Table illustrates how a many-to-many relationship can be broken into multiple one-to-many relationships. In the customer orders example illustrated by Tables 1-4 through 1-6, orders and inventory are related in a many-to-many relationship; multiple inventory items can correspond to a single order, and a single inventory item can appear on multiple orders. The Ordered_Items Table is used to implement a one-to-many mapping of inventory items to orders.

Views

Codd's View Update Rule (Rule 6) states that data can be presented to the user in different logical combinations, called views. All views must support the same range of data-manipulation capabilities as are available for a table.

Views are implemented in a relational database system by allowing the user to select data from the database to create temporary tables, known as *views*. These views are usually saved by name along with the selection command used to create them. They can be accessed in exactly the same way as normal tables.

Frequently, views are used to create a table that is a subset of an existing table. Table 1-7 is a typical example, showing rows from Table 1-2 (where Last_Name = 'Corleone', and City = 'New York').

Table 1-7: View of New York Corleones

ID	FIRST_NAME	MI	LAST_NAME	STREET	CITY	ST	ZIP
100	Michael	A	Corleone	123 Pine	New York	NY	10006
101	Fredo	X	Corleone	19 Main	New York	NY	10007
103	Francis	X	Corleone	17 Main	New York	NY	10005

Normalization

Normalization is the process of organizing the data in a database by making it conform to a set of rules known as the normal forms. The *normal forms* are a set of design guidelines that are designed to eliminate redundancies and to ensure consistent dependencies. Apart from wasting space, redundant data creates maintenance problems. For example, if you save a customer's address in two locations, you need to be absolutely certain to make any required changes in both locations.

It is important to ensure that data dependencies are consistent so that you can access and manipulate data in a logical and consistent manner. A glance at the examples shown in Tables 1-2 and 1-4 through 1-6 reveals how related data items are stored in the same table, separate from unrelated items.

Although normalization enhances the integrity of the data by minimizing redundancy and inconsistency, it does so at the cost of some impact on performance. Data-retrieval efficiency can be reduced, since applying the normalization rules can result in data being redistributed across multiple records. This can be seen from the examples shown in Tables 1-2 and 1-4 through 1-6, where information pertaining to a single order is distributed across four separate tables.

A database that conforms to the normalization rules is said to be in *normal form*. If the database conforms to the first rule, the database is said to be in *first normal form,* abbreviated as *1NF*. If it conforms to the first four rules, the database is considered to be in *fourth normal form (4NF)*.

First normal form

The requirements of the first normal form are as follows:

- All records have the same number of fields.
- All fields contain only a single data item.
- There must be no repeated fields.

The first of these requirements, that all occurrences of a record type must contain the same number of fields, is a built-in feature of all database systems.

The second requirement, that all fields contain only one data item, ensures that you can retrieve data items individually. This requirement is also known as the *atomicity* requirement. Requiring that each data item be stored in only one field in a record is important to ensure data integrity.

Finally, each row in the table must be identified using a unique column or set of columns. This unique identifier is the primary key.

Second normal form

The requirements of the second normal form are as follows:

- The table must be in first normal form.
- The table cannot contain fields that do not contain information related to the whole of the key.

The second normal form is only relevant when a table has a multipart key. In the example shown in table 1-8, which shows inventory for each warehouse, the primary key, which is the unique means of identifying a row, consists of two fields, the Name field and the Warehouse field.

Second normal form requires that a table should only contain data related to one entity, and that entity should be described by its primary key. The Warehouse Inventory table is intended to describe inventory items in a given warehouse, so all the data describing the inventory item itself is related to the primary key.

In the example of Table 1-8, the second row shows that there are 97 cases of Rice Krispies in warehouse #2, purchased at a unit cost of $1.95, and 103 cases of Rice Krispies in warehouse #7, purchased at a unit cost of $2.05. The warehouse address, however, describes only part of the key, namely, the warehouse, so it does not belong in the table. If this information is stored with every inventory item, there is a potential risk of discrepancies between the address saved for a given warehouse in different rows, since there is no clearly defined master reference. In addition, of course, storing the same data item in multiple locations is very inefficient in terms of space, and requires that any change to the data item be made to all rows containing the data item, rather than to a single master reference.

Table 1-8: Warehouse Inventory Table

Name	Warehouse	Address	Description	Qty	Cost
Corn Flakes	Warehouse #2	123 Pine	Cereal	178	1.95
Rice Krispies	Warehouse #2	123 Pine	Cereal	97	1.95
Rice Krispies	Warehouse #7	24 Holly	Cereal	103	2.05
Oatmeal	Warehouse #7	24 Holly	Cereal	15	0.98

The solution is to move the warehouse address to a Warehouse table linked to the Inventory table by a foreign key. The resulting tables would look like Tables 1-9 and 1-10. These tables are in the second normal form.

Table 1-9: Inventory Table in 2NF

Name	Warehouse	Description	Qty	Cost

Table 1-9: Inventory Table in 2NF

Name	Warehouse	Description	Qty	Cost
Corn Flakes	Warehouse #2	Cereal	178	1.95
Rice Krispies	Warehouse #2	Cereal	97	1.95
Rice Krispies	Warehouse #7	Cereal	103	2.05
Oatmeal	Warehouse #7	Cereal	15	0.98

Table 1-10: Warehouse Table in 2NF

Warehouse	Address
Warehouse #2	123 Pine
Warehouse #7	24 Holly

In summary, the second normal form requires that any data that is not directly related to the entire key should be removed and placed in a separate table or tables. These new tables should be linked to the original table using foreign keys. In the example of Tables 1-9 and 1-10, the Warehouse column is both part of the primary key of Table 1-9, and the foreign key pointing to Table 1-10.

Third normal form

The requirements of the third normal form are as follows:

- The table must be in second normal form.
- The table cannot contain fields that are not related to the primary key.

Third normal form is very similar to second normal form, with the exception that it covers situations involving simple keys rather than compound keys. In the example used to explain the second normal form, a compound key was used because inventory items of the same type, such as Rice Krispies, could have different attributes such as Warehouse number. If you are tracking unique items, such as employees, you can have a similar situation, but with a simple key, as shown in Table 1-11:

Table 1-11: Employee Table

Name	Department	Location
Jones	Sales	43 Elm
Smith	Production	17 Main
Williams	Shipping	123 Pine

In the example of Table 1-11, the Location column describes the location of the Department. The employee is located there because he or she belongs to that department. As in the example for the second normal form, columns that do not contain data describing the primary key should be removed to a separate table. In this instance, that means that you should create a separate Departments table, containing the Department name and location, using the Department column in the Employees table as a foreign key to point to the Departments table. The resulting tables are shown in Tables 1-12 and 1-13.

Table 1-12: Normalised Employee Table

Name	Department
Jones	Sales
Smith	Production
Williams	Shipping

Table 1-13: Departments Table

Department	Location
Sales	43 Elm
Production	17 Main
Shipping	123 Pine

Fourth normal form

The requirements of the fourth normal form are as follows:

- The table must be in third normal form.
- The table cannot contain two or more independent multivalued facts about an entity.

For example, if you wanted to keep track of customer phone numbers, you could create a new table containing a Customer_ID number column, a phone number column, a fax number column, and a cell-phone number column. As long as a customer has only one of each listed in the table, there is no problem. However, if a customer has two land line phones, a fax, and two cell phones, you might be tempted to enter the numbers as shown in Table 1-14.

Table 1-14: Phone Numbers Table which violates 4NF

CUSTOMER_ID	PHONE	FAX	CELL
100	123-234-3456	123-234-3460	121-345-5678
100	123-234-3457	<NULL>	121-345-5679

Since there is no relationship between the different phone numbers in a given row, this table violates the fourth normal form, in that there are two or more independent multivalued facts (or phone numbers) for the customer on each row. The *combinations* of land line, fax, and cell phone numbers on a given row are not meaningful.

The main problem with violating the fourth normal form is that there is no obvious way to maintain the data. If, for example, the customer decides to give up the cell phone listed in the first row, should the cell phone number in the second row be moved to the first row, or left where it is? If he or she gives up the land line phone in the second row and the cell phone in the first row, should all the phone numbers be consolidated into one row? Clearly, the maintenance of this database could become very complicated.

The solution is to design around this problem by deleting the phone, fax, and cell columns from the original table, and creating an additional table containing Customer_ID as a foreign key, and phone number and type as data fields (see Table 1-15). This will allow you to handle several phone numbers of different types for each customer without violating the fourth normal form.

Table 1-15: Phone Numbers Table

CUSTOMER_ID	NUMBER	TYPE
100	123-234-3456	PHONE
100	123-234-3457	PHONE
100	123-234-3460	FAX
100	121-345-5678	CELL
100	121-345-5679	CELL

Fifth normal form

The requirements of the fifth normal form are as follows:

- The table must be in fourth normal form.
- It must be impossible to break down a table into smaller tables unless those tables logically have the same primary key as the original table.

The fifth normal form is similar to the fourth normal form, except that where the fourth normal form deals with independent multivalued facts, the fifth normal form deals with interdependent multivalued facts. Consider, for example, a dealership handling several similar product lines from different vendors. Before selling any product, a salesperson must be trained on the product. Table 1-16 summarizes the situation.

Table 1-16: SalesPersons

Salesperson	Vendor	Product
Jones	Acme Widget	Printer
Jones	Acme Widget	Copier
Jones	Zeta Products	Printer
Jones	Zeta Products	Copier

This table contains a certain amount of redundancy, which can be removed by converting the data to the fifth normal form. Conversion to the fifth normal form is achieved breaking the table down into smaller tables, as shown in Tables 1-17, 1-18, and 1-19.

Table 1-17: SalesPersons by Vendor

Salesperson	Vendor
Jones	Acme Widget
Jones	Zeta Products

Table 1-18: SalesPersons by Product

Salesperson	Product
Jones	Printer
Jones	Copier

Table 1-19: Products by Vendor

Vendor	Product
Acme Widget	Printer
Acme Widget	Copier
Zeta Products	Printer
Zeta Products	Copier

Boyce-Codd normal form

Boyce-Codd normal form (BCNF) is a more rigorous version of the third normal form designed to deal with tables containing the following items:

- Multiple candidate keys
- Composite candidate keys
- Candidate keys that overlap

A relational table is in BCNF only if every column on which some of the columns are fully functionally dependent is a candidate key. In other words, if the table has a number of columns or groups of columns which could be used as the primary key (so-called candidate keys), then to be in BCNF, the table must be in third normal form for each of those candidate keys.

Normalization in Practice

Most databases can be considered to be adequately normalized when they are in the fifth normal form. In the fifth normal form, a database has the following important properties:

- All records have the same number of fields.
- All fields contain only a single data item.
- There are no repeated fields.
- All fields contain data related to the whole of the primary key.
- The table does not contain two or more independent multivalued facts about the key.
- The table does not contain two or more interdependent multivalued facts about the key.

Additional normal forms address situations that only apply in special situations. For example, Boyce-Codd normal form requires that a table be in third normal form for every column or group of columns which has the properties which could qualify it for use as the primary key. In practice, the database designer will usually designate a primary key, so the Boyce-Codd normal form will not be relevant, as other candidate keys will not be used, so the third normal form is adequate.

High Level Language

Codd's Sub Language Rule (Rule 5) and his High Level Language Rule (Rule 7) concern the availability of a language for use with the database. Descriptions of these rules are restated here:

- **Sub Language Rule** — An RDBMS must support a clearly defined data-manipulation language that comprehensively supports data manipulation and definition, view definition, integrity constraints, transactional boundaries, and authorization.
- **High Level Language Rule** — An RDBMS must be able to retrieve relational data sets. It has to be capable of insert, update, retrieve and delete data as a relational set.

The main features of this language are that it must have a linear syntax and must support the following functions:

- Data-definition operations (including view definitions)
- Data update and retrieval operations
- Data-integrity constraints
- Transaction management

- Data-security constraints

The standard implementation of these rules is the Structured Query Language (SQL).

Structured Query Language

The Structured Query Language (SQL) was first developed by IBM in the 1970s and was later the subject of several ANSI standards. As a result of the way that the requirements for a high-level database language are defined, SQL is usually considered to be composed of a number of sublanguages. These sublanguages are as follows:

- Data Definition Language (DDL) is used to create, alter, and drop tables and indexes.
- Data Manipulation Language (DML) is used to insert, update, and delete data.
- Data Query Language (DQL) is used to query the database using the SELECT command.
- Transaction Control Commands are used to start, commit, or rollback transactions.
- Data Control Language (DCL) is used to grant and revoke user privileges and to change passwords.

Despite the conventional division of SQL into a number of sublanguages, statements from any of these constituent sublanguages can be used together. The convention is really just a reflection of the way Codd's rules define the requirement for a high level language, with sublanguages for different functions.

The next three sections provide a brief outline of the sublanguages used to perform the basic database functions of creating databases and tables, populating the tables with data, and retrieving the data. These functions are performed by the DDL, the DML, and the DQL sublanguages.

Data Definition Language

Data definition operations are handled by SQL's Data Definition Language, which is used to create and modify a database. The SQL2 standard refers to DDL statements as "SQL Schema Statements." The SQL standard defines a Schema as a high level abstraction of a container object which contains other database objects.

A good example of the use of the DDL is the creation of a table. When a table is created, various parameters are set for each column. These include the following:

- **Data types.** These include CHARACTER,INTEGER, FLOAT, and so on.
- **Data constraints.** These include such restrictions as whether NULLS are permitted.
- **Default values.** Default values can be assigned for each column.

The basic form of the CREATE TABLE command is:

```
CREATE TABLE tableName
   ( columnName dataType[(size)] [constraints] [default value],...);
```

Integrity constraints and triggers

It is obvious from the earlier discussion of primary and foreign keys that the idea of linking tables through the use of keys can go completely haywire if a primary key has either a NULL value or a value that is not unique. Problems like this are handled using constraints. The main types of constraint are as follows:

- NULL or NOT NULL constraint specifies whether a field is required to contain valid data or whether it can be left empty.
- The UNIQUE constraint specifies that no two records can have the same value in a particular column.
- The PRIMARY KEY constraint specifies that this column is the primary key for the table.

In addition to defining constraints, the SQL language allows the user to specify security rules that are applied when specified operations are performed on a table. These rules are known as *triggers*, and work like stored procedures, with the exception that, instead of being called by name, they are triggered automatically by the occurrence of a database event such as updating a table.

A typical use of a trigger might be to check the validity of an update to an inventory table. The following code snippet shows a trigger that automatically rolls back or voids an attempt to increase the cost of an item in inventory by more than 15 percent:

```
CREATE TRIGGER FifteenPctRule ON INVENTORY FOR INSERT, UPDATE AS
DECLARE @NewCost money
DECLARE @OldCost money
SELECT @NewCost = cost FROM Inserted
SELECT @OldCost = cost FROM Deleted
IF @NewCost > (@OldCost * 1.15)
ROLLBACK Transaction
```

Transaction management and the SQL ROLLBACK command are discussed later in this chapter and in more detail in subsequent chapters.

Data Manipulation Language

The Data Manipulation Language comprises the SQL commands used to insert data into a table and to update or delete data. SQL provides the following three statements you can use to manipulate data within a database:

- INSERT
- UPDATE

- DELETE

The INSERT statement is used to insert data into a table, one row or record at a time. It can also be used in combination with a SELECT statement to perform bulk inserts of multiple selected rows from another table or tables.

The UPDATE command is used to modify the contents of individual columns within a set of rows. The UPDATE command is normally used with a WHERE clause, which is used to select the rows to be updated.

The DELETE command is used to delete selected rows from a table. Again, row selection is based on the result of an optional WHERE clause.

Data Query Language

The Data Query Language is the portion of SQL used to retrieve data from a database in response to a query. The SELECT statement is the heart of a SQL query. In addition to its use in returning data in a query, it can be used in combination with other SQL commands to select data for a variety of other operations, such as modifying specific records using the UPDATE command.

The most common way to use SELECT, however, is as the basis of data retrieval commands, or queries, to the database. The basic form of a simple query specifies the names of the columns to be returned and the name of the table they can be found in. A basic SELECT command looks like this:

```
SELECT columnName1, columnName2,.. FROM tableName;
```

In addition to this specific form, where the names of all the fields you want returned are specified in the query, SQL supports a wild-card form. In the wild-card form, an asterisk (*) is substituted for the column list, as shown here:

```
SELECT * FROM tableName;
```

The wild card tells the database management system to return the values for all columns.

The real power of the SELECT command comes from the use of the WHERE clause. The WHERE clause allows you to restrict the query to return the requested fields from only records that match some specific criteria. For example, you can query the Customers Table shown in Table 1-2 by using this statement:

```
SELECT * FROM Contact_Info WHERE Last_Name = 'Corleone';
```

The result of this query is to return all columns from any row containing the Last_Name "Corleone". The order in which the columns are returned is the order in which they are stored in the database; the row order is arbitrary.

Comparison operators

In addition to the equality operator used in the preceding example, SQL supports a full range of standard comparison operators, including special operators used to test for the presence or absence of a NULL value in a column:

- Equality (=)
- Inequality (<>)
- Greater Than (>) and Greater Than or Equal To (>=)
- Less Than (<) and Less Than or Equal To (<=)
- IS NULL
- IS NOT NULL

Comparison operations can be combined using the basic logical operators: AND, OR and NOT.

Another way of combining operations is to nest subqueries. The syntax for nesting subqueries uses parentheses to indicate nesting levels as shown below:

```
SELECT *
FROM Tables
WHERE
  (SUBQUERY
    (SUBQUERY
      (SUBQUERY)));
```

Sorting the results of a query

A common requirement when retrieving data from an RDBMS by using the SELECT statement is to sort the results of the query in alphabetical or numeric order on one or more of the columns. Sorting result is done using the ORDER BY clause in a statement like this:

```
SELECT First_Name, Last_Name, City, State
FROM CUSTOMERS
WHERE Last_Name = 'Corleone'
ORDER BY First_Name;
```

Joining tables

The information in a practical database is usually distributed across several tables, each of which contains sets of logically related data. A typical example of such a database is shown in Tables 1-2 and 1-4 through 1-6.

When a customer places an order, an entry is made in the Orders Table, assigning an order number and containing the Customer number and the order date. Then entries are added to the Ordered_Items Table, recording the order number, item number, and quantity.

One of the most powerful features of SQL is its ability to combine data from several tables by using JOINS. For example, the following SQL statement performs a JOIN on the ORDERS, CUSTOMERS, ORDERED_ITEMS and INVENTORY Tables to total the purchases each customer makes:

```
SELECT LAST_NAME + ',' + FIRST_NAME AS NAME,
    SUM(oi.QTY * COST * 1.6) AS PURCHASES
FROM ORDERS o, CUSTOMERS c, ORDERED_ITEMS oi,
     INVENTORY i
WHERE O.CUSTOMER_NUMBER = C.CUSTOMER_NUMBER AND
    O.ORDER_NUMBER = OI.ORDER_NUMBER AND
     OI.ITEM_NUMBER = I.ITEM_NUMBER
GROUP BY LAST_NAME + ',' + FIRST_NAME;
```

Here are the results of this query:

NAME	PURCHASES
Adams,Kay	11.14
Corleone,Francis	11.87
Corleone,Fredo	22.69
Corleone,Vito	21.52

This example also illustrates the use of *aliases* both for column names and for tables, as well as SQL's ability to perform arithmetic and String computations. Here the alias NAME has been assigned to the concatenation of the Last_Name and First_Name fields, and the alias PURCHASES to the calculated product of quantity, cost, and the 1.6 markup through the use of the expression:

```
SELECT LAST_NAME + ',' + FIRST_NAME AS NAME,
    SUM(oi.QTY * COST * 1.6) AS PURCHASES
```

The use of aliases and SQL's mathematical capabilities are discussed thoroughly in subsequent chapters.

Reporting functions

SQL supports a number of aggregation functions that can be used to provide statistical or summary information about groups of data elements. The standard aggregation functions include the following:

- Sum and Count
- Average and Standard Deviation
- Maximum and Minimum

Note Different SQL dialects expand on this basic set of aggregate functions. You are advised to refer to the documentation provided by the supplier of your particular RDBMS for details of the aggregate functions provided.

A good practical example of the use of aggregate functions is the creation of a simple sales report. The following query creates a result set that lists states and the total cost of goods sold and sales by state:

```
SELECT STATE, SUM(oi.QTY * COST) AS TOTAL,
    SUM(oi.QTY * COST * 1.6) AS SALES
FROM ORDERS o, CUSTOMERS c, ORDERED_ITEMS oi,
     INVENTORY i
WHERE O.CUSTOMER_NUMBER = C.CUSTOMER_NUMBER AND
    O.ORDER_NUMBER = OI.ORDER_NUMBER AND
     OI.ITEM_NUMBER = I.ITEM_NUMBER
GROUP BY STATE;
```

The resulting table looks like this:

STATE	TOTAL	SALES
NJ	20.41	32.65
NY	21.6	34.56

The last three sections have presented a brief discussion of creating databases and tables, populating the tables with data, and retrieving the data. For the very simplest of database operations, these capabilities may be sufficient. However, in real world applications, more complex situations arise in two main areas:

- In many practical applications, a complete operation cannot be expressed in a single SQL statement, so a means of handling multiple interdependent statements is required.
- There is frequently a need, particularly in larger installations, to provide some means of ensuring the security of an application.

These requirements are handled by the Transaction Control Commands and the Data Control Language, respectively. Since these topics require some explanation, the respective sublanguages are reviewed in the appropriate sections below.

Transaction Management and the Transaction Control Commands

Transaction management refers to the capability of an RDBMS to execute database commands in groups, known as *transactions*. A transaction is a group or sequence of commands, all of which must be executed in order and all of which must completed successfully.

The Transaction Control Commands are used to control transactions.

The ACID Test

A commonly used expression in data processing is the ACID test. The ACID test defines a set of properties that a database management system must have in order to be adequate for handling transactions. These properties are as follows:

- Atomicity
- Consistency
- Isolation
- Durability

A discussion of the preceding properties follows.

Atomicity

Transactions must be *atomic*. Specifically, a transaction must be executed in its entirety and committed as a whole or rolled back as a whole, so that either all changes that constitute a transaction take effect or none of them take effect. A classic example of an atomic transaction is a transfer of funds from a checking account to a savings account. Clearly, you want both the deduction from savings and the addition to checking to take place, failing which, neither should take place. When atomicity is not guaranteed, you have an accounting nightmare.

Consistency

The *consistency* requirement defines a transaction as *legal* only if it obeys user-defined integrity constraints. Essentially, these constraints define legal database states and proscribe transactions that cause transitions from a legal state to an illegal state. For example, if you are making a transfer of funds from a checking account to a savings account and your business rules require that such a transfer be logged to another table, any problems updating that table will violate the integrity constraint and will require that the entire transaction be rolled back.

Isolation

Isolation means that the effects of a transaction must be invisible to other transactions until the current transaction is complete. For example, if you are making a transfer of funds from a checking account to a savings account, the intermediate balances after savings have been debited, but before checking has been credited, must not be available to an outside transaction. If the intermediate balances are available to an outside transaction, you might, for example, generate an insufficient funds warning, since the funds will show up in neither account.

Durability

The *durability* requirement demands that, once committed, the results of a transaction be preserved in some form of long term storage. In other words, once a funds transfer has been made from savings to checking, the DBMS must save it to persistent storage.

Transaction Management in SQL

If anything goes wrong during the transaction, the database management system allows the entire transaction to be cancelled, or "Rolled Back." If, on the other hand, it completes successfully, the transaction can be saved to the database, or "Committed."

A transaction typically involves several related commands, as in the case of a bank transfer. If a client orders a transfer of funds from his savings account to his checking account, at least these two database-access commands must be executed:

- The savings account must be debited.
- The checking account must be credited.

If one of these commands is executed and the other is not, the funds will either vanish from the savings account without appearing in the checking account, or the funds will be credited to the checking account without being withdrawn from the savings account.

The solution is to combine logically related commands into groups that are committed as a single transaction. If a problem arises, the entire transaction can be rolled back, and the problem can be fixed without serious adverse impact on business operations.

SQL supports this requirement through the COMMIT and ROLLBACK commands. The COMMIT command commits changes made from the beginning of the transaction to the point at which the command is issued, and the ROLLBACK command undoes them.

In addition, most databases support the AUTOCOMMIT option, which tells the database management system to commit all commands individually as they are executed. This option can be turned on or off with the SET command. By default, the AUTOCOMMIT option is usually on.

Cross-Reference Chapter 3 provides a comprehensive overview of SQL; Chapters 5 through 9 give detailed examples of the use of SQL in the context of the JDBC Core API. Appendix A provides a comparison of common SQL dialects.

Database Security and the Data Control Language

Databases generally represent a significant investment of time and effort and are frequently a major corporate asset. As such, ensuring the security of a database is an important administrative consideration. The most important aspects of database security are as follows:

- Ensuring that database access is restricted to authorised and qualified personnel, generally by some extension of the password principle
- Ensuring the consistency of the database where many users are accessing and up-dating it simultaneously
- Ensuring the physical integrity of the database. At the very least, this involves making provision for back up and reloading.

Most database management systems incorporate proprietary tools to manage database security. In general, the access-control mechanisms are similar and use the SQL language.

Managing Database Users

A user, in database terms, is anyone who has access to the database. Most database management systems provide the capability of defining different users and groups of users with different access privileges and different operational roles. When a database is created, its creator has owner privileges. These allow the user to create the database and any of its components. After creation, the database may also be accessed by users who are assigned lower privileges. Data entry clerks, for example, may only have sufficient privileges to enter limited data into specific tables.

Creating a user

To add individual users to a database, the database administrator must create database users. This is done using the CREATE USER command. When you create a user, you can assign a password, certain basic permissions, and an expiration date, all in one command. You can also add the user to an existing user group.

Altering or dropping a user

During the lifetime of a database user, you may need to make modifications to his or her password or access expiration date. Similarly, you may want to modify a user's privileges. These functions are handled using the ALTER USER command.

Ultimately, you may need to remove an individual's access to the database entirely. This is done using the DROP USER command.

User Privileges

Database management systems define sets of privileges that can be assigned to users. These privileges correspond to actions that can be performed on objects in the database. This approach provides a fine degree of control of database access, allowing the database administrator to do anything he or she may need to do, while restricting clerical personnel to a lower and potentially less damaging level of access.

When a new database is created, the default owner of the database is the user who executes the CREATE command. To allow other users to work with the database, you need to assign them the privileges to do so. Privileges can be assigned either to individual users or to groups of users.

User privilege levels

User privileges can be assigned at two different levels. Users can be restricted both at the level of the types of actions they can perform, such as READ, MODIFY, or WRITE, and at the level of the types of database objects they can access.

Access level privileges can generally be assigned at the following levels:

- Global level access to all databases on a given server
- Database level access to all tables in a given database
- Table level access to all columns in a given table
- Column level access to single columns in a given table

It is obvious from the range of different access privileges provided that security is a major consideration in database implementation. Normally, the management of user privileges is an administrative function, handled by the database administrator.

Granting and revoking user privileges

The SQL GRANT command is used to grant users the necessary access privileges to perform various operations on the database. In addition to granting a user specified access privileges, the GRANT command can be used to allow the user to grant a privilege to other users. There is also an option allowing the user to grant privileges on all subtables and related tables.

The REVOKE command is used to revoke privileges granted to a user. Like the GRANT command, REVOKE can be applied at various levels.

Users Groups and Roles

In addition to defining individual users, many systems also allow the database administrator to organize users into logical groups with the same privileges. Like users, user groups and roles are managed using SQL commands.

A database role defines what operations a user or group can do on the database, such as "Create Databases," "Backup Databases," and so on. In other words, roles are simply predefined sets of user privileges.

Creating, altering and dropping groups

Groups are created in much the same way as individual users. Groups are also similar to individual users in that groups can be made part of other groups. When a group is made a part of another group, it inherits the permissions of that group, along with its own. In this way, you can create an entire hierarchy of groups and users and manipulate them in accordance with your system needs.

When a group is altered or dropped, only the group is affected. Any users in a group that is dropped simply lose their membership in the group. The users are otherwise unaffected. Similarly, when a group is altered by dropping a user, only the group is affected. The user simply loses his or her membership in the group but is otherwise unaffected.

Creating roles

User roles are simply a predefined set of user privileges. Most RDBMS systems support the following roles or their equivalents:

- Owner – A user who can read or write data and create, modify, and delete the database or its components
- Writer – A user allowed to read or write data
- Reader – Someone allowed to read data but not write to the database
- Public – The lowest possible status in terms of privileges

User roles are a neat administrative feature designed to save time for database administrators. Like groups, roles can be defined by the database administrator as required.

Database Architectures

Codd's Distribution Independence Rule (Rule 11) states that existing applications should continue to operate successfully when a distributed version of the DBMS is introduced or when existing distributed data is redistributed around the system. The need for distributed systems was seen even in the early days of computing.

In modern systems, distribution is accomplished in several different ways. The type of distribution Codd was talking about would now be considered internal to the RDBMS, so a database might be distributed across a sizeable cluster of systems and yet its distribution would be transparent to the Java database programmer, who would access it as a single RDBMS. From the perspective of the Java database programmer, a multitier architecture is a far more common form of distribution.

The system architectures that are most common in database applications are the two-tier and three-tier models. In other words, the Java application either accesses the database directly or as part of a middle tier server application. A new variation, which might be called *single tier*, is the Java Data Objects (JDO) based application, in which JDO supplies persistence, with no specific persistence code being written by the Java programmer. Container managed persistence in Enterprise JavaBeans Applications also abstracts the persistence code, though in a fundamentally different way, and as part of a multi-tier architecture.

Java Data Objects

In Java, as in any other object-oriented language, the programmer is accustomed to working primarily with objects. Relational databases, on the other hand, are organized around smaller data items, which might be considered similar to object attributes. For example, a customer object in Java might have a number of attributes such as firstName and lastName, stored individually in separate columns in a database record.

The JDO architecture supports the concept of *transparent persistence*, which is intended to hide the details of the underlying persistence mechanism from the application. The Java business logic is simply developed in the customary way. The business logic classes are then enhanced at the byte-code level to generate a persistence-capable version of the class. Almost all user-defined classes can be made persistent in this way.

Once the business logic classes have been compiled and enhanced, the application that uses the enhanced business classes can be developed. The persistence management of the business objects is transparent. In other words, the application developer never needs to fetch and store objects or their attributes at the JDBC/SQL level.

Note Although a JDO application looks and behaves like a single-tier application, the underlying persistence mechanism can be implemented

using a local RDBMS or a multitier EJB based architecture. In either case, completely transparent persistence is achieved.

Two-Tier Model

In the two-tier model, a Java application is designed to interact directly with a database. Application functionality is divided into these two layers:

- Application layer, including the JDBC driver, business logic, and user interface
- Database layer, including the RDBMS

The interface to the database is handled by a Java Database Connectivity (JDBC) Driver appropriate to the particular database management system being accessed. The JDBC Driver passes SQL statements to the database and returns the results of those statements to the application.

A client/server configuration like the one shown in Figure 1-2 is a special case of the two-tier model, where the database is located on another machine, referred to as the server. The application runs on the client machine, which is connected to the server over a network. Commonly, the network is an intranet, using dedicated database servers to support multiple clients, but it can just as easily be the Internet.

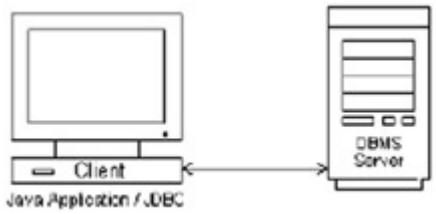


Figure 1-2: A two-tier client/server configuration is typical of office applications.

Part II of this book illustrates the use of basic JDBC and SQL functionality in the context of a basic two-tier application. That application uses simple Swing components to create a generic RDBMS graphical user interface (GUI). The inherent flexibility of a Java/JDBC approach to developing database applications enables you to access a wide range of RDBMS systems, including Oracle, Sybase, SQL Server, and MySQL, as well as MS Office applications, using this GUI.

Three-Tier Model

In the three-tier model, the client typically sends requests to an application server, forming the middle tier. The application server interprets these requests, and formats the necessary SQL statements to fulfill these requests, and sends them to the database. The database processes the SQL statements and sends the results back to the application server, which then sends them to the client.

These are some advantages of a three-tier architecture:

- Performance can be improved by separating the application server and database server.
- Business logic is clearly separated from the database.
- Client applications can use a simple protocol such as CGI to access services.

The three-tier model shown in Figure 1-3 is common in Web applications. In this scenario, the client tier is frequently implemented in a browser on a client machine; the middle tier is implemented in a Web server with a servlet engine; and the database management system runs on a dedicated database server.

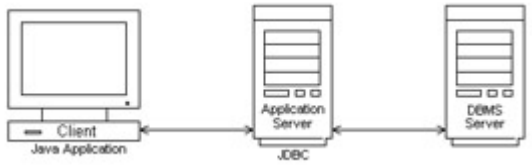


Figure 1-3: The three-tier model is typical of Web applications.

These are the main components of a three-tier architecture:

- **Client tier** — Typically, this is a thin presentation layer that may be implemented using a Web browser.
- **Middle tier** — This tier handles the business or application logic. This may be implemented using a servlet engine such as Tomcat or an application server such as JBOSS. The JDBC driver also resides in this layer.
- **Data source layer** — This component includes the RDBMS.

Part III of this book illustrates additional capabilities of the JDBC API. It provides a three-tier application that uses a Web browser as the client, an Apache/Tomcat server as the middle tier, and a relational database management system as the database tier.

Summary

This chapter has given an overview of how Relational Database Management Systems work. The key building blocks of relational databases have been introduced, and will be reviewed in more detail in subsequent chapters. The following key topics have been discussed:

- Creating and normalising databases and their constituent tables.
- Using primary and foreign keys to link tables
- Using the Structured Query Language
- Understanding transactions and transaction management

In addition, this chapter explored the underlying theory of relational databases developed by E.F. Codd, and summarized in Codd's rules and the ACID test. Although these specific topics are not discussed again, they are important in understanding why relational databases work the way they do.

Chapter 2 explores the design of a simple, but non-trivial database for an XML based application presented in Part IV. The design is derived from a specification for a legal invoicing system which uses XML to transmit invoices from legal firms to large corporate clients. The central topics discussed are the practical design of the tables required, and of the primary and foreign keys used to link them. Particular attention is also given to database integrity.

Chapter 2: Designing a Database

In This Chapter

Like most of the chapters in this book, this one is built around an example. In this case, the example is a time and materials tracking and billing system. The objective is to build a system capable of handling time and materials tracking and invoicing in accordance with the Legal Electronic Data Exchange Standard (LEDES). Being a published business, this is a good benchmark for a practical commercial application.

A primary consideration that went into defining the document structure was its compatibility with existing legacy systems. LEDES mimics the existing paper process, reflecting current time and materials invoicing practices in the legal profession.

Note Although the LEDES terminology derives from the legal profession, the system is recognizable as a special case of a time and materials billing system capable of handling multiple projects or matters within a single invoice.

Database Design Considerations

The most important consideration in designing a database is the application and its requirements. For instance, the examples discussed in Chapter 1 refer to a Customers Table, containing the names, addresses, and phone numbers of individual customers. This is a good approach when all the customers have unique addresses. However, when an application needs to store information about individuals at a corporation, many of those individuals work at the same location, so they have the same address. In this case, it makes sense to have separate tables for contacts and locations.

Like most organizations using time and materials billing, the legal profession focuses a lot of effort on tracking the hours expended on a given project. Although lawyers use slightly different terminology, and charge much higher rates, the process is much like billing for contract programming. Usually, lawyers bill on a project basis, though they call a project a *matter*. The legal term for tracking hours expended on a project is *capturing time*. Members of a law firm whose time is billable are referred to as *timekeepers*. Their time is billable at various predefined levels, ranging from partner to clerk.

On of the most important aspects of any kind of system engineering is identifying the needs of the users. In this application, the primary users are as follows:

- Lawyers capturing time
- Book keepers generating invoices

- Management tracking projects

In addition, there is a database-management requirement involving the addition of new clients and overall database maintenance. In other words, this is a fairly typical time and materials billing system.

The next step is to identify the underlying purpose of the application, since you can generally work out the rest of the application once you understand its underlying purpose. In this case, the underlying purpose is the generation of invoices for billable hours.

The Project Specification

Ideally, any application is designed in response to a detailed specification. Although it does not describe the entire accounting process in a typical law firm, the LEDES 2000 specification is a great help here, as it defines exactly what is required in an electronic invoice and how the invoice should be formatted. Originally, Price Waterhouse Coopers developed LEDES as an ASCII-based electronic billing standard. In its current form, LEDES 2000 defines an XML file format intended to serve as a standard file format that the legal industry can use for the electronic exchange of information. Initially, the focus is on billing information.

LEDES 2000 defines the following major data-content elements:

- Generating firm
- Destination client
- Alternative fee arrangements (time & expense, flat fee, contingency, and staged billing)
- Fee sharing
- Discount schedules
- Taxes
- Electronic funds transfer reference support
- Multiple clients
- Multiple matters within an invoice

The core of the LEDES specification is captured in the XML invoice document it specifies. Listing 2-1 is a slightly simplified example of a LEDES 2000 invoice. The full LEDES specification can be accessed at http://www.ledes.org/.

Listing 2-1: LEDES 2000 sample invoice

```
<?xml version="1.0"?>

<!DOCTYPE ledesxml SYSTEM "ledes2000.dtd">

<ledesxml>
```

```
<!-- Law firm originating invoice -->
<firm>
  <lf_tax_id>12-3456789</lf_tax_id>
  <lf_id>100001</lf_id>
 <lf_name>Dewey, Cheatham &amp;amp; Howe</lf_name>
  <lf_address>
    <address_info>
      <address_1>101 Penny Lane</address_1>
       <city>Philadelphia</city>
       <state_province>PA</state_province>
       <zip_postal_code>12345</zip_postal_code>
       <phone>123-456-7890</phone>
       <fax>123-456-7899</fax>
    </address_info>
  </lf_address>
  <lf_remit_address>
  </lf_remit_address>
  <lf_billing_contact_lname>Dewey</lf_billing_contact_lname>
  <lf_billing_contact_fname>Oliver</lf_billing_contact_fname>
  <lf_billing_contact_id>cont005</lf_billing_contact_id>
  <lf_billing_contact_phone>123-456-7891</lf_billing_contact_phone>
  <lf_billing_contact_fax>123-456-7899</lf_billing_contact_fax>
  <lf_billing_contact_email>
    Dewey@HoweDeweyCheatham.com
  </lf_billing_contact_email>
</firm>

<!-- Client receiving invoice -->
<client>
  <cl_id>cl0536</cl_id>
  <cl_name>Acme Insurance</cl_name>
  <cl_address>
    <address_info>
      <address_1>303 North Market Blvd</address_1>
       <city> Philadelphia </city>
       <state_province>PA</state_province>
       <zip_postal_code>12346</zip_postal_code>
       <phone>123-456-8000</phone>
       <fax>123-456-8009</fax>
    </address_info>
  </cl_address>
  <cl_tax_id>45-6789012</cl_tax_id>
```

```
<!-- Invoice Proper -->
<invoice>

  <!-- Invoice header -->
  <inv_id>i200011</inv_id>
  <inv_date>19990915</inv_date>
  <inv_due_date>19991015</inv_due_date>
  <inv_currency>USD</inv_currency>
  <inv_start_date>19990814</inv_start_date>
  <inv_end_date>19990914</inv_end_date>
  <inv_desc>Legal services August - September 1999</inv_desc>
  <inv_payment_terms>10/50</inv_payment_terms>
  <inv_generic_discount>0.10</inv_generic_discount>
  <inv_total_net_due>998.64</inv_total_net_due>

  <!-- Subject of Invoice -->
  <matter>
    <cl_matter_id>clm345690</cl_matter_id>
    <lf_matter_id>lfm439878</lf_matter_id>
   <matter_name>Kiwi Electronics vs. Mary Replogle</matter_name>
    <lf_managing_contact_lname>Cheatham</lf_managing_contact_lname>
    <lf_managing_contact_fname>Igor</lf_managing_contact_fname>
    <lf_contact_id>lfct00</lf_contact_id>
    <lf_contact_phone>415-123-4569</lf_contact_phone>
    <lf_contact_email>
      Cheatham@HoweDeweyCheatham.com
    </lf_contact_email>
    <cl_contact_lname>Norian</cl_contact_lname>
    <cl_contact_fname>Mike</cl_contact_fname>
    <cl_contact_id>clct01</cl_contact_id>
    <cl_contact_phone>916-921-4511</cl_contact_phone>
    <cl_contact_email>mnoy@acmeins.com</cl_contact_email>
    <eft_agreement_number>eft8746186</eft_agreement_number>
    <matter_billing_type>hour</matter_billing_type>
    <matter_final_bill>N</matter_final_bill>
    <matter_total_detail_fees>950.00</matter_total_detail_fees>
    <matter_total_detail_exp>48.64</matter_total_detail_exp>
    <matter_tax_on_fees>0.00</matter_tax_on_fees>
    <matter_tax_on_exp>0.00</matter_tax_on_exp>
    <matter_adj_on_fees>0.00</matter_adj_on_fees>
    <matter_adj_on_exp>0.00</matter_adj_on_exp>
    <matter_perc_shar_fees>0.35</matter_perc_shar_fees>
    <matter_perc_shar_exp>0.35</matter_perc_shar_exp>
    <matter_net_fees>950.00</matter_net_fees>
```

```
<matter_net_exp>48.64</matter_net_exp>
<matter_total_due>998.64</matter_total_due>

<!-- Individual Timekeeper Billing Summary -->
<tksum>
   <tk_id>tk002</tk_id>
   <tk_lname>Cheatham</tk_lname>
   <tk_fname>Igor</tk_fname>
   <tk_level>partner</tk_level>
   <tk_rate>400.00</tk_rate>
   <tk_hours>2.5</tk_hours>
   <tk_cost>1000.00</tk_cost>
</tksum>

<!-- Individual Timekeeper Billing Summary -->
<tksum>
   <tk_id>tk001</tk_id>
   <tk_lname>Dewey</tk_lname>
   <tk_fname>Oliver</tk_fname>
   <tk_level>partner</tk_level>
   <tk_rate>450.00</tk_rate>
   <tk_hours>0.2</tk_hours>
   <tk_cost>90.00</tk_cost>
</tksum>

<!-- Itemised Fees Section ( can contain several items ) -->
<fee>
   <charge_date>19990823</charge_date>
   <tk_id>tk002</tk_id>
 <charge_desc>Review and study file for hearing.</charge_desc>
   <acca_task>L230</acca_task>
   <acca_activity>A101</acca_activity>
   <cl_code_1>clc888</cl_code_1>
   <charge_type>U</charge_type>
   <units>1.0</units>
   <rate>400.00</rate>
   <base_amount>400.00</base_amount>
   <discount_type>Percent</discount_type>
   <discount_amount>0.00</discount_amount>
   <discount_percent>10</discount_percent>
   <total_amount>360.00</total_amount>
</fee>

<!-- Itemised Expense Section(can contain several items ) -->
```

```
        <expense>
          <charge_date>19990910</charge_date>
          <tk_id>tk002</tk_id>
         <charge_desc>special photocopy expense.</charge_desc>
          <acca_expense>e101</acca_expense>
          <charge_type>U</charge_type>
          <units>608</units>
          <rate>0.08</rate>
          <base_amount>48.64</base_amount>
          <total_amount>48.64</total_amount>
        </expense>

      </matter>
    </invoice>
  </client>

</ledesxml>
```

From the comments in Listing 2-1, it is easy to identify the main constituents of a LEDES 2000 invoice, which are as follows:

- Originating law firm data
- Client data
- Invoice header data, including information on:
  - Alternative fee arrangements (time & expense, flat fee, contingency, and staged billing)
  - Fee sharing
  - Discount schedules
  - Taxes
  - Electronic funds transfer reference support
- Matter invoiced
- Summary of timekeeper fees
- Itemized fees and expenses

In addition, you can see that the format can support multiple clients and multiple matters if required. All in all, LEDES 2000 is a flexible eXtensible Markup Language (XML) specification for electronic billing. The only simplification made in Listing 2-1 is to drop the XML element that defines the software vendor and version information, which would probably not be saved as a database item anyway.

Cross-Reference In Chapter 17, which is about using JDBC and XML together, you will find a more detailed discussion of the eXtensible Markup Language.

Designing the Tables

According to a widely quoted remark commonly attributed to C.J. Date, one of the gurus of the relational database world, the primary principles of database design are "nothing more than formalized common sense." Or, as David Adams and Dan Beckett express it in their book *Programming 4th Dimension: The Ultimate Guide*: "The purpose of formal normalization is to ensure that your common sense and intuition are applied consistently to the entire database design." Since Chapter 1 discusses database design from a theoretical viewpoint, this chapter uses a common-sense approach and ties the results back to the rules of normalization.

Cross-Reference See Chapter 1 for a discussion of normalization.

Client and contact data

The obvious first step is to design a Client Table. At first glance, it looks as if you can do this by simply mapping the relevant portion of the XML of Listing 2-1 to a table. However, bear in mind the following considerations:

- The client is frequently a corporation, represented by an individual or individuals involved in a specific matter.
- The client company may often assign different employees to handle different aspects of a given matter.
- Each individual may have a different phone number, mail drop, or cell phone, but all may have the same mailing address.
- A corporation may operate out of a number of different locations.

Since one of the guiding principles of database design is to avoid storing the same item of information in two or more places, these considerations mean that the information about a client has to be divided into a number of different tables. The best place to start is with the lowest level of data, in this case the address. Addresses are stored in a table by themselves, separate from, but linked to, the clients or individuals by a foreign key in the Client or Contact Tables (see Table 2-1).

Table 2-1: Address_Info Table

id	address_1	city	state	zip	country	phone	fax
1004	123 Penny Lane	New York	NY	1006	USA	555 - 123 - 4670	555 - 123 - 4690
1005	711 Quarter St	New York	NY	1007	USA	555 - 119 - 3232	555 - 119 - 3239

The next level of information concerning a client is the contact person. This table uses a foreign key, address_id, to link to an address. It also contains individual phone numbers and e-mail addresses.

Table 2-2 raises the question of handling data items that may also be stored elsewhere. For example, the partners may have their own fax lines, although other employees do not. An obvious way to handle this is to add a fax column to the Contacts Table. However, you should use this column only for personal fax numbers and should set it to NULL for employees who do not have their own fax numbers. You can then write your queries to return the shared fax number if no personal fax number is found. If you fail to do this and you insert the common fax number for each employee, you will be duplicating data.

Table 2-2: Contacts Table

id	fname	lname	company_id	address_id	email	phone	cell
1001	Oliver	Dewey	1001	1004	o.dewey@dsh.com	555-123-4567	444-123-3333
1002	Ichabod	Cheat-ham	1001	1004	i.cheatham@dsh.com	555-123-4568	444-123-3334
1003	Anne	Howe	1001	1004	a.howe@dsh.com	555-123-4569	444-123-3335

Having dealt with the lower levels of client data, you are now ready to create the Client Table itself. This has now become rather simple, since all it needs to contain is the client name, tax id, and address id, as illustrated in Table 2-3.

Table 2-3: Client Table

ID	FIRM_ID	NAME	ADDRESS_ID	TAX_ID
2001	cl0536	Acme Insurance	1001	45-6789012
2002	cl7324	Clark Plumbing	1002	52-6783716

The Firm_ID column is shown her to illustrate the kinds of apparently extraneous information you can expect to find when working with any legacy system. LEDES includes the Firm_ID field as a concession to legacy accounting systems their members are using. Many of these systems contain data fields which may not be pertinent to the needs of the LEDES system, but which are significant to the member firm.

The relationships among these tables is shown in Figure 2-1. The address_id columns in the Client and Contact Tables are foreign keys linking them to the primary key in Address_info.

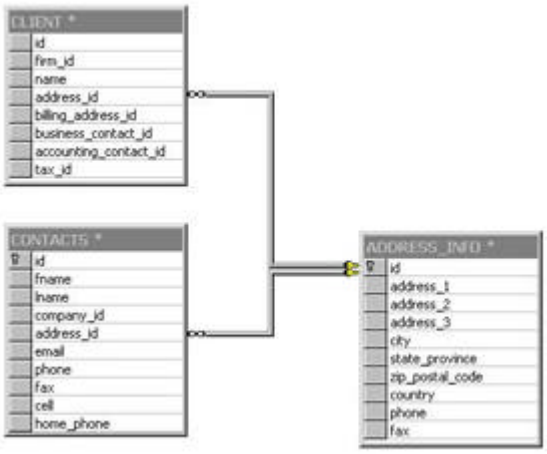


Figure 2-1: Foreign keys link the Client and Contacts Tables to the primary key of the Address_Info table.

The tables conform to the requirements of the first normal form discussed in Chapter 1, for the following reasons:

- All records in any one table have the same number of fields.
- All fields contain only a single data item.
- There are no repeated fields.

Finally, each row in the table is identified using a unique column or set of columns. This unique identifier is the primary key.

The tables conform to the requirements of the second and third normal forms, which are as follows:

- The tables are in first normal form.
- The tables cannot contain fields that are not related to the primary key.
- The tables contain no fields that do not contain information related to the whole of the key.

Since the second normal form applies to tables that have a multipart key, and none of these tables do, conformity is by default. However, multipart keys are not common.

Boyce-Codd normal form is a more rigorous version of the third normal form designed to deal with tables containing:

- Multiple candidate keys
- Composite candidate keys
- Candidate keys that overlap

As it turns out, the Client Table, with its Firm_ID and Tax_ID columns, has multiple candidate keys. Assuming that the legacy Firm_ID column is unique, and knowing that tax id codes should be unique, the Boyce-Codd normal form applies to this table.

In practice, you are unlikely to encounter a problem with BCNF, since the purpose of assigning a unique ID column rather than relying on supposedly unique legacy data is to prevent problems of this sort.

Law firm data

Having created the tables required to manage the clients, you can move on to setting up the tables for the law firm itself. However, after a moment's thought, you will probably realize that the tables you have created will handle all the data for the law firm, too.

Billable items

In a time and materials invoicing system, there are two kinds of billable items: fees and expenses. Fees are charged in a number of different ways, the most common of which is hourly. Expenses are simply charged on a unit basis, as in the case of photo copies, which are billed per page copied. In either case, the id of the law firm employee, or timekeeper, making the charge is provided.

The first table required for billable items, then, is the Timekeeper Table. This table includes a foreign key identifying the individual in the Contacts Table, as well as columns for level and hourly rate. The LEDES specification defines the following levels:

- Partner
- Associate
- Paralegal
- Legal Assistant
- Secretary
- Clerk
- Other

These levels are best stored in a Lookup Table of billing levels, accessed by a foreign key in the Timekeeper Table. Hourly rates, too, should be stored in a Lookup Table, to allow for increases. These two tables contain only an id column and a corresponding level or billing rate, so they are not shown here. The resulting Timekeeper Table might look like Table 2-4.

Table 2-4: Timekeeper Table

id	contact_id	level_code	default_rate_code
1000	2001	1	1
1001	2002	1	2
1002	2007	5	9

Notice how this structure allows for two partners to bill at different rates. It is also intended that the rate code be overridden if the terms of a contract require it.

The billable items are stored in a table that contains the date, a reference to the matter or project, and the id of the timekeeper, as well as information about the specific activity being billed. I have called the table Billable Items, as it is structured such that expense items can be inserted as easily as billable hours.

The Billable_Items Table shown in Table 2-5 contains foreign keys linking it to the Timekeeper Table and the Client_Matter table, as shown in Figure 2-2.

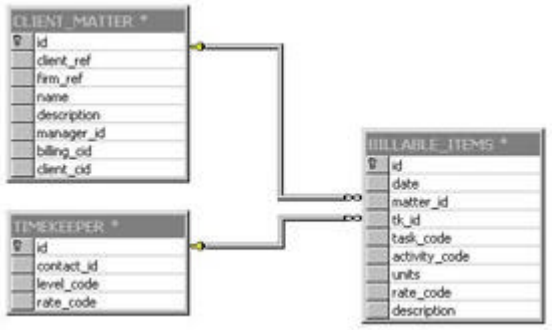


Figure 2-2: The Billable_Items table is linked to the Client_Matter and Timekeeper tables.

Table 2-5: Billable Items Table

id	date	matter_id	tk_id	task_code	activity_code	units	rate_code	description
1	4/12/02	7001	2002	L530	E112	300	0	Court fees
2	4/12/02	7001	2002	L110	A101	2.5	1	Review File

The task and activity columns refer to the industry standard Litigation Code Set developed by the American Bar Association, the American Corporate Counsel Association, and a sponsoring group of major corporate law departments. A copy of the Litigation Code Set can be purchased from the ABA Member Services Department, or viewed on line at:

http://http://www.abanet.org/litigation/litnews/practice/utbms.pdf

In the example of Table 2-5, E112 is the Litigation Code Set code for court fees, while the rate code 0 is used to handle fixed-cost items, as opposed to items billed on a per-unit basis. This permits the merging of unit billings with fixed cost billings without introducing additional columns to handle them separately.

If you add an extra column to handle fixed-cost billings, you introduce a possible ambiguity, because it becomes possible to enter both fixed and unit billings in a single row. This violates the requirements of the fourth normal form because it creates

nonmeaningful combinations of column values. By handling the situation through the rate code, you can use just one table, conforming to the requirements of the fourth normal form.

The tables also meet the requirements of the fifth normal form, which are as follows:

- The table must be in fourth normal form.
- It must be impossible to break down a table into smaller tables unless those tables logically have the same primary key as the original.

By separating address information into a table separate from the Contacts and Clients tables, you can see that if this separation is necessary to conform to the fifth normal form. The addresses do not logically share the same primary key as either contacts or clients.

Matter or Project Tables

Having designed the simpler tables, it is time to move on to handling the Client Matter, or Project, Tables. These tables encapsulate the information specific to the service the law firm is performing for the client. As such, they contain the following:

- Matter Data
  - Name
  - Client reference number
  - Law firm reference number
  - Law firm managing contact
  - Law firm billing contact
  - Client primary contact
- Billing Data
  - Billing type
  - Electronic funds transfer agreement number
  - Tax rate information
  - Fee sharing information
  - Discount agreements information
  - Invoice currency and payment terms
- Invoice Data
  - Date
  - Due date
  - Amount
  - Staffing

The Matter Table and Billing Rates Table are separate; in an ongoing relationship with a client, a law firm may establish a billing agreement that applies to a number of individual matters, so billing data is not strictly specific to a single matter. Conversely, a billing agreement may be renegotiated during the life of a matter.

The Client Matter Table illustrated in Table 2-6 contains the columns billing_cid and client_cid, which are foreign keys pointing to entries in the contacts table, and are labeled with a _cid suffix to denote contact_id in order to avoid confusion with client_id.

Table 2-6: Client Matter Table

id	client_id	client_ref	name	billing_rate	manager_id	billing_cid	client_cid
10001	1201	ref-3711	Jones v Biddle	2	1004	1007	2001
10002	1296	b7997	Jones v Biddle	1	1001	1007	2093

The Billing Rates Table shown in Table 2-7 includes a type code that simply points to a Lookup Table of billing types, including the following:

- Time and Materials
- Flat Fee
- Contingency
- Fee Sharing

Table 2-7: Billing Rates Table

id	type_code	discount_type	discount	tax_rate_fees	tax_rate_exp	terms
1	1	1	15	5	5	1
2	1	1	12.5	5	5	3

Discount types is also a reference to a Lookup Table containing the entries FLAT and PERCENT. Based on the selected discount type, the discount contains either a flat discount amount or a percentage discount rate. The terms column contains another lookup code pointing to a table of payment terms such as 10/30, which means that the billing firm accepts a 10 percent discount if the invoice is paid in full within 30 days.

Generating an Invoice

Generating an invoice involves retrieving a list of all open matters and summarizing the billable items outstanding against each open matter. For the purposes of this example, a Dedicated Billings Table will be created. This table has a one-to-one relationship with the Client Matter Table, as shown in Figure 2-4.

The process involved in creating an invoice is to scan the Billings Table for matters where the status indicates that the matter is still open. (When a client matter has been resolved, and the final invoice paid, the status is set to indicate that the matter is closed.) The links between the tables are shown in Figure 2-3.

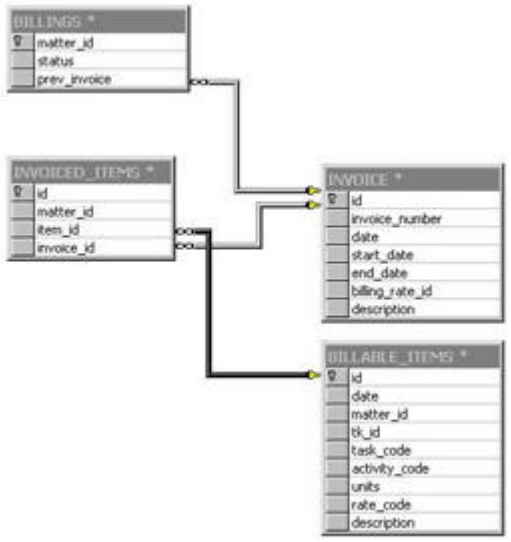


Figure 2-3: Invoices are generated by creating a list of billable items which have not been previously invoiced.

The next step is to compare the Invoiced_Items Table against the Billable_Items Table to find items associated with an open Client_Matter that have not been invoiced. Items that have not been invoiced are added to the Invoiced_Items Table, with their Invoice_ID set to indicate which invoice they were billed on. The Invoiced_Items Table is shown in Table 2-8.

Table 2-8: Invoiced Items Table

id	matter_id	item_id	invoice_id
10001	2006	2031	1007
10007	2119	2047	1063

Another way to handle this is to add an Invoice_Id column to the Billable_Items Table. The Invoice_Id is then updated when the item is invoiced. The advantage of this approach is that you are not adding a new table with a one-to-one relationship with an existing table. The disadvantage is that updating a table can be slow compared to adding a new row.

Table 2-9 shows the Invoice Table. The Invoice Number column provides a legacy system compatible invoice number, and the start date and end date columns identify the billing period covered by the invoice. The Billing Rate Id column is a foreign key

pointing to the Billing Rate Table holding information about payment terms, discounts, and so forth.

Table 2-9: Invoice Table

id	invoice_number	date	start_date	end_date	billing_rate_id	description
1	2001	4/14/02	3/1/02	3/31/02	1021	Services, March 2002
2	2002	4/14/02	3/1/02	3/31/02	1021	Services, March 2002

Invoices are generated by creating a list of billable items which have not been previously invoiced. Billable items which have not been previously invoiced are identified using the links between the tables shown in Figure 2-3.

The relationships between the main tables used to create an invoice are shown in Figure 2-4. Notice the one to one relationship between the Billings and Client_Matter tables mentioned earlier.

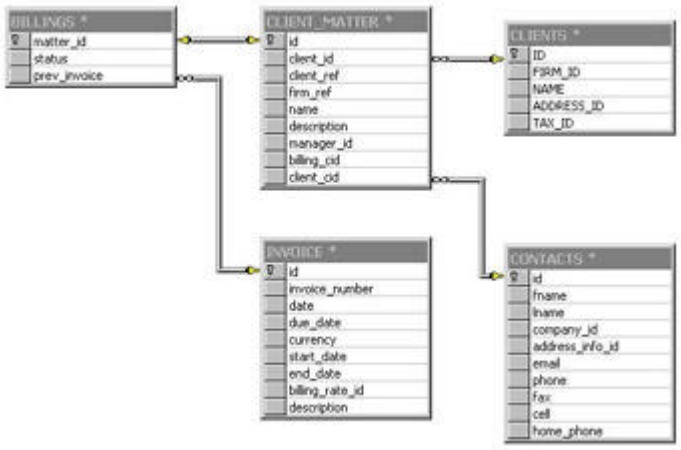


Figure 2-4: These tables are used to create the invoice header.

By combining the data from all these tables, you can generate an invoice containing all the information in Listing 2-1. In addition to itemizing the individual fee and expense items, the LEDES 2000 invoice format requires that fees be summarized by timekeeper. This is done by using the foreign key tk_id in the Billable Items Table.

The final step is to create the invoice header using data from the Clients and Contacts Tables. The procedure to create the invoice header is straightforward, and follows the same basic steps as have been outlined in describing the detail sections of the invoice.

This completes the table definitions required to implement the requirements of the LEDES specification. The next step is to ensure that the referential integrity requirements of the database have been met.

Referential Integrity

In addition to the definitions of the normal forms, the relational model defines certain integrity rules that are a necessary part of any relational database. There are two types of integrity rules: *general* and *database-specific*.

General Integrity Rules

The relational model specifies these two general integrity rules that apply to all databases:

- Entity integrity rule
- Referential integrity rule

The *entity integrity rule* states that primary keys cannot contain NULLs. Obviously, you can't use a NULL to uniquely reference a row, so this is just common sense. It's important to note that, if you use composite keys, this rule requires that none of the individual columns making up the composite key contain NULLs. Most databases enforce this rule automatically when a primary key is declared.

The *referential integrity rule* states that the database must not contain any unmatched foreign-key values. In other words, all references through foreign keys must point to primary keys identifying rows that actually exist.

The referential integrity rule also means that corrective action must be taken to prevent changes or deletions to a row referenced by a foreign key leaving that foreign key with no primary key to reference. This can be handled in the following ways:

- Such changes can be disallowed.
- Changes can be cascaded, so that deleting a row containing a referenced primary key results in deleting all linked rows in dependent tables.
- The dependent foreign-key values are set to NULL.

The specific action you take depends on the circumstances. Many relational database systems support the automatic implementation of one or more of these ways of handling attempted violations of the referential integrity rule. For example, an attempt to insert a row with a foreign key that cannot be found in the appropriate table results in a SQL exception message such as the following:

```
INSERT statement conflicted with COLUMN FOREIGN_KEY constraint
'FK_CONTACTS_ADDRESS_INFO'. The conflict occurred in database 'LEDES',
```

table 'ADDRESS_INFO', column 'id'.

Database-Specific Integrity Rules

Database-specific integrity rules are all other integrity constraints on a specific database. They are handled by the business logic of the application. In the case of the LEDES application discussed in this chapter, they include the following:

- The extensive use of lookup tables to manage such matters as billing and discount schedules
- Validation rules on time captured by employees of the law firm

Many of the integrity constraints can be handled by SQL *Triggers*, but some are be handled by the Java business logic. Triggers are SQL procedures triggered by events such as insertions or changes to the database.

Cross-Reference Triggers are discussed in Chapter 3.

Summary

This chapter has illustrated a common-sense application of the normal forms to the design of a database. The main topics covered are the following:

- Using primary and foreign keys to link tables
- Applying the normalization rules
- Explaining general and database-specific integrity rules

Chapter 3 presents an overview of the SQL language, which you use to work with your relational database.

Chapter 3: SQL Basics

In This Chapter

As discussed in Chapter 1, a clearly defined data-manipulation language is an important part of any Relational Database Management System. Codd defined the requirements of the language to include comprehensive support of data manipulation and definition, view definition, integrity constraints, transactional boundaries, and authorization. He also specified that the language must have the capability to insert, update, retrieve and delete data as a relational set.

The language that has been adopted across virtually the entire database world is the Structured Query Language (SQL). The purpose of this chapter is to provide a comprehensive overview of the Structured Query Language.

The SQL Language

Structured Query Language (SQL) is a development of an IBM product of the 1970s called Structured English Query Language (SEQUEL). Despite its name, SQL is far more than a simple query tool.

As discussed in Chapter 1, in addition to being used to query a database, SQL is used to control the entire functionality of a database system. To support these different functions, SQL can be thought of as a set of the following sublanguages:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Query Language (DQL)
- Data Control Language (DCL)

Unlike Java and most other computer languages, SQL is *declarative* rather than *procedural*. In other words, instead of writing a class to perform some task, in SQL you issue a statement that updates a table or returns a group of records.

The American National Standards Institute (ANSI) has published a series of SQL standards, notably SQL92 and SQL99 (also known as SQL-2 and SQL-3). These standards define several levels of conformance. SQL92 defines entry level, intermediate, and full; SQL99 defines Core SQL99 and Enhanced SQL99.You can get a copy of the ANSI SQL standard from the American National Standards Institute's Web store:

http://webstore.ansi.org/ansidocstore/dept.asp

The pertinent documents are:

- ANSI/ISO/IEC 9075-1-1999 Information Technology - Database Language - SQL Part 1: Framework (SQL/Framework)
- ANSI/ISO/IEC 9075-2-1999 Information Technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation)
- ANSI/ISO/IEC 9075-3-1999 Information Technology - Database Languages - SQL - Part 3: Call-level Interface (SQL/CLI)
- ANSI/ISO/IEC 9075-4-1999 Information Technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM)
- ANSI/ISO/IEC 9075-5-1999 Information Technology - Database Languages - SQL - Part 5: Host Language Bindings (SQL/Bindings)

One of the difficulties you encounter when working with SQL is that each provider uses a slightly different dialect of the language. In the main, these differences amount to enhancements, in that they add to the functionality of SQL. However, they do mean that your SQL statements may not be entirely portable from one implementation to another.

Cross-Reference Chapters 5 through 10 provide detailed examples of the use of SQL in the context of the Java Database Connectivity (JDBC) Core API. Appendix A provides a guide to common SQL commands.

SQL Data Types

SQL supports a variety of different data types that are listed in Table 3-1, together with JDBC data types to which they are mapped. It is important to realize that different SQL dialects support these data types in different ways, so you should read your documentation regarding maximum string lengths, or numeric values, and which data type to use for large-object storage.

Table 3-1: Standard SQL Data Types with Their Java Equivalents

SQL type	Java Type	Description
BINARY	byte[]	Byte array. Used for binary large objects.
BIT	boolean	Boolean 0 / 1 value
CHAR	String	Fixed-length character string. For a CHAR type of length n, the DBMS invariably assignd n characters of storage, padding unused space.
DATETIME	java.sql.Date	Date and Time as: yyyy-mm-dd hh:mm:ss
DECIMAL	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. These can be retrieved using either BigDecimal or String.

Table 3-1: Standard SQL Data Types with Their Java Equivalents

SQL type	Java Type	Description
FLOAT	double	Floating-point number, mapped to double
INTEGER	int	32-bit integer values
LONGVARBINARY	byte[]	Variable-length character string. JDBC allows retrieval of a LONGVARBINARY as a Java input stream.
LONGVARCHAR	String	Variable-length character string. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
NCHAR	String	National Character Unicode fixed-length character string
NUMERIC	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
NTEXT	String	Large string variables. Used for character large objects.
NVARCHAR	String	National Character Unicode variable-length character string
REAL	float	Floating-point number, mapped to float
SMALLINT	short	16-bit integer values
TIME	java.sql.Time	Thin wrapper around java.util.Date
TIMESTAMP	java.sql.Timestamp	Composite of a java.util.Date and a separate nanosecond value
VARBINARY	byte[]	Byte array
VARCHAR	String	Variable-length character string. For a VARCHAR of length n, the DBMS assigns upto n charcters of storage, as required.

Many SQL dialects also support additional data types, such as a MONEY or CURRENCY type. These are handled in Java using the most appropriate getter and setter methods.

Data of any SQL data type can be retrieved using the getObject() method. This is particularly useful if you don't know the data type, and can derive it elsewhere in the application. In addition, data of many types can be retrieved using getString(), and

various other getter methods you might not expect to work, since JDBC will attempt to perform the required data type

Data Definition Language

SQL's Data Definition Language (DDL) is used to create and modify a database. In other words, the DDL is concerned with changing the structure of a database. The SQL2 standard refers to DDL statements as "SQL Schema Statements" and specifies only aspects of the DDL that are independent of the underlying operating system and physical-storage media. In practice, all commercial RDBMS systems contain proprietary extensions to handle these aspects of the implementation.

The main commands in the DDL are CREATE, ALTER, and DROP. These commands, together with the database elements they can work with, are shown in Table 3-2.

Table 3-2: DDL Commands

COMMAND	DATA-BASE	TABLE	VIEW	INDEX	FUNC-TION	PROCE-DURE	TRIGGER
CREATE	YES	YES	YES	YES	YES	YES	YES
ALTER	NO	YES	YES	NO	NO	NO	NO
DROP	YES	YES	YES	YES	YES	YES	YES

Creating, Dropping, and Altering Databases and Tables

The basic SQL command used to create a database is straightforward, as you can see here:

```
CREATE DATABASE CONTACTS;
```

Most RDBMS systems support extended versions of the command, allowing you to specify the files or file groups to be used, as well as a number of other parameters such as log-file names. If you plan to use more than the basic command, refer to the documentation for your specific RDBMS.

The SQL command used to remove a database is as simple as the CREATE DATABASE command. The SQL DROP command is used:

```
DROP DATABASE CONTACTS;
```

Relational databases store data in tables. Most databases may contain a number of different tables, each containing different types of data, depending on the application. Tables are intended to store logically related data items together, so a database may contain one table for business contacts, another for projects, and so on.

A *table* is a set of data records, arranged as rows, each of which contains individual data elements or fields, arranged as columns. All the data in one column must be of the same type, such as integer, decimal, character string, or date.

In many ways, a table is like a spreadsheet. Each row contains a single record. Unlike the rows in a spreadsheet, however, the rows in a database have no implicit order. Table 3-3 illustrates the way tables are designed to contain rows of related, unordered data elements.

Table 3-3: Part of a Database Table

Contact_ID	First_Name	MI	Last_Name	Street	City	State	Zip
1	Alex	M	Baldwin	123 Pine St	Washington	DC	12345
2	Michael	Q	Cordell	1701 York Rd	Columbia	MD	21144

It is immediately obvious that all fields within a given column have a number of features in common:

- They are similar in *type*.
- They form part of a column that has a *name*.
- All fields in a column may be subject to one or more *constraints.*

When a table is created, data types and field lengths are set for each column. These assignments are set using a statement of the following form:

```
CREATE TABLE tableName
   ( columnName dataType[(size)] [constraints] [default value],...);
```

Note The table and column names must start with a letter and can be followed by letters, numbers, or underscores.

Integrity constraints

In addition to selecting data type and length, there are various constraints that may have to be applied to the data stored in a column. These constraints are called integrity constraints because they are used to ensure the consistency and accuracy of the data. They are as follows:

- NULL or NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY

NULL or NOT NULL

Unlike most languages, SQL makes specific provision for empty data fields by allowing you to set them to NULL. A SQL NULL is defined to be a representation of missing or inapplicable data that is systematic and distinct from all regular values and independent of data type. This means you can insert a NULL when the value for a field is unknown or not applicable without any risk that the NULL will be misinterpreted as a zero or a space. The NULL or NOT NULL constraint lets you specify whether a field is required to contain valid data or whether it can be left empty. Keys fields, for example, can never be NULL.

UNIQUE

The UNIQUE constraint is used to specify that all the values in a given column must be unique. It is used primarily when defining columns that are to be used as keys.

PRIMARY KEY

The primary key is used by the database-management systems as a unique identifier for a row. For example, a sales order management system might use the Customer_ID as the primary key in a table of customer names and addresses. This Customer_ID is inserted into the Orders Table as a foreign key, linking customer billing and shipping information to the order.

FOREIGN KEY

The DBMS uses the foreign key to link two tables. For example, when you create a table of customers, you might, for marketing reasons, wish to create a table of their spouses or significant others. The SQL command you use to do this is shown in the second listing under the next section, "Creating a Table."

Creating a table

Listing 3-1 displays the CREATE TABLE statement used to create the table shown in Table 3-3. The statement defines the table name, followed in parentheses by a series of column definitions. Column definitions simply list the column or field name, followed by the data type and the optional constraints. Column definitions are separated by commas, as shown in the example of Listing 3-1.

Listing 3-1: CREATE TABLE Statement

```
CREATE TABLE CONTACT_INFO
(CONTACT_ID        INTEGER          NOT  NULL     PRIMARY  KEY,
 FIRST_NAME        VARCHAR(20)     NOT  NULL,
 MI                CHAR(1)          NULL,
 LAST_NAME         VARCHAR(30)     NOT  NULL,
 STREET            VARCHAR(50)     NOT  NULL,
```

```
CITY            VARCHAR(30)    NOT NULL,
STATE           CHAR(2)        NOT NULL,
ZIP             VARCHAR(10)    NOT NULL);
```

The example of Listing 3-2 illustrates the creation of a foreign key. The column defined as a foreign key, SIGNIFICANT_OTHER, is used to link separate entries in the customers table.

Listing 3-2: Creating a table containing a foreign key

```
CREATE TABLE SIGNIFICANT_OTHERS(CUSTOMER_NUMBER INT NOT
 NULL PRIMARY KEY, SIGNIFICANT_OTHER INT,
 FOREIGN KEY (SIGNIFICANT_OTHER) REFERENCES CUSTOMERS);
```

Cross-Reference The use of Primary Keys and Foreign Keys to link tables was discussed in Chapter 1. Linking tables in JOINS is an important aspect of the use of SQL to retrieve data. Chapter 9 discusses JOINS in more detail.

Altering a table

The ALTER TABLE command is primarily used to add, alter, or drop columns. For example, to add a column for FAX numbers to the Customers Table, you can use the following command:

```
ALTER TABLE CUSTOMERS ADD FAX VARCHAR(20);
```

To change the column width, use this command:

```
ALTER TABLE CUSTOMERS ALTER COLUMN FAX VARCHAR(30);
```

Finally, to drop the column completely, use this command:

```
ALTER TABLE CUSTOMERS DROP COLUMN FAX;
```

Dropping a table

You remove a table from the database completely by using the DROP command. To drop the Customers Table, use the following command:

```
DROP TABLE CUSTOMERS;
```

Creating, Altering, and Dropping a View

A *view* is very similar to a table. Like a table, it has a name that can be used to access it in other queries. In fact, views are sometimes called *temporary tables*.

Creating a view

Rather than being created as a fundamental part of the underlying database, a view is created using a query, as shown here:

```
CREATE VIEW ViewCorleones AS
    SELECT *
    FROM CUSTOMERS
   WHERE Last_Name = 'Corleone'
```

Now you can execute a query just as if this view were a normal table:

```
SELECT *
FROM ViewCorleones
WHERE State = 'NJ'
```

This query would return this result set:

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
Vito	G	Corleone	23 Oak St	Newark	NJ	12345

Since a view is really nothing more than a named result set, you can create a view by joining multiple tables. One way to retrieve data from multiple tables is to use an INNER JOIN. The following code snippet shows how to use an INNER JOIN to create a view called "Orders_by_Name":

```
CREATE VIEW Orders_by_Name AS
SELECT c.LAST_NAME + ', ' + c.FIRST_NAME AS Name,
    COUNT(i.Item_Number) AS Items, SUM(oi.Qty * i.Cost)
     AS Total
FROM ORDERS o INNER JOIN
    ORDERED_ITEMS oi ON
    o.Order_Number = oi.Order_Number INNER JOIN
    INVENTORY i ON
     oi.Item_Number = i.Item_Number INNER JOIN
    CUSTOMERS c ON
     o.Customer_Number = c.CUSTOMER_NUMBER
GROUP BY c.LAST_NAME + ', ' + c.FIRST_NAME
```

In effect, any result set returned that a SELECT statement returns can be used to create a view. That means you can use nested queries, JOINS, or UNIONS as well as simple SELECTS.

Cross-Reference in depth later in this chapter. There are also extensive examples in subsequent chapters, particularly in Chapter 7.

Altering a view

Since a view is created using a SELECT command, views are altered using the ALTER command to issue a new SELECT command. For example, to alter the view you have just created, use the following command:

```
ALTER VIEW ViewCorleones AS
   SELECT FIRST_NAME,LAST_NAME
   FROM CUSTOMERS
   WHERE Last_Name = 'Corleone'
```

You can use a view for updating or deleting rows, as well as for retrieving data. Since the view is not a table in its own right, but merely a way of looking at a table, rows updated or deleted in the view are updated or deleted in the original table.

For example, you can use the view created earlier in this chapter to change Vito Corleone's street address, using this SQL statement:

```
UPDATE ViewCorleones
SET Street = '19 Main'
WHERE First_Name = 'Vito'
```

This example illustrates one of the advantages of using a view. A lot of the filtering required to identify the target row is done in the view, so the SQL code is simpler and more maintainable. In a nontrivial example, this can be a worthwhile improvement.

Note Views are, in a sense, queries that you can save by name, because database management systems generally save views by associating the SELECT statement used to create the view with the name of the view and execute the SELECT when you want to access the view. The downside is that this obviously adds some overhead each time you use a view.

Data Manipulation Language

The Data Manipulation Language (DML) is used to insert data into a table and, when necessary, to modify or delete data. SQL provides the three following statements you can use to manipulate data within a database:

- INSERT
- UPDATE
- DELETE

These statements are discussed in the following sections.

The INSERT Statement

The INSERT statement, in its simplest form, is used to insert data into a table, one row or record at a time. It can also be used in combination with a SELECT statement to perform bulk inserts of multiple selected rows from another table or tables. INSERT can only be used to insert entire rows into a table, not to insert individual fields directly into a row.

The basic form of the INSERT statement looks like this:

```
INSERT INTO tableName (colName1, colName2, ...) VALUES (value1, value2, ...);
```

To insert name and address information into the Customers Table, use an INSERT statement like this:

```
INSERT INTO Customers
(First_Name, MI, Last_Name, Street,City, State, ZIP, Phone)
VALUES
('Michael','X','Corleone','123 Green','New York','NY','12345','111-222-3333');
```

Notice how the field names have been specified in the order in which you plan to insert the data. You can also use a shorthand form, such as the following, if you know the column order of the table:

```
INSERT INTO Customers VALUES
('Michael','X','Corleone','123 Green','New York','NY','12345','111-222-3333');
```

When the Customers Table is defined, the MI field is defined as NULLABLE. The correct way to insert a NULL is like this:

```
INSERT INTO Contact_Info
(FName, MI, LName, Email)
VALUES
('Michael',NULL,'Corleone','offers@cosa_nostra.com');
```

Note String data is specified in quotes ('), as shown in the examples. Numeric values are specified without quotes.

There are some rules you need to follow when inserting data into a table with the INSERT statement:

- Column names you use must match the names defined for the column. Case is not significant.
- Values you insert must match the data type defined for the column they are being inserted into.
- Data size must not exceed the column width.
- Data you insert into a column must comply with the column's data constraints.

These rules are obvious, but breaking them accounts for a lot of SQL exceptions, particularly when you save data in the wrong field order. Another common error is to try and insert the wrong number of data fields.

Using INSERT ... SELECT

Another common use of the INSERT statement is to copy subsets of data from one table to another. In this case, the INSERT statement is combined with a SELECT statement, which queries the source table for the desired records. The advantage of this approach is that the whole process is carried out within the RDBMS, avoiding the overhead of retrieving records and reinserting them externally.

An example of a situation where you might use INSERT...SELECT is the creation of a table containing only the first and last names from the Customers Table. To insert the names from the original Customers Table, use a SQL INSERT...SELECT command to select the desired fields and insert them into the new Names Table. Here's an example:

```
INSERT INTO Names
SELECT First_Name, Last_Name FROM Customers;
```

Essentially, This command tells the database management system to perform two separate operations internally:

1. A SELECT to query the Customers Table for the FName and LName fields from all records
2. An INSERT to input the resulting record set into the new Names Table

By performing these operations within the RDBMS, the use of the INSERT...SELECT command eliminates the overhead of retrieving the records and reinserting them.

Using the WHERE clause with INSERT ... SELECT

The optional WHERE clause allows you to make conditional queries. For example, you can get all records in which the last name is "Corleone" and insert them into the Names Table with the following statement:

```
INSERT INTO Names
SELECT First_Name, Last_Name FROM Customers WHERE Last_Name =
'Corleone';
```

The UPDATE Statement

The UPDATE command is used to modify the contents of individual columns within a set of rows. The UPDATE command is normally used with a WHERE clause, which is used to select the rows to be updated.

A frequent requirement in database applications is the need to update records. For example, when a contact moves, you need to change his or her address. The way to do this is with the SQL UPDATE statement, using a WHERE clause to identify the record you want to change. Here's an example:

```
UPDATE Customers
SET Street = '55 Broadway', ZIP = '10006'
WHERE First_Name = 'Michael' AND Last_Name = 'Corleone';
```

This statement first evaluates the WHERE clause to find all records with matching First_Name and Last_Name. It then makes the address change to all of those records.

Caution If you omit the WHERE clause from the UPDATE statement, all records in the given table are updated.

Using calculated values with UPDATE

You can use the UPDATE statement to update columns with calculated values. For example, if you add stock to your inventory, instead of setting the Qty column to an absolute value, you can simply add the appropriate number of units with a calculated UPDATE statement like the following:

```
UPDATE Inventory
SET Qty = QTY + 24
WHERE Name = 'Corn Flakes';
```

When you use a calculated UPDATE statement like this, you need to make sure that you observe the rules for INSERTS and UPDATES mentioned earlier. In particular, ensure that the data type of the calculated value is the same as the data type of the field you are modifying, as well as being short enough to fit in the field.

Using Triggers to Validate UPDATES

In addition to defining constraints, the SQL language allows you to specify security rules that are applied when specified operations are performed on a table. These rules are known as *triggers*, as they are triggered automatically by the occurrence of a database event such as updating a table.

A typical use of a trigger might be to check the validity of an update to an inventory table. The following code snippet shows a trigger that automatically rolls back or voids an attempt to increase the cost of an item in inventory by more than 15 percent.

```
CREATE TRIGGER FifteenPctRule ON INVENTORY FOR INSERT, UPDATE AS
DECLARE @NewCost money
DECLARE @OldCost money
```

```
SELECT @NewCost = cost FROM Inserted
SELECT @OldCost = cost FROM Deleted
IF @NewCost > (@OldCost * 1.15)
ROLLBACK Transaction;
```

The SQL ROLLBACK command used in this code snippet is one of the Transaction Management commands. Transaction management and the SQL ROLLBACK command are discussed in the next section.

Using transaction management commands with UPDATE

Transaction management refers to the capability of a relational database management system to execute database commands in groups, known as *transactions*. A transaction is a group or sequence of commands, all of which must be executed in order and all of which must complete successfully. If anything goes wrong during the transaction, the database management system allows the entire transaction to be cancelled or "rolled back." If, on the other hand, it completes successfully, the transaction can be saved to the database or "committed."

In the SQL code snippet below, there are two update commands. The first attempts to set the cost of Corn Flakes to $3.05, and the cost of Shredded Wheat to $2.15. Prior to attempting the update, the cost of Corn Flakes is $2.05, so the update clearly violates the FifteenPctRule trigger defined above. Since both updates are contained within a single transaction, the ROLLBACK command in the FifteenPctRule trigger will execute, and neither update will take effect.


```
BEGIN transaction;
UPDATE Inventory
 SET  Cost = 3.05
 WHERE Name = 'Corn Flakes';
UPDATE Inventory
 SET Cost = 2.15
 WHERE Name = 'Shredded Wheat';
COMMIT transaction;
```

Although all SQL commands are executed in the context of a transaction, the transaction itself is usually transparent to the user unless the AUTOCOMMIT option is turned off. Most databases support the AUTOCOMMIT option, which tells the RDBMS to commit all commands individually as they are executed. This option can be used with the SET command:

```
SET AUTOCOMMIT [ON | OFF] ;
```

By default, the SET AUTOCOMMIT ON command is executed at startup, telling the RDBMS to commit all statements automatically as they are executed. When you start

to work with a transaction, turn Autocommit off; then issue the commands required by the transaction. Assuming that everything executes correctly, the transaction will be committed when the COMMIT command executes, as illustrated above. If any problems arise during the transaction, the entire transaction is cancelled by the ROLLBACK command.

Cross-Reference Transaction management and the ACID test are discussed in Chapter 1. The examples in Chapter 6 illustrate the use of the COMMIT and ROLLBACK commands.

Using UPDATE on Indexed Tables

When a table is indexed for rapid data retrieval, and particularly when a clustered index is used for this purpose, updates can be very slow unless you understand and use the indexes correctly. The reason for this is that the purpose of an index is to provide rapid and efficient access to a table. In most situations, speed of data retrieval is considered to be of paramount performance, so tables are indexed to enhance the efficiency of data retrieval.

A limiting factor in retrieving data rapidly and efficiently is the performance of the physical storage medium. Performance can be optimized for a specific index by tying the layout of the rows on the physical storage medium to that index. The index for which the row layout is optimized is commonly known as the clustered index.

If you fail to take advantage of indexes, and in particular, of the clustered index, when planning your update strategy, your updates may be very slow. Conversely, if your updates are slow, you would be well advised to add an index specifically to handle updates, or to modify your update strategy in light of the existing indexes.

The DELETE Statement

The last DML command is the DELETE command, which is used for deleting entire records or groups of records. Again, when using the DELETE command, you use a WHERE clause to identify the records to be deleted.

Using the DELETE command is very straightforward. For example, this is the command you use to delete records containing the First_Name: "Michael" and the Last_Name: "Corleone":

```
DELETE FROM Customers
WHERE First_Name = 'Michael' AND Last_Name = 'Corleone';
```

Without the WHERE clause, all rows throughout the entire table will be deleted. If you are using a complicated WHERE clause, it is a good idea to test it in a SELECT statement before using it in a DELETE command.

Caution INSERT, DELETE and UPDATE, can cause problems with other tables, as well as significant problems within the table you are working on. Delete with care.

Data Query Language

Probably the most important function of any database application is the ability to search for specific records or groups of records and return them in the desired form. In SQL, this capability is provided by the Data Query Language (DQL). The process of finding and returning formatted records is known as *querying the database.*

The SELECT Statement

The SELECT statement is the basis of data retrieval commands, or queries, to the database. In addition to its use in returning data in a query, the SELECT statement can be used in combination with other SQL commands to select data for a variety of other operations, such as modifying specific records using the UPDATE command.

The basic form of a simple query specifies the names of the columns to be returned and the name of the table or tables in which they can be found. A basic SELECT command looks like this:

```
SELECT columnName1, columnName2,.. FROM tableName;
```

Using this query format, you can retrieve the first name and last name of each entry in the Customers Table by using the following SQL command:

```
SELECT First_Name, Last_Name FROM Customers;
```

In addition to this form of the command, where the names of all the fields you want returned are specified in the query, SQL supports this wild card form:

```
SELECT * FROM tableName;
```

The wild card, "*", tells the database management system to return the values for all columns.

The WHERE Clause

Under normal circumstances, you probably do not want to return every row from a table. A practical query needs to be more restrictive, returning the requested fields from only records that match some specific criteria.

To make specific queries, use the WHERE clause. The WHERE clause was introduced earlier in this chapter under the section "Data Manipulation Language."

This clause lets you retrieve, for example, the records of all customers living in New York from the Customers Table shown in Table 3-4.

Table 3-4: The CUSTOMERS Table

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Michael	A	Corleone	123 Pine	New York	NY	10006
Fredo	X	Corleone	17 Main	New York	NY	10007
Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
Francis	X	Corleone	17 Main	New York	NY	10005
Vito	G	Corleone	23 Oak St	Newark	NJ	12345
Tom	B	Hagen	37 Chestnut	Newark	NJ	12345
Kay	K	Adams	109 Maple	Newark	NJ	12345
Francis	F	Coppola	123 Sunset	Hollywood	CA	23456
Mario	S	Puzo	124 Vine	Hollywood	CA	23456

The SQL query you use to retrieve the records of all customers living in New York is as follows:

```
SELECT * FROM Customers WHERE City = 'New York';
```

The result of this query returns all columns from any row with the CITY column containing "New York." The order in which the columns are returned is the order in which they are stored in the database; the row order is arbitrary.

To retrieve columns in a specific order, the column names must be specified in the desired order in your query. For example, to get the data in First_Name, Last_Name order, issue this query:

```
SELECT First_Name, Last_Name FROM Customers WHERE Last_Name = 'Corleone';
```

To get the order reversed, use this query:

```
SELECT Last_Name, First_Name FROM Customers WHERE Last_Name = 'Corleone';
```

Note Unlike rows in a spreadsheet, records in a database table have no implicit order. Any ordering you need has to be specified explicitly, using the SQL ORDER BY command.

SQL Operators

The queries discussed so far have been very simple, but in practice you will frequently be using queries that depend on the values of a number of fields in various combinations. SQL provides a number of operators to enable you to create complex queries based on value comparisons.

Operators are used in expressions to define how to combine the conditions specified in a WHERE clause to retrieve data or to modify data returned from a query. SQL has several types of operators:

For convenience, SQL operators can be separated into these five main categories:

- Comparison operators
- Logical operators
- Arithmetic operators
- Set operators
- Special-purpose operators

Comparison operators

One of the most important uses for operators in SQL is to define the tests used in WHERE clauses. SQL supports the following standard-comparison operators, as well as a special IS NULL operator, and its complement, IS NOT NULL, used to test for a NULL value in a column:

- Equality (=)
- Inequality (<>)
- Greater Than (>) and Greater Than or Equal To (>=)
- Less Than (<) and Less Than or Equal To (<=)
- IS NULL
- IS NOT NULL

Numeric and character comparisons

All the comparison operators in SQL work equally well on both numeric and character variables. This means that you can compare character variables using an equality test in exactly the same way as you test a numeric value. The query:

```
SELECT * FROM Customers WHERE Last_Name = 'Corleone';
```

is every bit as valid as the query:

```
SELECT * FROM Inventory WHERE Part_Number = 1903;
```

If you use the greater-than or less-than operators for comparisons of CHAR or VARCHAR values, the comparison is performed lexically. For example, to find

customers named "Michael," or whose names come after "Michael" in the alphabet, you can use this query:

```
SELECT *
FROM CUSTOMERS
WHERE first_name >= 'Michael';
```

This query returns a result set like the one shown in Table 3-5.

Table 3-5: Results of a Lexical String Comparison

ID	FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
100	Michael	A	Corleone	123 Pine	New York	NY	10006
102	Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
104	Vito	G	Corleone	23 Oak St	Newark	NJ	12345
105	Tom	B	Hagen	37 Chestnut	Newark	NJ	12345

Using the IS NULL operator

SQL's special NULL value represents an absence of data, so it can't be evaluated using the comparison operators. SQL provides special IS NULL and IS NOT NULL operators to test for NULL. If, for example, you add a column to the Customers Table for FAX numbers, leaving it NULL when a contact doesn't have a fax number, you can query the table for contacts with faxes as follows:

```
SELECT * FROM Customers WHERE FAX IS NOT NULL;
```

Using the LIKE and NOT LIKE operators

In addition to the comparison operators, SQL adds these dedicated operators for testing for a substring within CHAR and VARCHAR variables:

- LIKE
- NOT LIKE

The LIKE operator, and its negation, the NOT LIKE operator, combined with wildcards provide a very powerful tool for String comparison. The wildcards are as follows:

- Underscore (_), the single character wild card
- Percent (%), the multicharacter wild card

For example, to find all records in the Customers Table with last names starting with "C," write a query using LIKE as follows:

```
SELECT * FROM Customers WHERE Last_Name LIKE 'C%';
```

Similarly, to find all records where the last name contains the letter "o" in the second place, the query looks like this:

```
SELECT * FROM Customers WHERE Last_Name LIKE '_o%';
```

NOT LIKE works in much the same way as LIKE. For example, to find all records in the Customers Table with the last name NOT starting with the letter "C," write a query using NOT LIKE such as the following:

```
SELECT * FROM Customers WHERE Last_Name NOT LIKE 'C%';
```

Using the concatenation operator

The concatenation operator (+ or ||) is used to append one string to another string. For example, to return the last name, followed by the first name and separated by commas, use this query:

```
SELECT Last_Name + ', ' + First_Name AS NAME FROM Customers;
```

Caution The concatenation operator is one of the SQL features that vary from one flavor of SQL to another. SQL Server, MS Access, and Sybase, for example, accept '+', whereas Oracle accepts '||'.

Logical operators

It is frequently necessary to combine two or more comparisons in a WHERE clause. SQL provides these standard logical operators for this purpose:

- AND
- OR
- NOT

Using the AND operator

The AND operator is used to combine two or more comparisons, all of which must evaluate to TRUE for the comparison to be valid. If either expression is false, AND returns FALSE. For example, to find all records in the Customers Table with a last name of Corleone who live in New York, use this query:

```
SELECT * FROM Customers WHERE Last_Name = 'Corleone' AND City = 'New York';
```

Using the OR operator

The OR operator is used to combine two or more comparisons, any one of which can evaluate to TRUE for the comparison to be valid. For example, to find all records in the Customers Table who live in New York City or in New Jersey, use this query:

```
SELECT * FROM Customers WHERE City = 'New York' OR State = 'NJ';
```

Using the NOT operator

The NOT operator is used to invert the result of a comparison. For example, the previous example can be modified as follows to find all to find all customers with a last name of Corleone who do not live in New York City or in New Jersey:

```
SELECT * FROM Customers
WHERE Last_Name = 'Corleone' AND NOT ( City = 'New York' OR State = 'NJ' );
```

Combining logical operators using parentheses

Logical operators can be combined using parentheses (()). For example, the queries shown in the preceding two code snippets that use the AND and OR operators can be combined to form a query that returns all records in the Customers Table with a last name of Corleone who live in New York City or New Jersey:

```
SELECT * FROM Customers
WHERE Last_Name = 'Corleone' AND ( City = 'New York' OR State = 'NJ' );
```

Arithmetic operators

SQL supports the common arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/). In addition, SQL supports the modulo operator (%), which returns the remainder of the division of one integer by another.

Using arithmetic operators in the WHERE clause

The first and most obvious use of arithmetic operators is in the WHERE clause. The following example uses the LESS THAN operator to identify items in the inventory shown in Table 3-6 with a Qty below 24:

```
SELECT *
FROM INVENTORY
WHERE Qty < 24;
```

Table 3-6: Inventory

ID	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	130	1.95
1002	Rice Krispies	Cereal	97	1.87

Table 3-6: Inventory

ID	Name	Description	Qty	Cost
1003	Shredded Wheat	Cereal	103	2.05
1004	Oatmeal	Cereal	15	0.98
1005	Chocolate Chip	Cookies	217	1.26
1006	Fig Bar	Cookies	162	1.57
1007	Sugar Cookies	Cookies	276	1.03
1008	Cola	Soda	144	0.61
1009	Lemon Soda	Soda	96	0.57
1010	Orange Soda	Soda	84	0.71

This query returns the following result:

ID	Name	Description	Qty	Cost
1004	Oatmeal	Cereal	15	0.98

You can also perform a calculation in a WHERE clause. For example, if you normally mark up the cost of an item by 60% to get the sales price, you can perform this calculation in the WHERE clause. To list only items whose retail price is below 100.00, use this format:

```
SELECT Name,Description,Cost,Cost*1.6 AS Retail
FROM Inventory
WHERE Cost * 1.6 < 100;
```

Creating calculated result columns

Arithmetic operators are also very useful for creating a calculated result field. For example, you can calculate a retail price by marking up a cost as follows:

```
SELECT ID,Name,Description,Cost,Cost*1.6 AS Retail
FROM Inventory;
```

This query returns the additional column "Retail," as shown in Table 3-7.

Table 3-7: Calculated Result Fields

ID	Name	Description	Cost	Retail
1001	Corn Flakes	Cereal	1.95	3.12

Table 3-7: Calculated Result Fields

ID	Name	Description	Cost	Retail
1002	Rice Krispies	Cereal	1.87	2.99
1003	Shredded Wheat	Cereal	2.05	3.28
1004	Oatmeal	Cereal	0.98	1.57
1005	Chocolate Chip	Cookies	1.26	2.02
1006	Fig Bar	Cookies	1.57	2.51
1007	Sugar Cookies	Cookies	1.03	1.65
1008	Cola	Soda	0.61	0.98
1009	Lemon	Soda	0.57	0.91
1010	Orange	Soda	0.71	1.14

Caution When you create a calculated field in a result, you should always use AS to assign a name to the field, because there is no defined naming convention for calculated fields in SQL. Different variants of SQL assign different arbitrary names.

Using Aliases

In the preceding example, the key word AS was used in the expression command. Using the optional AS clause lets you assign a meaningful name to an expression, which makes referring back to the expression easier. An alias can be used as a normal column name when you need to refer to the column elsewhere in a statement, as you will see in further examples in future chapters. In this example, AS assigns the name, or alias, "Retail" to the calculated value column.

When assigning and using an alias, you must bear in mind the order in which SQL processes the various clauses constituting the command, since the output of one clause is the input to the next one. The order in which the subclauses of a SQL command are processed is shown in the following list:

- FROM clause
- WHERE clause
- GROUP BY clause
- HAVING clause
- SELECT clause
- ORDER BY clause

Since you use AS to assign an alias in the SELECT clause, the alias can't be used as part of the WHERE clause, since that has already been executed by the time

you get to the SELECT. It can, however, be used in an ORDER BY. For example, you can order the inventory table by retail as follows:

```
SELECT ID,Name,Description,Cost,Cost*1.6 AS Retail
FROM Inventory ORDER BY Retail;
```

Set operators

Set operators allow you to combine ResultSets returned by different queries into a single ResultSet. The main set operators are as follows:

- UNION and UNION ALL return the combined results of two queries.
- INTERSECT returns only the rows that both queries find.
- EXCEPT returns the rows from the first query that are not present in the second.

Using UNION and UNION ALL

UNION ALL returns the results of two queries; while UNION does the same thing, but it removes duplicate results. For example, you can use a UNION to combine the results of a query for all customers with the last name "Adams" with a query for all customers in New York with the last name "Corleone." Here's an example:

```
SELECT *
FROM Customers
WHERE Last_Name = 'Corleone' AND City = 'New York'
UNION
SELECT *
FROM Customers
WHERE Last_Name = 'Adams';
```

UNION, used by itself, returns the results of the two queries without any repetitions. UNION ALL, on the other hand, returns the results of the two queries including all repetitions.

Using INTERSECT and EXCEPT

The INTERSECT and EXCEPT operators adhere to the same syntax as the UNION operator. You should check with the documentation for the DBMS you are using to ensure that these operators are supported before using one of them.

Special-purpose operators

SQL also provides a number of operators to perform functions which, in most other languages, require special procedural code. Since SQL is not a procedural language, these are particularly useful features of the language.

The IN operator

The IN operator is a powerful way of comparing fields against a list. For example, to find contacts in New York State or New Jersey, you can use this query:

```
SELECT *
FROM Customers
WHERE State IN ('NY', 'NJ');
```

IN also works with numbers. If you want to select items from the Inventory Table by ID, use this query:

```
SELECT *
FROM Inventory
WHERE ID IN (1001, 1003, 1004);
```

The BETWEEN operator

The BETWEEN operator is used to select fields with values between specified limits. Referring again to the Inventory Table, you can query for items with costs in the $1.03 to $1.95 range using this query:

```
SELECT *
FROM Inventory
WHERE Cost BETWEEN 1.03 AND 1.95;
```

Note BETWEEN returns values *within* its defined range inclusive of the limits, so if you try the query against the Inventory Table, it will return rows with costs of 1.03 and 1.95.

The DISTINCT operator

A basic SELECT statement tells the database management system to return all records matching the query in the ResultSet. For example, you cab request all Last_Names from customers using this query:

```
SELECT Last_Name
FROM Customers;
```

Using the data shown in Table 3-4, this gives you five repetitions of "Corleone."

The DISTINCT operator tells the database management system not to return duplicate records in a ResultSet. For example, to return all Last_Names from the Customers Table with no duplicates, use this query:

```
SELECT DISTINCT Last_Name
FROM Customers;
```

When this operator is applied to the results, you see only the last name "Corleone" once, despite the fact that there are several different Corleones in the table.

Note There is also a keyword ALL, as in SELECT ALL, but since ALL is implied unless DISTINCT is used, the expression SELECT ALL is rarely, if ever, used.

The TOP operator

The TOP operator specifies that only the first *n* rows are to be output from the query result set, or, optionally, the top *n* percent of the rows. When specified with PERCENT, *n* must be an integer between 0 and 100:

```
SELECT TOP 25 PERCENT *
FROM Inventory;
```

The result set from this query is shown in Table 3-8.

Table 3-8: Top n Records

ID	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	130	1.95
1002	Rice Krispies	Cereal	97	1.87
1003	Shredded Wheat	Cereal	103	2.05

If the query includes an ORDER BY clause, the first *n* rows (or *n* percent of rows) ordered by the ORDER BY clause are output. If the query has no ORDER BY clause, the order of the rows is arbitrary.

Escape Sequences

Escape sequences are used to handle situations where a character has a particular meaning to SQL, and you want to use it in a different way. A typical example is the use of the apostrophe ('). A problem that arises frequently in normal free-form text is the use of the apostrophe. Since the apostrophe is, in effect, a single quote, SQL reads it as a CHAR or VARCHAR terminator and throws a SQL error when it tries to handle the rest of the String.

The solution is simple: simply double up the apostrophe. These two other characters sometimes need to be escaped:

- % percent
- _ underscore

They are handled by defining an escape character at the end of the query in which the characters are used. The escape character is defined in curly braces ({}) using the keyword escape:

{escape 'escape-character'}

For example, the following query finds names that begin with an underscore. It uses the backslash (\) character as an escape character:

```
SELECT name
FROM variables
WHERE Id LIKE `\_%' {escape \'};
```

Using Subqueries

The use of queries is not limited to situations where you want to return a result set to the user. It is frequently useful to create a result set so that you can use it within another SQL statement. A *subquery* is used as part of another SQL statement.

Subqueries can be nested inside any of the following types of SQL statements:

- SELECT or SELECT...INTO
- INSERT...INTO
- DELETE
- UPDATE
- Inside another query or subquery

Subqueries are used to provide an intermediate result set to be operated on by another part of the SQL statement. For instance, you can use a subquery to return cost data about all the cookies in your inventory and then use this cost data with a comparison operator in another query. In this case, you use a SELECT statement to provide a set of values to be evaluated in the WHERE or HAVING clause of the main statement.

Subqueries can be used in WHERE or HAVING clauses as the right-hand side of the following comparison and expressions:

- Comparisons using ANY, ALL or SOME
- Expressions using IN or NOT IN
- Expressions using EXISTS or NOT EXISTS

Using the ANY, SOME, and ALL operators

In many cases, a subquery used in a comparison returns more than one value, so special predicates are required to operate on the results of the subquery before making the comparison.

For example, if you want to find out which inventory items cost more than cookies, you can use a subquery like this:

```
(SELECT cost FROM inventory
  WHERE Description = 'Cookies');
```

The result of this subquery is several rows of cookie costs, so you need to select which cost you want to use. The ANY or SOME predicates, which are synonymous, can be used to retrieve records in the main query that satisfy the comparison with any records retrieved in this subquery:

```
SELECT * FROM INVENTORY
WHERE cost >= ANY
        (SELECT cost FROM inventory
         WHERE Description = 'Cookies');
```

This query returns all inventory items with a cost greater than or equal to the lowest-cost cookies in the Inventory Table, as shown here:

Item_Number	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	178	1.95
1002	Rice Krispies	Cereal	97	2
1003	Shredded Wheat	Cereal	103	2.05
1005	Chocolate Chip	Cookies	217	1.26
1006	Fig Bar	Cookies	162	1.57
1007	Sugar Cookies	Cookies	276	1.03

The ALL predicate can be used to retrieve only records in the main query that satisfy the comparison with all records retrieved in the subquery. If you change ANY to ALL in the preceding example, the query returns only those inventory items that cost more than all cookies:

Item_Number	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	178	1.95
1002	Rice Krispies	Cereal	97	2

Item_Number	Name	Description	Qty	Cost
1003	Shredded Wheat	Cereal	103	2.05
1006	Fig Bar	Cookies	162	1.57

Using the IN and NOT IN operators

As you recall from the section on operators earlier in this chapter, the IN predicate is used to compare values against a list. For example, to return all customers in either NY or NJ, you can use the IN predicate to check the customer's state against a list containing 'NY' and 'NJ'. Here's an example:

```
SELECT * FROM CUSTOMERS
WHERE STATE IN ('NY','NJ');
```

Also, you can use the IN predicate with a subquery to populate the list. The following code snippet uses a subquery to create a list of item numbers from the Orderd_Items Table and uses the IN predicate to return the corresponding inventory data:

```
SELECT *
FROM INVENTORY
WHERE Item_Number IN
          (SELECT  Item_Number
        FROM Ordered_Items
       WHERE  Order_Number = 2);
```

The result set this query returns looks like this:

Item_Number	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	178	1.95
1004	Oatmeal	Cereal	15	0.98
1005	Chocolate Chip	Cookies	217	1.26
1010	Orange	Soda	84	0.71

In addition, you can use the IN predicate with the NOT operator to select all inventory items that are not included in the select list. Note that you can only specify one SELECT list item when using the IN predicate, since the list is returned for comparison with a single item.

Using the EXISTS and NOT EXISTS predicates

EXISTS and NOT EXISTS are predicates. That is, they return true or false. They are used in true/false comparisons to determine whether the subquery returns any records. For example, you can use a subquery to return a result set of Ordered Items matched up by Order Number and Customer Number to the customer who has ordered them. Then you can find out what kinds of cookies he or she has ordered using the comparison Description = 'Cookies' with the EXISTS predicate, as shown here:

```
SELECT DISTINCT Name
FROM Inventory
WHERE Description = 'Cookies' AND EXISTS
        (SELECT *
           FROM Customers c, Ordered_Items oi, Orders o, Inventory i
            WHERE c.Customer_Number = o.Customer_Number AND
                  oi.Order_Number = o.Order_Number AND
                   oi.Item_Number = i.Item_Number);
```

This query returns this result set:

Name
Chocolate Chip
Fig Bar
Sugar Cookies

Notice the use of the asterisk (*) as the SELECT list for the subquery. Conventionally, you use an asterisk with the EXISTS predicate because EXISTS only returns true or false, so there is nothing to be gained by being more specific.

Note The EXISTS predicate stops the search as soon as it finds a single match and is therefore much faster and more efficient than a query that continues to check for additional rows that match.

Correlated subqueries

As a rule, the main FROM list should only contain tables that are referenced in the main SELECT. In this case, the main SELECT clause includes only inventory. You can also use table name aliases in a subquery to refer to tables listed in a FROM clause outside the subquery, as in the following example. This usage is known as a *correlated subquery:*

```
SELECT c.First_Name, c.Last_Name, o.Order_Number, i.Item_Number, i.Name
FROM Customers c, Inventory i, Orders o
WHERE i.Description = 'Cookies' AND EXISTS
```

```
(SELECT *
        FROM Ordered_Items oi
       WHERE c.Customer_Number = o.Customer_Number AND
             oi.Order_Number   = o.Order_Number AND
              oi.Item_Number     = i.Item_Number);
```

In this example, most of the tables the subquery accesses are defined in the main query. This query returns the following result set:

First_Name	Last_Name	Order_Number	Item_Number	Name
Fredo	Corleone	2	1005	Chocolate Chip
Francis	Corleone	3	1006	Fig Bar
Kay	Adams	5	1006	Fig Bar
Kay	Adams	5	1007	Sugar Cookies

Correlated subqueries depend on a value in the outer query. A reference to a table in the outer query is called a *correlated reference*. Correlated queries are executed repeatedly, once for each row of the table identified in the outer-level query, so they can be extremely inefficient. It is frequently worth rewriting correlated queries as joins where possible, though in some cases the SQL engine may be able to optimize the correlated subquery.

Nesting subqueries

Just as you can use a subquery within a query, you can also use a subquery within another subquery. Subqueries can be nested as deeply as your implementation of SQL allows. The syntax for nesting subqueries looks like this:

```
SELECT *
FROM Tables
WHERE
   ( SUBQUERY
      (SUBQUERY
         (SUBQUERY)));
```

Additional uses of subqueries

Just as you can use calculated values, or even literals, in place of simple data-field values in the SELECT clause of a query, you can also use the results subqueries return. This can be useful if you want to create a summary comparing the cost of an item against another value retrieved from the table, such as an average cost of all similar items. Here's an example:

```
SELECT Name, Cost,
  (SELECT AVG(Cost) FROM Inventory WHERE Description = 'soda') AS Average
FROM Inventory WHERE Description = 'soda';
```

Notice how the entire subquery replaces the column name, so that the AS clause used to name the column appears outside the parentheses defining the subquery. The results of this query look like this:

Name	Cost	Average
Cola	0.61	0.63
Lemon	0.57	0.63
Orange	0.71	0.63

Using a subquery with the INSERT command

You can use subqueries in the INSERT command just as easily as you can in a SELECT command. Consider an example where you might want to insert selected records from one table into another. One way to do this is to use a subquery to select the desired subset from the source table.

In the following example, a subquery is used to select the Customer_Numbers of customers from New Jersey. Then the appropriate fields are selected from customers with the selected Customer_Numbers, and inserted into the Employees Table:

```
INSERT INTO Employees (Employee_ID, First_Name, Last_Name)
   SELECT Customer_Number, First_Name, Last_Name
      FROM Customers
      WHERE Customer_Number IN
         (SELECT Customer_Number
            FROM Customers
            WHERE State = 'NJ');
```

Using a subquery with the UPDATE command

A more common usage of a subquery is with the UPDATE command. This example uses a subquery to select the Customer_Number of the customer to be updated from the Customers Table. You then use this customer number in the WHERE clause of the UPDATE command as shown here:

```
UPDATE Employees
   SET First_Name = 'Alfie'
      WHERE Employee_ID IN
         (SELECT Customer_Number
```

```
    FROM Customers
WHERE First_Name = 'Sonny');
```

One advantage of using a subquery is that you can easily test the subquery by itself to make sure you are getting the correct data set. Then, once it checks out OK, you can plug the subquery into the actual update command.

Using a subquery with the DELETE command

Finally, here's an example of the use of a subquery with DELETE. This example uses a subquery to select the Employee_IDs of all employees so that they can be deleted from the Customers Table:

```
DELETE FROM Customers
WHERE Customer_Number IN
        (SELECT Employee_ID FROM Employees);
```

Sorting the Results of a Query

A common requirement when retrieving data from a database is to sort the results of the query in alphabetic or numeric order on one or more of the columns. Results are sorted using the ORDER BY clause in a statement like this:

```
SELECT First_Name, Last_Name, City, State
FROM CUSTOMERS
WHERE Last_Name = 'Corleone'
ORDER BY First_Name;
```

This gives you a list of all the Corleones sorted in ascending order by first name, as shown in Table 3-9.

Table 3-9: Records Sorted Using ORDER BY

First_Name	Last_Name	City	State
Francis	Corleone	New York	NY
Fredo	Corleone	New York	NY
Michael	Corleone	New York	NY
Sonny	Corleone	Newark	NJ
Vito	Corleone	Newark	NJ

The default sort order is ascending. This can be changed to descending by adding the DESC keyword as shown here:

```
SELECT *
FROM CUSTOMERS
WHERE Last_Name = 'Corleone'
ORDER BY First_Name DESC;
```

Sorting on multiple columns is also easy to do by using a sort list. For example, to sort the data in ascending order based on Last_Name and then sort duplicates using the First_Name in descending order, the SQL statement is as follows:

```
SELECT First_Name, MI, Last_Name, Street, City, State, Zip
FROM CUSTOMERS
ORDER BY Last_Name, First_Name DESC;
```

Note When no ORDER BY clause is used, the order of the output of a query is undefined.

The rules for using ORDER BY are as follows:

- The ORDER BY clause must be the last clause in the SELECT statement.
- Default sort order is ascending.
- You can specify ascending order with the keyword ASC.
- You can specify descending order with the keyword DESC.
- You can use column names or expressions in the ORDER BY clause.
- The column names in the ORDER BY clause do not have to be specified in the select list.
- NULLS usually occur first in the sort order.

Summarizing the Results of a Query

Another common reporting requirement is to break down the data a query returns into various groups so that it can be summarized in some way. The GROUP BY clause enables you to combine database records to perform calculations such as averages or counts on groups of records.

The GROUP BY clause combines records with identical values in a specified field into a single record for this purpose, as shown in the following example:

```
SELECT Description, COUNT(Description) AS 'Count', AVG(Cost) AS 'Average
Cost'
FROM Inventory
GROUP BY Description;
```

The results of this query will be as follows:

Description	Count	Average Cost
Cereal	4	1.745

Description	Count	Average Cost
Cookies	3	1.2866
Soda	3	0.63

Notice that the name given to the "Count" column was quoted, since COUNT is a SQL keyword.

Because the GROUP BY clause combines all records with identical values in one column into a single record, each of the column names in the SELECT clause must be either a column specified in the GROUP BY clause or a column function such as COUNT() or AVG(). This means that you can't SELECT a list of individual customers by name and then count them as a group using GROUP BY. However, you can group on more than one column, just as you can use more than one column with the ORDER BY clause.

Every column name specified in the SELECT statement must also be mentioned in the GROUP BY clause. Not mentioning the column names in both places gives you an error. The GROUP BY clause returns a row for each unique combination column in the GROUP BY clause.

Aggregate Functions

Aggregate functions return a single value from an operation on a column of data. This differentiates them from the arithmetic, logical, and character operators discussed earlier in this chapter, which operate on individual data elements.

Most Relational Database Management Systems support the following aggregate functions:

- SUM Sum of column values
- AVG Average of column values
- STDEV Standard deviation of column values
- COUNT Count of rows in column
- MAX Maximum value in column
- MIN Minimum value in column

Aggregate functions are used to provide statistical or summary information about groups of data elements. These groups may be created specifically using the GROUP BY clause, or the aggregate functions may be applied to the default group, which is the entire result set.

Here's a good practical example of the use of most of the common aggregate functions:

```
SELECT DESCRIPTION, COUNT(DESCRIPTION) AS 'COUNT', AVG(COST)
    AS 'AVERAGE COST', MIN(COST) AS 'LOWEST COST', MAX(COST)
     AS 'HIGHEST COST'
FROM INVENTORY
GROUP BY DESCRIPTION;
```

This query generates the following results:

Description	Count	Average Cost	Lowest Cost	Highest Cost
Cereal	4	1.745	0.98	2.05
Cookies	3	1.2866	1.03	1.57
Soda	3	0.63	0.57	0.71

Note The fundamental difference between aggregate functions and standard functions is that the former use the entire column of data as their input and produce a single output.

Using the HAVING Clause to Filter Groups

There are going to be situations where you'll want to filter the groups themselves in much the same way as you filter records using the WHERE clause. For example, you may want to analyze your sales by state, but ignore states with a limited number of customers.

To filter groups, apply a HAVING clause after the GROUP BY clause. The HAVING clause lets you apply a qualifying condition to groups so that the RDBMS returns a result only for the groups that satisfy the condition.

HAVING clauses can contain one or more predicates connected by ANDs and ORs. Each predicate compares a property of the group (such as COUNT(State)) with either another property of the group or a constant.

The following example shows the use of the HAVING clause to compute a count of customers by state, filtering out results from states with only one customer:

```
SELECT DESCRIPTION, STATE, COUNT(STATE) AS 'COUNT',
    SUM(oi.QTY * i.COST) AS TOTAL
FROM CUSTOMERS c, ORDERS o, ORDERED_ITEMS oi,
     INVENTORY i
WHERE c.CUSTOMER_NUMBER = o.CUSTOMER_NUMBER AND
    o.ORDER_NUMBER = oi.ORDER_NUMBER AND
     i.ITEM_NUMBER = oi.ITEM_NUMBER
GROUP BY STATE, DESCRIPTION
HAVING COUNT(STATE) > 1;
```

This query yields a result set that looks like this:

DESCRIPTION	STATE	COUNT	TOTAL
Cereal	NJ	2	14.1
Cereal	NY	2	4.88
Cookies	NJ	2	5.74
Cookies	NY	2	11.32
Soda	NY	2	5.4

You can also apply a HAVING clause to the entire result set by omitting the GROUP BY clause. In this case, the DBMS treats the entire table as one group, so there is at most one result row. If the HAVING condition is not true for the table as a whole, no rows will be returned.

HAVING enables you to use aggregate functions in a comparison statement, providing for aggregate functions what WHERE provides for individual rows.

Using Indexes to Improve the Efficiency of SQL Queries

You can improve database performance significantly by using *indexes*. An index is a structure that provides a quick way to look up specific items in a table or view. In effect, an index is an ordered array of pointers to the rows in a table or view.

When you assign a unique id to each row as a key, you are predefining an index for that table. This makes it much faster for the DBMS to look up items by id, which is commonly required when you are doing joins on the id column.

SQL's CREATE INDEX statement allows you to add an index for any desired column or group of columns. If you need to do a search by customer name, for example, the fact that the table has a built-in index on the primary key doesn't help, so the DBMS has to do a brute force search of the entire table to find all customer names matching your query. If you plan on doing a lot of queries by customer name, it obviously makes sense to add an index to the customer name column or columns. Otherwise, the task is like looking up names in a phone list that hasn't been alphabetized.

The SQL command to add an index uses the CREATE INDEX key word, specifying a name for the index and defining the table name and the column list to index. Here's an example:

```
CREATE INDEX STATE_INDEX ON MEMBER_PROFILES(STATE);
```

To remove the index, use the DROP INDEX command as follows:

```
DROP INDEX MEMBER_PROFILES.STATE_INDEX;
```

Notice how the name of the index has to be fully defined by prefixing it with the name of the table to which it applies.

Formatting SQL Commands

The SQL engine ignores excess whitespace, so you can and should insert line breaks for clarity. Conventionally major clauses such as the FROM clause and the WHERE clause are placed on their own lines, unless the command is so brief as to be trivial. A good basic approach when you are not quite sure how to format a command is to go for readability.

Key words, table names, and column names are not case-sensitive, but the contents of the records within a table are case-sensitive. This means that with a little thought, you can use case to help make your SQL statements more readable.

Caution Although SQL ignores case in commands, table names, column names, and so on, case can matter when you are using a name in a WHERE clause, so 'Corleone' and 'CORLEONE' are not necessarily the same.

Using SQL Joins

Recall that the information in a practical database is usually distributed across several different tables, each of which contains sets of logically related data. The example introduced in Chapter 1 represents a typical database containing the four following tables:

- **Customers** contains customer number, name, shipping address, and billing information.
- **Inventory** contains item number, name, description, cost, and quantity on hand.
- **Orders** contains order number, customer number, order date, and ship date.
- **Ordered_Items** contains order number, item number, and quantity.

When a customer places an order, an entry is made in the Orders Table, assigning an order number and containing the customer number and the order date. Then entries are added to the Ordered_Items Table, recording order number, item number and quantity. To fill a customer order, combine the necessary information from each of these tables.

A few rows of each of these tables are shown in Tables 3-10 through 3-13.

Table 3-10: Customer Table

Customer_Number	First_Name	MI	Last_Name	Street	City	State	Zip

Table 3-10: Customer Table

Customer_Number	First_Name	MI	Last_Name	Street	City	State	Zip
100	Michael	A	Corleone	123 Pine	New York	NY	10006
101	Fredo	X	Corleone	17 Main	New York	NY	10007
102	Sonny	A	Corleone	123 Walnut	Newark	NJ	12346

Table 3-11: Inventory Table

Item_Number	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	130	1.95
1002	Rice Krispies	Cereal	97	1.87
1005	Chocolate Chip	Cookies	217	1.26
1006	Fig Bar	Cookies	162	1.57
1008	Cola	Soda	144	0.61
1010	Orange Soda	Soda	84	0.71

Table 3-12: Orders Table

Order_Number	Customer_Number	Order_Date	Ship_Date
2	101	12/8/01	12/10/01
3	103	12/9/01	12/11/01

Table 3-13: Ordered Items Table

ID	Order_Number	Item_Number	Qty
5000	2	1001	2
5001	2	1004	1
5004	3	1006	4
5005	3	1009	2

One of the most powerful features of SQL is its ability to combine data from several different tables using the JOIN statement. Using JOINS, you are able to produce a detailed invoice showing the customer name, shipping address, and billing information, with a detailed list of the items ordered, including description, quantity, unit price, and extended price (unit price * quantity).

Using keys in a JOIN

The most important thing to understand when discussing SQL JOINS is the use of primary and foreign keys. Database-management systems use the two following kinds of keys:

- Primary keys
- Foreign keys

In each of the four tables in the sample database, there is an identifier such as customer number or item number. These identifiers are the *primary keys* and are used to provide a unique reference to a given record. A primary key is a column that uniquely identifies the rest of the data in any given row. For example, in the Customers Table, the Customer_Number column uniquely identifies that customer. For this to work, no two rows can have the same key or, in this instance, Customer_Number.

A *foreign key* is a column in a table where that column is a primary key of another table. For example, the Orders Table contains one column for Order_Number, which is the primary key for the Orders Table, and another column for the Customer_Number, which is a foreign key. In effect, the foreign key acts as a pointer to a row in the Customers Table.

The purpose of these keys is to establish relationships across tables, without having to repeat data in every table. This concept encapsulates the power of relational databases.

Accessing data from multiple tables with Equi-Joins

SQL Joins work by matching up equivalent columns in different tables by comparing keys. The most common type of Join is an Equi-Join, where you look for items in one table which have the same item number as items in another.

Writing SQL JOIN Commands

There are two ways to write SQL JOIN statements. The first is through the specific use of the key word JOIN:

```
SELECT First_Name, Last_Name, Order_Number
FROM CUSTOMERS c INNER JOIN
    ORDERS o ON c.Customer_Number = o.Customer_Number;
```

This statement will return exactly the same results as the short form:

```
SELECT First_Name, Last_Name, Order_Number
FROM CUSTOMERS c, ORDERS o
WHERE c.Customer_Number = o.Customer_Number;
```

The result set which will be returned by either statement is:

First_Name	Last_Name	Order_Number
Fredo	Corleone	2
Francis	Corleone	3
Vito	Corleone	4
Kay	Adams	5

For example, the Ordered Items Table provides a link between the order number and the items in the Inventory Table. To get a detailed list of the inventory items corresponding to order number 2, you can write the following SQL JOIN command:

```
SELECT Orders.Order_number, Ordered_Items.Item_number,
    Ordered_Items.Qty, Inventory.Name,
     Inventory.Description
FROM Orders, Ordered_Items, Inventory
WHERE Orders.order_number = Ordered_Items.order_number AND
    Inventory.Item_Number = Ordered_Items.Item_Number AND
    Orders.order_number = 2;
```

Notice how the columns used in the WHERE clause comparison are the key columns of the various tables. This yields the following ResultSet:

Order_number	Item_number	Qty	Name	Description
2	1001	2	Corn Flakes	Cereal
2	1004	1	Oatmeal	Cereal

Non-Equi-Joins

In addition to Equi-Joins, you can do Non-Equi-Joins, Joins where the relationship is not equal, though they are not very common. For example, since there are only two orders in the Orders Table, you can get the other order using the Non-Equi-Join, as shown here:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer, oi.Qty,
    i.Name, i.Description, i.Cost * 1.6 AS Price_Each,
    i.Cost * 1.6 * oi.Qty AS Price
FROM Orders o, Customers c, Ordered_Items oi, Inventory i
WHERE o.Order_number = oi.Order_number AND
     c.Customer_Number = o.Customer_Number AND
    i.Item_Number = oi.Item_Number AND o.Order_number <> 2;
```

Inner and Outer Joins

The Joins discussed so far have been *Inner Joins*. An Inner Join exists between two tables and includes only rows with matching rows in the both tables. The easiest way to understand the terminology of Inner and Outer Joins is to look at Figure 3-1, where the Customer_Number columns in the Customers and Orders Tables have been overlapped or "Joined."

First_Name	MI	Last_Name	Customer Number	Order Number	Order_Dat	Ship_Date
Michael	A	Corleone	101	2	12/8/01	12/10/01
Fredo	X	Corleone	102	3	12/9/01	12/11/01
Sonny	A	Corleone	103	4	12/9/01	12/11/01
Francis	X	Corleone	104	5	12/10/01	12/12/01
Vito	G	Corleone	105			
Tom	B	Hagen	106			
Kay	K	Adams	107			
Francis	F	Coppola	108			
Mario	S	Puzo	109			

Figure 3-1: Tables joined on customer number

The two tables are shown in the rounded boxes; the joined fields are shaded.

Using an Inner Join, as shown in the last example, you can only list customers who have placed an order, so their customer numbers fall into the shaded area of Figure3-1. If you want a list of all customers, together with the dates of any orders they have placed, you can't get there with an Inner Join.

An *Outer Join* can include not only records that are inside the union of the sets or tables but records that are outside the union of the sets. In other words, in addition to the set members that share customer numbers, you can get customers in the lower, or "Outer," part of the joined tables.

These are the three different types of Outer Joins:

- LEFT OUTER JOIN (*=)
- RIGHT OUTER JOIN (=*)
- FULL OUTER JOIN

The terms LEFT, RIGHT, and FULL describe which of the tables' unmatched columns to include in the Join relative to the order in which the tables appear in the JOIN command.

LEFT OUTER JOIN

The LEFT OUTER JOIN operator includes all rows from the left side of the join. This includes all the customers who have not placed any orders, as shown here:

```
SELECT c.Last_Name, c.First_Name, o.Order_Date
FROM Customers c LEFT OUTER JOIN
```

```
    Orders o ON c.Customer_number = o.Customer_Number;
```

The result set this query generates is shown in Table 3-14. Note the NULLs listed under order date where the customer hasn't actually placed an order.

Table 3-14: Results of Left Outer Join

Last_Name	First_Name	Order_Date
Corleone	Michael	<NULL>
Corleone	Fredo	12/8/01
Corleone	Sonny	<NULL>
Corleone	Francis	12/9/01
Corleone	Vito	12/9/01
Hagen	Tom	<NULL>
Adams	Kay	12/10/01
Coppola	Francis	<NULL>
Puzo	Mario	<NULL>

RIGHT OUTER JOIN

It is important to note that "left" and "right" are completely dependent on the order of the tables in the SQL statement, so you can turn this into a RIGHT OUTER JOIN by reversing the order of the tables in the JOIN command. Here's an example:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer,
    o.Order_Date
FROM ORDERS o RIGHT OUTER JOIN
    CUSTOMERS c ON c.customer_number = o.customer_number;
```

OUTER JOIN commands can also be written in a shorthand similar to the form we use for our INNER JOIN. The form for the LEFT OUTER JOIN uses the "*=" operator, as shown here:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer,
    o.Order_Date
FROM CUSTOMERS c, ORDERS o
WHERE c.customer_number *= o.customer_number;
```

The form for the RIGHT OUTER JOIN uses the "=*" operator as follows:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer,
```

```
    o.Order_Date
FROM ORDERS o, CUSTOMERS c
WHERE o.customer_number =* c.customer_number;
```

Note In the shorthand version, the type of JOIN depends on both the order of the tables in the FROM clause and the position of the asterisk in the *= operator.

FULL OUTER JOIN

A "full outer join" includes all unmatched rows from both tables in the result. For example, to find any orders in the Orders Table with customer numbers that do not match any entries in our Customers Table, you can execute a Full Outer Join to show all the entries in both tables. Here's an example:

```
SELECT c.Last_Name, c.First_Name, o.Order_Date
FROM Customers c FULL OUTER JOIN
    Orders o ON c.Customer_number = o.Customer_Number;
```

The result set generated by this join is the same as the results shown in Table 3-14, since all orders have a corresponding customer. However, if, for some reason, an order placed on 12/12/01existed in the Orders Table with no corresponding entry in the Customers Table, the additional row shown at the bottom of Table 3-15 would be generated.

Table 3-15: Results of FULL OUTER JOIN

Last_Name	First_Name	Order_Date
Corleone	Michael	<NULL>
Corleone	Fredo	12/8/01
Corleone	Sonny	<NULL>
Corleone	Francis	12/9/01
Corleone	Vito	12/9/01
Hagen	Tom	<NULL>
Adams	Kay	12/10/01
Coppola	Francis	<NULL>
Puzo	Mario	<NULL>
<NULL>	<NULL>	12/12/01

Using NOT EXISTS

Now you know how to use INNER JOINS to find records from two tables with matching fields, and how to use OUTER JOINS to find all records, matching or nonmatching. Next, consider a case in which you want to find records from one table that don't have corresponding records in another.

Using the Customers and Orders Tables again, find all the customers who have not placed an order. The way to do this is to find customer records with customer numbers that do not exist in the Orders Table. This is done using NOT EXISTS:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer
FROM CUSTOMERS c
WHERE NOT EXISTS
        (SELECT *
      FROM orders o
      WHERE o.customer_number = c.customer_number);
```

Self-joins

A *self-join* is simply a normal SQL join that joins a table to itself. You use a self-join when rows in a table contain references to other rows in the same table. An example of this situation is a table of employees, where each record contains a reference to the employee's supervisor by Employee_ID. Since the supervisor is also an employee, information about the supervisor is stored in the Employees Table, as shown in Table 3-16, so you use a self-join to access it.

Table 3-16: Employees Table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SUPERVISOR
100	Michael	Corleone	104
101	Fredo	Corleone	100
102	Sonny	Corleone	100
103	Francis	Corleone	100
104	Vito	Corleone	99
105	Tom	Hagen	100
106	Kay	Adams	100
107	Francis	Coppola	100
108	Mario	Puzo	100

Since a join implicitly requires two table names, identifying the tables to be joined, you can create a self-join by using table-name aliases to give each reference to the table a separate name. To get a list of employees and their supervisors, create a self-join

by creating two separate references to the Employees Table, using two different aliases:

```
SELECT e.Last_Name, e.First_Name,
    boss.Last_Name + ', ' + boss.First_Name AS Boss
FROM EMPLOYEES e, EMPLOYEES boss
WHERE e.supervisor = boss.employee_id
```

The preceding SQL code is effectively creating what looks like two identical tables, E and Boss, and joining them using an Inner Join. This approach allows you to get the employee information from one reference to the table and supervisor information from the other, as shown here:

Last_Name	First_Name	Boss
Corleone	Michael	Corleone, Vito
Corleone	Fredo	Corleone, Michael
Corleone	Sonny	Corleone, Michael
Corleone	Francis	Corleone, Michael
Hagen	Tom	Corleone, Michael
Adams	Kay	Corleone, Michael
Coppola	Francis	Corleone, Michael

You can turn this into an Outer Self-Join very easily, as follows:

```
SELECT e.last_name, e.first_name,
    boss.last_name + ', ' + boss.first_name AS Boss
FROM EMPLOYEES e, employees boss
WHERE e.supervisor *= boss.employee_id;
```

This returns one additional row, since the Employee_ID of Vito's supervisor does not appear in the Employees Table. His boss appears as <NULL>, as shown here:

Last_Name	First_Name	Boss
Corleone	Michael	Corleone, Vito
Corleone	Fredo	Corleone, Michael
Corleone	Sonny	Corleone, Michael
Corleone	Francis	Corleone, Michael

Last_Name	First_Name	Boss
Corleone	Vito	<NULL>
Hagen	Tom	Corleone, Michael
Adams	Kay	Corleone, Michael
Coppola	Francis	Corleone, Michael

Cartesian Products

Cartesian products, or cross products, are something you normally want to avoid. The Cartesian product of a Join occurs when every record in one table is joined on every record of the other, so the Cartesian product of two tables 100-rows long is 10,000 rows.

Cartesian products are normally an error, caused by a bad or nonexistent WHERE clause. In the case of a small table like the ones in our examples, this is not a major problem; but on a large database, the time taken to generate cross products of thousands of rows can be significant.

Using the UNION Operator to Combine Queries

Another way to combine data from two separate sources is to use the UNION operator. The default action of the UNION operator is to combine the results of two or more queries into a single query and to eliminate any duplicate rows. When ALL is used with UNION, duplicate rows are not eliminated.

In the following example, the first query returns the names and addresses of all the Corleones; the second returns all customers in New Jersey. The UNION operator combines the results, removing the duplicate records that are generated for Corleones in New Jersey:

```
SELECT First_Name, Last_Name, Street, City, State
FROM Customers
WHERE Last_Name = 'Corleone'
UNION
SELECT First_Name, Last_Name, Street, City, State
FROM Customers
WHERE State = 'NJ'
ORDER BY Last_Name, First_Name;
```

You can use ORDER BY, as shown, to sort the combined answer set by adding the ORDER BY clause after the last query. Here is the result:

First_Name	Last_Name	Street	City	State
Kay	Adams	109 Maple	Newark	NJ
Francis	Corleone	17 Main	New York	NY
Fredo	Corleone	19 Main	New York	NY
Michael	Corleone	123 Pine	New York	NY
Sonny	Corleone	123 Walnut	Newark	NJ
Vito	Corleone	23 Oak St	Newark	NJ
Tom	Hagen	37 Chestnut	Newark	NJ

You do not have to use the same columns in each query. Only the column counts and column types need to match. However, if you create a UNION of two result sets with different columns, you have to apply the ORDER BY clause using the column number.

EXCEPT operator

The EXCEPT operator creates a result set by including all rows that the first query returns but not rows that the second query returns. The default version eliminates all duplicate rows; EXCEPT ALL does not. The following statement will return the names and addresses of all Corleones except those living in New Jersey:

```
SELECT First_Name, Last_Name, Street, City, State
FROM Customers
WHERE Last_Name = 'Corleone'
EXCEPT
SELECT First_Name, Last_Name, Street, City, State
FROM Customers
WHERE State = 'NJ'
```

INTERSECT operator

The INTERSECT operator creates a result set by including only rows that exist in both queries and eliminating all duplicate rows. When you use ALL with INTERSECT, the duplicate rows are not eliminated. The following statement will return the names and addresses of Corleones living in New Jersey:

```
SELECT First_Name, Last_Name, Street, City, State
FROM Customers
```

```
WHERE Last_Name = 'Corleone'
INTERSECT
SELECT First_Name, Last_Name, Street, City, State
FROM Customers
WHERE State = 'NJ';
```

Data Control Language

The Data Control Language (DCL) provides the tools to manage the database and control such aspects as user-access privileges. Since a database usually represents a significant investment in time and effort, managing users is an important aspect of database management.

A *user* is anyone who has access to the database. Users can be granted different privileges, ranging from read-only access to a limited portion of the database, all the way up to unlimited access to the entire RDBMS.

Managing Users

To add individual users to a database, the database administrator must create database users. This is done using the CREATE USER command. When you create a user, you can assign a password, certain basic permissions and an expiration date, all in one command. You can also add the user to an existing user group.

After creating a user, you may need to modify his or her privileges, perhaps to add the right to modify or delete certain tables or to change the user's password. These functions are handled using the ALTER USER command.

Finally, you may need to remove an individual's access to the database entirely. This is done using the DROP USER command.

User privileges

Relational Database Management Systems define sets of privileges that can be assigned to users. These privileges correspond to actions that can be performed on objects in the database. User privileges can be assigned at two different levels. Users can be restricted both at the level of the types of actions they can perform, such as READ, MODIFY, or WRITE, and at the level of the types of database objects they can access.

Access-level privileges can generally be assigned at the following levels:

- Global level access to all databases on a given server
- Database level access to all tables in a given database
- Table-level access to all columns in a given table

- Column-level access to single columns in a given table

Normally, the management of user privileges is an administrative function that the database administrator handles.

Frequently, user privileges are assigned by defining a user's *role*. Database roles are simply predefined sets of user privileges. Like users, user groups and roles are managed using SQL commands. Most RDBMSes support the following roles or their equivalents:

- **Owner** – A user who can read or write data and create, modify, and delete the database or its components
- **Writer** – A user who is allowed to read or write data
- **Reader** – Someone who is allowed to read data but not write to the database
- **Public** – The lowest possible in terms of privileges

User roles are a neat administrative feature designed to save time for the database administrator. Like groups, roles can be defined by the database administrator as required.

Managing user groups

In addition to defining individual users, many systems allow the database administrator to organize users into logical groups with the same privileges. Groups are created in much the same way as individual users. The general syntax for CREATE GROUP is as follows:

```
CREATE GROUP group_name WITH USER user1, user2
```

Like users, groups are dropped using the DROP command, as shown here:

```
DROP GROUP group_name
```

To add a user to a group, use the ALTER GROUP ADD command; to delete users, use the ALTER GROUP DROP command, as shown here:

```
ALTER GROUP group_name ADD USER username [, ... ]
ALTER GROUP group_name DROP USER username [, ... ]
```

A significant difference between adding and dropping groups as opposed to adding and dropping individual users is that when a group is altered or dropped, only the group is affected. Any users in a group that is dropped simply lose their membership in the group. The users are otherwise unaffected. Similarly, when a group is altered by dropping a user, only the group is affected. The user simply loses his or her membership in the group but is otherwise unaffected.

Granting and revoking user privileges

The SQL GRANT command is used to grant users the necessary access privileges to perform various operations on the database. In addition to granting a user specified access privileges, the GRANT command can be used to allow the user to grant a privilege to other users. There is also an option allowing the user to grant privileges on all subtables and related tables. These two versions of the GRANT command look like this:

```
GRANT privilege ON table_name TO user_name;
GRANT SELECT ON PRODUCTS WITH GRANT OPTION TO jdoe;
```

The REVOKE command is used to revoke privileges granted to a user. Like the GRANT command, this command can be applied at various levels.

The REVOKE command is used to revoke privileges from users so that they cannot do certain tasks on the database. Just like the GRANT command, this command can be applied at various levels. It is important to note that the exact syntax of this command might differ as per your database. For example, the following command revokes the SELECT privileges from John Doe, on the Products Table.

```
REVOKE SELECT ON PRODUCTS FROM jdoe
```

Creating and Using Stored Procedures

A stored procedure is a saved collection of SQL statements that can take and return user-supplied parameters. You can think of a stored procedure as a method or function, written in SQL. There are obviously a number of advantages to using stored procedures, including:

- Stored procedures are precompiled, so they will execute fast.
- Stored procedures provide a standardised way of performing common tasks.

Almost any SQL statement can be used as a stored procedure. All that is required is to provide a procedure name and a list of variables:

```
CREATE PROCEDURE procedure_name
   @parameter data_type,
   @parameter data_type = default_value,
   @parameter data_type OUTPUT
AS
   sql_statement [ ...n ]
```

Variable names are specified using an at sign @ as the first character. Otherwise the name must conform to the rules for identifiers. Variable names cannot be used in

place of table names, column names, or the names of other database objects. They can only be used to pass values to and from the stored procedure.

In addition to the variable name, you must specify a data type. All data types can be used as a parameter for a stored procedure. You can also specify a default value for the variable, as shown in the example.

If you want to return a value to the caller, you must specify the variable used for the return value using the OUTPUT keyword. You can then set this value in the body of the stored procedure.

The AS keyword is used to identify the start of the SQL statement forming the body of the stored procedure. A very simple stored procedure with no parameter variables might look like:

```
CREATE PROCEDURE LIST_ORDERS_BY_STATE
AS
SELECT
     o.Order_Number,
   c.Last_Name + ', ' + c.First_Name AS Name,
     c.State
FROM Customers c,Orders o
  WHERE c.Customer_Number = o.Customer_Number
  ORDER BY c.State,c.Last_Name;
```

To execute this stored procedure, you simply invoke it by name. The following code snippet shows how:

```
LIST_ORDERS_BY_STATE;
```

The stored procedure will return a result set which looks like:

Order_Number	Name	State
5	Adams, Kay	NJ
4	Corleone, Vito	NJ
2	Corleone, Fredo	NY
3	Corleone, Francis	NY

Using Input Parameters in a Stored Procedure

The following code snippet shows how you can use input parameters in a stored procedure. This particular store procedure was designed to handle the input from an

HTML form. Notice that the variable names are not required to be the same as the column names:

```
CREATE PROCEDURE INSERT_CONTACT_INFO
@FName VARCHAR(20), @MI CHAR(1), @LName VARCHAR(30),
@Street VARCHAR(50), @City VARCHAR(30), @ST CHAR(2),
@ZIP VARCHAR(10), @Phone VARCHAR(20), @Email VARCHAR(50)
AS
INSERT INTO CONTACT_INFO
        (First_Name, MI, Last_Name,
         Street, City,  State, ZIP, Phone, Email)
VALUES
        (@FName, @MI, @LName,
         @Street, @City, @ST, @ZIP, @Phone, @Email);
```

The SQL statement used to call this procedure is very similar to the statement shown in the previous example. The only difference is the use of the input parameters obtained from the HTML form:

```
INSERT_CONTACT_INFO 'Charles', 'F', 'Boyer', '172 Michelin',
'Detroit', 'MI', '76543', '900-555-1234', 'charles@boyer.net'
```

Using Output Parameters in a Stored Procedure

Creating a stored procedure which uses output parameters is also quite straightforward. The example shows a stored procedure which returns a validation message when a UserName, Password pair is checked against a table:

```
CREATE PROCEDURE CHECK_USER_NAME
     @UserName varchar(30),
     @Password varchar(20),
    @PassFail varchar(20) OUTPUT
AS
    IF EXISTS(Select * From Customers
       WHERE Last_Name = @UserName
         AND
        First_Name = @Password)
         BEGIN
           SELECT @PassFail = "PASS"
         END
     ELSE
         BEGIN
           SELECT @PassFail = "FAIL"
         END
```

You can check the output from this stored procedure by declaring a variable such as @PFValue and passing it to the stored procedure as an OUTPUT, as shown below. In this example, the result is stored to a new table, PWCHECK:

```
DECLARE @PFValue VARCHAR(20)
EXECUTE CHECK_USER_NAME 'Corleone', 'Michael', @PFValue OUTPUT
INSERT INTO PWCHECK
                VALUES ('Corleone', 'Michael', @PFValue)
```

Summary

This chapter provides a brief but fairly comprehensive overview of SQL. You should now be able to create and populate a database and to use SQL to perform fairly complex queries.

Specifically, you learn about using SQL when:

- Creating and populating databases and tables
- Querying a database
- Using primary and foreign keys to join tables
- Managing database security

Chapter 4 discusses Java Database Connectivity (JDBC), which enables you to use your knowledge of SQL in a Java application. Much of the rest of the book explains how to do this in the context of a variety of practical applications.

Chapter 4: Introduction to JDBC

In This Chapter

- Understanding DriverManager and different types of JDBC drivers
- Using JDBC DataSources for simple, pooled, and distributed connections
- Using Statements, PreparedStatements, and CallableStatements
- Using transactions, isolation levels, and SavePoints
- Using ResultSets and Rowsets
- Using MetaData
- Mapping of SQL data types in JDBC

JDBC is a Java Database Connectivity API that lets you access virtually any tabular data source from a Java application. In addition to providing connectivity to a wide range of SQL databases, JDBC allows you to access other tabular data sources such as spreadsheets or flat files. Although JDBC is often thought of as an acronym for *Java Database Connectivity*, the trademarked API name is actually JDBC.

What Is JDBC?

JDBC is a Java Database Connectivity API that lets you access virtually any tabular data source from a Java application. In addition to providing connectivity to a wide range of SQL databases, JDBC allows you to access other tabular data sources such as spreadsheets or flat files. Although JDBC is often thought of as an acronym for *Java Database Connectivity*, the trademarked API name is actually JDBC.

JDBC defines a low-level API designed to support basic SQL functionality independently of any specific SQL implementation. This means the focus is on executing raw SQL statements and retrieving their results. JDBC is based on the X/Open SQL Call Level Interface, an international standard for programming access to SQL databases, which is also the basis for Microsoft's ODBC interface.

The JDBC 2.0 API includes two packages: java.sql, known as the JDBC 2.0 core API; and javax.sql, known as the JDBC Standard Extension. Together, they contain the necessary classes to develop database applications using Java. As a core of the Java 2 Platform, the JDBC is available on any platform running Java.

The JDBC 3.0 Specification, released in October 2001, introduces several features, including extensions to the support of various data types, additional MetaData capabilities, and enhancements to a number of interfaces.

The JDBC Extension Package (javax.sql) was introduced to contain the parts of the JDBC API that are closely related to other pieces of the Java platform that are themselves optional packages, such as the Java Naming and Directory Interface (JNDI) and the Java Transaction Service (JTS). In addition, some advanced features

that are easily separable from the core JDBC API, such as connection pooling and rowsets, have been added to javax.sql. Putting these advanced facilities into an optional package instead of into the JDBC 2.0 core API helps to keep the core JDBC API small and focused.

The main strength of JDBC is that it is designed to work in exactly the same way with any relational database. In other words, it isn't necessary to write one program to access an Oracle database, another to access a Sybase database, another for SQL Server, and so on. JDBC provides a uniform interface on top of a variety of different database-connectivity modules. As you will see in Part II of this book, a single program written using JDBC can be used to create a SQL interface to virtually any relational database. The three main functions of JDBC are as follows:

- Establishing a connection with a database or other tabular data source
- Sending SQL commands to the database
- Processing the results

Listing 4-1 provides a simple example of the code required to access an Inventory database containing a table called Stock, which contains the names, descriptions, quantities, and costs of various items. The three steps required to use JDBC to access data are clearly illustrated in the code.

Listing 4-1: Simple example of JDBC functionality

```
package java_databases.ch04;

import java.sql.*; // imports the JDBC core package

public class JdbcDemo{
  public static void main(String args[]){
     int qty;
     float cost;
     String name;
     String desc;
    // SQL Query string
    String query = "SELECT Name,Description,Qty,Cost FROM Stock";

     try {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // load the JDBC driver
       Connection con = DriverManager.getConnection ("jdbc:odbc:Inventory");
       // get a connection
       Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query); // execute query
       while (rs.next()) { // parse the results
          name = rs.getString("Name");
```

```
            desc = rs.getString("Description");
            qty  = rs.getInt("Qty");
            cost = rs.getFloat("Cost");
            System.out.println(name+", "+desc+"\t: "+qty+"\t@ $"+cost);
          }
          con.close();
        }
        catch(ClassNotFoundException e){
          e.printStackTrace();
        }
        catch(SQLException e){
          e.printStackTrace();
        }
    }
}
```

The example illustrates the following main steps required to access a database and retrieve data from a ResultSet using the JDBC API:

- Load a JDBC driver.
- Get a connection to the database.
- Create a statement.
- Execute a SQL query.
- Retrieve data from the ResultSet.

The ResultSet provides the methods necessary to loop through the results and get the individual database fields using methods appropriate to their respective types. Here's an example:

Steiner 10 x 50	Binoculars	10	$799.95
Steiner 8 x 30	Binoculars	30	$299.95
PYGMY-2	Night Vision Monocular	20	$199.95

The JDBC API defines standard mappings between SQL data types and Java/JDBC data types, including support for SQL99 advanced data types such as BLOBs and CLOBs, ARRAYs, REFs, and STRUCTs.

Note This example uses the JDBC.ODBC bridge. JDBC supports a wide range of different drivers of four distinctly different types. These are discussed in the section on driver types later in this chapter.

The JDBC API can be used directly from your application or as part of a multi-tier server application as shown in the next section.

Two-Tier and Three-Tier Models

The JDBC API supports both two-tier and three-tier models for database access. In other words, JDBC can either be used directly from your application or as part of a middle-tier server application.

Two-Tier Model

In the two-tier model, a Java application interacts directly with the database. Functionality is divided into these two layers:

- **Application layer**, including the JDBC driver, business logic, and user interface
- **Database layer**, including the RDBMS

The interface to the database is handled by a JDBC driver appropriate to the particular database management system being accessed. The JDBC driver passes SQL statements to the database and returns the results of those statements to the application.

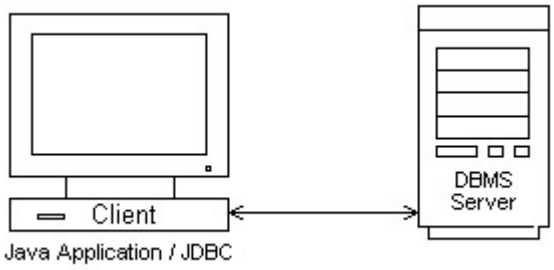


Figure 4-1: Two-tier client/server configuration

A client/server configuration is a special case of the two-tier model, where the database is located on another machine, referred to as the *server*. The application runs on the client machine, which is connected to the server over a network. Commonly, the network is an intranet, using dedicated database servers to support multiple clients, but it can just as easily be the Internet.

Part II of this book illustrates the use of basic JDBC and SQL functionality in the context of a basic two-tier application using simple Swing components to create a generic RDBMS GUI. The inherent flexibility of a Java/JDBC approach to developing database applications enables you to access a wide range of RDBMS systems, including Oracle, Sybase, SQL Server, and MySQL as well as MS Office applications, using this GUI.

Three-tier Model

In the three-tier model illustrated in Figure 4-2, commands are sent to an application server, forming the middle tier. The application server then sends SQL statements to the database. The database processes the SQL statements and sends the results back to the application server, which then sends them to the client.

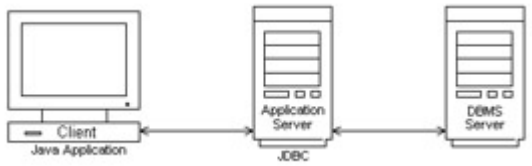


Figure 4-2: Three-tier model typical of Web applications

These are some advantages of three-tier architecture:

- Performance can be improved by separating the application server and database server.
- Business logic is clearly separated from the database.
- Client applications can use a simple protocol such as CGI to access services.

The three-tier model is common in Web applications, where the client tier is frequently implemented in a browser on a client machine, the middle tier is implemented in a Web server with a servlet engine, and the database management system runs on a dedicated database server.

The main components of a three-tier architecture are as follows:

- **Client tier**, typically a thin presentation layer that may be implemented using a Web browser
- **Middle tier**, which handles the business logic or application logic. This may be implemented using a servlet engine such as Tomcat or an application server such as JBOSS. The JDBC driver also resides in this layer.
- **Data-source layer**, including the RDBMS

Part III of this book illustrates additional capabilities of the JDBC API in a three-tier application that uses a Web browser as the client, an Apache/Tomcat server as the middle tier, and a relational database management system as the database tier.

SQL Conformance

Although SQL is the standard language for accessing relational databases, different RDBMS systems support a large number of different dialects of SQL. These differences range from such minor details as whether a SQL statement needs a closing semicolon to major variations such as the absence of support for stored procedures or some types of joins in some database systems.

Another major difference is that many database management systems offer a lot of advanced functionality that SQL standards do not cover. These advanced features may be implemented in ways that are not consistent across different database systems. A very important design requirement of the JDBC API is that it must support SQL as it is rather than as the standards define it.

One way the JDBC API deals with this problem is by allowing any SQL String to be passed to an underlying DBMS driver. This feature means that an application is free to use whatever functionality a given DBMS might offer. The corollary is that some database management systems return an error response to some commands.

The JDBC API supports this ability to pass any SQL String to a database management system through an escape mechanism that provides a standard JDBC syntax for several of the more common areas of SQL divergence. For example, there are escapes for date literals and for stored procedure calls.

An additional support mechanism is provided by way of the DatabaseMetaData interface, which provides descriptive information about the DBMS. This is especially useful in cross-platform applications, where it can help you to adapt your application to the requirements and capabilities of different database management systems.

Just as there are variations in the implementation of the SQL standard, there can be variations in the level of a JDBC driver's compliance to the definition of the API. The concept of JDBC compliance is discussed in the next section.

JDBC Compliance

Sun created the "JDBC COMPLIANT™" designation to indicate that you can rely on a vendor's JDBC implementation to conform to a standard level of JDBC functionality. Before a vendor can use this designation, the vendor's driver must pass Sun's JDBC conformance tests. These conformance tests check for the existence of all of the classes and methods defined in the JDBC API, and, as far as possible, they check that the SQL Entry Level functionality is available.

The java.sql.Driver method jdbcCompliant() reports whether the driver is JDBC Compliant. A driver may only report "true" when this method is called if it passes the JDBC compliance tests; otherwise, it is required to return false. This method is not intended to encourage the development of non-JDBC compliant drivers. It exists merely in recognition of the fact that some vendors are interested in using the JDBC API and framework for lightweight databases that do not support full database functionality or for special databases such as document-information retrieval, where a SQL implementation may not be feasible.

Sun defines the three following levels of JDBC compliance:

- JDBC 1.0 API Compliance, which requires implementation of the following interfaces:
  - java.sql.Driver
  - java.sql.DatabaseMetaData (excluding those portions defined in the JDBC 2.0 and 3.0 extensions)
  - java.sql.ResultSetMetaData (excluding portions defined in the JDBC 2.0 and 3.0 extensions)
  - java.sql.Connection
  - java.sql.Statement
  - java.sql.CallableStatement
  - java.sql.PreparedStatement
  - java.sql.ResultSet
- JDBC 2.0 API Compliance, which requires:
  - JDBC 1.0 API Compliance
  - Full implementation of the DatabaseMetaData interface extensions defined in JDBC 2.0
  - Implementation of additional JDBC 2.0 ResultSet methods
- JDBC 3.0 API Compliance, which requires:
  - JDBC 2.0 API Compliance
  - Implementation of java.sql.ParameterMetaData
  - Implementation of java.sql.Savepoint
  - Full implementation of the DatabaseMetaData interface extensions defined in JDBC 3.0

Driver developers can ascertain that their drivers meet the JDBC Compliance standards by using the test suite available with the JDBC API.

Having discussed how variations in SQL implementations, and variations in JDBC compliance are handled, it is time to move on to the actual workings of the JDBC API. The next section discusses how JBC actually works.

How Does JDBC Work?

The key interfaces in the JDBC Core API are as follows:

- java.sql.DriverManager. In addition to loading JDBC drivers, the DriverManager is responsible for returning a connection to the appropriate driver. When getConnection() is called, the DriverManager attempts to locate a suitable driver for the URL provided in the call by polling the registered drivers.
- java.sql.Driver. The Driver object implements the acceptsURL(String url)method, confirming its ability to connect to the URL the DriverManager passes.
- java.sql.Connection. The Connection object provides the connection between the JDBC API and the database management system the URL specifies. A Connection represents a session with a specific database.
- java.sql.Statement. The Statement object acts as a container for executing a SQL statement on a given Connection.

- java.sql.ResultSet. The ResultSet object controls access to the results of a given Statement in a structure that can be traversed by moving a cursor and from which data can be accessed using a family of getter methods.

The DriverManager

The java.sql.DriverManager provides basic services for managing JDBC drivers. During initialization, the DriverManager attempts to load the driver classes referenced in the "jdbc.drivers" system property. Alternatively, a program can explicitly load JDBC drivers at any time using Class.forName(). This allows a user to customize the JDBC drivers their applications use.

A newly loaded driver class should call registerDriver() to make itself known to the DriverManager. Usually, the driver does this internally.

When getConnection() is called, the DriverManager attempts to locate a suitable driver from among those loaded at initialization and those loaded explicitly using the same classloader as the current applet or application. It does this by polling all registered drivers, passing the URL of the database to the drivers' acceptsURL() method.

There are three forms of the getConnection() method, allowing the user to pass additional arguments in addition to the URL of the database:

```
public static synchronized Connection getConnection(String url) throws
SQLException
public static synchronized Connection getConnection(String url,
                                                    String user,
                                                    String password)
throws SQLException

public static synchronized Connection getConnection(String url,
                                                    Properties info)
throws SQLException
```

Note When searching for a driver, JDBC uses the first driver it finds that can successfully connect to the given URL. It starts with the drivers specified in the sql.drivers list, in the order given. It then tries the loaded drivers in the order in which they are loaded.

JDBC drivers

To connect with individual databases, JDBC requires a driver for each database. JDBC drivers come in these four basic varieties.

- Types 1 and 2 are intended for programmers writing applications.

- Types 3 and 4 are typically used by vendors of middleware or databases.

A more detailed description of the different types of drivers follows.

JDBC driver types

The four structurally different types of JDBC drivers are as follows:

- Type 1: JDBC-ODBC bridge plus ODBC driver
- Type 2: Native-API partly Java driver
- Type 3:JDBC-Net pure Java driver
- Type 4: Native-protocol pure Java driver

These types are discussed in the following sections.

Type 1: JDBC-ODBC bridge plus ODBC driver

The JDBC-ODBC bridge product provides JDBC access via ODBC drivers. ODBC (Open Database Connectivity) predates JDBC and is widely used to connect to databases in a non-Java environment. ODBC is probably the most widely available programming interface for accessing relational databases.

The main advantages of the JDBC-ODBC bridge are as follows:

- It offers the ability to connect to almost all databases on almost all platforms.
- It may be the only way to gain access to some low-end desktop databases and applications.

Its primary disadvantages are as follows:

- ODBC drivers must also be loaded on the target machine.
- Translation between JDBC and ODBC affects performance.

Type 2: Native-API partly Java driver

Type 2 drivers use a native API to communicate with a database system. Java native methods are used to invoke the API functions that perform database operations.

A big advantage of Type 2 drivers is that they are generally faster than Type 1 drivers. The primary disadvantages of Type 2 drivers are as follows:

- Type 2 drivers require native code on the target machine.
- The Java Native Interface on which they depend is not consistently implemented amongdifferent vendors of Java virtual machines.

Type 3:JDBC-Net pure Java driver

Type 3 drivers translate JDBC calls into a DBMS independent net protocol that is then translated to a DBMS protocol by a server.

Advantages of Type 3 drivers are the following:

- Type 3 drivers do not require any native binary code on the client.
- Type 3 drivers do not need client installation.
- Type 3 drivers support several networking options, such as HTTP tunneling.

A major drawback of Type 3 drivers is that they can be difficult to set up since the architecture is complicated by the network interface.

Type 4: Native-protocol pure Java driver

The Type 4 driver is a native protocol, 100-percent Java driver. This allows direct calls from a Java client to a DBMS server. Because the Type 4 driver is written in 100-percent Java, it requires no configuration on the client machine other than telling your application where to find the driver. This allows a direct call from the client machine to the DBMS server. Many of these protocols are proprietary, so these drivers are provided by the database vendors themselves.

Native protocol pure Java drivers can be significantly faster than the JDBC ODBC bridge. In Part II of this book, performance of the Opta2000 driver from I-Net is compared with the performance of the JDBC-ODBC bridge in a simple SQL Server application. Although this comparison is not intended to be anything more than a trivial indicator of the difference between the two, the Opta2000 driver's performance is clearly faster.

Cross-Reference To learn more about available drivers, you can visit the Web site Sun maintains at:

http://java.sun.com/products/jdbc/industry.html

This Web site provides an up-to-date listing of JDBC-driver vendors.

JDBC DataSource

The DataSource interface, introduced in the JDBC 2.0 Standard Extension API, is now, according to Sun, the preferred alternative to the DriverManager class for making a connection to a particular source of data. This source can be anything from a relational database to a spreadsheet or a file in tabular format.

A DataSource object can be implemented in these three significantly different ways, adding important and useful capabilities to the JDBC API:

- The basic DataSource that produces standard Connection objects that are not pooled or used in a distributed transaction
- A DataSource that supports connection pooling. Pooled connections are returned to a pool for reuse by another transaction.
- A DataSource that supports distributed transactions accessing two or more DBMS servers

With connection pooling, connections can be used over and over again, avoiding the overhead of creating a new connection for every database access. Reusing connections in this way can improve performance dramatically, since the overhead involved in creating new connections is substantial.

Distributed transactions are discussed later in this chapter. They involve tables on more than one database server. The JDBC DataSource can be implemented to produce connections for distributed transactions. This kind of DataSource implementation is almost always implemented to produce connections that are pooled as well.

DataSource objects combine portability and ease of maintenance with the ability to provide connection pooling and distributed transactions. These features make DataSource objects the preferred means of getting a connection to a data source.

DataSources and the Java Naming and Directory Interface

A DataSource object is normally registered with a Java Naming and Directory Interface (JNDI) naming service. This means an application can retrieve a DataSource object by name from the naming service independently of the system configuration.

JNDI provides naming and directory functionality to Java applications. It is defined to be independent of any specific directory-service implementation so that a variety of directories can be accessed in a common way.

The JNDI naming services are analogous to a file directory that allows you to find and work with files by name. In this case, the JNDI naming service is used to find the DataSource using the logical name assigned to it when it is registered with the JNDI naming service.

The association of a name with an object is called a *binding*. In a file directory, for example, a file name is bound to a file. The core JNDI interface for looking up, binding, unbinding, renaming objects, and creating and destroying subcontexts is the Context interface.

Context interface methods include the following:

- bind(String name,Object obj) — Binds a name to an object

- listBindings(String name)— Enumerates the names bound in the named context, along with the objects bound to them.
- lookup(String name)— Retrieves the named object

Obviously, using JNDI improves the portability of an application by removing the need to hard code a driver name and database name, in much the same way as a file directory improves file access by overcoming the need to reference disk cylinders and sectors.

Deploying and Using a Basic Implementation of DataSource

A JDBC DataSource maintains information about how to locate the data as a set of properties, such as the data-source name, the server name on which it resides, and the port number.

Deploying a DataSource object consists of three tasks:

- Creating an instance of the DataSource class
- Setting its properties
- Registering it with a JNDI naming service

The first step is to create the BasicDataSource object and set the ServerName, DatabaseName, and Description properties:

```
com.dbaccess.BasicDataSource ds = new com.dbaccess.BasicDataSource();
ds.setServerName("jupiter");
ds.setDatabaseName("CUSTOMERS");
ds.setDescription("Customer database");
```

The BasicDataSource object is now ready to be registered with a JNDI naming service. The JNDI API is used in the following way to create an InitialContext object and to bind the BasicDataSource object ds to the logical name jdbc/customerDB:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/customerDB", ds);
```

The prefix jdbc is a JNDI subcontext under the initial context, much like a subdirectory under the root directory. The subcontext jdbc is reserved for logical names to be bound to DataSource objects, so jdbc is always the first part of a logical name for a data source.

To get a connection using a DataSource, simply create a JNDI Context, and supply the name of the DataSource object to its lookup() method. The lookup() method returns the DataSource object bound to that name, which can then be used to get a Connection:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/customerDB");
Connection con = ds.getConnection("myUserName", "myPassword");
```

Note The BasicDataSource object described repesents a vendor's implementation of the basic DataSource, which may have a vendor specific name. The Opta2000 driver, for example, calls it a TdsDataSource. The Connection object that the basic implementation of the DataSource.getConnection method returns is identical to a Connection object that the DriverManager.getConnection method returns.

Using a DataSource object is optional unless you are writing applications that include connection pooling or distributed transactions. In such cases, as discussed in the next few paragraphs, the use of a DataSource object with built-in connection pooling or distributed-transaction capabilities offers obvious advantages.

Connection Pooling

Creating and destroying resources frequently involves significant overhead and reduces the efficiency of an application. Resource pooling is a common way of minimizing the overhead of creating a new resource for an operation and discarding it as soon as the operation is terminated. When resource pooling is used, a resource that is no longer needed after a task is completed is not destroyed but is added to a resource pool instead, making it available when required for a subsequent operation.

Because establishing a connection is expensive, reusing connections in this way can improve performance dramatically by cutting down on the number of new connections that need to be created.

The JDBC 2.0 API introduces the ConnectionPoolDataSource interface. This object is a factory for PooledConnection objects. Connection objects that implement this interface are typically registered with a JNDI service.

To deploy a DataSource object to produce pooled connections, you must first deploy a ConnectionPoolDataSource object, setting its properties appropriately for the data source to which it produces connections:

```
ConnectionPoolDataSource cpds = new ConnectionPoolDataSource();
cpds.setServerName("Jupiter");
cpds.setDatabaseName("CUSTOMERS ");
cpds.setPortNumber(9001);
cpds.setDescription("Customer database");
```

The ConnectionPoolDataSource object is then registered with the JNDI naming service:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/pool/customerDB ", cpds);
```

Note The logical name associated with cpds has the subcontext pool added under the subcontext jdbc, which is similar to adding a subdirectory to another subdirectory in a hierarchical file system.

After the ConnectionPoolDataSource object has been registered with a JNDI naming service, deploy a DataSource object implemented to work with it.

Only two properties need to be set for the DataSource object, since the information required for connection has already been set in the ConnectionPoolDataSource object. These are as follows:

- dataSourceName
- description

The dataSourceName is then set to the logical name of the ConnectionPoolDataSource, as shown here:

```
PooledDataSource ds = new PooledDataSource();
ds.setDescription("Customer database pooled connection source");
ds.setDataSourceName("jdbc/pool/customerDB ");
Context ctx = new InitialContext();
ctx.bind("jdbc/customerDB", ds);
```

You have now deployed a DataSource object that an application can use to get pooled connections to the database.

Caution It is especially important to close pooled connections in a finally block, so that even if a method throws an exception, the connection will be closed and put back into the connection pool.

Another situation in which using a DataSource object is required is when you need to implement distributed transactions. In such cases, as discussed in the next few paragraphs, the use of a DataSource object with built-in distributed-transaction capabilities is the best solution.

Distributed Transactions

In a three-tier architecture, it is sometimes necessary to access data from more than one database server in a distributed transaction. This situation can be handled very effectively using a DataSource implemented to produce connections for distributed transactions in the middle tier.

As with connection pooling, two classes must be deployed:

- An XADataSource, which produces XAConnections supporting distributed transactions
- A DataSource object that is implemented to work with it

DataSources implemented to produce connections for distributed transactions are almost always implemented to produce connections that are pooled as well. The XAConnection interface, in fact, extends the PooledConnection interface.

The XADataSource object needs to be deployed first. This is done by creating an instance of XATransactionalDS and setting its properties, as shown here:

```
XATransactionalDS xads = new XATransactionalDS();
 xads.setServerName("Jupiter");
 xads.setDatabaseName("CUSTOMERS");
 xads.setPortNumber(9001);
xads.setDescription("Customer database");
```

Next, the XATransctionDS needs to be registered with the JNDI naming service, as shown here:

```
Context ctx = new InitialContext();
 ctx.bind("jdbc/xa/CustomerDB", xads);
```

Note The logical name associated with xads has the subcontext xa added under the subcontext jdbc, in the same way as the subcontext pool is added in the connection-pooling example.

Finally, the DataSource object is implemented to interact with xads, and other XADataSource objects are deployed:

```
 TransactionalDS ds = new TransactionalDS();
 ds.setDescription("Customers distributed transaction connections
source");
  ds.setDataSourceName("jdbc/xa/CustomerDB ");
 Context ctx = new InitialContext();
  ctx.bind("jdbc/CustomerDB", ds);
```

Now that instances of the TransactionalDS and XATransactionalDS classes have been deployed, an application can use the DataSource to get a connection to the CUSTOMERS database. This connection can then be used in distributed transactions. The following code to get the connection is very similar to the code to get a pooled connection:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/CustomerDB");
Connection  con = ds.getConnection("myUserName", "myPassword");
```

Distributed Transaction Management

The primary difference between using a regular connection and using a connection intended for distributed transactions is that all distributed transactions are committed or rolled back by a separate transaction manager in the middle tier. So the application should not do anything that can interfere with what the transaction manager is doing. This means that application code should never call these methods:

- Connection.commit
- Connection.rollback
- Connection.setAutoCommit(true)

A connection created for distributed transactions can, of course, also be used for nondistributed transactions, in which case these restrictions do not apply.

Note A Connection object that can be used for distributed transactions has its auto-commit mode turned off by default, unlike a regular connection for which the default is to have its auto-commit mode turned on.

Connection

A Connection object represents a connection with a database. A connection session includes the SQL statements that are executed and the results that are returned over that connection. A single application can have one or more connections with a single database, or it can have connections with many different databases.

Opening a connection

The standard way to establish a connection with a database is to call the method getConnection() on either a DataSource or a DriverManager. The Driver method connect uses this URL to establish the connection.

A user can bypass the JDBC management layer and call Driver methods directly. This can be useful in the rare case that two drivers can connect to a database and the user wants explicitly to select a particular driver. Usually, however, it is much easier to just let the DataSource class or the DriverManager class open a connection.

Database URLs

A URL (Uniform Resource Locator) is an identifier for locating a resource on the Internet. It can be thought of as an address. A JDBC URL is a flexible way of identifying a database so that the appropriate driver recognizes it and establishes a connection with it. JDBC URLs allow different drivers to use different schemes for naming databases. The odbc subprotocol, for example, lets the URL contain attribute values.

The standard syntax for JDBC URLs is shown here:

jdbc:<subprotocol>:<subname>

The three parts of a JDBC URL are broken down as follows:

- Jdbc — The protocol. The protocol in a JDBC URL is always jdbc.
- <subprotocol> — The name of the driver or connectivity mechanism, which may be supported by one or more drivers
- <subname> — A unique identifier for the database

For example, this is the URL to access the contacts database through the JDBC-ODBC bridge:

jdbc:odbc:contacts

The odbc subprotocol

The odbc subprotocol has the special feature of allowing any number of attribute values to be specified after the database name, as shown here:

jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]*

Attributes passed in this way may include user id and password, for example.

Having established a connection to the database, you are now in a position to execute a SQL statement. The next section discusses SQL statements.

SQL Statements

Once a connection is established, it is used to pass SQL statements to the database. Since there are no restrictions imposed on the kinds of SQL statements that may be sent to a DBMS using JDBC, the user has a great deal of flexibility to use database-specific statements or even non-SQL statements.

The JDBC core API provides these three classes for sending SQL statements to the database:

- Statement. A Statement object is used for sending simple SQL statements. Statements are created by the method createStatement().
- PreparedStatement. A PreparedStatement is a SQL statement that is precompiled and stored in a PreparedStatement object. This object can then be used to execute this statement multiple times.
- CallableStatement. CallableStatement is used to execute SQL stored procedures. CallableStatements are created by the method prepareCall().

Statement

A Statement object is used for executing a static SQL statement and obtaining the results it produces. Statement defines these three methods for executing SQL statements, which handle SQL commands returning different kinds of results:

- executeUpdate(String sql): Execute a SQL INSERT, UPDATE, or DELETE statement, which returns either a count of rows affected or zero.
- executeQuery(String sql): Execute a SQL statement that returns a single ResultSet.
- execute(String sql): Execute a SQL statement that may return multiple results.

The executeUpdate method is used for SQL commands such as INSERT, UPDATE, and DELETE, which return a count of rows affected rather than a ResultSet; or for DDL commands such as CREATE TABLE, which returns nothing, in which case the return value is zero.

The executeQuery method is used for SQL queries returning a single ResultSet. The ResultSet object is discussed in detail later in this chapter.

A significant difference introduced in JDBC 3.0 is the ability for a Statement to have more than one ResultSet open. If you are using a driver that does not implement JDBC 3.0, a Statement can have only one ResultSet open at a time; if you need to interleave data from different ResultSets, each must be generated by a different Statement; otherwise, any execute method closes the current ResultSet prior to executing.

The execute method is used to execute a SQL statement that may return multiple results. In some situations, a single SQL statement may return multiple ResultSets and/or update counts. The execute method returns boolean true if the SQL statement returns a ResultSet and false if the return is an update count. The Statement object defines the following supporting methods:

- getMoreResults
- getResultSet
- getUpdateCount

These methods let you navigate through multiple results. You can use getResultSet() or getUpdateCount() to retrieve the result and getMoreResults() to move to any subsequent results.

An example of the use of a simple statement is shown in Listing 4-1.

PreparedStatement

PreparedStatements are nothing more than statements that are precompiled. *Precompilation* means that these statements can be executed more efficiently than simple statements, particularly in situations where a Statement is executed repeatedly in a loop.

PreparedStatements can contain placeholders for variables known as IN parameters, which are set using setter methods. A typical setter method looks like this:

```
public void setObject(int parameterIndex, Object x) throws SQLException
```

An example of this is the following line, which sets integer parameter #1 equal to 2:

```
pstmt.setInt(1, 2);
```

Use of PreparedStatements is fairly intuitive, as shown in Listing 4-2.

Listing 4-2: Using a PreparedStatement

```
package java_databases.ch04;
import java.sql.*;

public class PreparedStmt{
  public static void main(String args[]){
     int qty;
     float cost;
     String name;
     String desc;
    String query = ("SELECT * FROM Stock WHERE Item_Number = ?";
     try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con =
                DriverManager.getConnection ("jdbc:odbc:Inventory");
       PreparedStatement pstmt = con.prepareStatement(query);
       pstmt.setInt(1, 2);
       ResultSet rs = pstmt.executeQuery();
       while (rs.next()) {
          name = rs.getString("Name");
          desc = rs.getString("Description");
          qty  = rs.getInt("Qty");
         cost = rs.getFloat("Cost");
          System.out.println(name+", "+desc+"\t: "+qty+"\t@ $"+cost);
       }
     }
     catch(ClassNotFoundException e){
        e.printStackTrace();
```

```
        }
        catch(SQLException  e){
            e.printStackTrace();
        }
    }
}
```

The JDBC PreparedStatement provides setter methods for all SQL data types. The setter methods used for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter.

You can explicitly convert an input parameter to a particular JDBC type by using the method setObject. This method can take a third argument, which specifies the target JDBC type. The driver converts the Java Object to the specified JDBC type before sending it to the database. If no JDBC type is given, the driver will simply map the Java Object to its default JDBC type. This allows an application to be generic, accepting input for a parameter at run time.

The methods setBytes and setString are capable of sending unlimited amounts of data. You can also handle large amounts of data by setting an IN parameter to a Java input stream.

JDBC provides these three methods for setting IN parameters to input streams:

- setBinaryStream(for streams containing uninterpreted bytes)
- setAsciiStream (for streams containing ASCII characters)
- setUnicodeStream (for streams containing Unicode characters)

When the statement is executed, the JDBC driver makes repeated calls to the input stream, reading its contents and sending them to the database as the actual parameter value.

The setNull method allows you to send a NULL value to the database as an IN parameter. You can also send a NULL to the database by passing a Java null value to a setXXX method.

CallableStatement

The CallableStatement object allows you to call a database stored procedure from a Java application. A CallableStatement object contains a call to a stored procedure; it does not contain the stored procedure itself, as the stored procedure is stored in the database. In the example of Listing 4-3, we create and use a stored procedure.

Listing 4-3: Creating and using a stored procedure

```
package java_databases.ch04;
```

```
import java.sql.*;

public class CallableStmt{
  public static void main(String args[]){
     int orderNo;
     String name;
    String storedProc = "create procedure SHOW_ORDERS_BY_STATE "+
                        "@State CHAR (2) "+
                        "as "+
                        "select c.Last_Name+', '+c.First_Name AS Name,"+
                        "o.Order_Number "+
                        "from CUSTOMERS c, ORDERS o "+
                        "where c.Customer_Number = o.Customer_Number "+
                        "AND c.State = @State "+
                        "order by c.Last_Name;";

     try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con = DriverManager.getConnection ("jdbc:odbc:Customers");
       Statement stmt = con.createStatement();
        stmt.executeUpdate(storedProc);
        CallableStatement cs = con.prepareCall("{call SHOW_ORDERS_BY_STATE(?)}");
        cs.setString(1,"NJ");
       ResultSet rs = cs.executeQuery();
       while (rs.next()) {
          name = rs.getString("Name");
          orderNo  = rs.getInt("Order_Number");
          System.out.println(name+": "+orderNo);
       }
     }
     catch(ClassNotFoundException e){
        e.printStackTrace();
     }
     catch(SQLException e){
        e.printStackTrace();
     }
  }
}
```

Notice the JDBC escape sequence used to call the stored procedure. This allows stored procedures to be called in a standard way for all database management systems.

CallableStatement extends PreparedStatement, so a CallableStatement object can take input parameters as a PreparedStatement object can. A CallableStatement can also take output parameters or parameters that are for both input and output.

This escape syntax used to call a stored procedure has two forms: one that includes a result parameter and one that does not. Here's an example:

```
{?= call <procedure-name>[<arg1>,<arg2>, ...]}
{call <procedure-name>[<arg1>,<arg2>, ...]}
```

Question marks (?) serve as placeholders for parameters defined in the stored procedure using the @Name convention as shown in the example. IN parameter values are set using the set methods inherited from PreparedStatement. If used, the result parameter must be registered as an OUT parameter using the registerOutParameter() method before one of the execute methods is called. Consider this example:

```
cstmt.registerOutParameter(1, java.sql.Types.VARCHAR);
```

OUT parameter values can be retrieved after execution using get methods appropriate to the data types of the values.

Listing 4-4 is an example of a simple stored procedure that checks a user name and password against the database, returning the String "PASS" if a match is found or "FAIL" otherwise:

Listing 4-4: Stored procedure with input and output parameters

```
CREATE PROCEDURE CHECK_USER_NAME
 @UserName varchar(30),
 @Password varchar(20),
 @PassFail varchar(20) OUTPUT
As
IF EXISTS(Select * From Customers
Where UserName = @UserName
 And
Password = @Password)
 SELECT @PassFail = "PASS"
else
 SELECT @PassFail = "FAIL"
```

Because of limitations imposed by some relational database management systems, it is recommended that, for maximum portability, all of the results generated by the execution of a CallableStatement object be retrieved before OUT parameters are retrieved.

In cases where a CallableStatement object returns multiple ResultSet objects, all of the results should be retrieved using the method getMoreResults before OUT parameters are retrieved.

Listing 4-5 illustrates how you would retrieve an output parameter from a stored procedure in a JDBC application.

Note Stored procedures can contain more than one SQL statement, in which case they produce multiple results, in which case the execute method should be used.

Listing 4-5: Getting an output parameter from a stored procedure

```
package java_databases.ch04;
import java.sql.*;

public class CheckPassword{
  public static void main(String args[]){
     try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con =
         DriverManager.getConnection ("jdbc:odbc:Customers","user","pwd");

       CallableStatement cs =
          con.prepareCall("{call CHECK_USER_NAME(?,?,?)}");
        cs.setString(1,"Corleone");
        cs.setString(2,"Vito");
       cs.registerOutParameter(3, java.sql.Types.VARCHAR);
        cs.executeUpdate();
        System.out.println(cs.getString(3));
        con.close();
     }
     catch(ClassNotFoundException e){
        e.printStackTrace();
     }
     catch(SQLException e){
        e.printStackTrace();
     }
  }
}
```

There are many situations in which it is important that a group of SQL statements be executed in its entirety. A classic example is a series of statements that debit one bank account and credit another. In this situation, it is highly undesirable from the bank's viewpoint to credit the second account if, for some reason, the system fails to

debit the first account. The issue of managing a series of SQL statements as a single transaction is discussed in the next section.

Transactions

The capability to group SQL statements for execution as a single entity is provided through SQL's transaction mechanism. A *transaction* consists of one or more statements that are executed, completed, and either committed or rolled back as a group. When the method commit or rollback is called, the current transaction ends, and another one begins.

In the context of database transactions, the term commit means that the change is made permanently in the database, and the term rollback means that no change is made in the database.

A new JDBC connection is usually in auto-commit mode by default, meaning that when a statement is completed, the method commit is called on that statement automatically. The commit occurs when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a ResultSet, the statement completes when the last row of the ResultSet has been retrieved or the ResultSet has been closed. In advanced cases, a single statement may return multiple results as well as output parameter values. Here, the commit occurs when all results and output parameter values have been retrieved.

Auto-commit mode is controlled by this method:

```
public void setAutoCommit(boolean autoCommit) throws SQLException
```

If auto-commit mode has been disabled, a transaction will not terminate until either the commit method or the rollback method is called explicitly, so it will include all the statements that have been executed since the last invocation of the commit or rollback method. In this case, all the statements in the transaction are committed or rolled back as a group.

When a SQL statement makes changes to a database, the commit method makes those changes permanent, and it releases any locks the transaction holds. The rollback method, on the other hand, simply discards those changes.

Clearly, in a situation such as our preceding bank-transfer example, we can prevent one step in the funds transfer if the other fails by disabling auto-commit and grouping both updates into one transaction. If both updates are successful, the commit method is called, making the effects of both updates permanent; if one fails or both fail, the rollback method is called, restoring the account balances that existed before the updates were executed.

Most JDBC drivers support transactions. In fact, a JDBC-compliant driver must support transactions. DatabaseMetaData supplies information describing the level of transaction support a DBMS provides.

Transaction Isolation Levels

The basic concept of transaction management allows us to manage simple conflicts arising from events such as a failure to complete a linked series of SQL commands. However, other types of conflict can occur that require additional levels of management.

For example, consider the case of a multiuser application, where one transaction has initiated a transfer between two accounts but has not yet committed it, when a second transaction attempts to access one of the accounts in question. If the first transaction is rolled back, the value the second transaction reads will be invalid. Depending on the application, this situation may be acceptable, but in many financial applications, it would probably be quite unacceptable.

JDBC defines five levels of transaction isolation that provide different levels of conflict management. The lowest level specifies that transactions are not supported at all, and the remainder map to the four isolation levels that SQL-92 defines. These are as follows:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

The highest specifies that while one transaction is operating on a database, no other transactions may make any changes to the data that transaction reads.

SQL-92 isolations levels are stated in terms of three prohibited operation sequences, called *phenomena*. These are as follows:

- **Dirty Read.** This occurs if one transaction can see the results of the actions of another transaction before it commits.
- **Non-Repeatable Read (also called Fuzzy Read).** This occurs if the results of one transaction can be modified or deleted by another transaction before it commits.
- **Phantom Read**. This occurs if the results of a query in one transaction can be changed by another transaction before it commits.

The SQL-92 isolation levels are defined in terms of which of these phenomena can occur for a given isolation level, as shown in Table 4-1.

Table 4-1: SQL-92 Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	YES	YES	YES
Read Committed	NO	YES	YES
Repeatable Read	NO	NO	YES
Serializable	NO	NO	NO

From Table 4-1, you can see that if Read Committed is supported, you will never experience a DIRTY READ, but you might experience a NON-REPEATABLE READ or a PHANTOM READ. Similarly, if Repeatable Read is supported, you will never experience a DIRTY READ or a NON-REPEATABLE READ, but you might experience a PHANTOM READ.

Table 4-1 lists isolation levels in terms of the level of isolation afforded, in that Read Uncommitted is less restrictive than Read Committed. Typically, the higher the level of isolation, the greater the locking overhead and the lower the concurrency between users, so the slower the application executes. This means a trade off occurs between performance and data consistency when making a decision about what isolation level to use.

The current level of isolation can be queried using this method:

```
public int getTransactionIsolation()
```

This method returns the following isolation level codes:

- TRANSACTION_NONE: Transactions are not supported.
- TRANSACTION_READ_COMMITTED: Dirty reads are prevented; nonrepeatable reads and phantom reads can occur.
- TRANSACTION_READ_UNCOMMITTED: Dirty reads, nonrepeatable reads, and phantom reads can occur.
- TRANSACTION_REPEATABLE_READ: Dirty reads and nonrepeatable reads are prevented; phantom reads can occur.
- TRANSACTION_SERIALIZABLE: Dirty reads, nonrepeatable reads, and phantom reads are prevented.

Control over the isolation level of a connection is provided by this method:

```
con.setTransactionIsolation(TRANSACTION_ISOLATION_LEVEL_XXX);
```

For example, you can instruct the DBMS to allow a value to be read before it has been committed with the following code:

```
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

When you create a new Connection object, its transaction isolation level is usually set to the default for the underlying database. You can call the method setIsolationLevel to change the transaction isolation level, and the new level is in effect for the rest of the connection session.

You can also change the transaction isolation level for just one transaction by setting it before the transaction begins and resetting it after the transaction ends.

Caution Changing the transaction isolation level during a transaction is not usually recommended because it triggers an immediate call to the commit method, causing any changes up to that point to be made permanent.

Transaction Savepoints

Transaction savepoints are JDBC 3.0 enhancements that offer finer control over transaction commit and rollback. During a transaction, a named savepoint may be inserted between operations to act as a marker, so that the transaction may be rolled back to that marker, leaving all of the operations before the marker in effect.

The following example shows a Savepoint being set after the first update, and the transaction being rolled back to that Savepoint, removing two subsequent updates. (The arguments update1, update2 and update3 represent SQL commands.)

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();

stmt.executeUpdate(update1);

Savepoint savePoint1 = con.setSavepoint("SavePoint1");

stmt.executeUpdate(update2);
stmt.executeUpdate(update3);

con.rollback(savePoint1);
con.commit();
```

Multithreading

The JDBC specifications require that operations on all the java.sql objects be thread safe. This means that they must be able to handle a situation where several threads call the same object simultaneously. Some drivers may provide this full concurrency, and others may execute one statement and wait until it completes before sending the next.

Note One specific use of multithreading is to cancel a long-running statement. This is done by using one thread to execute the statement and another to cancel it with its Statement.cancel() method.

It is sometimes more efficient to submit a set of update statements to the database for processing as a batch. Support for batch updates is part of the JDBC 2.0 API, as discussed in the next section.

Batch Updates

A *batch update* is a set of update statements submitted to the database for processing as a batch. This can be more efficient than sending update statements separately. Support for batch updates is part of the JDBC 2.0 API.

Under the JDBC 1.0 API, updates must be submitted to the database individually, so that even though multiple update statements can be part of the same transaction, they are processed individually.

Under the JDBC 2.0 API, the Statement, PreparedStatement and CallableStatement support a batch list, which may contain statements for updating, inserting, or deleting a row. The batch list may also contain DDL statements such as CREATE TABLE and DROP TABLE.

Note Only statements that produce an update count can be used in a batch update. Statements that return a ResultSet object, such as a SELECT statement, cannot be used in a batch.

Commands used to manage batch updates include the following:

- AddBatch (add SQL commands to the batch list)
- clearBatch (empty the batch list)
- executeBatch (execute all statements in the list as a batch)

Note The batch list associated with a Statement is initially empty.

The code for inserting new rows as a batch might look like this:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO CUSTOMERS VALUES('Homer', 'Simpson')");
stmt.addBatch("INSERT INTO CUSTOMERS VALUES('Bart', 'Simpson')");
stmt.addBatch("INSERT INTO CUSTOMERS VALUES('Marge', 'Simpson')");

int [] updateCounts = stmt.executeBatch();
con.commit();
```

```
con.setAutoCommit(true);
```

The DBMS executes the commands in the batch list in the order in which they are added, returning an array of integer update counts.

The array of update counts represents the results of successfully executed commands in the batch list in the order in which they are executed.

You will get a BatchUpdateException if any of the SQL statements in the batch do not execute successfully.

Since the array of update counts represents the results of successfully executed commands, you can easily identify the problem command from the length of the returned array.

Note You should always disable auto-commit mode during a batch update so that, if any errors occur, they can be handled properly. As shown in the example, you need to issue a specific commit() command to commit the updates.

BatchUpdateException extends SQLException, adding an array of update counts similar to the array the executeBatch method returns. You can retrieve this array using the getUpdateCounts() method, as shown here:

```
try {
    //...
} catch(BatchUpdateException b) {
    System.err.print("Update counts:  ");
   int [] updateCounts = b.getUpdateCounts();
  for (int i = 0; i < updateCounts.length; i++) {
       System.err.println(updateCounts[i]);
    }
}
```

Since the update counts are in the same order as the commands, you can tell which commands in the batch have executed successfully.

Cross-Reference Exception handling, SQLExceptions, and SQLWarnings are discussed in more detail at the end of this chapter.

The results returned by a SQL query are held in a java.sql.ResultSet. This objects is discussed in the next section.

ResultSets

A ResultSet is the data a SQL Query returns, consisting of all the rows that satisfy the conditions of that query arranged in rows accessible through the ResultSet's methods.

ResultSets are arranged as a table, with the column headings and values returned in the order specified in the statement, satisfying the conditions of the query. For example, if this is your query:

```
SELECT Name,Description,Qty,Cost FROM Stock
```

Your result set might look like Table 4-2.

Table 4-2: Organization of a ResultSet

Name	Description	Qty	Cost
Steiner	10 x 50 Binoculars	10	799.95
Steiner	8 x 30 Binoculars	30	299.95
PYGMY-2	Night Vision Monocular	20	199.95

A ResultSet maintains a cursor that points to the row of data accessible through the getter methods of the ResultSet. Each time the ResultSet.next() method is called, the cursor moves down one row.

Initially, the cursor is positioned before the first row, so you need to call next() to set the cursor on the first row, making it the current row. Since next()returns a boolean true if a valid data row is available, this design approach makes for a clean while loop for accessing row data, as shown in Listing 4-6.

Listing 4-6: Retrieving a ResultSet

```
package java_databases.ch04;

import java.sql.*;

public class PrintResultSet{
  public static void main(String args[]){
    String query = "SELECT Name,Description,Qty,Cost FROM Stock";
    PrintResultSet p = new PrintResultSet(query);
  }
  public PrintResultSet(String query){
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection ("jdbc:odbc:Inventory");
      Statement stmt = con.createStatement();
```

```
        ResultSet rs = stmt.executeQuery(query);

         System.out.println("Name\tDescription\tQty\tCost");

        while (rs.next()) {
            System.out.print(rs.getString("Name")+"\t");
            System.out.print(rs.getString("Description")+"\t");
            System.out.print(rs.getInt("Qty")+"\t");
            System.out.println(rs.getFloat("Cost"));
        }
      }
      catch(ClassNotFoundException e){
         e.printStackTrace();
      }
      catch(SQLException e){
         e.printStackTrace();
      }
   }
}
```

ResultSet rows are retrieved in sequence from the top row down as the cursor moves down one row with each successive call to next().

A cursor remains valid until the ResultSet object or its parent Statement object is closed.

Note Column names used as input to ResultSet getter methods are case insensitive.

Data is retrieved from a ResultSet using getter methods that reference the column containing the data. The ResultSet getter methods provide for data-type specific retrieval of column values from the current row. Within each row, column values may be retrieved in any order.

The getter methods the ResultSet object provides are listed in Table 4-3. Each getter method has two variants: one that references the column by name and one by column number. For brevity, only one variant of each is listed in Table 4-3.

Caution Columns are numbered from left to right, starting with column 1, *not* column 0.

Table 4-3: ResultSet getter Methods

Data Type	Method
BigDecimal	getBigDecimal(String columnName, int scale)

Table 4-3: ResultSet getter Methods

Data Type	Method
boolean	getBoolean(String columnName)
byte	getByte(String columnName)
byte[]	getBytes(String columnName)
double	getDouble(String columnName)
float	getFloat(String columnName)
int	getInt(String columnName)
java.io.InputStream	getAsciiStream(String columnName)
java.io.InputStream	getUnicodeStream(String columnName)
java.io.InputStream	getBinaryStream(String columnName)
java.sql.Date	getDate(String columnName)
java.sql.Time	getTime(String columnName)
java.sql.Timestamp	getTimestamp(String columnName)
long	getLong(String columnName)
Object	getObject(String columnName)
short	getShort(String columnName)
String	getString(String columnName)

Data can be retrieved using either the column name or the column number. For example, the previous example uses column names. However, you can get the same results using column numbers, assuming you know what they are. Here's an example:

```
ResultSet rs = stmt.executeQuery("SELECT First_Name, Last_Name FROM Customers");
while (rs.next()){
   System.out.println(rs.getString(2)+•, •+rs.getString(1));
}
```

The option of using column names is provided so that if you specify column names in a query, you can use the same names as the arguments to getter methods.

A problem that can arise using column names is that if you do a join on two tables, it is possible for a SQL query to return a result set that has more than one column with the same name. If you then use the column name as the parameter to a getter method, it will return the value of the first matching column name.

Some database management systems, such as Oracle, get around this by letting you use fully qualified columns names of the form table_name.column_name to resolve this situation; but others, such as MSAccess, do not, so if there are multiple columns with the same name, using the column index is more portable. It may also be slightly more efficient to use column numbers.

Caution Column values should be read only once. Subsequent reads are not guaranteed to return valid data.

Scrollable ResultSets

One of the features added in the JDBC 2.0 API is the ScrollableResultSet, which supports the ability to move a result set's cursor in either direction. In addition to methods that move the cursor backwards and forwards, there are methods for getting the cursor position and moving the cursor to a particular row.

Creating a Scrollable ResultSet

The type of ResultSet a java.sql.Statement object returns is defined when the Statement is created by the Connection.createStatement method. These are the two forms of the Connection.createStatement method:

public Statement **createStatement**() throws SQLException
public Statement **createStatement**(int rsType, int rsConcurrency) throws SQLException

The first of these methods is discussed in the examples earlier in this chapter. The second variant allows the user to create scrollable and updateable ResultSets.

The first argument, rsType, must be one of the three following constants added to the ResultSet interface to indicate the type of a ResultSet object:

- TYPE_FORWARD_ONLY
- TYPE_SCROLL_INSENSITIVE
- TYPE_SCROLL_SENSITIVE

Specifying the constant TYPE_FORWARD_ONLY creates a nonscrollable result set (that is, one in which the cursor moves forward only). If you also specify CONCUR_READ_ONLY for the second argument, you will get the default ResultSet identical to the ResultSet created with the no-argument variant.

To get a scrollable ResultSet object, you must specify either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE. The difference between these two types of scrollable result sets is that a result set defined using TYPE_SCROLL_INSENSITIVE does not reflect changes made while it is still open, and one that is TYPE_SCROLL_SENSITIVE does. Of course, you can always see

changes, regardless of the type of result set, by closing the result set and reopening it.

The second argument must be one of the two ResultSet constants for specifying whether a result set is read-only or updateable: CONCUR_READ_ONLY or CONCUR_UPDATABLE. If you specify a result-set type, you must also specify whether the result set is read-only or updateable.

The ResultSet.getType() method checks whether the ResultSet object is scrollable:

```
if(rs.getType()==ResultSet.TYPE_FORWARD_ONLY)
  System.out.println("FORWARD_ONLY");
else
 System.out.println("SCROLLABLE");
```

Cursor Control

Once you have a scrollable ResultSet object, you can use it to move the cursor around in the result set. As with a ResultSet that is not scrollable, the cursor is initially positioned before the first row.

In addition to the ResultSet.next() method, which is used to move the cursor forward, one row at a time, scrollable ResultSets support the method ResultSet.previous(), which moves the cursor back one row.

Both methods return false when the cursor goes beyond the result set (to the position after the last row or before the first row), which makes it possible to use them in a while loop. Listing 4-7 replaces the default ResultSet of Listing 4-6 with a scrollable ResultSet navigated using both next() and previous().

Listing 4-7: Scrollable ResultSet

```
public void printResultSet(String query){
   try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     Connection con = DriverManager.getConnection ("jdbc:odbc:Inventory");
      Statement stmt = con.createStatement(
                              ResultSet.TYPE_SCROLL_INSENSITIVE,
                               ResultSet.CONCUR_READ_ONLY);
      ResultSet rs = stmt.executeQuery(query);
       ResultSetMetaData md    = rs.getMetaData();

      int nColumns = md.getColumnCount();
       for(int i=1;i<=nColumns;i++){
          System.out.print(md.getColumnLabel(i)+((i==nColumns)?"\n":"\t"));
```

```
            if(i==2)System.out.print("\t");
        }
        while (rs.next()) {
            for(int i=1;i<=nColumns;i++){
                System.out.print(rs.getString(i)+((i==nColumns)?"\n":"\t"));
            }
        }
        while (rs.previous()) {
            for(int i=1;i<=nColumns;i++){
                System.out.print(rs.getString(i)+((i==nColumns)?"\n":"\t"));
            }
        }
    }
    catch(ClassNotFoundException e){
        e.printStackTrace();
    }
    catch(SQLException e){
        e.printStackTrace();
    }
}
```

As you can see from the screen shot of Figure 4-3, the example first prints three rows in first-to-last order and prints them again in reverse:

```
Name    Description             Qty     Cost
Steiner 10 x 50 Binoculars      10      799.9500
Steiner 8 x 30  Binoculars      30      299.9500
PYGMY-2 Night Vision Monocular  20      199.9500
PYGMY-2 Night Vision Monocular  20      199.9500
Steiner 8 x 30  Binoculars      30      299.9500
Steiner 10 x 50 Binoculars      10      799.9500
press any key to exit...
```

Figure 4-3: Printing rows from a scrollable result set

Moving the Cursor to a Designated Row

In addition to using the next() and previous() methods to scroll forward and backward, you can move the cursor to a particular row by using the following methods:

- first()
- last()
- beforeFirst()
- afterLast()
- absolute(int rowNumber)
- relative(int rowNumber)

The effect of the first four of these is apparent from the method names.

The method absolute(int rowNumber) moves the cursor to the row number indicated in the argument. If the number is positive, the cursor moves to the given row number from the beginning. If the number is negative, the cursor moves to the given row number from the end.

> **Note** Row numbers count from 1, so calling absolute(1)puts the cursor on the first row, and calling absolute(-1) puts the cursor on the last row.

The method relative(int rowNumber) lets you specify how many rows to move from the current row and in which direction to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows.

Getting the Cursor Position

In addition to positioning the cursor, you can get its position by using one of these methods:

- isFirst()
- isLast()
- isBeforeFirst()
- isAfterLast()
- getRow()

Again, the behavior of these methods is apparent from their names.

> **Note** The method isAfterLast() returns false when the cursor is not after the last row and when the result set is empty, so a returned value of false from the method isAfterLast() cannot be used to indicate that data is available.

Updatable ResultSets

An UpdatableResultSet is, as the name suggests, updatable. You can make updates to the values in the ResultSet itself, and these changes are reflected in the database.

To create an UpdatableResultSet object, you need to call the createStatement method with the ResultSet constant CONCUR_UPDATABLE as the second argument. The Statement object created produces an updatable ResultSet object when it executes a query.

> **Note** An updatable ResultSet object does not necessarily have to be scrollable.

Once you have an UpdatableResultSet object, you can insert a new row, delete an existing row, or modify one or more column values.

Caution Specifying that a result set be updatable does not guarantee that the result set you get will actually be updatable. Drivers that do not support updatable result sets will return one that is read-only. In addition, to get an updatable result set, the query must generally specify the primary key as one of the columns selected, and it should select columns from only one table.

Since requesting an UpdatableResultSet does not guarantee that you will actually get one, depending on the driver in use, you should check whether the ResultSet is updatable using ResultSet.getConcurrency(). Listing 4-8 illustrates opening a scrollable updatable ResultSet and using getConcurrency to ensure that it is updatable.

Listing 4-8: Opening an updatable ResultSet

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection ("jdbc:odbc: Contacts");
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData md = rs.getMetaData();

if(rs.getConcurrency()==ResultSet.CONCUR_UPDATABLE)
   System.out.println("UPDATABLE");
else
  System.out.println("READ_ONLY");

int nColumns = md.getColumnCount();
for(int i=1;i<=nColumns;i++){
   System.out.print(md.getColumnLabel(i)+((i==nColumns)?"\n":"\t"));
}
while (rs.next()) {
   rs.updateString("Street", "123 Main");
   rs.updateRow();
  for(int i=1;i<=nColumns;i++){
      System.out.print(rs.getString(i)+((i==nColumns)?"\n":"\t"));
   }
}
```

If the driver does not support the definition of UpdatableResultSet, the Statement object may throw a SQL "Optional feature not implemented" exception.

Updating a ResultSet

To appreciate the simplicity of using UpdatableResultSet instead of SQL UPDATES, it is worth looking first at what is involved in using Statement.executeUpdate() to change a customer address. The code to make this change looks like this:

```
stmt.executeUpdate(
    "UPDATE Customers SET Street = '123 Main Street' +
    "WHERE First_Name = 'Vito' AND Last_Name = 'Corleone'");
```

This is simple enough when you know how to identify the record to be updated, but consider how much more complicated it would be if your application were displaying the ResultSet in a JTable. Unless you go to considerable trouble to keep track of the current record, it is quite difficult to identify to the RDBMS which record to update.

Using an UpdatableResultSet simplifies the situation considerably. All you need to do is set the cursor to the desired row and change the column value using a data-type-specific update method. Here's an example:

```
rs.updateString("Street", "123 Main");
```

Since updates made to an UpdatableResultSet always affect the current row, you must make sure you have moved the cursor to the correct row prior to making an update.

Most of the ResultSet.update methods take two parameters: the column to update and the new value to put in that column. As with the getter methods, the column may be specified using either the column name or the column number.

Table 4-4 summarizes the update methods for the UpdatableResultSet, showing only the variant using column name as the specifier for reasons of space.

Note A special update method, updateNull(), is used for setting column values to NULL.

Table 4-4: ResultSet Update Methods

Data Type	Method
BigDecimal	updateBigDecimal(String columnName, BigDecimal x)
boolean	updateBoolean(String columnName, boolean x)
byte	updateByte(String columnName, byte x)
byte[]	updateBytes(String columnName, byte[] x)
double	updateDouble(String columnName, double x)
float	updateFloat(String columnName, float x)
int	updateInt(String columnName, int x)

Table 4-4: ResultSet Update Methods

Data Type	Method
java.io.InputStream	updateAsciiStream(String columnName, InputStream x, int length)
java.io.InputStream	updateUnicodeStream(String columnName, InputStream x, int length)
java.io.InputStream	updateBinaryStream(String columnName, InputStream x, int length)
java.sql.Date	updateDate(String columnName, Date x)
java.sql.Time	updateTime(String columnName, Time x)
java.sql.Timestamp	updateTimestamp(String columnName, Timestamp x)
long	updateLong(String columnName, long x)
Object	updateObject(String columnName, Object x)
Object	updateObject(String columnName, Object x, int scale)
short	updateShort(String columnName, short x)
String	updateString(String columnName, String x)
NULL	updateNull(String columnName)

Note that after updating a column value in the ResultSet, you must call the ResultSet's updateRow() method to make a permanent change in the database before moving the cursor, since changes made using the update methods do not take effect until updateRow() is called.

Caution If you move the cursor to another row before calling updateRow(), the updates will be lost, and the row will revert to its previous column values.

You can specifically cancel updates any time before calling updateRow() by calling the cancelRowUpdates() method. Once you have called updateRow(), however, the cancelRowUpdates() method no longer works.

Inserting a New Row

In addition to supporting updates, an UpdatableResultSet supports the insertion and deletion of entire rows. The ResultSet object has a row called the *insert row*, which is, in effect, a dedicated row buffer in which you can build a new row.

The new row is created in a manner very similar to the row updates discussed earlier. Simply follow these steps:

1. Move the cursor to the insert row, which is done by calling the method moveToInsertRow().

2. Set a new value for each column in the row by using the appropriate update method.
3. Call the method insertRow() to insert the new row into the result set and, simultaneously, into the database.

Listing 4-9 demonstrates the use of the UpdatableResultSet to insert a new row into a database.

Listing 4-9: Using UpdatableResultSet to insert a new row

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection ("jdbc:odbc:Contacts");
Statement stmt = con.createStatement(
                        ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(query);
rs.moveToInsertRow();

rs.updateInt("Contact_ID", 150);
rs.updateString("First_Name", "Nigel");
rs.updateString("Last_Name",    "Thornebury");

rs.insertRow();
```


If you insert a row without supplying a value for every column in the row, the default value for the column will be used if there is one. Otherwise, if the column accepts SQL NULL values, a NULL will be inserted. Failing either of those, a SQLException will be thrown.

You will also get a SQLException if a required table column is missing in the ResultSet you use to insert the row, so the query used to get the ResultSet object should generally select all columns, though you will probably want to use a WHERE clause to limit the number of rows returned by your SELECT statement.

Caution If you move the cursor from the insert row before calling the method insertRow(), you will lose all of the values you have added to the insert row.

To move the cursor from the insert row back to the result set, you can use any of the methods that put the cursor on a specific row: first, last, beforeFirst, afterLast, and absolute. You can also use the methods previous and relative because the result set maintains a record of the current row while accessing the insert row.

In addition, you can use a special method: moveToCurrentRow(), which can be called only when the cursor is on the insert row. This method moves the cursor from the insert row back to the row that was previously the current row.

Deleting a Row

Deleting a row in an UpdatableResultSet is very simple. All you have to do is move the cursor to the row you want to delete and call the method deleteRow().

The example in the following code snippet shows how to delete the third row in a result set by getting the ResultSet object, moving the cursor to the third row, and using the deleteRow() method:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection ("jdbc:odbc:Contacts");
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(query);
rs.absolute(3);
rs.deleteRow();
```

Caution Be aware that different JDBC drivers handle deletions in different ways. Some remove a deleted row so that it is no longer visible in a result set, and others insert a blank row where the deleted row used to be.

When you make a change to a ResultSet, the change may not necessarily be visible. The next section explains the reasons.

Seeing Changes in ResultSets

Changes made to a ResultSet are not necessarily visible, either to the ResultSet itself or to other open transactions. In this context, the terms *visible* and *not visible* have the following meanings:

- An update is visible if the updated value can be retrieved by calling the appropriate getter method after making an update.
- An update is not visible if the getter method still returns the initial column value.

Similarly, an inserted row is visible if it appears in the ResultSet after calling insertRow(). Deletions are visible if deleted rows are either removed from the result set or if deleted rows leave a *hole* in the result set.

There are a number of factors affecting the visibility of changes, including the following:

- JDBC driver implementation
- Transaction isolation level in effect
- Result-set type

An application can determine if the changes a result set makes are visible to the result set itself by calling these DatabaseMetaData methods:

- ownUpdatesAreVisible(int ResultSet.TYPE_XXX)
- ownDeletesAreVisible(int ResultSet.TYPE_XXX)
- ownInsertsAreVisible(int ResultSet.TYPE_XXX)

The DatabaseMetaData interface also provides the following methods that allow an application to determine whether a JDBC driver can detect changes for a particular result-set type:

- insertsAreDetected(ResultSet.TYPE_XXX)
- deletesAreDetected(ResultSet.TYPE_XXX)
- updatesAreDetected(ResultSet.TYPE_XXX)

If these methods return true, the following methods can be used to detect changes to a ResultSet:

- wasInserted()
- wasDeleted()
- wasUpdated()

Remember that if you modify data in a ResultSet object, the change will always be visible if you close the ResultSet and reopen it by executing the same query again after the changes have been made.

Another way to get the most recent data is to use the method refreshRow(), which gets the latest values for a row straight from the database. This is done by positioning the cursor to the desired row and calling refreshRow(), as shown here:

```
rs.absolute(3);
rs.refreshRow();
```

Note The result set should be TYPE_SCROLL_SENSITIVE; if you use the method refreshRow() with a ResultSet object that is TYPE_SCROLL_INSENSITIVE, refreshRow() does nothing.

Another way to get data from a database is to use a RowSet object. RowSets add JavaBeans support to the functionality of the ResultSet, as explained in the next section.

RowSets

A RowSet is an object that contains a set of rows from a result set or some other source of tabular data, like a file or spreadsheet. RowSet is an extension of ResultSet,

with the added feature that it adds JavaBeans support to the JDBC API. Similarly, the RowSetMetaData interface extends the ResultSetMetaData interface.

Being JavaBeans, RowSets follow the JavaBeans model for setting and getting properties and for event notification, so they are easy to combine with other components in an application.

RowSets make it easy to send tabular data over a network. They can also be used as a wrapper, providing scrollable result sets or updatable result sets when the underlying JDBC driver does not support them.

There are two main types of RowSets: connected and disconnected.

- A connected RowSet, like a ResultSet, maintains a connection to a data source for as long as the RowSet is in use.
- A disconnected RowSet gets a connection to a data source to load data or to propagate changes back to the data source, but most of the time it does not have a connection open.

While it is disconnected, a RowSet does not need a JDBC driver or the full JDBC API, so its footprint is very small.

Because it is not continually connected to its data source, a disconnected RowSet stores its data in memory. It maintains MetaData about the columns it contains and information about its internal state. It also includes methods for making connections, executing commands, and reading and writing data to and from the data source.

Implementations of RowSets include the following:

- JDBCRowSet — A connected RowSet that serves mainly as a thin wrapper around a ResultSet object to make a JDBC driver look like a JavaBeans component
- CachedRowSet — A disconnected RowSet that caches its data in memory
- WebRowSet — A connected RowSet that uses the HTTP protocol internally to talk to a Java servlet that provides data access

Creating a Rowset and Setting Properties

Since RowSets are JavaBeans, they contain setter and getter methods for retrieving and setting properties.

These methods include the following:

- setCommand — The SQL command to be executed
- setConcurrency — Read only or updatable
- setType — Scrollable or foward only
- setDataSourceName — Used with DataSource access

- setUrl — used with DriverManager access
- setUsername
- setPassword
- setTransactionIsolation

You need only set those properties that are needed for your particular use of a RowSet.

The following lines of code make the CachedRowSet object crset scrollable and updatable.

```
CachedRowSet crset = new CachedRowSet();
 crset.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
 crset.setConcurrency(ResultSet.CONCUR_UPDATABLE);
crset.setCommand("SELECT * FROM Customers");
 crset.setDataSourceName("jdbc/customers");
 crset.setUsername("myName");
 crset.setPassword("myPwd");
 crset.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
 crset.addRowSetListener(listener);
```

If the DriverManager were being used to make a connection, you would set the properties for a JDBC URL, a user name, and a password. The preferred means of getting a connection is to use a DataSource object with the owner's user name and the owner's password.

Now that the CachedRowSet has been created and initialized, all that remains is to call the execute() method; the RowSet uses the information in its properties to make a connection and execute the query. The data in the RowSet can then be accessed and updated.

Rowset Events

A RowSetEvent is generated when something important happens in a RowSet, such as a change in a column value. Being JavaBeans, RowSets can use the Java event model to notify listeners when the RowSet is changed.

These are the RowSetListener methods:

- rowChanged (Called when the RowSet is changed)
- rowSetChanged(Called when a RowSet is inserted, updated, or deleted)
- cursorMoved (Called when a RowSet's cursor is moved)))

In addition to obtaining the data stored in the database, it is frequently ver useful to be able to obtain data about the database and its contents. This capability is supported by the MetaData objects discussed in the next section.

MetaData

MetaData is information about the database or its contents made available by the JDBC API.

These are the main types of MetaData accessible from JDBC:

- DatabaseMetaData
- ResultSetMetaData
- ParameterMetaData

DatabaseMetaData

The DatabaseMetaData interface provides information about the underlying database as a whole. The interface defines over 150 different methods providing the following types of information about the database:

- General information about the data source
- Data-source limits
- Levels of transaction support
- Feature support
- Information about the SQL objects that the source contains

Many of the DatabaseMetaData methods return information in ResultSets, allowing you to use ResultSet methods such as getString and getInt to retrieve this information. If a given form of MetaData is not available, these methods should throw a SQLException.

Some of the DatabaseMetaData methods take arguments that are String patterns conforming to the normal wild-card rules for SQL Strings. For pattern String arguments, "%" means match any substring of zero or more characters, and "_" means match any one character. If a search pattern argument is set to null, that argument's criteria will be ignored in the search.

If a driver does not support a MetaData method, a SQLException will normally be thrown. In the case of methods that return a ResultSet, either a ResultSet (which may be empty) is returned or a SQLException is thrown.

A DatabaseMetaData object is created using the Connection.getMetaData() method. It can then be used to get information about the database, as in the following example, which gets the names of the tables in the database:

```
Connection con = DriverManager.getConnection ("jdbc:odbc:Customers");
DatabaseMetaData dbmd = con.getMetaData();
ResultSet rs = dbmd.getTables(null,null,"%",new String[]{"TABLE"});
```

General information about the underlying database is accessible from the DatabaseMetaData interface by using methods such as these:

- getURL()
- getUserName()
- getDatabaseProductName()
- getSQLKeywords()
- nullsAreSortedHigh() and nullsAreSortedLow()

Useful methods for retrieving information about supported functionality include the following:

- supportsBatchUpdates()
- supportsStoredProcedures()
- supportsFullOuterJoins()
- supportsPositionedDelete()

These methods are provided to determine limits the database imposes:

- getMaxRowSize()
- getMaxStatementLength()
- getMaxConnections()
- getMaxColumnsInTable()

Useful methods for retrieving information about SQL objects and their attributes include the following:

- getSchemas()
- getCatalogs()
- getTables()
- getPrimaryKeys()
- getProcedures()

The transaction-support capabilities of the database management system can be queried using these methods:

- supportsMultipleTransactions()
- getDefaultTransactionIsolation()
- supportsSavePoints()

Note Many of the DatabaseMetaData methods have been added or modified in JDBC 2.0 and JDBC 3.0, so if your driver is not JDBC 2.0 or JDBC 3.0

compliant, a SQLException may be thrown.

ResultSetMetaData

Information about the columns in a ResultSet is available by calling the getMetaData() method. The ResultSetMetaData object returned gives the number, types, and properties of its ResultSet object's columns.

Some of the methods available to access ResultSetMetaData are as follows:

- getColumnCount() — Returns the number of columns in the ResultSet
- getColumnDisplaySize(int column)— Returns the column's normal max width in chars
- getColumnLabel(int column) — Returns the column title for use in printouts and displays
- getColumnName(int column) — Returns the column name
- getColumnType(int column) — Returns the column's SQL data-type index
- getColumnTypeName(int column)— Returns the name of the column's SQL data type
- getPrecision(int column)— Returns the number of decimal digits in the column
- getScale(int column) — Returns the number of digits to right of the decimal point
- getTableName(int column) — Returns the table name
- isAutoIncrement(int column) — Returns true if the column is automatically numbered
- isCurrency(int column) — Returns true if the column value is a currency
- isNullable(int column)— Returns true if the column value can be set to NULL

Listing 4-10 illustrates the use of the ResultSetMetaData methods getColumnCount and getColumnLabel in an example where the column names and column count are unknown.

Listing 4-10: Using ResultSetMetaData

```
public void printResultSet(String query){
    try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection
("jdbc:odbc:Inventory");
       Statement stmt = con.createStatement();
       ResultSet rs = stmt.executeQuery(query);
       ResultSetMetaData md = rs.getMetaData();

       int nColumns = md.getColumnCount();
        for(int i=1;i<=nColumns;i++){

System.out.print(md.getColumnLabel(i)+((i==nColumns)?"\n":"\t"));
        }
       while (rs.next()) {
          for(int i=1;i<=nColumns;i++){
```

```
                System.out.print(rs.getString(i)+((i==nColumns)?"\n":"\t"));
            }
        }
    }
    catch(ClassNotFoundException e){
        e.printStackTrace();
    }
    catch(SQLException e){
        e.printStackTrace();
    }
}
```

Notice in particular the use of the getColumnLabel method. This method returns the preferred display name for the column, defaulting to the column name if no specific label is assigned.

ParameterMetaData

The PreparedStatement method getMetaData() retrieves a ResultSetMetaData object containing a description of the columns that will be returned when the PreparedStatement is executed. Here's an example:

```
PreparedStatement ps = con.PrepareStatement("SELECT * FROM CUSTOMERS");
ResultSetMetaData md = ps.getMetaData();
int cols = md.getColumnCount();
```

The method getParameterMetaData() returns a ParameterMetaData object containing descriptions of the IN and OUT parameters the PreparedStatement uses, as shown here:

```
PreparedStatement ps = con.PrepareStatement("SELECT * FROM CUSTOMERS");
ParameterMetaData pd = ps.getParameterMetaData();
int pType = pd.getParameterType(1);
```

Note Support for ParameterMetaData is provided as part of the JDBC 3.0 API, and requires JDK 1.4

JDBC Mapping of SQL Data Types

The JDBC Core API provides automatic type conversion between SQL data types and Java data types. Table 4-5 summarizes these conversions.

Table 4-5: Standard Mapping from SQL Types to Java

SQL type	Java Type	Description
CHAR	String	Fixed-length character string. For a CHAR type

Table 4-5: Standard Mapping from SQL Types to Java

SQL type	Java Type	Description
		of length n, the DBMS invariably assigns n characters of storage, padding unused space.
VARCHAR	String	Variable-length character string. For a VARCHAR of length n, the DBMS assigns up to n charcters of storage, as required.
LONGVARCHAR	String	Variable-length character string. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
NUMERIC	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
DECIMAL	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
BIT	boolean	Yes / No value
TINYINT	byte	8 bit integer values
SMALLINT	short	16 bit integer values
INTEGER	int	32 bit integer values
BIGINT	long	64 bit integer values
REAL	float	Floating point number, mapped to float
FLOAT	double	Floating point number, mapped to double
DOUBLE	double	Floating point number, mapped to double
BINARY	byte[]	Retrieve as byte array.
VARBINARY	byte[]	Retrieve as byte array.
LONGVARBINARY	byte[]	Retrieve as byte array. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
DATE	java.sql.Date	Thin wrapper around java.util.Date
TIME	java.sql.Time	Thin wrapper around java.util.Date
TIMESTAMP	java.sql.Timestamp	Composite of a java.util.Date and a separate nanosecond value

Cross-Reference In addition to the data types supported by the JDBC Core API, JDBC 2.0 and JDBC 3.0 have introduced support for other data

types. These are discussed in the next few paragraphs.

Some databases allow for certain columns to be given automatically generated key values. In this case, an insert statement is not responsible for supplying a value for the column. The database generates a unique value for the column and inserts the value. This is often used for generating unique primary keys. A problem with this approach is that it may be difficult to get the value after the insert is executed. The JDBC 3.0 specification defines a more functional Statement interface that provides access to these values after an insert.

Assume a table called USERS with three columns. The FIRST_NAME column and LAST_NAME column are varchars. The USER_ID column is auto-generated and should contain a unique identifier for each user in the table. Here's an example:

```
Statement stmt = conn.createStatement();
String SQLInsert = "INSERT INTO Users (First_Name, Last_Name) "+
                   "VALUES('Robert',  'Conners')");

stmt.executeUpdate(SQLInsert);
ResultSet rs = stmt.getGeneratedKeys();
```

SQL3 Data Types

The JDBC 2.0 Extension API adds support for the new data types commonly referred to as SQL3 types. The JDBC 3.0 Extension API extends this support. These new data types support the two following major new features:

- Very large data objects
- Object relational data types

The SQL3 data types are being adopted in the next version of the ANSI/ISO SQL standard. The JDBC API extensions provide interfaces that represent the mapping of these SQL3 data types into the Java programming language. With these new interfaces, you can work with SQL3 data types the same way you do other data types.

Object Relational Databases

Object relational databases are simply an extension to normal relational database management systems supporting the use of an object-oriented-design approach to the database world.

For example, in a normal RDBMS, you might create a table of names and addresses, containing these columns:

First_Name	VARCHAR (20)
MI	CHAR(1)
Last_Name	VARCHAR(30)
Street	VARCHAR(50)
City	VARCHAR(30)
State	CHAR(2)
Zip	CHAR(10)

In another application, you might create a second table of names and addresses, perhaps this time with different field sizes or even additional fields.

From a design viewpoint, the ability to define a class or structure can be used across the board is very attractive.

An object relational database provides the necessary tools to support this approach with User Defined Data Types (UDTs).

Using SQL3 Data Types

The new SQL3 data types that the JDBC 2.0 Extension supports include the following:

- BLOB (Binary Large Object), which can store very large amounts of data as raw bytes
- CLOB (Character Large Object), which can store very large amounts of character data
- ARRAY, which can store an array as a column value
- User Defined Types
- Structured, object relational types
- The DISTINCT type

The following list provides the JDBC 2.0 interfaces that map SQL3 types. We discuss them in more detail later in this chapter.

- A Blob instance maps an SQL BLOB value.
- A Clob instance maps an SQL CLOB value.
- An Array instance maps an SQL ARRAY value.
- A Struct instance maps an SQL structured type value.
- A Ref instance maps an SQL REF value.

SQL3 data types are retrieved, stored, and updated in the same way as other data types, using the methods shown in Table 4-6.

Table 4-6: SQL3 Data Type Reference Methods

SQL3 type	get	set	update
BLOB	getBlob	seBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
Structured type	getObject	setObject	updateObject
REF (structured type)	getObject	setObject	updateObject

Note At the time of this writing, the update methods are scheduled for future release. Until then, you can use the method updateObject, which works just as well.

Here's an example of accessing one of these new data types. The following code fragment retrieves a CLOB value, Notes, from a patient's medical records.

```
ResultSet rs = stmt.executeQuery(
                "SELECT Notes FROM Patients WHERE SSN = 123-45-6789");
rs.next();
Clob notes = rs.getClob("Notes");
```

Because a SQL BLOB, CLOB, or ARRAY object may be very large, an instance of any of these types is actually a SQL locator or logical pointer to the object in the database that the instance represents. JDBC provides the tools to manipulate them without having to bring all of their data from the database server to your client machine. This feature can make performance significantly faster.

If you want to bring the data of a BLOB or CLOB value to the client, you can use the following methods in the Blob and Clob interfaces provided for this purpose:

- getAsciiStream()(Gets the CLOB value designated by this Clob object as a stream of ASCII bytes)
- getCharacterStream() (Gets the Clob contents as a Unicode stream)
- getSubString(long pos, int length)(Returns a copy of the specified substring in the CLOB value designated by this Clob object)
- length()(Returns the number of characters in the CLOB value designated by this Clob object)
- position(Clob searchstr, long start) (Determines the character position at which the specified Clob object searchstr appears in this Clob object)
- position(String searchstr, long start) (Determines the character position at which the specified substring searchstr appears in the CLOB)

Both Blob and Clob objects provide methods for materializing the object's value on the client, for getting the length of the object, and for performing searches within the object's value.

The JDBC 3.0 API Extensions add methods to alter the values of BLOBS and CLOBS directly, using these methods:

- Blob.setBytes()
- Clob.setString()

A JDBC Array object materializes the SQL ARRAY it represents as either a result set or a Java array.

For example, after retrieving the SQL ARRAY value in the column Meds as a java.sql.Array object, the following code fragment materializes the ARRAY value on the client. It then iterates through Medications, the Java array that contains the elements of the SQL ARRAY value.

```
ResultSet rs = stmt.executeQuery(
                    "SELECT MEDS FROM Patients WHERE SSN = 123-45-6789");
while (rs.next()) {
   Array Medications = rs.getArray("MEDS");
   String[] meds = (String[])Medications.getArray();
  for (int i = 0; i < meds.length; i++) {
        . . . // code to display medications
    }
}
```

The ResultSet method getArray returns the value stored in the column MEDS of the current row as the java.sql.Array object Medications, as shown here:

```
Array Medications = rs.getArray("MEDS");
```

The variable Medications contains a locator, which means that it is a logical pointer to the SQL ARRAY on the server; it does not contain the elements of the ARRAY itself.

In the following line, getArray is the Array.getArray method, returning a Java Object that is cast to an array of String objects before being assigned to the variable meds.

```
String[] meds = (String[])Medications.getArray();
```

Thus, the Array.getArray method materializes the SQL ARRAY elements on the client as an array of String objects we can iterate through and display.

Creating User Defined Data Types

SQL allows the user to create user defined data types or UDTs with the CREATE TYPE statement. There are two main kinds of data type which the user can create:

- The structured data type
- The DISTINCT type

Creating a structured data type

The following SQL statement creates the new data type ADDRESS and registers it with the database as a data type, so it is available for use as the data type for a table column or as an attribute of a structured type:

```
CREATE TYPE ADDRESS
(
      STREET  VARCHAR(40),
      APT_NO  INTEGER,
      CITY  VARCHAR(40),
      STATE  CHAR(2),
     ZIP  CHAR(5)
);
```

In this definition, the new type ADDRESS has five attributes, which are equivalent to fields in a Java class. The attribute STREET is a VARCHAR(40); the attribute APT_NO is an INTEGER; the attribute CITY is a VARCHAR(40); the attribute STATE is a CHAR(2); and the attribute ZIP is a CHAR(5).

Creating a DISTINCT type

A DISTINCT type can be thought of as a structured type with only one attribute. DISTINCT types are always based on another data type, which must be a predefined type; they cannot be based on another UDT. DISTINCT types are retrieved or set using the appropriate method for the underlying type.

For example, a Social Security Number (SSN), which is never going to be used for arithmetic operations, and may be a good candidate for special handling, can be created using the command. Here's an example:

```
CREATE TYPE SSN AS CHAR(9);
```

This is the equivalent SQL Server command:

```
EXEC sp_addtype SSN, 'VARCHAR(9)'
```

Now that User Defined Data Types for Address and Social Security Number have been created, they can be used to define a new UDT, as shown here:

```
CREATE TYPE EMPLOYEE
(
    EMP_ID INTEGER,
    LAST_NAME VARCHAR(40),
    FIRST_NAME VARCHAR(40),
    RESIDENCE ADDRESS,
    SOCIAL SSN
);
```

This definition can be created in a JDBC application by opening a connection and creating a Statement in the normal way, then executing the following code to send the definition of the structured type EMPLOYEE to the database.

```
String createEmployee = "CREATE TYPE EMPLOYEE ("+
                        "EMP_ID INTEGER,"+
                        "LAST_NAME VARCHAR(40),"+
                        "FIRST_NAME VARCHAR(40),"+
                        "RESIDENCE ADDRESS,"+
                        "SOCIAL SSN);";
stmt.executeUpdate(createEmployee);
```

On occasion, your code may generate errors. Java handles these errors by throwing SQLExceptions, as discussed in the next section.

Exceptions and Logging

There are several types of exceptions which can be thrown during data base access. The most common is the SQLException.

SQLException

The SQLException class extends java.lang.Exception to provide information on database-access errors. Each SQLException provides the following information:

- The Java exception message String, available using the getMessage() method
- The SQLState String, which follows the XOPEN SQLState conventions, available using the getSQLState() method
- A vendor-specific, integer-error code, available using the getErrorCode() method. Normally, this isthe actual error code that the underlying database returns.

SQLException also lets you get the next exception, which can be used to provide additional error information.

SQLWarning

The SQLWarning class extends SQLException to provide information on database-access warnings. Warnings are silently chained to the object whose method causes the warning to be reported and are returned by the getWarnings() method of that class.

In addition to the inherited methods of SQLException, SQLWarning provides methods to get the next SQLWarning for additional information or to add a warning to the chain.

BatchUpdateException

A BatchUpdateException provides information about problems arising during batch updates. BatchUpdateException extends SQLException, adding an array of update counts similar to the array returned by the executeBatch method. You can retrieve this array by using the getUpdateCounts() method as follows:

```
int [] updateCounts = b.getUpdateCounts();
```

Since the update counts are in the same order as the commands, you can tell which commands in the batch have executed successfully.

Logging

In all but the simplest applications, it is worth incorporating some degree of error and event logging. The most basic form of logging, of course, is the use of System.err and System.out to report exceptions and significant events.

In a practical application, simply dumping exception messages to the system console is generally inadequate. It is preferable to use dedicated logging files or perhaps even a database to manage event logs and error logs.

It is easy to implement a file-based error and event-logging system by simply redirecting the basic System.err stream and by defining a PrintWriter for use by the Exception class for dumping a StackTrace.

Listing 4-11 extends the example of Listing 4-1 to demonstrate two different ways to log exceptions to an error-logging file:

- Define a PrintWriter for use with the printStackTrace() method.
- Redirect System.err to a logging file by using System.setErr().

Listing 4-11: Logging errors to a file

```
package java_databases.ch04;

import java.io.*;
```

```
import java.sql.*;
import java.util.*;

public class Logging{
  public static void main(String args[]){
    PrintWriter errLog = null;
    PrintStream stderr = null;
     try{
      FileOutputStream errors = new FileOutputStream ("StdErr.txt",
true);
       stderr = new PrintStream (errors);
       errLog = new PrintWriter(errors,true);
     }
    catch (Exception e){
      System.out.println ("Redirection error:  Unable to open SystemErr.txt");
     }
    System.setErr ( stderr );

     int qty;
     float cost;
     String name;
     String desc;
    String query = "SELECT Name,Description,Qty,Cost,Sell_Price FROM Stock";

     try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con = DriverManager.getConnection
("jdbc:odbc:Inventory");
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
           name = rs.getString("Name");
           desc = rs.getString("Description");
            qty  = rs.getInt("Qty");
           cost = rs.getFloat("Cost");
            System.out.println(name+", "+desc+"\t: "+qty+"\t@ $"+cost);
         }
     }
     catch(ClassNotFoundException e){
        e.printStackTrace(errLog);
     }
     catch(SQLException e){
        System.err.println((new GregorianCalendar()).getTime());
        System.err.println("Thread: "+Thread.currentThread());
```

```
            System.err.println("ErrorCode:  "+e.getErrorCode());
            System.err.println("SQLState:    "+e.getSQLState());
            System.err.println("Message:      "+e.getMessage());
            System.err.println("NextException:  "+e.getNextException());
            e.printStackTrace(errLog);
            System.err.println();
        }
        try{
            stderr.close  ();
        }
        catch  (Exception  e){
            System.out.println("Redirection  error:    Unable  to  close SystemErr.txt");
        }
    }
}
```

A practical point worth noting is that the example saves current time and the current thread as part of the logged-error information.

Caution Remember to open your error-logging file for append. Otherwise, you see only the last error. Also, it is a good idea to set autoFlush = true as shown, so that errors are written to the file immediately.

This query is used in Listing 4-11:

```
String query = "SELECT Name,Description,Qty,Cost,Sell_Price FROM Stock";
```

This query attempts to SELECT a nonexistent column, so a SQL Exception is thrown, resulting in logging the following error messages to the error log:

```
Sun Dec 30 14:43:44 EST 2001
Thread: Thread[main,5,main]
ErrorCode: -3010
SQLState:    07001
Message:    [Microsoft][ODBC Microsoft Access Driver] Too few parameters.
Expected 1.
NextException: null
java.sql.SQLException: [Microsoft][ODBC Microsoft Access Driver] Too few
parameters. Expected 1.
  at sun.jdbc.odbc.JdbcOdbc.createSQLException(JdbcOdbc.java:6031)
  at sun.jdbc.odbc.JdbcOdbc.standardError(JdbcOdbc.java:6188)
  at sun.jdbc.odbc.JdbcOdbc.SQLExecDirect(JdbcOdbc.java:2494)
  at sun.jdbc.odbc.JdbcOdbcStatement.execute(JdbcOdbcStatement.java:314)
  at
```

```
sun.jdbc.odbc.JdbcOdbcStatement.executeQuery(JdbcOdbcStatement.java:229)
 at java_databases.ch04.Logging.main(Logging.java:30)
```

Summary

Part I is an introduction to database management systems, SQL, and JDBC, providing a theoretical overview of the topics as a basis for the more detailed explanations in subsequent chapters.

This chapter provides an overview of the use of the JDBC API. In this chapter, you learn about the building blocks of a JDBC-based application:

- Using the DriverManager and different types of JDBC drivers
- Using JDBC DataSources for dimple, pooled, and distributed connections
- Using connections
- Using Statements, PreparedStatements and CallableStatements
- Using transactions, isolation levels and SavePoints
- Handling batch updates
- Using ResultSets and Rowsets
- Using MetaData
- JDBC Mapping of SQL Data Types
- Exceptions and loggin

Part II: Using JDBC and SQL in a Two-Tier Client/Server Application

Chapter List

Part Overview

Part II expands the overviews of Part I by presenting a series of application examples that cover two major topics in depth: The JDBC core API and SQL basics. These topics are covered in the context of a series of Swing-based desktop applications. Each chapter starts with a detailed discussion of a major element of the SQL language, followed by a presentation of a JDBC application using the SQL commands discussed.

Individual chapters are dedicated to using basic SQL commands to create, populate, and query databases, as well as to using the various SQL operators to build more complex queries. The Java examples use the JDBC core API to connect to a database and execute the SQL commands.

Another chapter is devoted to showing how to perform SQL joins and compound queries. Inner and outer joins, self-joins, and unions are discussed, as are ordering and grouping the results of these joins.

The final chapter in Part II brings together the examples in the previous chapters to create a Swing GUI that can be used as a control panel for any database system. This chapter goes on to explain how JDBC can be used with any RDBMS system by simply plugging in the appropriate drivers. The examples compare the effects on performance of plugging in a commercial pure Java driver in place of Sun's JDBC-ODBC bridge.

Chapter 5: Creating a Table withJDBC and SQL

This chapter discusses various ways in which JDBC and SQL enable you to create tables and manipulate the content therein.

Creating the Database

Before we can create a table, we need to create a database. This has to be done using the Database Management System itself, because JDBC requires an existing database to make a connection.

DBMS packages that support a GUI, such as MS Access, SQL Server, Sybase, and Oracle, provide a simple graphical way to do this, generally in the form of a wizard, which guides you through the necessary steps. If you are running a command line DBMS such as MySQL, start the package; at the command prompt, type the following:

```
CREATE DATABASE CONTACTS;
```

Although the material in this book applies to any JDBC driver, assume that you are using the JDBC-ODBC bridge. Once you have created the database, register it with the ODBC Data Source Administrator utility. If, in fact, you are using a different driver, the examples still work fine; all you need to do is to specify the name of the driver you are using when you register the driver with the DriverManager.

Assuming that you are, in fact, using the JDBC-ODBC bridge, you will need to register your newly created database with the ODBC Data Source Administrator utility. If, in fact, you are using a different driver, the examples still work fine: all you need to do is to specify the name of the driver you are using when you register the driver with the DriverManager.

Once you have created a database, you are ready to start creating tables. The SQL commands used to create tables are discussed in the next section.

Using Tables

Relational databases store data in tables. A given database may contain one or more tables, depending on the application. Tables are intended to store logically related data items together, so a database may contain one table for business contacts, another for projects, and so on.

Each table in a database is like a spreadsheet. When you create a table, you tell the RDBMS how many columns each row has. Each record in the database consists of one row in this table.

A database is more restrictive than a spreadsheet in that all the data in one column must be of the same type, such as integer, decimal, character string, or date. Another difference between a spreadsheet and a database is that unlike the rows in a spreadsheet, the rows in a database have no implicit order. This is significant; although you may insert records in some order, there is no guarantee that they will be returned in that order when you query the database.

Cross-Reference

The design of relational databases and the organization of tables is

discussed in Chapter 1.

Records and Fields, Rows and Columns

A *table* (see Table 5-1) is a set of data records, arranged as rows, each of which contains individual data elements or fields, arranged as columns. Here and in subsequent chapters in this part of the book, we are working with a simple Name and Address Table. Each row in this table is a record containing information about a single individual or entity.

Successive fields within the record contain different pieces of information about the person or entity, such as first name, middle initial, last name, and so on. These fields are arranged logically in columns, so that the first column contains first names, the second, middle initials, and so on.

Table 5-1: Example of a Table

First_Name	MI	Last_Name	Street	City	State	Zip
Alex	M	Baldwin	123 Pine St	Washington	DC	12345
Michael	Q	Cordell	1701 York Rd	Columbia	MD	21144

It is immediately obvious that all fields within a given column have the following features in common:

- They are similar in *type;* for example, all M.I. fields contain zero or one character, and all zips are numeric.
- They form part of a column that has a *name*.
- As you will see shortly, all fields in a column may be subject to one or more constraints.

Note

The table and column names must start with a letter and can be followed by letters, numbers, or underscores. Do not use any SQL reserved keywords as names for tables or column names (such as "select," "create," "insert," and so on).

Create this table using the `SQL CREATE` command. Before you can do this, there are some decisions you need to make regarding data types, field lengths, and constraints.

SQL Data Types

As we see in Chapter 2, SQL supports a variety of data types. Table 5-2 lists SQL data types with the corresponding java.sql data types.

Table 5-2: Standard Mapping from SQL Types to Java

SQL type	Java type	Description
CHAR	String	Fixed-length character string. For a CHAR type of length n, the DBMS invariably assigns n characters of storage, padding unused space.
VARCHAR	String	Variable-length character string. For a VARCHAR of length n, the DBMS assigns up to n charcters of storage, as required.

Table 5-2: Standard Mapping from SQL Types to Java

SQL type	Java type	Description
LONGVARCHAR	String	Variable-length character string. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
NUMERIC	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
DECIMAL	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
BIT	boolean	Yes/No value
TINYINT	byte	8 bit integer values
SMALLINT	short	16 bit integer values
INTEGER	int	32 bit integer values
BIGINT	long	64 bit integer values
REAL	float	Floating point number, mapped to float
FLOAT	double	Floating point number, mapped to double
DOUBLE	double	Floating point number, mapped to double
BINARY	byte[]	Retrieve as byte array
VARBINARY	byte[]	Retrieve as byte array
LONGVARBINARY	byte[]	Retrieve as byte array. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
DATE	java.sql.Date	Thin wrapper around java.util.Date
TIME	java.sql.Time	Thin wrapper around java.util.Date
TIMESTAMP	java.sql.Timestamp	Composite of a java.util.Date and a separate nanosecond value

As you can see from Table 5-2, most of the fields we will be using can be handled using the VARCHAR type. The zip code is perhaps also best handled using a VARCHAR, since we will not be using it for arithmetic; nine-digit zips are frequently entered with a hyphen as a separator.

Note

VARCHAR is preferrable to CHARACTER because when you use CHARACTER(n), the DBMS always assigns n characters to the field, padding the field to fill unallocated space; when you use VARCHAR(n), the DBMS assigns up to n characters, as required.

Integrity Constraints

In addition to selecting data type and length, there are various integrity constraints you may need to apply to the data stored in a column. Integrity constraints are important to ensure consistency and accuracy.

NULL or NOT NULL

In addition to assigning a data type to a field, SQL lets you specify whether a field is required to contain valid data or whether it can be left empty. In our example, you may decide that you require first name and last name, but you may not be particularly concerned about middle initials. In this case, set the constraints for first name and last name to `NOT NULL` and the constraint for middle initial to `NULL`.

Note

Most database systems default to NULL.

UNIQUE

The `UNIQUE` constraint specifies that no two records can have the same value in a particular column. They must each be unique. An employee id, for example, should be unique.

PRIMARY KEY

The primary key is used by the database management systems as a unique identifier for a row. Probably the best choice for a primary-key field is an integer, because integers are much faster to process than, for example, long strings when processing the table. This is one reason why Oracle provides a ROWID field that is incremented for each row that is added, and MSAccess offers an AutoNumber option, making the field always a unique key by default.

Note

`NULL`, `UNIQUE`, and `PRIMARY KEY` are the constraints most commonly used, but various database management systems offer custom constraints, such as Oracle's CHECK, which lets you define syntactic and logical checks to be performed on field values prior to insertion.

This brief review of data types, constraints and keys should have given you enough background to start creating a table. The use of SQL to create tables is covered in the next section.

Creating a Table

Now that you know enough about the data you intend to store in your table, you are ready to give your table a name and write the SQL command to create it. Tables are created using the `CREATE TABLE` statement with a table name, followed in parentheses (()) by a series of column definitions. Here's an example:

```
CREATE TABLE tableName ( columnName dataType [constraints],...);
```

Column definitions simply list the column or field name, followed by the data type and the optional constraints. Column definitions are separated by commas, as shown here:

```
CREATE TABLE CONTACT_INFO
(CONTACT_ID     INTEGER      NOT NULL   PRIMARY KEY,
 FIRST_NAME     VARCHAR(20)   NOT NULL,
```

```
MI            CHAR(1)      NULL,
LAST_NAME      VARCHAR(30)   NOT NULL,
STREET         VARCHAR(50)   NOT NULL,
CITY           VARCHAR(30)   NOT NULL,
STATE          CHAR(2)      NOT NULL,
ZIP            VARCHAR(10)   NOT NULL);
```

Caution

Notice the semicolon terminating the command. Most dialects of SQL work with semicolons, but some, such as Transact-SQL, require the keyword GO. We use semicolons in our examples.

The next section will show you how to use the SQL `CREATE TABLE` from a Java application.

Creating a Table Using JDBC

The JDBC API is made up of a small number of important classes and interfaces that handle the tasks of loading a suitable driver for your database, connecting to the database, creating and executing a SQL command, and handling any returned records. The primary classes we use for this example are as follows:

- DriverManager
- Driver
- Connection
- Statement

DriverManager

The DriverManager is responsible for loading JDBC drivers and for returning a connection to the appropriate driver. The DriverManager locates a suitable driver for the URL provided in the `getConnection()`call by polling the registered drivers.

Driver

For the examples in this book, we use the JDBC ODBC bridge. The first thing we do is load the `sun.jdbc.odbc.JdbcOdbcDriver` by name, using `Class.forName()`.Then we register it with the DriverManager, using this command:

```
DriverManager.registerDriver(new JdbcOdbcDriver());
```

Connection

We next request a connection to the database from the DriverManager using the following command:

```
getConnection(jdbc:driverName:databaseName);
```

The DriverManager polls all registered drivers to find the first one that can create a connection to the URL. Variations on this command let you give the database a user name and password or pass a Java Properties object with the URL:

```
getConnection(String url,String user,String password);
```

```
getConnection(String url, Properties info);
```

A connection represents a session with a specific database, providing the context in which our SQL statements are executed and results are returned.

Statement

The term `Statement` refers to the Java class that passes the SQL Query to the database via the connection rather than to the SQL Query itself. A `Statement` object is used for executing a static SQL statement and obtaining the results it produces.

The actual SQL command you pass to the database is the command you have just created when we were discussing the `CREATE` command. JDBC does not put any restrictions on the SQL commands you send to the database, but you must ensure that the data source you are connecting to supports whatever SQL you are using. JDBC allows any query string to be passed to an underlying DBMS driver, so an application may use as much SQL functionality as desired at the risk of receiving an error on some DBMSs. In fact, an application query need not even be SQL, or it may be a specialized derivative of SQL. If the database engine reports a problem, a SQLException will be thrown, providing information on the database-access error.

Listing 5-1 contains the code for creating a table using JDBC.

Listing 5-1: Creating a table using JDBC

```
package jdbc_bible.part2;


import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;


public class TableMaker{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String url = "jdbc:odbc:";



  static String SQLCreate =
       "CREATE TABLE CONTACT_INFO ("+
```

```
        "CONTACT_ID    INTEGER     NOT NULL   PRIMARY KEY,"+
        "FIRST_NAME    VARCHAR(20)  NOT NULL,"+
        "MI           CHAR(1)     NULL,"+
        "LAST_NAME     VARCHAR(30)  NOT NULL,"+
        "STREET        VARCHAR(50)  NOT NULL,"+
        "CITY          VARCHAR(30)  NOT NULL,"+
        "STATE         CHAR(2)     NOT NULL,"+
        "ZIP           VARCHAR(10)  NOT NULL"+
        ");";

  public TableMaker(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
      Class.forName(jdbcDriver);

    }
    catch(ClassNotFoundException e){
      System.err.print(e.getMessage());
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
  }
  public void execute(String SQLCommand){
    url += dbName;
    Connection con;
    Statement stmt;
    try {
      con = DriverManager.getConnection(url);
```

```
      stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
    finally {
      try {
          if (con != null) {
             con.close();
          }
          if (stmt !=null) {
            stmt.close();
          }
    } catch (Exception ex) { // ignore }
   }
  }
  public static void main(String[] args) {
    TableMaker tableMaker = new TableMaker();
    tableMaker.execute(SQLCreate);
  }
}
```

Compile and execute the example, and you should be able to see the new table in your database. Using a GUI-based system, you are able to see the table when you open the database. With a command line DBMS like MySQL, you need to type the following at the command prompt:

```
SHOW TABLES;
```

In addition to creating a table, you may find it necessary to alter an existing table. This can be done using the `ALTER TABLE` command.

Altering a Table with ALTER TABLE

Now that you have built your table, it looks as if you should have included fields for phone number and e-mail address. Many database management systems let you use SQL to modify tables with the `ALTER TABLE` command. The `ALTER TABLE` command enables you to do these two things:

- Add a column to an existing table

- Modify a column that already exists

This is the syntax for the `ALTER TABLE` command:

```
ALTER TABLE tableName ADD columnName dataType;
```

For example, to add a phone number field to the `CONTACT_INFO` table, use this command:

```
ALTER TABLE CONTACT_INFO ADD PHONE VARCHAR(16);
```

You can use `MODIFY` to change a column constraint from `NOT NULL` to `NULL` using this command:

```
ALTER TABLE tableName MODIFY columnName dataType NULL;>
```

In much the same way, you can use `MODIFY` as follows to change the width of a column using:

```
ALTER TABLE tableName MODIFY columnName dataType;>
```

Caution

You can always increase the width of a column, but you can't reduce the width below that of the widest value anywhere in the column. Similarly, you can only change a column's constraints from NOT NULL to NULL if there are no NULL values in the column.

Note

Implementations of the MODIFY clause tend to be specific to a database management system. Some allow the use of the MODIFY clause; others do not.

These are the SQL statements that are required to insert phone number and e-mail address columns:

```
ALTER TABLE CONTACT_INFO ADD PHONE VARCHAR(16);
ALTER TABLE CONTACT_INFO ADD EMAIL VARCHAR(50) NOT NULL;
```

Just as you can create a table using JDBC, you can also alter a table using JDBC. As you can see from the example of Listing 5-2, the code to alter a table looks very much like the `TableMaker.java` example, with the exception that a new method, `execute(String[] SQLCommand) has been added.` This method loops through an array of SQL commands to execute each of the `ALTER TABLE` commands.

Listing 5-2: Altering a table using JDBC

```
IMPORT JAVA.SQL.*;
IMPORT SUN.JDBC.ODBC.JDBCODBCDRIVER;


PUBLIC CLASS TABLEMODIFIER{
  STATIC STRING JDBCDRIVER = "SUN.JDBC.ODBC.JDBCODBCDRIVER";
  STATIC STRING DBNAME = "CONTACTS";
  STATIC STRING URL = "JDBC:ODBC:CONTACTS";
```

```
STATIC STRING[] SQLALTER = {
  "ALTER TABLE CONTACT_INFO ADD PHONE VARCHAR(16);",
  "ALTER TABLE CONTACT_INFO ADD EMAIL VARCHAR(50);",
};
PUBLIC TABLEMODIFIER(){
  REGISTERDRIVER();
}
PUBLIC VOID REGISTERDRIVER(){
  TRY {
    CLASS.FORNAME(JDBCDRIVER);
    DRIVERMANAGER.REGISTERDRIVER(NEW JDBCODBCDRIVER());
  }
  CATCH(CLASSNOTFOUNDEXCEPTION E){
    SYSTEM.ERR.PRINT(E.GETMESSAGE());
  }
  CATCH(SQLEXCEPTION E){
    SYSTEM.ERR.PRINTLN(E.GETMESSAGE());
  }
}
PUBLIC VOID EXECUTE(STRING[] SQLCOMMAND){
  TRY {
    CONNECTION CON = DRIVERMANAGER.GETCONNECTION(URL);
    STATEMENT STMT = CON.CREATESTATEMENT();
    FOR(INT I=0;I<SQLCOMMAND.LENGTH;I++){
      STMT.EXECUTE(SQLCOMMAND[I]);
    }
    CON.CLOSE();
  }
  CATCH(SQLEXCEPTION E){
    SYSTEM.ERR.PRINTLN(E.GETMESSAGE());
  }
}
PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
  TABLEMODIFIER TABLEMODIFIER = NEW TABLEMODIFIER();
```

```
    TABLEMODIFIER.EXECUTE(SQLALTER);
  }
}
```

From time to time, you may need to delete a table. Deleting tables, like creating and altering tables, is easy to do using SQL and JDBC.

Deleting or Dropping a Table

Deleting a table with SQL is done using the `DROP TABLE` command. The `DROP TABLE` command deletes a table along with all its associated views and indexes. Here's the syntax for the `DROP TABLE` command:

```
DROP TABLE table_name;
```

To drop the `CONTACT_INFO`, issue this command:

```
DROP TABLE CONTACT_INFO;
```

Since the code used to drop a table is very similar to the code used to create or alter a table, a dedicated Java example for `DROP TABLE`, is not provided. The Table Builder example discussed in the next section includes an example of the code required to drop a table in listing 5-3.

Creating a Swing-based Table Builder

To illustrate the topics covered in this chapter, we build a Swing-based Table Builder. This application forms the basis of a complete database management console, which, with only minor modifications, works with any database management system.

The Table Builder uses Model View Controller (MVC) architecture, both for clarity, and because MVC designs are easier to understand, build, and maintain. The first step is to create the controller portion of the MVC architecture.

The controller responds to user inputs from the various view elements and commands the model to execute the user's commands. User inputs come from JMenus, dialog boxes, and JInternalFrames.

The sequence of events is as follows:

1. User selects a database.
2. User assigns a table name.
3. User defines the required fields for the table.
4. `SQL CREATE TABLE` command is issued to create the desired table.

The controller interacts with the model using classes based on the code we look at in Listing 5-1.

The view portion of the MVC architecture is handled by the usual `JMenu` items, together with a `JOptionPane` to deal with single-value inputs such as database name and table name, and dedicated `JInternalFrames` to handle anything more complicated. The view components interact with the controller only for initialization and to return the data they are designed to collect. The model portion of

the MVC architecture performs the actual JDBC functions, connecting to the database and issuing the `CREATE TABLE` command.

Controller

Base your controller on a `JFrame`, which also hosts the view components. If you want to be a purist, you can create the `JFrame` separately and subclass it for the controller; but as the only view-related code in this class is in the constructor, I feel it is acceptable to use the `JFrame` as the controller.

In addition to constructing the `JFrame`, the constructor adds a `JDesktopPane` to handle the `JInternalFrames`, a `JMenu` class, and the `ActionListener` for the `JMenu`.

The `selectDatabase()` method is called when the user selects the "Database" menu option. This method prompts the user for the database name using a JOptionPane; after saving the database name, it enables the New Table menu item and the Drop Table menu item.

When either the New Table menu item or the Drop Table menu item is selected, a JOptionPane is displayed to get the table name, and then, depending on the menu selection, either the TableBuilderFrame is displayed to allow the user to create the table, or a JOptionFrame is displayed to confirm that the table should be dropped.

The `displayTableBuilderFrame()`method is called when the user responds to a prompt for the table name. It launches the `TableBuilderFrame`, setting an `ActionListener`, `CommandListener`, to receive the completed `CREATE TABLE` command from the `JInternalFrame`.

The `CommandListener` ultimately passes the `CREATE TABLE` command to the JDBC `SQLToolkit` class, which connects to the database and creates the table. (See Listing 5-3.)

Listing 5-3: Swing-based Table Builder — the main JFrame

```
package jdbc_bible.part2;


import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;


public class DBManager extends JFrame{
  JMenuBar menuBar = new JMenuBar();
  JDesktopPane desktop = new JDesktopPane();
  String database = null;
  String tableName = null;
```

```
String menuSelection = null;
TableBuilderFrame tableMaker = null;
DatabaseUtilities dbUtils = null;

TableMenu tableMenu = new TableMenu();

MenuListener menuListener = new MenuListener();

public DBManager(){
  setJMenuBar(menuBar);
  setTitle("JDBC Database Bible");
  getContentPane().setLayout(new BorderLayout());
  getContentPane().add(desktop,BorderLayout.CENTER);
  setSize(new Dimension(640,480));

  menuBar.add(tableMenu);
  tableMenu.setMenuListener(menuListener);

  setVisible(true);
}

private void displayTableBuilderFrame(){
  tableName = JOptionPane.showInputDialog(this,"Table:",
            "Select table",JOptionPane.QUESTION_MESSAGE);
  tableMaker = new TableBuilderFrame(tableName);
  tableMaker.setCommandListener(new CommandListener());
  desktop.add(tableMaker);
  tableMaker.setVisible(true);
}

private void selectDatabase(){
  database = JOptionPane.showInputDialog(this,"Database:",
           "Select database",JOptionPane.QUESTION_MESSAGE);
  dbUtils = new DatabaseUtilities();
```

```
    dbUtils.setExceptionListener(new ExceptionListener());

    tableMenu.enableMenuItem("New Table",true);
    tableMenu.enableMenuItem("Drop Table",true);
  }

  private void executeSQLCommand(String SQLCommand){
    dbUtils.execute(SQLCommand);
  }

  private void dropTable(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
              "Select table",JOptionPane.QUESTION_MESSAGE);
    int option = JOptionPane.showConfirmDialog(null,
              "Dropping table "+tableName,
              "Database "+database,
               JOptionPane.OK_CANCEL_OPTION);
    if(option==0){
      executeSQLCommand("DROP TABLE "+tableName);
    }
  }

class MenuListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      String menuSelection = event.getActionCommand();
      if(menuSelection.equals("Database")){
        selectDatabase();
      }else if(menuSelection.equals("New Table")){
        displayTableBuilderFrame();
      }else if(menuSelection.equals("Drop Table")){
        dropTable();
      }else if(menuSelection.equals("Exit")){
        System.exit(0);
      }
```

```
    }
  }

  class ExceptionListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      String exception = event.getActionCommand();
      JOptionPane.showMessageDialog(null, exception,
       "SQL Error", JOptionPane.ERROR_MESSAGE);
    }
  }

  class CommandListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      String SQLCommand = event.getActionCommand();
      executeSQLCommand(SQLCommand);
    }
  }

  public static void main(String args[]){
    DBManager dbm = new DBManager ();
  }
}
```

View

The view is handled primarily by these two classes:

- `TableMenu`.
- `TableBuilderFrame`

TableMenu

`TableMenu` displays and handles inputs from a basic `JMenu` used to select a database and identify a table. Being the first menu on the `JMenuBar`, the `TableMenu` also handles the `Exit` function.

`TableMenu` extends a simple base class `DBMenu` (see Listing 5-4), which provides common functionality. The main purpose of `DBMenu` is to simplify menu creation and to provide a common point for hooking an event listener into the menu items so that they do not have to be set up individually from the controller.

Listing 5-4: DBMenu (the base class for TableMenu)

```
package jdbc_bible.part2;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DBMenu extends JMenu{
  JMenuItem dbItem;
  JMenuItem newItem;
  JMenuItem openItem;
  JMenuItem exitItem;

  ActionListener menuListener = null;
  MenuItemListener itemListener = new MenuItemListener();

  public DBMenu(){
  }
  public void enableMenuItem(String itemName,boolean enable){
    Component c[] = getMenuComponents();
    for(int i=0;i<c.length;i++){
      if(c[i] instanceof JMenuItem){
        JMenuItem menuItem = (JMenuItem)c[i];
        if(menuItem.getText().equals(itemName))menuItem.setEnabled(enable);
      }
    }
  }
  public void setMenuListener(ActionListener menuListener){
    this.menuListener = menuListener;
  }
   class MenuItemListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
     String action = event.getActionCommand();
```

```
    if(menuListener!=null)menuListener.actionPerformed(event);
    }
  }
}
```

DBMenu is supported by DBMenuItem, which extends JMenuItem to provide a simple base class for the creation of the JMenuItems. This DBMenuItem base class is shown in Listing 5-5:

Listing 5-5: DBMenuItem (a convenience class for easy JMenuItem creation)

```
package jdbc_bible.part2;

import java.awt.event.*;
import javax.swing.*;

public class DBMenuItem extends JMenuItem{
  public DBMenuItem(String name,char hotkey,
        ActionListener itemListener,boolean enabled){
    super(name,(int)hotkey);
    setActionCommand(name);
    setEnabled(enabled);
    addActionListener(itemListener);
  }
}
```

Using these convenience classes, the creation of our custom menus becomes very simple, as you can see from Listing 5-6.

Listing 5-6: Table Menu

```
package jdbc_bible.part2;
import javax.swing.*;

public class TableMenu extends DBMenu{
  JMenuItem dbItem;
  JMenuItem newItem;
  JMenuItem openItem;
```

```
  JMenuItem exitItem;

  public TableMenu(){
  setText("Table");
  setActionCommand("Table");
  setMnemonic((int)'T');

  dbItem   = new DBMenuItem("Database", 'D',itemListener,true);
  newItem  = new DBMenuItem("New Table",'T',itemListener,false);
  openItem = new DBMenuItem("Drop Table",'D',itemListener,false);
  exitItem = new DBMenuItem("Exit",'X',itemListener,true);

    add(dbItem);
    addSeparator();
    add(newItem);
    add(openItem);
    addSeparator();
    add(exitItem);
  }
}
```

TableBuilderFrame

`TableBuilderFrame` is the heart of the MVC view. `TableBuilderFrame` extends `JInternalFrame,` containing a `JTable` used to set up the fields for the database table, a `JTextArea`, which provides a preview of the generated SQL command, and a Create Table button, which fires an `ActionEvent` to the controller, sending it the generated SQL command.

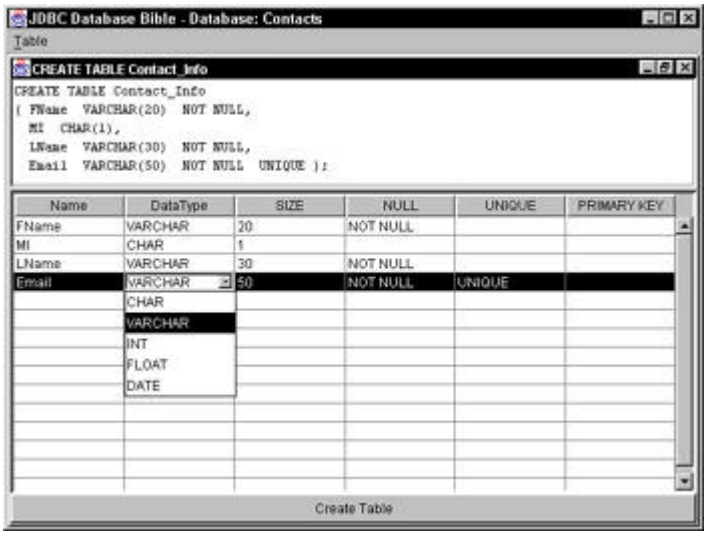


Figure 5-1: TableBuilderFrame generates SQL from table entries.

`TableBuilderFrame` is, in turn, built around a `Jtable`, which has been customized by adding `JComboBox` components as column editors for such fields as DataType.

The method `setCommandListener()` is called by the MVC controller so that `TableBuilderFrame` can pass the controller the generated SQL command when the Create Table button bar at the bottom of the frame is pressed by the user.

Listing 5-8: TableBuilderFrame

```
package jdbc_bible.part2;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
* TableBuilderFrame: a display which builds SQL CREATE statements
* <p/>
* TableBuilder, which extends JTable, is a key component.
*/
class TableBuilderFrame extends JInternalFrame{

  protected int nRows = 15;
  protected int nColumns = 6;
```

```
protected JTable table;
protected JTextArea SQLPane = new JTextArea();
protected JButton createButton = new JButton("Create Table");
protected ActionListener commandListener = null;

protected String tableName = null;
protected String SQLCommand = "";
protected String SQLCommandRoot = "";

public TableBuilderFrame(String tableName){
  setSize(600,400);
  setLocation(10,10);
  setClosable(true);
  setMaximizable(true);
  setIconifiable(true);
  setResizable(true);
  getContentPane().setLayout(new BorderLayout());
  this.tableName=tableName;
  SQLCommandRoot = "CREATE TABLE "+tableName;
  setTitle(SQLCommandRoot);
  init();
  setVisible(true);
}

// initialise the JInternalFrame
private void init(){
  table = createTable(nRows);
  TableChangeListener modelListener = new TableChangeListener ();
  table.getModel().addTableModelListener(modelListener);
  JScrollPane sqlScroller = new JScrollPane(SQLPane);
  JScrollPane tableScroller = new JScrollPane(table);
  JSplitPane splitter = new
    JSplitPane(JSplitPane.VERTICAL_SPLIT,sqlScroller,tableScroller);
  splitter.setDividerLocation(100);
```

```
    getContentPane().add(splitter,BorderLayout.CENTER);
    getContentPane().add(createButton,BorderLayout.SOUTH);
    createButton.addActionListener(new ButtonListener());
  }
  private JTable createTable(int nRows){
    String[] dataTypes = {"CHAR","VARCHAR","INT","FLOAT","DATE"};
    String[] defNull   = {"","NULL","NOT NULL"};
    String[] defUnique = {"","UNIQUE"};
    String[] defPriKey = {"","PRIMARY KEY"};
    String[] colNames  =
      {"Name","DataType","SIZE","NULL","UNIQUE","PRIMARY KEY"};
    String[][] rowData = new String[nRows][colNames.length];
    for(int i=0;i<nRows;i++){
      for(int j=0;j<colNames.length;j++)rowData[i][j]="";
    }
    JComboBox dTypes = new JComboBox(dataTypes);
    JComboBox nullDefs = new JComboBox(defNull);
    JComboBox uniqueDefs = new JComboBox(defUnique);
    JComboBox primaryKDefs = new JComboBox(defPriKey);
    JTable table = new JTable(rowData,colNames);
    table.getColumnModel().getColumn(1).
      setCellEditor(new DefaultCellEditor(dTypes));
    table.getColumnModel().getColumn(3).
      setCellEditor(new DefaultCellEditor(nullDefs));
    table.getColumnModel().getColumn(4).
      setCellEditor(new DefaultCellEditor(uniqueDefs));
    table.getColumnModel().getColumn(5).
      setCellEditor(new DefaultCellEditor(primaryKDefs));
    return table;
  }
  public String parseTable(){
    String tableValues = "";
    int rows = table.getRowCount();
    int cols = table.getColumnCount();
```

```
    if(rows>=0&&cols>=0){
      tableValues += "\n( ";
      for(int i=0;i<rows;i++){
        String rowData = "";
        for(int j=0;j<cols;j++){
          String field = (String)table.getValueAt(i,j);
          if(field!=null){
            if(field.length()==0)break;
            if(j==2)rowData+="(";
            else if(i>0||j>0)rowData += "  ";
            rowData += field;
            if(j==2)rowData+=")";
          }
        }
        if(rowData.length()==0)break;
        tableValues += rowData+",\n";
      }
    }
    if(tableValues.endsWith(",\n")){
      int tvLen = tableValues.length()-2;
      if(tvLen>0)tableValues = tableValues.substring(0,tvLen);
    }
    tableValues += " );";
    return tableValues;
  }

  // CommandListener is set by the MVC Controller module as a call back to
  // return the SQL command
  public void setCommandListener(ActionListener commandListener){
    this.commandListener=commandListener;
  }
  // Listener for the CreateButton
  class ButtonListener implements ActionListener{
```

```
    public void actionPerformed(ActionEvent event){
      String action = event.getActionCommand();
      if(commandListener!=null){
        ActionEvent evt = new ActionEvent(this,0,SQLCommand);
        commandListener.actionPerformed(evt);
      }
    }
  }

  // Listener for Edit events on the JTable
  class TableChangeListener implements TableModelListener{
    public TableChangeListener (){
    }
    public void tableChanged(TableModelEvent event){
      SQLCommand = SQLCommandRoot+parseTable();
      SQLPane.setText(SQLCommand);
    }
   }
}
```

Model

The model portion of the MVC model is nothing more than the JDBC class we build earlier in the chapter. This version has been edited slightly to remove the embedded SQL command strings and the `main()` method we use to test it.

We have also changed the exception handling to fire an ActionEvent to an ExceptionListener registered by the controller, which pops up a JOptionPane to display exceptions from the SQLToolkit rather than printing them to the console. (See Listing 5-9.)

Listing 5-9: DatabaseUtilities — the JDBC code

```
package jdbc_bible.part2;


import java.awt.event.*;
import java.sql.*;
import java.util.Vector;
import sun.jdbc.odbc.JdbcOdbcDriver;
```

```
public class DatabaseUtilities{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String urlRoot = "jdbc:odbc:";
  private ActionListener exceptionListener = null;

  public DatabaseUtilities(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      reportException(e.getMessage());
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public void execute(String SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
```

```
      reportException(e.getMessage());
    }
  }
  public void setExceptionListener(ActionListener exceptionListener){
    this.exceptionListener=exceptionListener;
  }
  private void reportException(String exception){
    if(exceptionListener!=null){
      ActionEvent evt = new ActionEvent(this,0,exception);
      exceptionListener.actionPerformed(evt);
    }else{
      System.err.println(exception);
    }
  }
}
```

Summary

Having read this chapter, you should have a good understanding of the following topics:

- How a relational database stores data in the rows and columns that make up tables
- Records and fields are and how they relate to rows and columns
- Integrity constraints
- Creating a table with `SQL CREATE`
- Removing a table with `SQL DROP`
- Modifying a table with `SQL ALTER`
- Using JDBC and Swing to create a Table Builder

In Chapter 6, we discuss the `SQL INSERT` command and use it to populate our Name and Address Table with data.

Chapter 6: Inserting, Updating,and Deleting Data

In This Chapter

The preceding chapter explains how to create a database and how to add, delete, and modify database tables. This chapter explains how to insert data into a table and, when necessary, modify or delete data.

Related topics covered in this chapter include a review of transaction control, which is important to understand when you are inserting and deleting interdependent data items. Transaction control is also covered at a more theoretical level in Chapter 1.

The use of the SQL commands is illustrated in the context of a series of Java examples, including a discussion of the use of the JDBC `DatabaseMetaData` object to obtain information about a database.

Inserting Data Using SQL INSERT

Once you have created a database and its constituent tables, it is important to know how to add, delete and modify its contents. SQL provides the three following statements you can use to manipulate data within a database:

- `INSERT`
- `UPDATE`
- `DELETE`

INSERT

The `INSERT` statement, in its simplest form, is used to insert data into a table, one row or record at a time. It can also be used in combination with a `SELECT` statement to perform bulk inserts of multiple selected rows from another table or tables. `INSERT` can only be used to insert entire rows into a table, not to insert individual fields directly into a row.

UPDATE

The `UPDATE` command is used to modify the contents of individual columns within a set of rows. The `UPDATE` command is normally used with a `WHERE` clause. As this chapter explains, the `WHERE` clause is used to select the rows to be updated. Clearly, it is important to choose the rows you are updating correctly; otherwise, you may find yourself updating records you have not planned on changing.

DELETE

`DELETE` is used to delete selected rows from a table. As in the case with the `UPDATE` command, row selection is based on the result of an optional `WHERE` clause. Again, you need to be careful when you make the selection, or you may delete records you mean to leave intact.

The INSERT Statement

The basic form of the `INSERT` statement looks like this:

```
INSERT INTO tableName (colName1, colName2, ...) VALUES (value1, value2, ...);
```

To insert name and address information into the Contact_Info Table we create in Chapter 5, use a SQL INSERT statement like this:

```
INSERT INTO Contact_Info
(FName, MI, LName, Email)
VALUES
('Michael','X','Corleone','offers@cosa_nostra.com');
```

Notice how the field names have been specified in the order in which you plan to insert the data. This insert will work just as well if you use the following command:

```
INSERT INTO Contact_Info
(Email, LName, FName, MI)
VALUES
('offers@cosa_nostra.com','Corleone','Michael','X');
```

You can also use a shorthand form if you know the column order of the table. Here's an example:

```
INSERT INTO Contact_Info
VALUES
('Michael','X','Corleone','offers@cosa_nostra.com');
```

Note

String data is specified in single quotes ('), as shown in the examples. Numeric values are specified without quotes.

Follow these rules when inserting data into a table with the INSERT statement:

- The column names you use must match the names defined for the column.
- The values you insert must match the data type defined for the column they are being inserted into. You can't, for example, put string data into a numeric field.
- The data size must not exceed the column width, so you can't put 30 character names into 20 character fields.
- The data you insert into a column must comply with the column's data constraints; for example, you can't put the last names of all members of the Corleone family into a column if you have constrained that column as UNIQUE.

These rules are obvious, but breaking them accounts for a lot of SQL exceptions, particularly when you save data in the wrong field order. Another common error is to try and insert the wrong number of data fields.

When the Contact_Info Table is defined, the MI field is defined as NULLABLE. The correct way to insert a NULL is this:

```
INSERT INTO Contact_Info
(FName, MI, LName, Email)
```

```
VALUES
('Michael',NULL,'Corleone','offers@cosa_nostra.com');
```

Caution

NULL values are not the same as spaces. A NULL value means that the value is empty. It is neither a zero, in the case of an integer, nor a space, in the case of a string.

Using INSERT with JDBC

The code required to use INSERT with JDBC is illustrated in Listing 6-1. This example is similar in appearance to the code of Listing 5-1, which illustrates how to create a table using JDBC. This helps illustrate how the JDBC API provides a means of passing any desired SQL command to a database management system.

Listing 6-1: Using INSERT with JDBC

```
package jdbc_bible.part2;

import java.awt.event.*;
import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;

public class DataInserter{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String urlRoot = "jdbc:odbc:";

  public DataInserter(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
```

```
    catch(ClassNotFoundException e){
      System.err.println(e.getMessage());
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
  }
  public void execute(String SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
  }
  public static void main(String args[]){
    DataInserter inserter = new DataInserter();
    String SQLCommand = "INSERT INTO CONTACT_INFO "+
          "(First_Name,MI,Last_Name,Street,City,State,Zip) "+
          "VALUES "+
          "('Michael','J','Corleone','86 Horsehead
Blvd','NY','NY','12345');";

    inserter.execute(SQLCommand);
  }
}
```

If you compile and execute the example, you should be able to see the new record in your Contact_info Table. Using a DBMS with a GUI-based management console, you are able to see the table and its contents when you open the database. With a command line DBMS such as MySQL, you need to go to the command prompt and type the following command:

```
SELECT * FROM Contact_Info;
```

Using INSERT ... SELECT

The INSERT statement illustrated in the example of Listing 6-1 is primarily intended for inserting records into a table one at a time. For applications such as storing information from membership applications or entering employee records, this is the perfect solution. However, there are times when you want to copy subsets of data from one table to another. On these occasions, doing the transfer one record at a time introduces a lot of overhead because each record has to be individually retrieved from one table and inserted into another other.

SQL allows you to handle these situations by combining the INSERT command with a SELECT command, which queries the database for the desired records. The advantage of this approach is that the whole process is carried out within the RDBMS, avoiding the overhead of retrieving records and reinserting them externally.

The SELECT statement

The SELECT statement is used to query the database for specific rows. This is the basic form of the SELECT statement:

```
SELECT
Field1, Field2, ...
FROM
TableName
[ WHERE ... ];
```

In place of a comma-delimited list of field names, you can supply the asterisk wildcard character, *, to request all fields:

```
SELECT * FROM TableName;
```

Cross-Reference

The SELECT statement is discussed in detail in Chapter 7.

An example of a situation where you might use INSERT...SELECT is the creation of a table containing only the first and last names from the Contact_Info Table. As illustrated in Chapter 5, the SQL command to create the table is as follows:

```
CREATE TABLE Names
(First_Name VARCHAR(20), Last_Name LName VARCHAR(30));
```

To insert the corresponding data from the original Contact_Info Table, use a SQL INSERT...SELECT command to select the desired fields from the Contact_Info Table, and insert them into the new Names Table. Here's an example:

```
INSERT INTO Names
SELECT First_Name, Last_Name FROM Contact_Info;
```

Essentially, this command tells the database management system to perform these two separate operations internally:

A SELECT (to query the Contact_Info Table for the FName and LName fields from all records)

An INSERT (to input the resulting record set into the new Names Table

By performing these operations within the RDBMS, the use of the INSERT...SELECT command eliminates the overhead of retrieving the records and reinserting them.

The WHERE clause

The optional WHERE clause allows you to make conditional queries; for example, you can get all records where the last name is "Corleone" and insert them into the Names Table with this statement:

```
INSERT INTO Names
SELECT First_Name, Last_Name FROM Contact_Info WHERE Last_Name =
'Corleone';
```

Using INSERT ... SELECT with JDBC

As with any other SQL command, it is easy to use INSERT ... SELECT with JDBC. If you substitute the code snippet of Listing 6-2 for the main() of Listing 6-1 and run the example again, you will create a Name Table populated with the first and last names.

Listing 6-2: Using INSERT ... SELECT with JDBC

```
  public static void main(String args[]){
    DataInserter inserter = new DataInserter();

    String SQLCommand = "INSERT INTO NAMES "+
          "SELECT First_Name,Last_Name FROM CONTACT_INFO "+
          "WHERE Last_Name = 'Corleone'; ";
   inserter.execute(SQLCommand);
  }
}
```

Once you have data in a table, you are likely to have to update it to reflect changes in data fields like addresses or inventory item count. The next section shows how to use the SQL UPDATE command to modify data in a table.

The UPDATE Statement

A frequent requirement in database applications is updating records. For example, when a contact moves, you need to change his or her address. Do this with the SQL UPDATE statement, using a WHERE clause to identify the record you want to change. Here's an example:

```
UPDATE Contact_Info
SET Street = '55 Broadway', ZIP = '10006'
WHERE First_Name = 'Michael' AND Last_Name = 'Corleone';
```

This statement first evaluates the WHERE clause to find all records with matching First_Name and Last_Name. It then makes the address change to all of those records.

Caution

If you omit the WHERE clause from the UPDATE statement, all records in the given table are updated.

Using Calculated Values with UPDATE

You can also use the `UPDATE` statement to update columns with calculated values. For example, if you add stock to your inventory, instead of setting the Qty column to an absolute value, you can simply add the appropriate number of units with a calculated `UPDATE` statement like this:

```
UPDATE Inventory
SET Qty = QTY + 24
WHERE Name = 'Corn Flakes';
```

When you use a calculated `UPDATE` statement like this, you need to make sure that you observe the rules for `INSERTS` and `UPDATES` mentioned earlier. In particular, ensure that the data type of the calculated value is the same as the data type of the field you are modifying, as well as short enough to fit in the field.

Common Problems with UPDATE

Two common problems can result from the use of calculated values:

- Truncation can result from number conversions, such as conversion from a real number to an integer.
- Overflow occurs when the resulting value is larger than the capacity of the column. This causes the database system to return an error.

Problems of this type can be avoided if you observe the rules for `INSERTS` and `UPDATES` mentioned earlier.

Listing 6-3: Using UPDATE with JDBC

```
package jdbc_bible.part2;


import java.awt.event.*;
import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;
```

```
public class DataUpdater{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String urlRoot = "jdbc:odbc:";
  private ActionListener exceptionListener = null;

  public DataUpdater(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      System.err.println(e.getMessage());
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
  }
  public void execute(String SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
```

```
    }
  }
  public static void main(String args[]){
    DataUpdater inserter = new DataUpdater();
    String SQLCommand = "UPDATE CONTACT_INFO "+
                        "SET STREET = '58 Broadway', ZIP = '10008' "+
                        "WHERE First_Name = 'Michael' AND "+
                       "Last_Name ='Corleone';";

    inserter.execute(SQLCommand);
  }
}
```

Once again, the basic Java code used to issue the SQL command remains unchanged. To try it out, compile and execute the example; you should be able to see the modified record in your Contact_Info Table.

Transaction Management with COMMIT and ROLLBACK

Transaction management refers to the capability of a relational database management system to execute database commands in groups, known as transactions. A *transaction* is a group or sequence of commands, all of which must be executed in order and must complete successfully. If anything goes wrong during the transaction, the database management system will allow the entire transaction to be cancelled or "rolled back." If, on the other hand, it completes successfully, the transaction can be saved to the database or "committed."

A transaction typically involves several related commands, as in the case of a bank transfer. If *Client A* orders a transfer of funds to *Client B*, at least two database-access commands must be executed:

- Client A's account must be debited.
- Client B's account must be credited.

If one of these commands is executed but the other is not, the funds will either vanish from Client A's account without appearing in Client B's account, or, perhaps worse from the viewpoint of the bank, the funds will be credited to Client B's account without being withdrawn from Client A's account, leaving the bank in the hole.

This situation obviously becomes dramatically more complicated in the real world, where a large financial institution, with hundreds or thousands of users all accessing the database at the same time, can potentially have vast numbers of incomplete transactions active at any given moment.

The solution is to combine logically related commands into groups that are committed as a single transaction. If a problem arises, the entire transaction can be rolled back and the problem fixed without serious adverse impact on business operations.

The primary commands used in managing transactions are `COMMIT` and `ROLLBACK`. As their names suggest, the `COMMIT` command commits changes made from the beginning of the transaction to the point at which the command is issued, and the `ROLLBACK` command undoes them. In addition, most databases support the `AUTOCOMMIT` option, which tells the RDBMS to commit all commands individually, as they are executed. This option can be used with the `SET` command. For example:

```
SET AUTOCOMMIT [ON | OFF] ;
```

By default, the `SET AUTOCOMMIT ON` command is executed at startup, telling the RDBMS to commit all statements automatically as they are executed. If you do not want these commands to be automatically executed, set the `AUTOCOMMIT` option to off as follows:

```
SET AUTOCOMMIT OFF;
```

When you start to work with a transaction, turn `Autocommit` off; then issue the commands required by the transaction, and, assuming that everything executes correctly, commit the transaction using this command:

```
COMMIT;
```

If any problems should arise during the transaction, you can cancel the entire transaction by using the following command:

```
ROLLBACK;
```

Note

Transaction-management syntax varies considerably from one database management system to the next, but the basic syntax shown previously is supported by all common database management systems.

The use of `COMMIT` and `ROLLBACK` in a JDBC example is very straightforward. Here's a modification to the example of Listing 6-3, which specifically turns `Autocommit` on. Simply insert the `con.setAutoCommit(true)` line into the `stmt.execute(SQLCommand)` method, as shown:

```
  public void execute(String SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      con.setAutoCommit(true);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
      System.err.print(e.getMessage());
    }
  }
```

Adding the `setAutoCommit(true)` line tells the database management system to commit all changes automatically. If you compile and execute the modified code, you should get exactly the same results as you do when you run the original example.

Now modify the code to turn `Autocommit` off, using `setAutoCommit(false)`, as shown here:

```
public void execute(String SQLCommand){
  String url = urlRoot+dbName;
  try {
    Connection con = DriverManager.getConnection(url);
    con.setAutoCommit(false);
    Statement stmt = con.createStatement();
    stmt.execute(SQLCommand);
    con.close();
  }
  catch(SQLException e){
    System.err.print(e.getMessage());
  }
}
```

This time, when you run the example, it throws an "Invalid Transaction State" exception, and the update has not been made. The exception is thrown because we have not terminated the transaction before closing the connection.

Now alter the code in the try block to the following; the change is made as before, because we have specifically told the database management system to commit the change:

```
  try {
    Connection con = DriverManager.getConnection(url);
    con.setAutoCommit(false);
    Statement stmt = con.createStatement();
    stmt.execute(SQLCommand);
    con.commit();
    con.close();
  }
```

If you change the try block by replacing the `con.commit()` with `con.rollback()`, the change will be rolled back, so no change will be visible. This time, however, no exception is thrown, as you can see here:

```
  try {
    Connection con = DriverManager.getConnection(url);
```

```
      con.setAutoCommit(false);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      //con.commit();
      con.rollback();
      con.close();
    }
```

You can check to see if the `UPDATE` has been executed by inserting a `SELECT` statement to read the updated value of the Street field after the update command is executed but before it is rolled back. The try block now looks like this:

```
    try {
      Connection con = DriverManager.getConnection(url);

      con.setAutoCommit(false);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);

      String query = "SELECT Street FROM Contact_Info "+
                     "WHERE First_Name = 'Michael' AND Last_Name ='Corleone';";
      ResultSet rs = stmt.executeQuery(query);
      rs.next();
      System.out.println("Street = "+rs.getString(1));

      con.rollback();
      con.close();
    }
```

When you run this version, it shows that the new value of Street matches the update, but when you look in the database, the previous value is still there because the change has been rolled back.

Cross-Reference

`RecordSets` and the `SELECT` are discussed in detail in Chapter 7.

The DELETE Statement

The last data-manipulation command is `DELETE`, which is used for deleting entire records or groups of records. Again, when using the `DELETE` command, you use a `WHERE` clause to identify the records to be deleted.

Use of the DELETE command is very straightforward. For example, this is the command you use to delete records containing the First_Name: "Michael" and the Last_Name: "Corleone":

```
DELETE FROM Contact_Info
WHERE First_Name = 'Michael' AND Last_Name = 'Corleone';
```

Caution

INSERT, DELETE, and UPDATE can cause referential integrity problems with other tables, as well as significant problems within the table you are working on. Delete with care.

A Swing-Based Table Editor

To illustrate the topics covered in this chapter, the Swing-based table builder created in Chapter 5 is extended by the addition of a table editor (see Figure 6-1). The table editor is based on components derived from components built in Chapter 5. A new Edit menu (with Insert, Update, Delete, JMenuItems) and a new JTable in a JInternalFrame (for handling the Insert, Edit, and Delete functions) are also added.

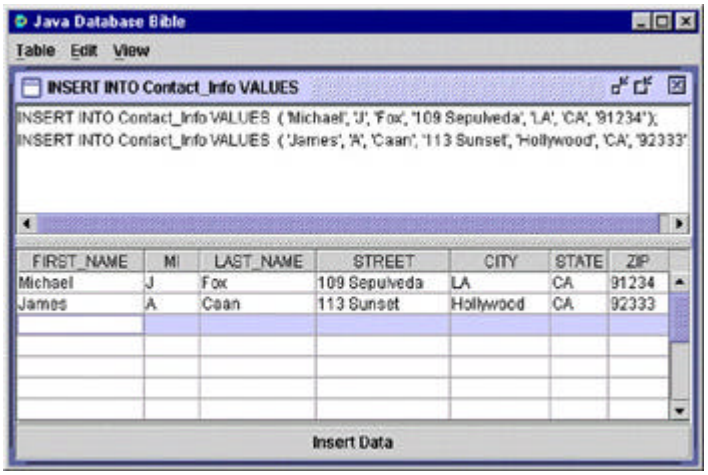


Figure 6-1: Inserting data with SQL INSERT

The events are as follows:

1. The user selects a database.
2. The user selects an action: Insert, Update, or Delete.
3. The user selects the table.
4. A TableEdit frame is displayed for user interaction.
5. A SQL command is created dynamically and executed on command.

The first step in building the table editor is to create the Edit menu by subclassing the DBMenu convenience class. The DBMenuItems Insert, Update, and Delete to the Edit menu are added and hooked into the Jframe, which forms the basis of the MainFrame class.

Listing 6-4: Edit menu with insert, update, and delete items

```
package jdbc_bible.part2;

import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;

public class EditMenu extends DBMenu{
  JMenuItem insertItem;
  JMenuItem updateItem;
  JMenuItem deleteItem;
  JMenuItem exitItem;

  public EditMenu(){
    setText("Edit");
    setActionCommand("Edit");
    setMnemonic((int)'E');

    insertItem   = new DBMenuItem("Insert",'I',itemListener,false);
    updateItem  = new DBMenuItem("Update",'U',itemListener,false);
    deleteItem = new DBMenuItem("Delete",'D',itemListener,false);

    add(insertItem);
    add(updateItem);
    add(deleteItem);
  }
}
```

As discussed in Chapter 5, the `DBMenu` base class and the `DBMenuItem` class are simply convenience classes for building menus. Using these convenience classes simplifies the menu code considerably.

TableEditFrame

`TableEditFrame`, shown in Listing 6-5, is very similar to the `TableBuilderFrame` discussed in Chapter 5. It extends `JInternalFrame` and contains a `JTable` used to set up the fields for the

database table. It also contains a `JTextArea`, which provides a preview of the generated SQL command, and an "Insert Data" button.

Listing 6-5: TableEditFrame

```
package jdbc_bible.part2;

import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
import java.util.EventListener;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

/**
* TableEditFrame extends JInternalFrame to create a display which builds
* SQL CREATE statements
* <p/>
* TableBuilder, which extends JTable, is a key component.
*/

class TableEditFrame extends JInternalFrame{
  protected JTable table;
  protected JTextArea SQLPane = new JTextArea();
  protected JButton insertButton = new JButton("Insert Data");
  protected DatabaseUtilities dbUtils;
  protected String tableName = null;
  protected String colNames[] = null;
  protected String dataTypes[] = null;
  protected String SQLCommand[] = null;
  protected String SQLCommandRoot = "";
```

```
public TableEditFrame(String tableName, DatabaseUtilities dbUtils){
  setSize(600,400);
  setLocation(10,10);
  setClosable(true);
  setMaximizable(true);
  setIconifiable(true);
  setResizable(true);
  getContentPane().setLayout(new BorderLayout());
  this.tableName=tableName;
  this.dbUtils=dbUtils;
  SQLCommandRoot = "INSERT INTO "+tableName+" VALUES ";
  setTitle(SQLCommandRoot);
  init();
  setVisible(true);
}

// initialise the JInternalFrame
private void init(){
  colNames = dbUtils.getColumnNames(tableName);
  dataTypes = dbUtils.getDataTypes(tableName);
  table = createTable(colNames,15);
  TableChangeListener modelListener = new TableChangeListener();
  table.getModel().addTableModelListener(modelListener);
  JScrollPane sqlScroller = new JScrollPane(SQLPane);
  JScrollPane tableScroller = new JScrollPane(table);
  JSplitPane splitter = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                    sqlScroller,tableScroller);
  splitter.setDividerLocation(100);
  getContentPane().add(splitter,BorderLayout.CENTER);
  getContentPane().add(insertButton,BorderLayout.SOUTH);
  insertButton.addActionListener(new ButtonListener());
}
```

```
protected JTable createTable(String[] colNames,int nRows){
  String[][] rowData = new String[nRows][colNames.length];
  for(int i=0;i<nRows;i++){
    for(int j=0;j<colNames.length;j++)rowData[i][j]="";
  }
  JTable table = new JTable(rowData,colNames);
  return table;
}



public Vector parseTable(){
  int rows = table.getRowCount();
  int cols = table.getColumnCount();
  Vector tableValues = new Vector();

  if(rows>=0&&cols>=0){
    for(int i=0;i<rows;i++){
      String rowData = "";
      for(int j=0;j<cols;j++){
        String field = (String)table.getValueAt(i,j);
        if(field.length()>0){
          field = fixApostrophes(field);
          if(j>0)rowData += ", ";
          if(dataTypes[j].equals("CHAR")||
             dataTypes[j].equals("VARCHAR"))
            rowData += "'"+field+"'";
          else
            rowData += field;
        }
      }
      if(rowData.length()==0)break;
       tableValues.addElement(" ( " + rowData + " );\n");
    }
  }
```

```
    return tableValues;
  }

  private String fixApostrophes(String in){
    int n=0;
    while((n=in.indexOf("'",n))>=0){
      in = in.substring(0,n)+"'"+in.substring(n);
      n+=2;
    }
    return in;
  }

  // Listener for the Insert Button
  class ButtonListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      dbUtils.execute(SQLCommand);
    }
  }

  // Listener for Edit events on the JTable
  class TableChangeListener implements TableModelListener{
    public TableChangeListener(){
    }
    public void tableChanged(TableModelEvent event){
      Vector rowData = parseTable();
      SQLCommand = new String[rowData.size()];
      SQLPane.setText("");
      for(int i=0;i<rowData.size();i++){
        if(rowData.elementAt(i)==null)break;
        SQLCommand[i] = SQLCommandRoot+(String)rowData.elementAt(i);
        SQLPane.append(SQLCommand[i]);
      }
    }
  }
```

```
}
```

The `parseTable()` method has been modified slightly and now returns a vector of Strings. This change supports the ability to issue several `SQL INSERT` commands as a result of a one-button click.

An additional change has been made to the `TableChangeListener`, which now accesses the `DatabaseUtilities` class directly rather than through the event system. Again, this has been done to support the ability to issue several SQL commands in response to a button click.

The Controller Class

The `DatabaseManager` class is shown in Listing 6-6. It is based on the class used in Chapter 5. It incorporates additional code to hook in the new menu and a new method, `displayTableEditFrame()`, to display the new `JInternalFrame, TableEditFrame.`

Listing 6-6: DatabaseManager — Controller class

```
package jdbc_bible.part2;


import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;


public class DBManager extends JFrame{
  JMenuBar menuBar = new JMenuBar();
  JDesktopPane desktop = new JDesktopPane();
  String database = null;
  String tableName = null;
  String menuSelection = null;
  TableBuilderFrame tableMaker = null;
  TableEditFrame tableEditor = null;    // added for Chapter 6
  DatabaseUtilities dbUtils = null;


  TableMenu tableMenu = new TableMenu();
  EditMenu editMenu = new EditMenu();   // added for Chapter 6
```

```
MenuListener menuListener = new MenuListener();

public DBManager(){
  setJMenuBar(menuBar);
  setTitle("JDBC Database Bible");
  getContentPane().setLayout(new BorderLayout());
  getContentPane().add(desktop,BorderLayout.CENTER);
  setSize(new Dimension(640,480));

  menuBar.add(tableMenu);
  tableMenu.setMenuListener(menuListener);

  menuBar.add(editMenu);                  // added for Chapter 6
  editMenu.setMenuListener(menuListener);

  setVisible(true);
}

private void displayTableBuilderFrame(){
  tableName = JOptionPane.showInputDialog(this,"Table:",
            "Select table",JOptionPane.QUESTION_MESSAGE);
  tableMaker = new TableBuilderFrame(tableName);
  tableMaker.setCommandListener(new CommandListener());
  desktop.add(tableMaker);
  tableMaker.setVisible(true);
}

private void displayTableEditFrame(){  // added for Chapter 6
  tableName = JOptionPane.showInputDialog(this,"Table:",
            "Select table",JOptionPane.QUESTION_MESSAGE);
  tableEditor = new TableEditFrame(tableName,dbUtils);
  desktop.add(tableEditor);
  tableEditor.setVisible(true);
}
```

```
  private void selectDatabase(){
    database = JOptionPane.showInputDialog(this,
              "Database:","Select database",
              JOptionPane.QUESTION_MESSAGE);
    dbUtils = new DatabaseUtilities();
    dbUtils.setExceptionListener(new ExceptionListener());

    tableMenu.enableMenuItem("New Table",true);
    tableMenu.enableMenuItem("Drop Table",true);

    editMenu.enableMenuItem("Insert",true);
    editMenu.enableMenuItem("Update",true);
    editMenu.enableMenuItem("Delete",true);
  }

  private void executeSQLCommand(String SQLCommand){
    dbUtils.execute(SQLCommand);
  }

  private void dropTable(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
              "Select table",JOptionPane.QUESTION_MESSAGE);
    int option = JOptionPane.showConfirmDialog(null,
              "Dropping table "+tableName,
              "Database "+database,
              JOptionPane.OK_CANCEL_OPTION);
    if(option==0){
      executeSQLCommand("DROP TABLE "+tableName);
    }
  }

  class MenuListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
```

```
        String menuSelection = event.getActionCommand();
        if(menuSelection.equals("Database")){
          selectDatabase();
        }else if(menuSelection.equals("New Table")){
          displayTableBuilderFrame();
        }else if(menuSelection.equals("Drop Table")){
          dropTable();
        }else if(menuSelection.equals("Insert")){
          displayTableEditFrame();
        }else if(menuSelection.equals("Exit")){
          System.exit(0);
        }
      }
    }

    class ExceptionListener implements ActionListener{
      public void actionPerformed(ActionEvent event){
        String exception = event.getActionCommand();
        JOptionPane.showMessageDialog(null, exception,
         "SQL Error", JOptionPane.ERROR_MESSAGE);
      }
    }

    class CommandListener implements ActionListener{
      public void actionPerformed(ActionEvent event){
        String SQLCommand = event.getActionCommand();
        executeSQLCommand(SQLCommand);
     }
    }

    public static void main(String args[]){
      DBManager dbm = new DBManager();
    }
  }
```

~~One of the most useful tools provided by JDBC is the capability to retrieve information a~~bout the data returned in a `ResultSet`. This information is obtained using the JDBC `ResultSetMetaData` object reviewed in the next section.

JDBC ResultSetMetaData

In addition, two methods have been added that use the `ResultSetMetaData` class to get information about the table being edited. The two `MetaData` objects that follow are capable of returning the table information required:

- `DatabaseMetaData`, which returns information at the database level
- `ResultSetMetaData`, which returns information at the `ResultSet` level

The reason for using the `ResultSetMetaData` object in this example is to restrict the column information to just the columns being displayed and to defer discussion of the `DatabaseMetaData` object until `ResultSets` have been discussed, since it makes heavy use of `ResultSets` to return information.

JDBC `ResultSetMetaData` provides access to different kinds of information about the data in a table, including column names and data types. Some of the most useful `ResultSet MetaData` methods are the following:

- `int getColumnCount()`
- `String getColumnName(int column)`
- `String getColumnTypeName(int column)`

The following usage is very straightforward. To get the names of all columns in a table, for example, a simple query is executed to return a `ResultSet` used to get the `ResultSetMetaData`. This is then queried for the desired information.

```
String SQLCommand = "SELECT * FROM "+tableName+";";


try {
  Connection con = DriverManager.getConnection(url);
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery(SQLCommand);
  ResultSetMetaData md = rs.getMetaData();


  String[] columnNames = new String[md.getColumnCount()];
  for(int i=0;i<columnNames.length;i++){
    columnNames[i] = md.getColumnLabel(i+1);
  }
  con.close();
}
```

Cross-Reference

`ResultSetMetaData` methods and usage are discussed in Chapter 4; usage examples are in Chapter 10.

In the expanded version of the `DatabaseUtilities` class shown in Listing 6-7, a second version of the `execute()` method has been added. This new version accepts a String array argument so that it can loop through a number of `SQL INSERT` commands.

Listing 6-7: DatabaseUtilities — JDBC code

```
package jdbc_bible.part2;

import java.awt.event.*;
import java.sql.*;
import java.util.Vector;
import sun.jdbc.odbc.JdbcOdbcDriver;

public class DatabaseUtilities{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String urlRoot = "jdbc:odbc:";
  private ActionListener exceptionListener = null;

  public DatabaseUtilities(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      reportException(e.getMessage());
    }
```

```
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public void execute(String SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public void execute(String[] SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      for(int i=0;i<SQLCommand.length;i++){
        stmt.execute(SQLCommand[i]);
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public String[] getColumnNames(String tableName){
    Vector dataSet = new Vector();
    String[] columnNames = null;
    String url = urlRoot+dbName;
```

```
    String SQLCommand = "SELECT * FROM "+tableName+";";

    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(SQLCommand);
      ResultSetMetaData md = rs.getMetaData();

      columnNames = new String[md.getColumnCount()];
      for(int i=0;i<columnNames.length;i++){
        columnNames[i] = md.getColumnLabel(i+1);
       }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
    return columnNames;
  }
  public String[] getDataTypes(String tableName){
    Vector dataSet = new Vector();
    String[] dataTypes = null;
    String url = urlRoot+dbName;
    String SQLCommand = "SELECT * FROM "+tableName+";";

    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(SQLCommand);
      ResultSetMetaData md = rs.getMetaData();

      dataTypes = new String[md.getColumnCount()];
      for(int i=0;i<dataTypes.length;i++){
        dataTypes[i] = md.getColumnTypeName(i+1);
```

```
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
    return dataTypes;
  }
  public void setExceptionListener(ActionListener exceptionListener){
    this.exceptionListener=exceptionListener;
  }
  private void reportException(String exception){
    if(exceptionListener!=null){
      ActionEvent evt = new ActionEvent(this,0,exception);
      exceptionListener.actionPerformed(evt);
    }else{
      System.err.println(exception);
    }
  }
}
```

Summary

In this chapter, you learned about:

- Using `SQL INSERT` to populate a table
- Using `SQL UPDATE` to modify the contents of a table
- Using `SQL DELETE` to delete records from a table
- Using the `SELECT` clause and how to use it with `INSERT`
- Using the `WHERE` clause and how to use it with `UPDATE` and `DELETE`
- Applying the basics of transaction control with `COMMIT` and `ROLLBACK`
- Using JDBC `ResultSetMetaData` to get information about a table
- Using JDBC and Swing to create a JDBC/SQL table editor

Chapter 7 discusses retrieving data from a database by using the `SELECT` command.

Chapter 7: Retrieving Data withSQL Queries

In This Chapter

One of the most important functions of any database application is finding the records in the database tables and returning them in the desired form. The process of finding and returning formatted records is known as *querying* the database. This chapter will explore the use of the SQL SELECT command to query the database created and populated in Chapters 5 and 6.

The SELECT Statement

The SELECT statement is the heart of a SQL query. In addition to its use in returning data in a query, it can be used in combination with other SQL commands to select data for a variety of other operations, such as modifying specific records using the UPDATE command.

The most common use of SELECT, however, is as the basis of data-retrieval commands, or queries, to the database. A simple query specifies the names of the columns to be returned and the name of the table they can be found in. A basic SELECT command looks like this:

```
SELECT columnName1, columnName2,.. FROM tableName;
```

The SQL command for selecting the First Name and Last Name of each entry in the Contact_Info table would be as follows:

```
SELECT First_Name, Last_Name FROM Contact_Info;
```

In addition to this specific form, where the names of all the fields you want returned are specified in the query, SQL also supports the following wild-card form:

```
SELECT * FROM tableName;
```

The wild card, *, tells the database-management system to return the values for all columns.

Using the WHERE Clause

The real power of the SELECT command comes from the WHERE clause, which allows you to query the database for specific data. You will have noticed that each of the commands shown above returns values for all rows. A practical query needs to be more restrictive, returning the requested fields from only those records that match specific criteria. For example, the WHERE clause enables you to retrieve all records with a Last_Name Corleone from the Contact_Info table shown in Table 7-1.

Table 7-1: The CONTACT_INFO Table

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Michael	A	Corleone	123 Pine	New York	NY	10006
Fredo	X	Corleone	17 Main	New York	NY	10007
Sonny	A	Corleone	123 Walnut	Newark	NJ	12346

Table 7-1: The CONTACT_INFO Table

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Francis	X	Corleone	17 Main	New York	NY	10005
Vito	G	Corleone	23 Oak St	Newark	NJ	12345
Tom	B	Hagen	37 Chestnut	Newark	NJ	12345
Kay	K	Adams	109 Maple	Newark	NJ	12345
Francis	F	Coppola	123 Sunset	Hollywood	CA	23456
Mario	S	Puzo	124 Vine	Hollywood	CA	23456

To retrieve all records containing the last name Corleone, you could use the following query:

```
SELECT * FROM Contact_Info WHERE Last_Name = 'Corleone';
```

The result of this query will be to return all columns from any row containing the Last_Name Corleone. The order in which the columns are returned will be the order in which they are stored in the database, although the row order is arbitrary.

Note

Unlike rows in a spreadsheet, records in a database table have no implicit order. You must specify explicitly any ordering you need.

To retrieve columns in a specific order, the column names must be specified in the query. For example, to get the data in First_Name, Last_Name order, use the following query:

```
SELECT First_Name, Last_Name FROM Contact_Info WHERE Last_Name =
'Corleone';
```

To get the order reversed, use the following query:

```
SELECT Last_Name, First_Name FROM Contact_Info WHERE Last_Name =
'Corleone';
```

Formatting SQL Commands

The SQL engine ignores excess white space, so you can and should insert line breaks for clarity. Conventionally, major clauses such as the `FROM` clause and the `WHERE` clause are placed on their own lines, unless the command is so brief as to be trivial. For example, many Relational Database Management Systems (RDBMS) such as SQL Server, format commands in the SQL pane automatically to conform to this style. A good basic approach when you are not quite sure how to format a command is to aim for readability. Remember, somebody will have to maintain what you write, so readability is important.

Key words, table names, and column names are not case sensitive, but the contents of the records within a table are case sensitive. This means that with a little thought, you can use case to help make your SQL statements more readable.

Caution

Although SQL ignores case in commands, table names, column names, and so on, case can matter when you are using a name in a `WHERE` clause. Thus, 'Corleone' and 'CORLEONE' are not necessarily the same. You should read the documentation for your particular DBMS.

While the simple SELECT statements discussed so far in this chapter give you an idea of what can be done in a SQL query, you are likely to need to use more complex queries in practice. The next section discusses creating more complex queries.

Using Operators in More Complex WHERE Clauses

The queries discussed so far have been very simple, but in practice you will frequently be using queries that depend on the values of a number of fields in various combinations. SQL provides a number of operators that enable you to create more complex queries based on value comparisons.

In practice, many queries will require the evaluation of more than a single condition or test. In such cases *operators* are used in the `WHERE` clause to specify a combination of conditions which must be evaluated. SQL has the following types of operators:

- `DISTINCT`
- `TOP`
- Comparison operators
- Character comparison
- Logical
- Arithmetic
- `IN` and `BETWEEN`
- Set operators

Note

There is also a keyword `ALL`, as in `SELECT ALL`, but since `ALL` is implied unless `DISTINCT` is used, the expression SELECT ALL is rarely, if ever, used in practice.

The DISTINCT Operator

A basic `SELECT` statement tells the database-management system to return all records matching the query in the `ResultSet`. For example, you could request all Last_Names from Contact_Info using this query:

```
SELECT Last_Name
FROM Contact_Info;
```

Using the data shown in Table 7-1 would give you Corleone repeated five times.

The `DISTINCT` operator tells the database-management system not to return duplicate records in a `ResultSet`. For example, to return all Last_Names from the Contact_Info table with no duplicates, you would use this query:

```
SELECT DISTINCT Last_Name
FROM Contact_Info;
```

When this operator is applied to the results, you would only see the Last_Name Corleone once, despite the fact that there are several different Corleones in the table.

Note

The `DISTINCT` operator is very resource intensive, so you might want to consider filtering duplicates when iterating over the `ResultSet`.

The TOP Operator

The `TOP` operator specifies that only the first *n* rows are to be output from the query result set, or, optionally, the top *n* percent of the rows. When specified with `PERCENT`, *n* must be an integer between 0 and 100, as shown in the following code:

```
SELECT TOP 25 PERCENT *
FROM Inventory;
```

The result set from running this query against a table containing 12 rows is shown in Table7-2.

Table 7-2: ResultSet Containing the TOP 25 Percent of the Table

ID	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	130	1.95
1002	Rice Krispies	Cereal	97	1.87
1003	Shredded Wheat	Cereal	103	2.05

If the query includes an `ORDER BY` clause, the first *n* rows (or *n* percent of rows) ordered by the `ORDER BY` clause are output. If the query has no `ORDER BY` clause, the order of the rows is arbitrary.

Cross-Reference

The `ORDER BY` clause is discussd in Chapter 8.

Comparison Operators

SQL supports the following standard comparison operators, as well as a special operator used to test for a `NULL` value in a column:

- Equality (=)
- Inequality (<>)
- Greater than (>) and greater than or equal to (>=)
- Less than (<) and less than or equal to (<=)
- IS NULL
- IS NOT NULL

Using the equals and not equals operators

Comparison operators work on strings as well as on numbers. Thus, to find all records in the Contact_Info database with a Last_Name of Corleone, you would use an *equals* (=) query like the following:

```
SELECT * FROM Contact_Info WHERE Last_Name = 'Corleone';
```

Conversely, to find all records in the Contact_Info database with any other Last_Name, you would use a *not equals* (<>) query like this:

```
SELECT * FROM Contact_Info WHERE Last_Name <> 'Corleone';
```

Using the greater than and less than operators

The greater than (>) and less than (<) operators can also be used for lexical comparison of `CHAR` and `VARCHAR` values, so to find all records in the Contact_Info database with a Last_Name that comes after Corleone alphabetically, you would use a query like this:

```
SELECT * FROM Contact_Info WHERE Last_Name > 'Corleone';
```

Similarly, you can combine the greater than and equals operators to find all records in the Contact_Info database with a last name including or after Corleone in the alphabet. Here's an example:

```
SELECT * FROM Contact_Info WHERE Last_Name >= 'Corleone';
```

Using the IS NULL Operator

As mentioned in Chapters 5 and 6, in the discussions about creating and populating database tables, the value in a field can sometimes be `NULL`, indicating that there is nothing in the field. It is important to understand that this really does mean nothing, rather than, for example, a value of zero in the case of a numeric field, or white space in the case of a `CHAR` or `VARCHAR` field.

Since the `NULL` represents an absence of data, it can't be evaluated using Greater Than, Equals, or Less Than. SQL provides a special `IS NULL` operator to test for `NULL`. If, for example, you added a column to the Contact_Info table for Cell Phone numbers, leaving it `NULL` when you don't have a contact's cell phone number, you could query the table for contacts without cell phones using this code:

```
SELECT * FROM Contact_Info WHERE Cell_Phone IS NULL;
```

Using the IS NOT NULL operator

Another common requirement is to find records where a specific field is `IS NOT NULL`. For example, to query the Contact_Info table for contacts with cell phones you could use this code:

```
SELECT * FROM Contact_Info WHERE Cell_Phone IS NOT NULL;
```

Note

You can't test for `NULL` using equality (=) or inequality (<>) operators, since, by definition, there is nothing in the field.

CHAR and VARCHAR Operators

In addition to letting you use the comparison operators to work with strings, SQL adds these dedicated string operators for use with `CHAR` and `VARCHAR` variables:

- `LIKE`
- `NOT LIKE`
- String concatenation

Using the LIKE and NOT LIKE operators

The `LIKE` operator — and its negation, the `NOT LIKE` operator — combined with the wild card provide a very powerful tool for string comparison. The wild cards are as follows:

- Underscore (_), the single character wild card
- Percent (%), the multi-character wild card

For example, to find all records in the Contact_Info table with last name starting with "C," you would write a query using `LIKE` as follows:

```
SELECT * FROM Contact_Info WHERE Last_Name LIKE 'C%';
```

Similarly, to find all records where the Last_Name contains the letter "o" in the second position, the query would look like this:

```
SELECT * FROM Contact_Info WHERE Last_Name LIKE '_o%';
```

`NOT LIKE` works in very much the same way as `LIKE`. For example, to find all records in the Contact_Info table with last name `NOT` starting with the letter "C," you would write a query using `NOT LIKE` as follows:

```
SELECT * FROM Contact_Info WHERE Last_Name NOT LIKE 'C%';
```

Using the concatenation operator

The concatenation operator is used to concatenate two strings. It is represented by the symbol, +, in SQL, Access, and Sybase; Oracle accepts || as the concatenation operator. For example, to return the last name followed by the first name separated by commas, you would use the following query:

```
SELECT Last_Name + ', ' + First_Name AS NAME FROM Contact_Info;
```

Caution

The concatenation operator is one of the SQL features that varies from one flavor of SQL to another (as mentioned above). It is frequently worth checking the documentation for the version of SQL you are using when you encounter problems.

Logical Operators

SQL provides several logical operators to combine two or more conditions in the `WHERE` clause of a SQL statement. These logical operators are as follows:

- `AND`
- `OR`
- `NOT`

Using the AND operator

The `AND` operator is used to combine two or more comparisons, all of which must evaluate to `TRUE` for the comparison to be valid. If any of the expressions are false, `AND` returns `FALSE`. For example, to find all records in the Contact_Info table with a Last_Name of Corleone who live in New York, you would use this query:

```
SELECT * FROM Contact_Info WHERE Last_Name = 'Corleone' AND City = 'New York';
```

Using the OR operator

The OR operator is used to combine two or more comparisons, any one of which can evaluate to TRUE for the comparison to be valid. For example, to find all records in the Contact_Info table who live in New York City or in New Jersey, you would use this query:

```
SELECT * FROM Contact_Info WHERE City = 'New York' OR State = 'NJ';
```

Combining logical operators using parentheses

Like arithmetic operators, logical operators can be combined using parentheses (()). For example, to find all to find all records in the Contact_Info table with a last name of Corleone who live in New York City or in New Jersey, you would use this query:

```
SELECT * FROM Contact_Info
WHERE Last_Name = 'Corleone' AND ( City = 'New York' OR State = 'NJ' );
```

Using the NOT operator

The NOT operator is used to reverse the result of a comparison. If the condition it applies to evaluates to TRUE, using the NOT operator makes it FALSE. Conversely, if the condition after the NOT is FALSE, it becomes TRUE when you use the NOT operator. For example, to find all to find all records in the Contact_Info table with a last name of Corleone who do not live in New York City or in New Jersey, you would use this query:

```
SELECT * FROM Contact_Info
WHERE Last_Name = 'Corleone' AND NOT ( City = 'New York' OR State = 'NJ' );
```

Arithmetic Operators

SQL supports the common arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/). In addition, SQL supports the modulo operator (%), which returns the remainder from the division of one integer by another.

Note

The modulo operator only works with integers. Dividing a float by a valid divisor always gives a float, and thus, no remainder.

Table 7-3 shows a very simple inventory. This inventory will be used in discussing how to work with arithmetic operators.

Table 7-3: Inventory

ID	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	130	1.95
1002	Rice Krispies	Cereal	97	1.87
1003	Shredded Wheat	Cereal	103	2.05
1004	Oatmeal	Cereal	15	0.98

Table 7-3: Inventory

ID	Name	Description	Qty	Cost
1005	Chocolate Chip	Cookies	217	1.26
1006	Fig Bar	Cookies	162	1.57
1007	Sugar Cookies	Cookies	276	1.03
1008	Cola	Soda	144	0.61
1009	Lemon Soda	Soda	96	0.57
1010	Orange Soda	Soda	84	0.71

The first, and most obvious use of arithmetic operators is in the `WHERE` clause. The following example uses the `LESS THAN` operator to identify items that are running low:

```
SELECT *
FROM INVENTORY
WHERE Qty < 24;
```

The preceding query will return the following result:

ID	Name	Description	Qty	Cost
1004	Oatmeal	Cereal	15	0.98

Creating calculated result fields

Another very useful application of arithmetic operators is to create a calculated result field as part of the results returned from a query. For example, you can calculate a retail price by marking up a cost as follows:

```
SELECT ID,Name,Description,Cost,Cost*1.6 AS Retail
FROM Inventory;
```

This query returns the additional column (or field) "Retail," as shown in Table 7-4.

Table 7-4: Calculated Result Fields

ID	Name	Description	Cost	Retail
1001	Corn Flakes	Cereal	1.95	3.12
1002	Rice Krispies	Cereal	1.87	2.992
1003	Shredded Wheat	Cereal	2.05	3.28
1004	Oatmeal	Cereal	0.98	1.568
1005	Chocolate Chip	Cookies	1.26	2.016

Table 7-4: Calculated Result Fields

ID	Name	Description	Cost	Retail
1006	Fig Bar	Cookies	1.57	2.512
1007	Sugar Cookies	Cookies	1.03	1.648
1008	Cola	Soda	0.61	0.976
1009	Lemon	Soda	0.57	0.912
1010	Orange	Soda	0.71	1.136

Aliases

In the preceding example, the expression command uses the key word AS. Using the optional AS clause enables you to assign a meaningful name, or *alias*, to an expression, which makes it easier to refer back to the expression later on. An alias can be used as a normal column name when you need to refer to the column elsewhere in a statement, as you will see in examples later in the book. In this example, AS assigned the name (alias) "Retail" to the calculated value column.

When assigning and using an alias, you must bear in mind the order in which SQL processes the various clauses constituting the command, since the output of one clause is the input to the next one. The order in which the subclauses of a SQL command are processed is shown in the following list:

- FROM clause
- WHERE clause
- GROUP BY clause
- HAVING clause
- SELECT clause
- ORDER BY clause

Since you used AS to assign an alias in the SELECT clause, you can't use the alias as part of the predicate in a WHERE clause, since the WHERE clause has already been executed by the time you get to the SELECT. The alias can, however, be used in an ORDER BY, if, for example, you wanted to order the inventory table by Retail, as shown here:

```
SELECT ID,Name,Description,Cost,Cost*1.6 AS Retail
FROM Inventory ORDER BY Retail;
```

Tip

When you create a calculated field in a result, you should always use AS to assign a name to the field. This is because there is no defined naming convention for calculated fields in SQL. Different variants of SQL assign different arbitrary names.

Arithmetic operators can also be used in the WHERE clause. For example, to list only items whose retail price is below 100, you would use the following code:

```
SELECT Name,Description,Cost,Cost*1.6 AS Retail
FROM Inventory
```

```
WHERE Cost * 1.6 < 100;
```

You can also create more complex calculations as required. The following query will return the profit on each item as well as the retail price:

```
SELECT Name,Description,Cost,Cost*1.6 as Retail,Cost*1.6 - Cost AS Profit
FROM Inventory
WHERE Cost * 1.6 < 100;
```

The preceding code will generate the results in Table 7-5.

Table 7-5: More Complex Calculated Columns

Name	Description	Cost	Retail	Profit
Cola	Soda	0.61	0.976	0.366
Lemon	Soda	0.57	0.912	0.342

Miscellaneous Operators: IN and BETWEEN

The `IN` operator provides a simple way to compare fields against a list. For example, to find contacts in New York State or New Jersey, you can use this query:

```
SELECT *
FROM Contact_Info
WHERE State IN ('NY', 'NJ');
```

`IN` also works with numbers. For example, if you wanted to select items from the inventory table by `ID`, you could use this query:

```
SELECT *
FROM Inventory
WHERE ID IN (1001, 1003, 1004);
```

The `BETWEEN` operator, as its name suggests, helps you select fields with values that fall between specified limits. Referring again to the Inventory table (Table 7-3), you can query for items with costs in the $1.03–$1.95 range using the query. Here's an example:

```
SELECT *
FROM Inventory
WHERE Cost BETYOUEN 1.03 AND 1.95;
```

Note

`BETWEEN` returns values *within* its defined range inclusive of the limits, so if you try the query against the Inventory table, it will return rows with costs of 1.03 and 1.95.

Set Operators

Set operators allow you to combine `ResultSets` returned by different queries into a single `ResultSet`. These are the main set operators:

- `UNION` returns the combined results of two queries.
- `INTERSECT` returns only the rows found by both queries.
- `EXCEPT` returns the rows from the first query that are not present in the second.

Caution

The `INTERSECT` and `EXCEPT` operators are not supported by all SQL dialects. `UNION`, together with the variant `UNION ALL`, works on most SQL versions.

Using UNION and UNION ALL

`UNION ALL` returns the results of two queries. `UNION` does the same thing, but it removes duplicate results. Let's say you wanted to invite all the New York and New Jersey Corleones to a party and introduce them to Kay Adams. You could use a `UNION` to combine the two queries into one guest list. Here's an example:

```
SELECT *
FROM Contact_Info
WHERE Last_Name = 'Corleone' AND (City = 'New York' OR
    State = 'NJ')
UNION
SELECT *
FROM contact_info
WHERE first_name = 'Kay';
```

`UNION`, used by itself, returns the results of the two queries without any repetitions. `UNION ALL`, on the other hand, returns the results of the two queries including all repetitions.

Using INTERSECT and EXCEPT

The `INTERSECT` and `EXCEPT` operators adhere to the same syntax as the `UNION` operator. You should check with the documentation for the DBMS you are using to ensure that these operators are supported before committing to using one of them.

Escape Sequences

Escape sequences are of valuable use in situations where a character has a particular meaning in SQL, and you want to use that character in a different way. A typical example is the use of the apostrophe (').

A problem that arises when handling names is the use of the apostrophe in names of Irish origin. Since the apostrophe is, in effect, a single quote (), SQL reads it as a `CHAR` or `VARCHAR` terminator and throws a SQL error when it tries to handle the rest of the string. This problem also arises fairly frequently in normal free-form text.

The solution is simple: you simply double up the apostrophe, as you have seen in the method `fixApostrophes()`in earlier chapters. Here's an example:

```
String fixApostrophes(String in){
  int n=0;
   while((n=in.indexOf("'",n))>=0){
     in = in.substring(0,n)+"'"+in.substring(n);
     n+=2;
   }
  return in;
}
```

This simple fix is worth implementing, as it's very annoying for an Irishman to be told he has spelled his name incorrectly (as frequently happens to me when logging on to a Web site). It's also quite surprising how frequently apostrophes appear in normal text (as this paragraph demonstrates).

Two other characters that require escape sequences are:

- Percent (%)
- Underscore (_)

These are handled by defining an escape character at the end of the query in which the characters are used.

The escape character is defined in curly braces ({}) using the keyword `escape`, as follows:

```
{escape 'escape-character'}
```

For example, the following query finds names that begin with an underscore. It uses the backslash character as an escape character:

```
SELECT name
FROM variables
WHERE Id LIKE `\_%' {escape '\'};
```

Subqueries

A *query* is a SQL command that uses the `SELECT` keyword to return an array of data fields from one or more tables. A *subquery* is simply a query that is used as part of another SQL statement. Subqueries can be nested inside any of the following types of SQL statements:

- `SELECT` or `SELECT...INTO`
- `INSERT...INTO`
- `DELETE`
- `UPDATE`
- Inside another subquery

You can use a subquery in a WHERE or HAVING clause or, in rarer instances, you can use a subquery instead of an expression in the field list of a SELECT statement.

In a subquery you use a SELECT statement to provide a set of one or more specific values to evaluate in the WHERE or HAVING clause expression, or to provide the returned values of a SELECT command directly, as part of the SELECT list.

Subqueries can be used in WHERE or HAVING clauses as the right-hand side of:

- A comparison using ANY, ALL, or SOME
- An expression using IN or NOT IN
- An expression using EXISTS or NOT EXISTS

Using the ANY, SOME, and ALL Predicates

In many cases, a subquery used in a comparison will return more than one value. Because of this, you need special predicates to operate on the results of the subquery before making the comparison. For example, if you want to find out which inventory items cost more than cookies, you could use a subquery like this:

```
(SELECT cost FROM inventory
 WHERE Description = 'Cookies');
```

The result of this subquery will be several rows of cookie costs, so you will need to select which cost you want to use. The ANY or SOME predicates, which are synonymous, can be used to retrieve records in the main query that satisfy the comparison with any records retrieved in the subquery. The following example returns all inventory items with a cost greater than the lowest cost cookies in the Inventory table:

```
SELECT * FROM INVENTORY
WHERE cost >= ANY
       (SELECT cost FROM inventory
        WHERE Description = 'Cookies');
```

The ALL predicate can be used to retrieve only those records in the main query that satisfy the comparison with all records retrieved in the subquery. If you changed ANY to ALL in the preceding example, the query would return only those inventory items that cost more than all cookies, as illustrated in Figure 7-1.

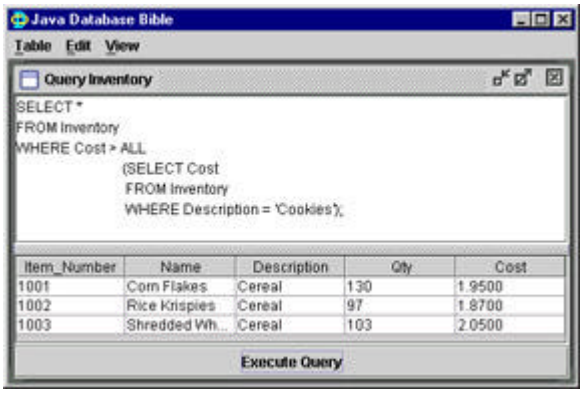


Figure 7-1: Subquery using ALL

Note

When creating a subquery, the entire subquery should be enclosed in parentheses.

Using the IN and NOT IN Predicates

The `IN` predicate is used to retrieve the records in the main query that have a matching record in the data set returned by the subquery. This usage is similar to the simple `SELECT` used with an `IN` list, as shown here:

```
SELECT * FROM CUSTOMERS
WHERE STATE IN ('NY','NJ');
```

In this example, the query returns all Customers where the State field is listed in the parenthesized `IN` list, or, in other words, where the State field equals either NY or NJ.

The example shown in Figure 7-2 returns all items from the Inventory table whose item numbers can be found in the Ordered_Items table, with Order_Number = 2.

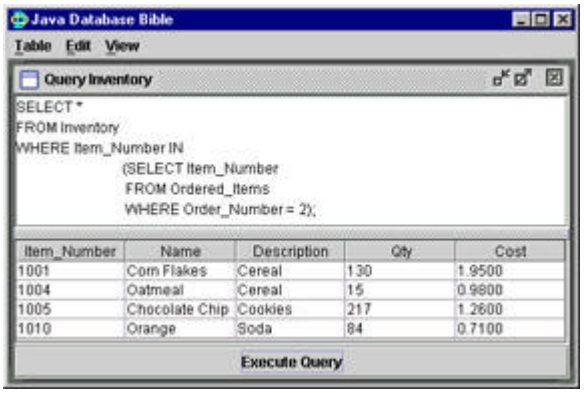


Figure 7-2: Subquery using IN

The `NOT IN` predicate, of course, reverses the selection.

Note

You can only specify one `SELECT` list item when using the `IN` predicate, since the list is returned for comparison with a single item.

Using the EXISTS and NOT EXISTS Predicates

The EXISTS and NOT EXISTS predicates are used in true/false comparisons to determine whether the subquery returns any records. You will use EXISTS in a subquery to find out what kinds of cookies have been ordered by members of the Corleone family.

The main query shown in Figure 7-3 selects the first and last names of the family member, and the name of the preferred type of cookie for all instances of a cookie preference returned by executing an EXISTS subquery on the tables.

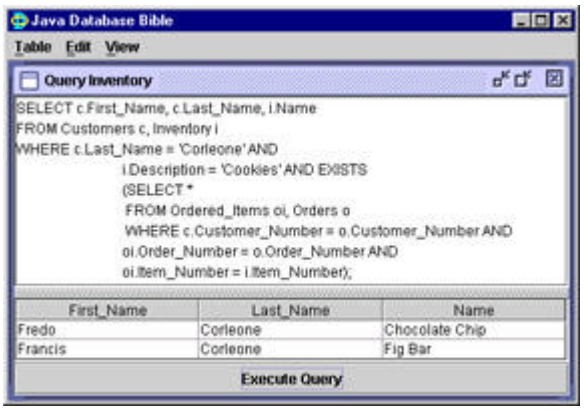


Figure 7-3: Subquery using EXISTS

Note

Conventionally, you use an asterisk(*) with the EXISTS predicate because EXISTS only returns true or false so there is nothing to be gained by being more specific.

As a rule, the main FROM list should only contain tables that are referenced in the main SELECT statement. In this case, you listed Customers and Inventory in the main SELECT statement and used their aliases in combination with the Orders and Ordered_Items tables in the subquery.

The EXISTS statement stops the search as soon as it finds a single match. The EXIST statement is therefore much faster and more efficient than a query that continues to check for additional rows that match.

Cross-Reference

You can also use table name aliases in a subquery to refer to tables listed in a FROM clause outside the subquery, as in the example in Figure 7-3. This capability, known as a correlated subquery, is discussed later in this chapter

Nesting Subqueries

Just as you can use a subquery within a query, you can also use a subquery within another subquery. Subqueries can be nested as deeply as your implementation of SQL allows. The syntax for nesting subqueries looks like this:

```
SELECT *

FROM Tables
```

```
WHERE
 (SUBQUERY
  (SUBQUERY
   (SUBQUERY)));
```

For example, to send out special notices to customers who spend more than the average amount of money, you could build a customer list by creating a query using two nested subqueries, as shown in Figure 7-4.

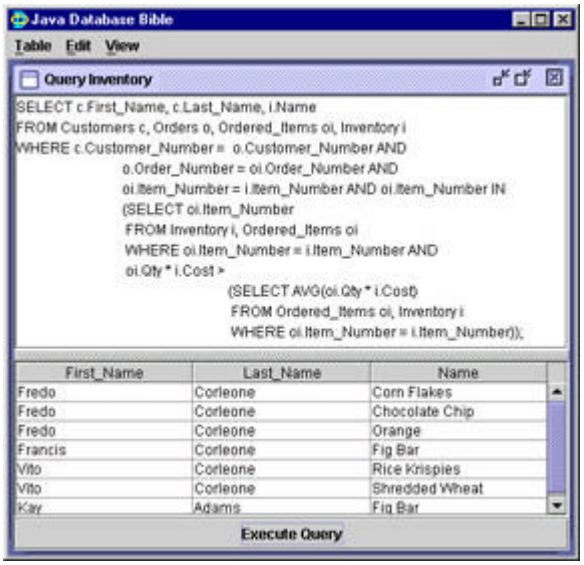


Figure 7-4: Using nested subqueries

Testing Subqueries

Recall that one of the nice things about subqueries is that they are easy to test as queries before plugging them into larger queries. For example, the subquery that calculates the average cost of a purchase is very straightforward. Here's an example:

```
(SELECT AVG(oi.Qty * i.Cost)
FROM Ordered_Items oi, Inventory i
WHERE oi.Item_Number = i.Item_Number)
```

In Figure 7-5 you see the two subqueries combined to generate a list of all purchases above the average cost. Note that the additional columns oi.Order_Number and oi.Qty * i.Cost AS 'Total' have been added to make it easier to check the queries.

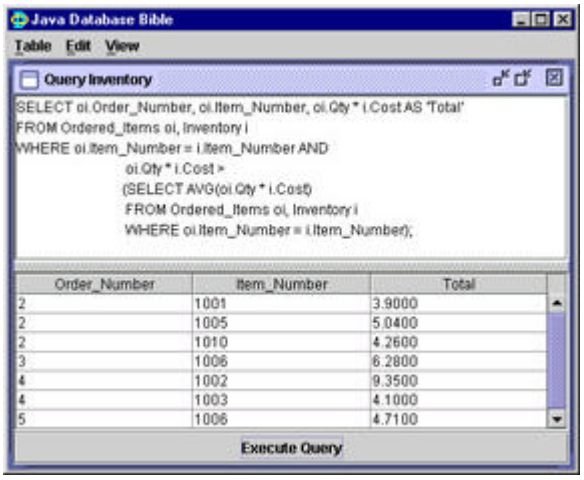


Figure 7-5: Subquery to find above average purchases

The next section discusses a number of additional ways in which subqueries can be used.

Additional Uses of Subqueries

Earlier examples discussed how you can use calculated values, or even literals, in place of simple data field values in the `SELECT` clause of a query, as in the following example:

```
SELECT 'Average Cost' AS Name, AVG(oi.Qty * i.Cost)
    AS 'AVG'
FROM Ordered_Items oi, Inventory i
WHERE oi.Item_Number = i.Item_Number;
```

This query returns the following result:

Name	AVG
Average Cost	3.7045

Using a Subquery in the SELECT List

You can also use results returned by subqueries in the `SELECT` list of a query. This can be useful if you want to create a summary table of items by category, as might be the case with the Inventory table. If, for example, you wanted to tabulate the average cost of various types of products in the inventory, you could use a query with subqueries in the command line, like the example shown in Figure 7-6.

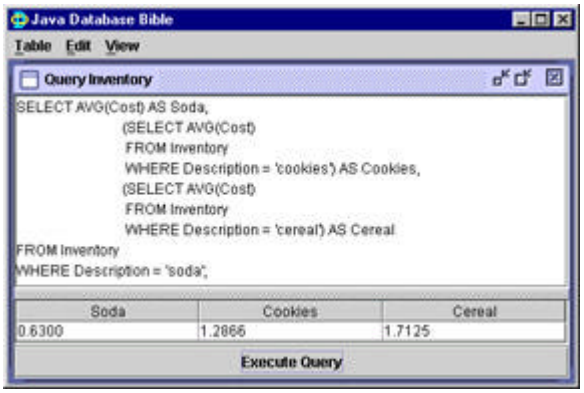


Figure 7-6: Using dubqueries in the SELECT clause

Notice how the entire subquery replaces the column name, so that the alias clauses used to name the columns appear outside the parentheses defining the subqueries.

Using a Subquery with the INSERT Command

You can use subqueries in the `INSERT` command just as easily as you can in a `SELECT` command. Consider an example where you might want to insert selected records from one table into another. One way to do this is to use a subquery to select the desired subset from the source table.

The following example uses a subquery to select the Customer_Numbers of customers from New Jersey. Then the appropriate fields are selected from Customers with the selected Customer_Numbers and are inserted into the Employees table.

```
INSERT INTO Employees
    (Employee_ID, First_Name, Last_Name)
SELECT Customer_Number, First_Name, Last_Name
FROM Customers
WHERE Customer_Number IN
       (SELECT Customer_Number
     FROM Customers
     WHERE State = 'NJ');
```

Using a Subquery with the UPDATE Command

A more common usage of the subquery is with the `UPDATE` command. One advantage of using a subquery is that you can very easily test the subquery by itself to make sure you are getting the correct data set. Then, once it checks out OK, you can plug it into the actual update.

The following example uses a subquery to select the Customer_Number of the customer to be updated from the Customers table. You then use this customer number in the `WHERE` clause of the `UPDATE` command.

```
UPDATE Employees
```

```
SET First_Name = 'Alfie'
WHERE Employee_ID IN
       (SELECT Customer_Number
      FROM Customers
      WHERE First_Name = 'Sonny');
```

Using a Subquery with the DELETE Command

Finally, here's an example of the use of a subquery with the `DELETE` command. This example uses a subquery to select the Customer_Numbers of all the customers with a Last_Name of Corleone. This list of Customer_Numbers is used in the `DELETE` command to identify the customers to be deleted from the Customers table. In this instance, you will get a list of all Corleones in the Customer table, regardless of whether they are employees. You then use this customer number list in the `WHERE` clause of the `DELETE` command and delete any employees in the list.

```
DELETE FROM Employees
WHERE Employee_ID IN
     (SELECT Customer_Number
      FROM Customers
      WHERE Last_Name = 'Corleone');
```

Correlated Subqueries

Most of the subqueries discussed so far are self-contained, in that they refer only to tables defined within the subquery itself. This self-contained aspect of subqueries has the advantage of making them easy to check out as stand-alone queries. However, sometimes it's useful to use outside references in a subquery.

Correlated subqueries are subqueries that depend on a value in the outer query. A reference to a table in the outer query is called a *correlated reference*. The following example presents a correlated query in the reference to the Customers table. In the following code, Customers appears in the `FROM` clause of the outer query, but not in the `FROM` clause of the subquery:

```
SELECT c.First_Name, c.Last_Name, i.Name, i.Item_Number
FROM Customers c, Inventory i
WHERE c.Last_Name = 'Corleone' AND
    i.Description = 'Cookies' AND EXISTS
       (SELECT *
      FROM Ordered_Items oi, Orders o
      WHERE c.Customer_Number = o.Customer_Number AND
          oi.Order_Number = o.Order_Number AND
          oi.Item_Number = i.Item_Number);
```

Correlated queries are executed repeatedly (once for each row of the table identified in the outer-level query), so they can be extremely inefficient. It is frequently worthwhile to rewrite correlated queries as joins wherever possible, though in some cases the SQL engine may be able to optimize the correlated subquery.

The next section explains how the SQL queries discussed in this chapter can be used in a JDBC application.

JDBC ResultSets

The JDBC `ResultSet` holds the data, arranged in rows and columns, returned by a query. A `ResultSet` maintains a cursor that points to the current row of data. The cursor moves down one row each time the `next()`method is called. You access the data by sequencing through the rows and requesting data from the columns using `getter` methods, either by column name or by column number. In general, using the column number will be more efficient than using the column name .

Caution

Columns are numbered from 1, not from 0.

The JDBC `ResultSet` provides getter methods that convert column data from SQL data types to the specified Java types. Each `getter` method comes in these two flavors:

- `getXXx(String columnName)`
- `getXXX(int columnNumber)`

For clarity only one variant is shown in the `getter` method summary in Table 7-6.

Table 7-6: ResultSet getter Methods

Data Type	Method
BigDecimal	`getBigDecimal(String columnName, int scale)`
boolean	`getBoolean(String columnName)`
byte	`getByte(String columnName)`
byte[]	`getBytes(String columnName)`
double	`getDouble(String columnName)`
float	`getFloat(String columnName)`
int	`getInt(String columnName)`
java.io.InputStream	`getAsciiStream(String columnName)`
java.io.InputStream	`getUnicodeStream(String columnName)`
java.io.InputStream	`getBinaryStream(String columnName)`
java.sql.Date	`getDate(String columnName)`
java.sql.Time	`getTime(String columnName)`

Table 7-6: ResultSet getter Methods

Data Type	Method
java.sql.Timestamp	`getTimestamp(String columnName)`
long	`getLong(String columnName)`
Object	`getObject(String columnName)`
short	`getShort(String columnName)`
String	`getString(String columnName)`

Caution

Each column can be read only once with getter method. Subsequent reads return unpredictable results.

A `ResultSet` maintains a cursor that points to the current row of data. Initially the cursor is positioned before the first row. The `next()` method moves the cursor to the next row and must be called before the first `getter` method is called.

When you access data with a basic, nonscrollable `ResultSet`, the table rows are retrieved sequentially. `ScrollableResultSets` add absolute positioning and reverse scrolling capabilities to the basic `ResultSet`. Within a row, you can access the column values in any order.

For the "`getter`" methods, the JDBC driver attempts to convert the underlying data to the specified Java type and returns a suitable Java value. Column names used as input to "`getter`" methods are case insensitive, in accordance with normal SQL rules.

Caution

When performing a "`getXXX`" using a column name, if several columns have the same name, the value of the first matching column will be returned.

A basic `ResultSet` is automatically closed by the statement that generated it when that statement is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results. This behavior may be modified in some of the JDBC Extension API `ResultSets`, as discussed in Chapter 4.

ResultSetMetaData

The `ResulSetMetaData` object returned by the `getMetaData()` method provides information about a `ResultSet`'s columns, such as number, types, and properties,. Chapter 4 discusses the `ResulSetMetaData` object in some detail.

The following are some of the methods available to access `ResultSetMetaData`:

- `getColumnCount()` — returns the number of columns in the ResultSet
- `getColumnLabel(int column)` — returns the column title for use in printouts and displays
- `getColumnName(int column)` — returns the column name
- `getColumnTypeName(int column)` – returns the name of the column's SQL data type

With just these four methods you have enough information to display the results of any query in a meaningful way.

Using SELECT to return RecordSets with JDBC

The procedure for retrieving data from a database is very similar to the procedure you used to insert data, with the exception that, since this is a query, you need to define a `ResultSet` to hold the returned data. In addition to the `ResultSet`, you are also defining a `ResultSetMetaData` object, which will hold information about the `ResultSet`. You will use this object to get the number of columns returned, since the `getData` method does not have any information regarding the query it is executing.

For the purposes of the example in Listing 7-1, you will simply loop through the `ResultSet` and print the data to the system console.

Listing 7-1: Data Retrieval using JDBC

```
package java_databases.part2;

import java.awt.event.*;
import java.sql.*;
import java.util.Vector;
import sun.jdbc.odbc.JdbcOdbcDriver;

public class DataRetriever{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String urlRoot = "jdbc:odbc:";
  private ActionListener exceptionListener = null;

  public DataRetriever(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
```

```
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      reportException(e.getMessage());
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public String[][] executeQuery(String SQLQuery){
    Vector dataSet = new Vector();
    String url = urlRoot+dbName;

    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(SQLQuery);
      ResultSetMetaData md = rs.getMetaData();

      int nColumns = md.getColumnCount();
      while(rs.next()){
        String[] rowData = new String[nColumns];
        for(int i=0;i<nColumns;i++){
          rowData[i] = rs.getObject(i+1).toString();
        }
        dataSet.addElement(rowData);
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
    String[][] records = new String[dataSet.size()][];
```

```
    for(int i=0;i<records.length;i++){
      records[i]=(String[])dataSet.elementAt(i);
    }
    return records;
  }
  public void setExceptionListener(ActionListener exceptionListener){
    this.exceptionListener=exceptionListener;
  }
  private void reportException(String exception){
    if(exceptionListener!=null){
      ActionEvent evt = new ActionEvent(this,0,exception);
      exceptionListener.actionPerformed(evt);
    }else{
      System.err.println(exception);
    }
  }

  public static void main(String args[]){
    DataRetriever retriever = new DataRetriever();
    retriever.setDatabaseName("Contacts");
    String[][] records =
      retriever.executeQuery("SELECT * FROM Contact_Info");
    for(int i=0;i<records.length;i++){
      String[] record = records[i];
      for(int j=0;j<record.length;j++){
        if(j>0)System.out.print("\t");
        System.out.print(record[j]);
      }
      System.out.println();
    }
  }
}
```

The main difference between the code required to retrieve data from the table and the code you used to insert it is the use of the `ResultSet` and `ResultSetMetaData` objects. The other difference is

that you need to use the `executeQuery()` method of the `Statement` object rather than the `execute()` method when you expect a `ResultSet` to be returned. Initially, the `ResultSet`'s cursor is positioned before the first row of the returned data, so you need to execute the `ResultSet.next()` method to point the cursor to the first row.

The `ResultSet.next()` method returns a `boolean false` if it advances the cursor beyond the end of the `ResultSet`. This makes it suitable as the basis of a `WHILE` loop to loop through the entire `ResultSet`, as shown here in Listing 7-1. Listing 7-1 also shows the use of `ResultSetMetaData` to get the number of columns in the `ResultSet`.

Caution

`ResultSet` columns count from 1, not from 0, so an exception will be thrown if you forget this and try to use a loop which counts columns from column 0.

The next section continues the development of the JDBC Swing example started in Chapters 5 and 6, adding the capability to execute queries.

A Swing-Based SQL Query Pane

To illustrate the topics covered in this chapter, the Swing-based Table Builder will be extended by the addition of a Query Pane (see Figure 7-7). The Query Pane is based on components you built in Chapter 5. You will add a new View Menu to allow us to display the Query Pane, and a new JInternalFrame for handling the Queries.

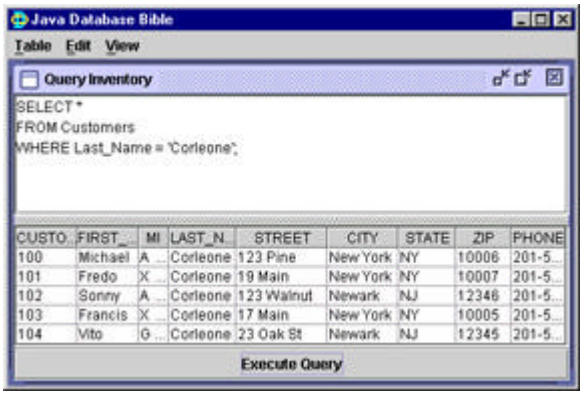


Figure 7-7: SQL Query Pane

The View Menu

The View menu extends the DBMenu class, adding DBMenuItems for the ResultSet that you are working with in this chapter. Listing 7-2 shows the necessary code.

Listing 7-2: View menu with ResultSet item

```
package jdbc_bible.part2;
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;

public class ViewMenu extends DBMenu{
  JMenuItem resultSetItem;
  JMenuItem scrollableResultSetItem;
  JMenuItem updatableResultSetItem;
  JMenuItem rowSetItem;

  public ViewMenu(){
   setText("View");
   setActionCommand("View");
   setMnemonic((int)'V');

   resultSetItem   = new DBMenuItem("ResultSet",'R',itemListener,false);
   scrollableResultSetItem
        = new DBMenuItem("Scrollable ResultSet",'S',itemListener,false);
   updatableResultSetItem
        = new DBMenuItem("Updatable ResultSet",'U',itemListener,false);
   rowSetItem  = new DBMenuItem("RowSet",'W',itemListener,false);

    add(resultSetItem);
    add(scrollableResultSetItem);
    add(updatableResultSetItem);
    add(rowSetItem);
  }
}
```

TableQueryFrame

`TableQueryFrame` is very similar to the `TableBuilderFrame` discussed in Chapter 5. It extends `JInternalFrame` and contains a `Jtable`, which is used to display the fields returned in the `ResultSet`, a `JTextArea` that provides an editable text area in which you can create queries, and an

"Execute Query" button. Otherwise, this class is simpler than its counterparts in preceding chapters as you no longer need a `parseTable()` method or a `TableChangeListener`.

The `JTable` is preloaded using the SQL query as shown in the following:

```
"SELECT TOP 5 * FROM " + tableName;
```

You use the `TOP 5` limitation to prevent having to load a huge `JTable` in cases where the database table is large. Obviously, you can change this to suit your own application.

The `TableQueryFrame` class is different from its counterparts in previous chapters primarily because it is driven by the `JTextArea` rather than by the `JTable`. The `JTextArea` is used to enter free form SQL queries for execution when the "Execute Query" button is clicked.

The sequence of events involved in using the `TableQueryFrame` (shown in Listing 7-3) example is as follows:

1. User selects a database.
2. User selects "View ResultSet".
3. User selects the table.
4. A `TableQueryFrame` is displayed showing the top five records from the table.
5. A SQL command is typed into the `JTextArea` and executed on command.

This example extends the examples of Chapters 5 and 6 to create a Swing-based application that can connect to any database-management system. This example can be used to create and populate tables, and to execute any of the queries discussed in this chapter. The TableQueryFrame code is shown in Listing 7-3.

Listing 7-3: TableQueryFrame

```
package jdbc_bible.part2;


import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
import java.util.EventListener;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;


/**
* TableQueryFrame extends JInternalFrame to create a display which builds SQL
```

```
 * CREATE statements
 */
 class TableQueryFrame extends JInternalFrame{

   protected JTable table;
   protected JScrollPane tableScroller;
   protected JTextArea SQLPane = new JTextArea();
   protected JButton queryButton = new JButton("Execute Query");
   protected DatabaseUtilities dbUtils;

   protected String tableName = null;
   protected String colNames[] = null;
   protected String dataTypes[] = null;
   protected String SQLQuery = null;
   protected String SQLCommandRoot = "";

   public TableQueryFrame(String tableName, DatabaseUtilities dbUtils){
     System.out.println(tableName+",  "+dbUtils);
     setSize(600,400);
     setLocation(10,10);
     setClosable(true);
     setMaximizable(true);
     setIconifiable(true);
     setResizable(true);
     getContentPane().setLayout(new BorderLayout());
     this.tableName=tableName;
     this.dbUtils=dbUtils;
     setTitle("Query "+tableName);
     init();
     setVisible(true);
   }

   // initialize the JInternalFrame
   private void init(){
```

```
    colNames = dbUtils.getColumnNames(tableName);
    dataTypes = dbUtils.getDataTypes(tableName);
    SQLQuery = "SELECT TOP 5 * FROM "+tableName;
    Vector dataSet = dbUtils.executeQuery(SQLQuery);
    table = createTable(colNames,dataSet);
    JScrollPane sqlScroller = new JScrollPane(SQLPane);
    tableScroller = new JScrollPane(table);
    JSplitPane splitter = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                        sqlScroller,tableScroller);
    splitter.setDividerLocation(100);
    getContentPane().add(splitter,BorderLayout.CENTER);
    getContentPane().add(queryButton,BorderLayout.SOUTH);
    queryButton.addActionListener(new ButtonListener());
  }
  protected JTable createTable(String[] colNames,Vector dataSet){
    int nRows = dataSet.size();
    String[][] rowData = new String[nRows][colNames.length];
    for(int i=0;i<nRows;i++){
      Vector row = (Vector)dataSet.elementAt(i);
      for(int j=0;j<row.size();j++)
        rowData[i][j]=((Object)row.elementAt(j)).toString();
    }
    JTable table = new JTable(rowData,colNames);
    return table;
  }
  // Listener for the Query Button
  class ButtonListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      SQLQuery = SQLPane.getText();
      JViewport viewport = tableScroller.getViewport();
      viewport.remove(table);
      colNames = dbUtils.getColumnNamesUsingQuery(SQLQuery);
      Vector dataSet = dbUtils.executeQuery(SQLQuery);
      table = createTable(colNames,dataSet);
```

```
      viewport.add(table);
    }
  }
}
```

Changes to the `DBManager` class (Listing 7-4) are once again minimal, amounting to no more than adding the hooks for the menu.

Listing 7-4: DBManager

```
package jdbc_bible.part2;

import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;

public class DBManager extends JFrame{
  JMenuBar menuBar = new JMenuBar();
  JDesktopPane desktop = new JDesktopPane();
  String database = null;
  String tableName = null;
  String menuSelection = null;
  TableBuilderFrame tableMaker = null;
  TableEditFrame tableEditor = null;    // added for Chapter 6
  TableQueryFrame tableQuery = null;    // added for Chapter 7
  DatabaseUtilities dbUtils = null;

  TableMenu tableMenu = new TableMenu();
  EditMenu editMenu = new EditMenu();   // added for Chapter 6
  ViewMenu viewMenu = new ViewMenu();   // added for Chapter 7

  MenuListener menuListener = new MenuListener();

  public DBManager(){
```

```
    setJMenuBar(menuBar);
  setTitle("Java Database Bible");
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(desktop,BorderLayout.CENTER);
    setSize(new Dimension(480,320));

    menuBar.add(tableMenu);
    tableMenu.setMenuListener(menuListener);

    menuBar.add(editMenu);                  // added for Chapter 6
    editMenu.setMenuListener(menuListener);

    menuBar.add(viewMenu);                  // added for Chapter 7
    viewMenu.setMenuListener(menuListener);

    setFont(new Font("Dialog",Font.PLAIN,18));
    setVisible(true);
    Font font = getGraphics().getFont();
    System.out.println(font);
  }

  private void displayTableBuilderFrame(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
              "Select table",JOptionPane.QUESTION_MESSAGE);
    tableMaker = new TableBuilderFrame(tableName);
    tableMaker.setCommandListener(new CommandListener());
    desktop.add(tableMaker);
    tableMaker.setVisible(true);
  }

  private void displayTableEditFrame(){  // added for Chapter 6
    tableName = JOptionPane.showInputDialog(this,"Table:",
              "Select table",JOptionPane.QUESTION_MESSAGE);
    tableEditor = new TableEditFrame(tableName,dbUtils);
```

```
    desktop.add(tableEditor);
    tableEditor.setVisible(true);
  }

  private void displayTableQueryFrame(){  // added for Chapter 7
    tableName = JOptionPane.showInputDialog(this,"Table:",
              "Select table",JOptionPane.QUESTION_MESSAGE);
    tableQuery = new TableQueryFrame(tableName,dbUtils);
    desktop.add(tableQuery);
    tableQuery.setVisible(true);
  }

  private String[] parseKeyValueString(String kvString){
    String[] kvPair = null;

    int equals = kvString.indexOf("=");
    if(equals>0){
      kvPair = new String[2];
      kvPair[0] = kvString.substring(0,equals).trim();
      kvPair[1] = kvString.substring(equals+1).trim();
    }
    return kvPair;
  }

  private void selectDatabase(){
    database = JOptionPane.showInputDialog(this,"Database:",
             "Select database",JOptionPane.QUESTION_MESSAGE);
    dbUtils = new DatabaseUtilities();
    dbUtils.setDatabaseName(database);
    dbUtils.setExceptionListener(new ExceptionListener());

    tableMenu.enableMenuItem("New Table",true);
    tableMenu.enableMenuItem("Drop Table",true);
```

```
    editMenu.enableMenuItem("Insert",true);
    editMenu.enableMenuItem("Update",true);
    editMenu.enableMenuItem("Delete",true);

    viewMenu.enableMenuItem("ResultSet",true);
  }

  private void executeSQLCommand(String SQLCommand){
    dbUtils.execute(SQLCommand);
  }

  private void dropTable(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
              "Select table",JOptionPane.QUESTION_MESSAGE);
  int option = JOptionPane.showConfirmDialog(null,
              "Dropping table "+tableName,
              "Database "+database,
              JOptionPane.OK_CANCEL_OPTION);
  if(option==0){
      executeSQLCommand("DROP TABLE "+tableName);
    }
  }

  class MenuListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      String menuSelection = event.getActionCommand();
      if(menuSelection.equals("Database")){
        selectDatabase();
      }else if(menuSelection.equals("New Table")){
        displayTableBuilderFrame();
      }else if(menuSelection.equals("Drop Table")){
        dropTable();
      }else if(menuSelection.equals("Insert")){
        displayTableEditFrame();
```

```
      }else if(menuSelection.equals("ResultSet")){ // added for Chapter 7
        displayTableQueryFrame();
      }else if(menuSelection.equals("Exit")){
        System.exit(0);
      }
    }
  }

  class ExceptionListener implements ActionListener{
   public void actionPerformed(ActionEvent event){
    String exception = event.getActionCommand();
    JOptionPane.showMessageDialog(null, exception,
        "SQL Error", JOptionPane.ERROR_MESSAGE);
    }
  }

  class CommandListener implements ActionListener{
   public void actionPerformed(ActionEvent event){
    String SQLCommand = event.getActionCommand();
      executeSQLCommand(SQLCommand);
    }
  }

   public static void main(String args[]){
     DBManager dbm = new DBManager();
   }
}
```

It now remains to add the necessary JDBC code to run the query, as discussed in the next section.

JDBC Code

In the extended version of the `DatabaseUtilities` class in Listing 7-5, the method `executeQuery(String SQLQuery)` has been added to return a `Vector` of `Vectors` containing the row data from the table. The choice of a `Vector` of `Vectors` is driven partly by the inherent flexibility it offers, and partly to demonstrate an approach that differs slightly from Listing 7-1. The method `getColumnNamesUsingQuery(String SQLCommand)` has also been added. This method returns a

String array of column names pertinent to the query, rather than all the column names for the entire table.

Listing 7-5: DatabaseUtilities

```
package jdbc_bible.part2;

import java.awt.event.*;
import java.sql.*;
import java.util.Vector;
import sun.jdbc.odbc.JdbcOdbcDriver;

public class DatabaseUtilities{
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String dbName = "Contacts";
  static String urlRoot = "jdbc:odbc:";
  private ActionListener exceptionListener = null;

  public DatabaseUtilities(){
    registerDriver();
  }
  public void setDatabaseName(String dbName){
    this.dbName=dbName;
  }
  public void registerDriver(){
    try {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      reportException(e.getMessage());
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
```

```
  }
  public void execute(String SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      stmt.execute(SQLCommand);
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public void execute(String[] SQLCommand){
    String url = urlRoot+dbName;
    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      for(int i=0;i<SQLCommand.length;i++){
        stmt.execute(SQLCommand[i]);
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
  }
  public String[] getColumnNames(String tableName){
    Vector dataSet = new Vector();
    String[] columnNames = null;
    String url = urlRoot+dbName;
    String SQLCommand = "SELECT * FROM "+tableName+";";

    try {
```

```
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(SQLCommand);
      ResultSetMetaData md = rs.getMetaData();

      columnNames = new String[md.getColumnCount()];
      for(int i=0;i<columnNames.length;i++){
        columnNames[i] = md.getColumnLabel(i+1);
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
    return columnNames;
  }
  public String[] getColumnNamesUsingQuery(String SQLCommand){
    Vector dataSet = new Vector();
    String[] columnNames = null;
    String url = urlRoot+dbName;

    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(SQLCommand);
      ResultSetMetaData md = rs.getMetaData();

      columnNames = new String[md.getColumnCount()];
      for(int i=0;i<columnNames.length;i++){
        columnNames[i] = md.getColumnLabel(i+1);
      }
      con.close();
    }
    catch(SQLException e){
```

```
      reportException(e.getMessage());
    }
    return columnNames;
  }
  public String[] getDataTypes(String tableName){
    Vector dataSet = new Vector();
    String[] dataTypes = null;
    String url = urlRoot+dbName;
    String SQLCommand = "SELECT * FROM "+tableName+";";

    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(SQLCommand);
      ResultSetMetaData md = rs.getMetaData();

      dataTypes = new String[md.getColumnCount()];
      for(int i=0;i<dataTypes.length;i++){
        dataTypes[i] = md.getColumnTypeName(i+1);
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
    return dataTypes;
  }
  public Vector executeQuery(String SQLQuery){
    Vector dataSet = new Vector();
    String url = urlRoot+dbName;

    try {
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
```

```
      ResultSet rs = stmt.executeQuery(SQLQuery);
      ResultSetMetaData md = rs.getMetaData();

      int nColumns = md.getColumnCount();
      while(rs.next()){
        Vector rowData = new Vector();
        for(int i=1;i<=nColumns;i++){
          rowData.addElement(rs.getObject(i));
        }
        dataSet.addElement(rowData);
      }
      con.close();
    }
    catch(SQLException e){
      reportException(e.getMessage());
    }
    return dataSet;
  }
  public void setExceptionListener(ActionListener exceptionListener){
    this.exceptionListener=exceptionListener;
  }
  private void reportException(String exception){
    if(exceptionListener!=null){
      ActionEvent evt = new ActionEvent(this,0,exception);
      exceptionListener.actionPerformed(evt);
    }else{
      System.err.println(exception);
    }
  }
}
```

Summary

In this chapter you learned how to build and use queries and subqueries. You also learned how to use queries and subqueries in a `SELECT` command as well as in the `INSERT`, `DELETE`, and `UPDATE` commands. Other topics discussed were:

- What a Query is and how to create and execute one

- Using `SELECT FROM` to retrieve all rows and columns from a table
- Using the `WHERE` clause to retrieve rows matching a specific query
- Using the `ORDER BY` clause to sort the returned data
- SQL Operators
- Escape sequences
- Subqueries using the keywords:
  - `EXISTS` and NOT EXISTS
  - ANY and ALL
  - `IN` and `NOT IN`
- Nested and correlated subqueries
- JDBC `ResultSets` and `ResultSetMetaData`

The next chapter discusses using joins to retrieve data from more than one table.

Chapter 8: Organizing Search Results and Using Indexes

In This Chapter

This chapter discusses various ways of organizing and analyzing the data returned by SQL queries. These include sorting the data by one or more columns, grouping the data and performing statistical analysis, and filtering the grouped results.

The chapter also addresses the use of indexes to make your queries more efficient. Using indexes wisely can result in a very significant improvement in performance, while using indexes incorrectly can result in very poor performance.

The final topic discussed in this chapter is the use of Views. Views provide a means of creating temporary tables based on a particular query.

Using ORDER BY to Sort the Results of a Query

A common requirement when retrieving data from an RDBMS by using the `SELECT` statement is to sort the results of the query in alphabetic or numeric order on one or more of the columns. You sort the results by using the `ORDER BY` clause in a statement like this:

```
SELECT First_Name, Last_Name, City, State
FROM CUSTOMERS
WHERE Last_Name = 'Corleone'
ORDER BY First_Name;
```

This gives you a list of all the Corleones sorted in ascending order by first name, as shown in Table 8-1:

Table 8-1: Records Sorted Using ORDER BY

First_Name	Last_Name	City	State
Francis	Corleone	New York	NY
Fredo	Corleone	New York	NY
Michael	Corleone	New York	NY
Sonny	Corleone	Newark	NJ
Vito	Corleone	Newark	NJ

The default sort order is ascending. This can be changed to descending order by adding the `DESC` keyword as shown in the next example:

Note The keywords ASC and DESC can be used to specify ascending or descending sort

order.

```
SELECT *
FROM CUSTOMERS
WHERE Last_Name = 'Corleone'
ORDER BY First_Name DESC;
```

Sorting on multiple columns is also easy to do by using a sort list. For example, to sort the data in ascending order based on Last_Name and then sort duplicates using the First_Name in descending order, the sort list is as follows:

```
ORDER BY Last_Name, First_Name DESC;
```

The entire SQL statement to sort the data in ascending order based on Last_Name and then sort duplicates using the First_Name in descending order is shown below .

```
SELECT First_Name, MI, Last_Name, Street, City, State, Zip
FROM CUSTOMERS
ORDER BY Last_Name, First_Name DESC;
```

Note

When no ORDER BY clause is used, the order of the output of a query is undefined.

These are the rules for using `ORDER BY`:

- `ORDER BY` must be the last clause in the `SELECT` statement.
- Default sort order is ascending.
- You can specify ascending order with the keyword ASC.
- You can specify descending order with the keyword DESC.
- You can use column names or expressions in the `ORDER BY` clause.
- The column names in the `ORDER BY` clause do not have to be specified in the select list.
- `NULLS` usually occur first in the sort order.

Note

The DatabaseMetaData object provides a number of methods:

```
boolean nullsAreSortedAtStart()

boolean nullsAreSortedAtEnd()
```

These methods can be used to determine the sort order for NULLs when in doubt.

Another common reporting requirement is to break down the data a query returns into various groups so that the data can be analyzed in some way. The `GROUP BY` clause, discussed in the next section, enables you to combine database records to perform calculations such as averages or counts on groups of records.

The GROUP BY Clause

The GROUP BY clause combines records with identical values in a specified field into a single record for this purpose, as shown in Figure 8-1, illustrating how to use GROUP BY to compute a count of customers by state.

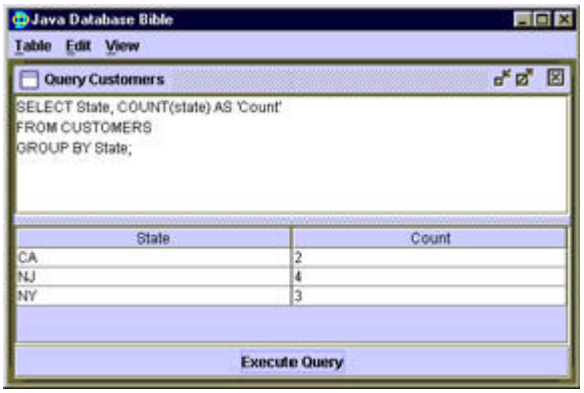


Figure 8-1: Using GROUP BY to count customers by state

Because the GROUP BY clause combines all records with identical values in one column into a single record, each of the column names in the SELECT clause must be either a column specified in the GROUP BY clause or a column function such as COUNT( ) or SUM( ).

This means that you can't SELECT a list of individual customers by name and then count them as a group by using GROUP BY. However, you can group on more than one column, just as you can use more than one column with the ORDER BY clause. You can see an example of the use of GROUP BY on more than one column in Figure 8-2.

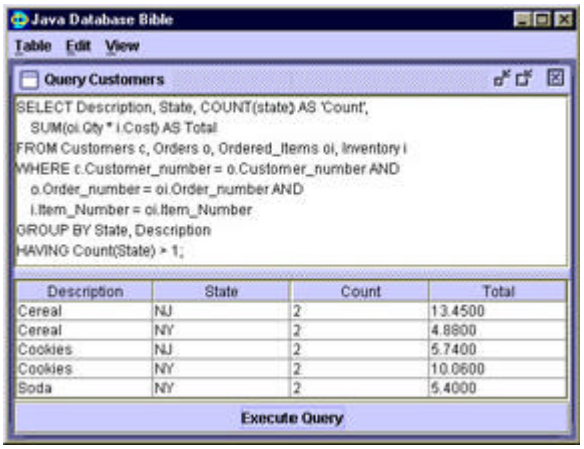


Figure 8-2: Using GROUP BY on multiple columns

Note

Every column name specified in the SELECT statement is also mentioned in the GROUP BY clause. Not mentioning the column names in both places gives you an error. The GROUP BY clause returns a row for each unique combination of description and state.

The most important uses of the `GROUP BY` clause is to group data for analytical purposes. The functions used to analyze groups of data are called *aggregate functions*. The aggregate functions are discussed in the next section.

Aggregate Functions

Aggregate functions return a single value from an operation on a column of data. This differentiates them from the arithmetic, logical, and character functions discussed in Chapter 7, which operate on individual data elements. Most relational database management systems support the aggregate functions listed in Table 8-2.

Table 8-2: Commonly Supported Aggregate Functions

Sum	`SUM`
Average	`AVG`
Count	`COUNT`
Standard Deviation	`STDEV`
Maximum	`MAX`
Minimum	`MIN`

Aggregate functions are used to provide statistical or summary information about groups of data elements. These groups may be created specifically using the `GROUP BY` clause, or the aggregate functions may be applied to the default group, which is the entire result set.

A good practical example of the use of aggregate functions is the creation of a simple sales report. In Figure 8-3, the query creates a result set listing distinct customers and calculating the number and total cost of the items they have bought.

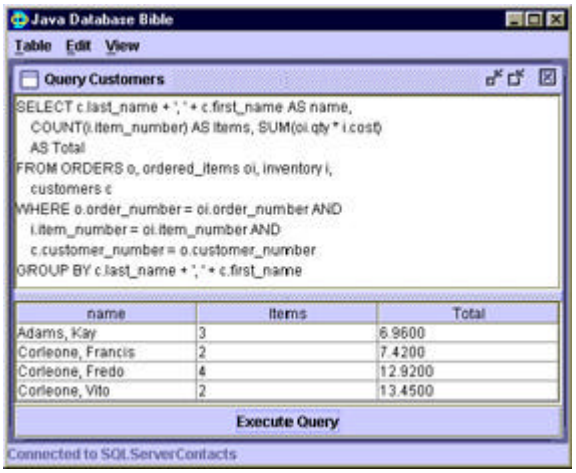


Figure 8-3: Using aggregate functions

Since this example groups order data by customer, each row of the result set represents a single customer so that customer information can be displayed. The aggregate functions act on all the purchases customers have made, so they, too, can be included in the `SELECT` list:

Note

The fundamental difference between aggregate functions and standard functions is that aggregate functions use the entire column of data as their input and produce a single output, whereas standard functions operate on individual data elements .

In addition to using the `GROUP BY` clause to group your results, you may also wish to narrow your set of groups down to a smaller subset. You can filter grouped data by using the `HAVING` clause, which is discussed in the next section.

Using the HAVING Clause to Filter Groups

There are going to be situations where you'll want to filter the groups themselves in much the same way as you filter records using the `WHERE` clause. For example, you may want to analyze your sales by state but ignore states with a limited number of customers.

SQL provides a way of filtering groups in a result set using the `HAVING` clause. The `HAVING` clause works in much the same way as the `WHERE` clause, except that it applies to groups within a returned result set, rather than to the entire table or group of tables forming the subject of a `SELECT` statement.

To filter groups, apply a `HAVING` clause after the `GROUP BY` clause. The `HAVING` clause lets you apply a qualifying condition to groups so that the database management system returns a result only for the groups that satisfy the condition. Incidentally, you can also apply a `HAVING` clause to the entire result set by omitting the `GROUP BY` clause. In this case, DBMS treats the entire table as one group, so there is at most one result row. If the `HAVING` condition is not true for the table as a whole, no rows will be returned.

`HAVING` clauses can contain one or more predicates connected by `AND`s and `OR`s. Each predicate compares a property of the group (such as `COUNT(State)`) with either another property of the group or a constant.

Figure 8-4 shows the use of the HAVING clause to compute a count of customers by state, filtering results from states that contain only one customer.

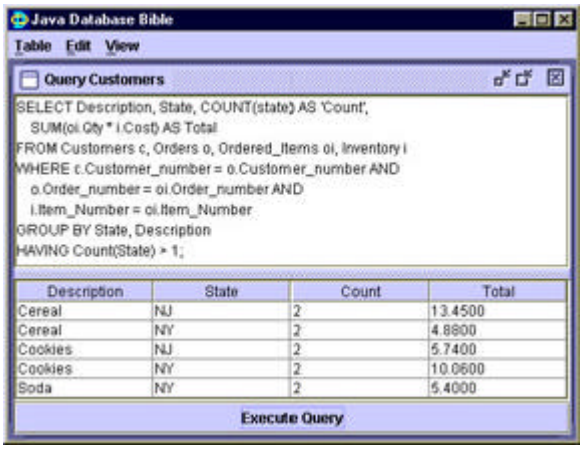


Figure 8-4: Using the HAVING clause

The main similarity between the `HAVING` clause and the `WHERE` clause is that both allow you to use a variety of filters in a query. The main difference is that the `HAVING` clause applies to groups within a returned result set, while the `WHERE` clause applies to the entire table or group of tables forming the subject of a `SELECT` statement.

Using Indexes to Improve the Efficiency of SQL Queries

You can improve database performance significantly by using *indexes*. An index is a structure that provides a quick way to look up specific items in a table or view. In effect, an index is an ordered array of pointers to the rows in a table or view.

When you assign a unique id to each row as a key, you are predefining an index for that table. This makes it much faster for the DBMS to look up items by id, which is commonly required when you are doing joins on the id column.

SQL's `CREATE INDEX` statement allows you to add an index for any desired column or group of columns. When you need to do a search by customer name, for example, the unique row id buys you nothing; the DBMS has to do a brute-force search of the entire table to find all customer names matching your query. If you plan on doing a lot of queries by customer name, it obviously makes sense to add an index to the customer name column or columns. Otherwise, you are in the position of someone working with a phone list that hasn't been alphabetized.

The SQL command to add an index uses the `CREATE INDEX` keyword, specifying a name for the index and defining the table name and the column list to index. Here's an example:

```
CREATE INDEX STATE_INDEX ON MEMBER_PROFILES(STATE);
```

To remove the index, use the `DROP INDEX` command.

```
DROP INDEX MEMBER_PROFILES.STATE_INDEX;
```

Notice how the name of the index has to be fully defined by prefixing it with the name of the table to which it applies.

The example in Listing 8-1 is a simple JDBC, with a couple of lines of additional code that calculate the start and stop times of the query so that the elapsed can be calculated. By commenting out the `CREATE INDEX` and `DROP INDEX` lines, speed improvement can easily be calculated.

Listing 8-1: Creating and dropping indexes

```
package java_databases.ch04;

import java.sql.*;

public class PrintIndexedResultSet{
  public static void main(String args[]){
    String query =
    "SELECT STATE, COUNT(STATE) FROM MEMBER_PROFILES GROUP BY STATE";
    PrintIndexedResultSet p = new PrintIndexedResultSet(query);
  }
  public PrintIndexedResultSet(String query){
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection ("jdbc:odbc:Members");
      Statement stmt = con.createStatement();
      stmt.executeUpdate("CREATE INDEX STATE_INDEX ON
MEMBER_PROFILES(STATE)");

      java.util.Date startTime = new java.util.Date();

      ResultSet rs = stmt.executeQuery(query);
      ResultSetMetaData md   = rs.getMetaData();

      int nColumns = md.getColumnCount();
      for(int i=1;i<=nColumns;i++){
        System.out.print(md.getColumnLabel(i)+((i==nColumns)?"\n":"\t"));
      }

      while (rs.next()) {
```

```
        for(int i=1;i<=nColumns;i++){
          System.out.print(rs.getString(i)+((i==nColumns)?"\n":"\t"));
        }
      }
      java.util.Date endTime = new java.util.Date();
      long elapsedTime = endTime.getTime() - startTime.getTime();
      System.out.println("Elapsed time: "+elapsedTime);

      stmt.executeUpdate("DROP INDEX MEMBER_PROFILES.STATE_INDEX");
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
  }
}
```

The example in Listing 8-1 is run against a membership database containing approximately 150,000 members and shows an improvement of 2:1 in elapsed time for the query shown.

Once you have executed a SQL query and obtained a sorted, grouped set of data from the database, it is frequently very useful to be able to save the query for reuse. One of the ways SQL lets you do this is by using Views.

Views

A *view* is similar to a table, but rather than being created as a fundamental part of the underlying database, it is created from the results of a query. In fact, you can think of a view as a temporary table. Like a table, a view has a name that can be used to access it in other queries. Because views work like tables, they can be a very useful tool in simplifying SQL queries. For example, you could create a view based on a complex `JOIN`, and then work with that view as a temporary table rather than embedding the `JOIN` as a subquery and working with the underlying tables.

The basic syntax used to create a view is as follows:

```
CREATE VIEW Orders_by_Name AS SELECT ...
```

The `SELECT` statement in the code is the `SELECT` you use in the query you want to save as a view, as shown here:

```
CREATE VIEW ViewCorleones AS
   SELECT *
   FROM CUSTOMERS
   WHERE Last_Name = 'Corleone'
```

Now you can execute a query just as if this view were a normal table, as follows:

```
SELECT *
FROM ViewCorleones
```

The result set this query returns looks like this:

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Michael	A	Corleone	123 Pine	New York	NY	10006
Fredo	X	Corleone	17 Main	New York	NY	10007
Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
Francis	X	Corleone	17 Main	New York	NY	10005
Vito	G	Corleone	23 Oak St	Newark	NJ	12345

As with any other table, you can use more complex queries. Here's an example:

```
SELECT *
FROM ViewCorleones
WHERE State = 'NJ'
```

This query returns the following result set:

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
Vito	G	Corleone	23 Oak St	Newark	NJ	12345

You can use a view for updating or deleting rows, as well as for retrieving data. Since the view is not a table in its own right, but merely a way of looking at a table, rows updated or deleted in the view are updated or deleted in the original table. For example, you can use the view to change Fredo Corleone's street address by using the following SQL statement:

```
UPDATE ViewCorleones
SET Street = '19 Main'
WHERE First_Name = 'Fredo'
```

This example illustrates one of the advantages of using a view. A lot of the filtering required to identify the target row is done in the view, so the SQL code is simpler and more maintainable. In a nontrivial example, this can be a worthwhile improvement.

Figure 8-5 shows how using the view to the change Fredo Corleone's street address propagates through to the Customers Table.

Figure 8-5: Updating a view updates the underlying table.

Recall that a view is really nothing more than a named result set made accessible as if it were a table. Creating a view from a complicated query is just as easy as creating one from a simple query.

One way to retrieve data from multiple tables is to use an `INNER JOIN`. The next example shows how to use an `INNER JOIN` to retrieve data from four different tables, creating a view called "Orders_by_Name":

```
CREATE VIEW Orders_by_Name AS
SELECT c.LAST_NAME + ', ' + c.FIRST_NAME AS Name,
    COUNT(i.Item_Number) AS Items, SUM(oi.Qty * i.Cost)
    AS Total
FROM ORDERS o INNER JOIN
    ORDERED_ITEMS oi ON
    o.Order_Number = oi.Order_Number INNER JOIN
    INVENTORY i ON
    oi.Item_Number = i.Item_Number INNER JOIN
    CUSTOMERS c ON
    o.Customer_Number = c.CUSTOMER_NUMBER
GROUP BY c.LAST_NAME + ', ' + c.FIRST_NAME
```

Cross-Reference

`JOINS` are discussed in Chapter 9.

You can now query this view in the normal way to get a summary of customer orders by name as shown in the following table.

Name	Items	Total
Adams, Kay	3	6.96

Name	Items	Total
Corleone, Francis	2	7.42
Corleone, Fredo	4	12.92
Corleone, Vito	2	13.45

This result set always reflects the underlying table; if Fredo Corleone were to buy a huge supply of chocolate chip cookies, the next time you run the same query, you might see a result set like this one:

Name	Items	Total
Adams, Kay	3	6.96
Corleone, Francis	2	7.42
Corleone, Fredo	4	135.14
Corleone, Vito	2	13.45

Note

Views are a way of saving queries by name, which can be very useful for creating reports or updates you want to use on a regular basis. The database management system generally saves the view by associating the `SELECT` statement with the view name and executing it when you want to access the view.

Summary

In this chapter, you learn to perform the following tasks:

- Sorting the data you retrieve from a database
- Grouping the results for analysis
- Performing statistical analyses on the data you retrieve from a database
- Create and use indexes to improve performance
- Saving your queries as views

In the next chapter, you learn to retrieve data from more than one table.

Chapter 9: Joins and Compound Queries

In This Chapter

One of the most powerful features of SQL is its ability to combine data from several tables into a single result set. When tables are combined in this way, the operation performed is called a JOIN. There are two primary types of JOIN, and a number of different ways in which they can be performed.

Another way to combine data from different tables into a single result set is to use the UNION operator. This chapter discusses the different types of JOINS, and the use of the UNION operator.

Joining Tables

Chapter 2 explained how an efficient and reliable database design frequently requires the information in a practical database will be distributed across several tables, each of which contains sets of logically related data. A typical example might be a database containing these four tables:

- **Customers,** containing customer number, name, shipping address, and billing information
- **Inventory,** containing item number, name, description, cost, and quantity on hand
- **Orders,** containing order number, customer number, order date, and ship date
- **Ordered_Items,** containing order number, item number, and quantity

When a customer places an order, an entry is made in the Orders Table, assigning an order number and containing the customer number and the order date. Then entries are added to the Ordered_Items table, recording order number, item number, and quantity. To fill a customer order, you need to combine the necessary information from each of these tables.

Using `JOIN`, you are able to combine data from these different tables to produce a detailed invoice. This invoice will show the customer name, shipping address, and billing information from the Customers table, combined with a detailed list of the items ordered from the Ordered_Items table, supported by detailed description, quantity, and unit price information from the inventory table.

Cross-Reference

Primary and Foreign Keys are also discussed in Chapter 1, which provides a more theoretical overview of Relational Database Management Systems.

Types of Joins

There are two major types of Joins: Inner Joins and Outer Joins. The difference between these two types of Joins goes back to the basic Set Theory underlying relational databases. You can imagine the keys of two database tables, A and B as intersecting sets, as shown in Figure 9-1.

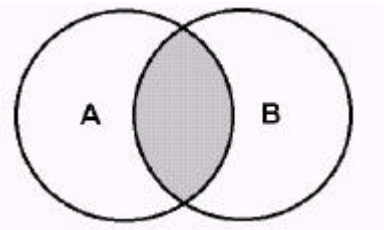


Figure 9-1: Primary and Foreign keys are used to define intersecting data sets.

Inner Joins

The Inner Join of these two sets is the intersection of the sets. For example, to retrieve all the information required to invoice a client, you would require the name and address information from table A, joined on the order information from table B. The intersection of these sets is the set of primary keys in the Customers table and the set of foreign keys in the order information table that match the required Customer_ID. The Inner Join of the two tables is the subset of the rows in the name and address table, which has the required Customer_ID, and the subset of the rows in the order information table, which references that Customer_ID. This is shown as the shaded portion of Figure 9-1.

Outer Joins

There are three kinds of Outer Joins:

- **Full Outer Joins**, which, in Set Theory terms, are Unions of the sets. A Full Outer Join includes all of both joined sets. This would correspond to the entire area of both circles in Figure 9-1.
- **Left Outer Joins,** which are the entire set on the left, plus the contents of the intersection. This would correspond to the entire area of the left circle A, plus the shaded area in Figure 9-1.
- **Right Outer Joins,** which are the entire set on the left, plus the contents of the intersection. This would correspond to the entire area of the right circle B, plus the shaded area in Figure 9-1.

It is important to note that it is really the keys that form the members of the sets, since only the keys are alike. The row data itself, being different from one table to another, can't intersect with row data from another table. This observation underscores the importance of keys in linking tables, which is reviewed in the next section.

Note

Although this discussion of Joins in terms of sets was illustrated using only two sets, the concept applies to any number of tables or sets.

Since the use of JOINS is heavily dependent on using keys, the next section reviews what primary and foreign keys are, and how they are used.

Keys

First, it is important to understand *keys*. In each of the four tables in the example, there is an identifier such as customer number or item number. These identifiers are called keys and are used primarily to provide a unique reference to a given record. Database management systems use two kinds of keys:

- Primary keys
- Foreign keys

Primary Keys

A *primary key* is a column that uniquely identifies the rest of the data in any given row. For example, in the Customers Table, the Customer_Number column uniquely identifies that customer. For this to work, no two rows can have the same key (or, in this instance, Customer_Number), so a key is a good example of the use of the `UNIQUE` constraint. A clear benefit of using a unique integer as a row identifier is that a list of integers is far faster to search than an array of First Name/Last Name character variables. Another obvious benefit of using unique keys is that your system can support more than one customer with the same name, as the Customer_Number is your primary means of identifying customers.

Foreign Keys

A *foreign key* is a column in a table where that column is a primary key of another table. For example, the Orders Table contains one column for Order_Number, which is its own primary key, and another column for Customer_Number, which is a foreign key.

The purpose of these *keys* is to establish relationships across tables, without having to repeat data in every table. This concept encapsulates the power of relational databases. You see many examples of the use of both primary keys and foreign keys in the Joins you work with in this chapter.

Using Inner Joins

An Inner Join between two or more tables, as discussed, represents the intersection of the sets of keys matching some query. The most common form of query used in creating an Inner Join involves the selection of rows that have a key equal to some particular value. A typical example might be to find data from a number of tables where the Customer_ID equals that of a specific customer. Joins using this equality test are called Equi-Joins, and are discussed in the next section.

Using Equi-Joins

SQL Joins work by matching up equivalent columns in different tables by comparing keys. The most common type of Join is an *Equi-Join*, where you look for items in one table that have the same item number as items in another. The first example demonstrates how Equi-Joins work.

The examples throughout Part II have used variations on the Customers table shown in Table 9-1 and the Inventory table shown in Table 9-2. These tables form the basis of an order management database.

Table 9-1: Customer Table

Customer_Number	First_Name	MI	Last_Name	Street	City	State	Zip
100	Michael	A	Corleone	123 Pine	New York	NY	10006
101	Fredo	X	Corleone	17	New	NY	1000

Table 9-1: Customer Table

Customer_Number	First_Name	MI	Last_Name	Street	City	State	Zip
				Main	York		7
102	Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
103	Francis	X	Corleone	17 Main	New York	NY	10005
104	Vito	G	Corleone	23 Oak St	Newark	NJ	12345
105	Tom	B	Hagen	37 Chestnut	Newark	NJ	12345
106	Kay	K	Adams	109 Maple	Newark	NJ	12345
107	Francis	F	Coppola	123 Sunset	Hollywood	CA	23456
108	Mario	S	Puzo	124 Vine	Hollywood	CA	23456

Table 9-2: Inventory Table

Item_Number	Name	Description	Qty	Cost
1001	Corn Flakes	Cereal	130	1.95
1002	Rice Krispies	Cereal	97	1.87
1003	Shredded Wheat	Cereal	103	2.05
1004	Oatmeal	Cereal	15	0.98
1005	Chocolate Chip	Cookies	217	1.26
1006	Fig Bar	Cookies	162	1.57
1007	Sugar Cookies	Cookies	276	1.03
1008	Cola	Soda	144	0.61
1009	Lemon Soda	Soda	96	0.57
1010	Orange Soda	Soda	84	0.71

In addition to the Customers table and the Inventory table you need a table that lists the orders by order number, using one record per order, and containing the customer number of the customer

placing the order, together with information such as order date and ship date. The Orders Table, as shown in Table 9-3, maps orders to customers.

Table 9-3: Orders Table

Order_Number	Customer_Number	Order_Date	Ship_Date
2	101	12/8/01	12/10/01
3	103	12/9/01	12/11/01

Finally, you need a table listing every item in each order. This table contains the order number, item number, and quantity for each ordered item, as shown in Table 9-4.

Table 9-4: Ordered Items Table

ID	Order_Number	Item_Number	Qty
5000	2	1001	2
5001	2	1004	1
5002	2	1005	3
5003	2	1010	6
5004	3	1006	4
5005	3	1009	2

The structure of these tables follows the basic principle of keeping related data items together and separated from unrelated items. There is never, for example, a direct relationship between inventory items and customers. The customer interacts with inventory through the mechanism of placing an order. The order links to the inventory through the Ordered_Items Table and to the customer via the customer number. The Ordered_Items Table provides a link between the order number and the items in the Inventory Table.

Once the data has been divided logically among these four tables, as illustrated in Tables 9-1 through 9-4, you can write the following SQL command to get a list of the products in order number 2:

```
SELECT Orders.Order_number, Ordered_Items.Item_number,
    Ordered_Items.Qty, Inventory.Name,
    Inventory.Description
FROM Orders, Ordered_Items, Inventory
WHERE Orders.order_number = Ordered_Items.order_number AND
    Inventory.Item_Number = Ordered_Items.Item_Number AND
    Orders.order_number = 2;
```

Notice how the columns used in the `WHERE` clause comparison are the key columns of the various tables. The dotted notation allows you to tell the database management system which table to look in

for each use of a given field, so the WHERE clause tells the DBMS to return data from the various tables where the Order_Number fields match up and equal 2. This gives the following ResultSet:

Order_number	Item_number	Qty	Name	Description
2	1001	2	Corn Flakes	Cereal
2	1004	1	Oatmeal	Cereal
2	1005	3	Chocolate Chip	Cookies
2	1010	6	Orange	Soda

Although this approach of prefixing the column name by the table name works well, it is rather verbose. Conventionally, SQL queries are made using short aliases for the table names. The use of aliases is discussed in the next section.

Using an alias for the table name in a query

Conventionally, SQL queries are made using short aliases for the table names. Frequently, the alias is a single letter, as shown here:

```
SELECT o.Order_number, oi.Item_number, oi.Qty, i.Name,
    i.Description
FROM Orders o, Ordered_Items oi, Inventory i
WHERE o.Order_number = oi.Order_number AND
    i.Item_Number = oi.Item_Number AND o.Order_number = 2;
```

The alias is defined in the FROM clause, since that is where the tables are identified, and is used throughout the rest of the query.

Caution

The most important aspect of using aliases succesfully is to understand the order in which parts of a SQL statement are executed. The use of aliases is discussed in more detail in chapter 7.

Figure 9-2 shows the results produced by executing this query using the Swing Database Query tool built in Chapter 7.

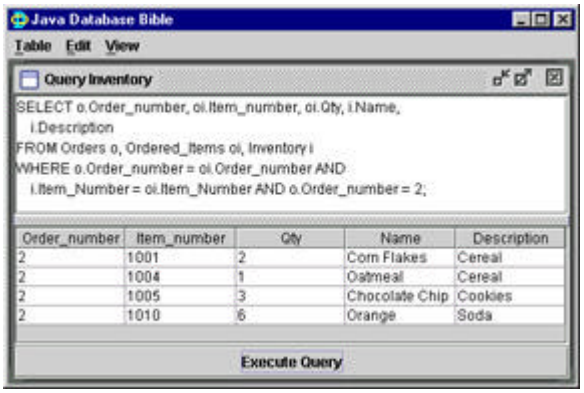


Figure 9-2: Using aliases to simplify queries

Figure 9-3 illustrates a slightly more complex query, involving all four tables and using calculated results in columns, with names assigned in the query.

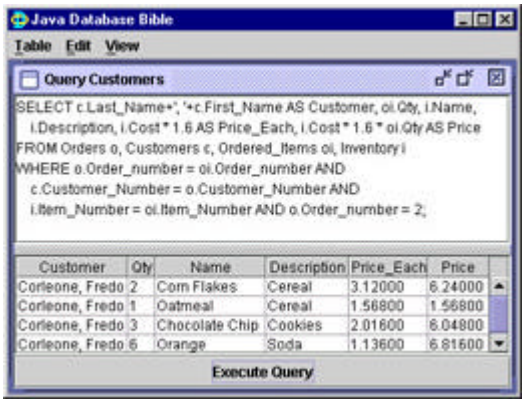


Figure 9-3: Returning calculated results from a Join

The example shown in Figure 9-3 does not take into account the possibility that you might want to write a query where a customer is listed only once. To handle situations where this is the case, you need a way to eliminate duplicate names from a result set.

Using DISTINCT to eliminate duplicates

There are many situations in which you may not want data to be repeated in a result set. For example, if you are planning a special sale on cookies, you might want to send a mailer to only customers who have bought cookies. Obviously, you want a list where each customer appears only once. This means that you need to tell SQL to eliminate duplicate names.

To find which orders include cookies, perform an Equi-Join on the Inventory, Ordered_Items, and Orders Tables. Then join the results on customers to get the name and address information for the mailer.

The basic Join looks like this:

```
SELECT c.first_name, c.last_name, c.street, c.city, c.state,
    c.zip
FROM ORDERS o, customers c, ordered_items oi,
    inventory i
WHERE i.description = 'Cookies' AND
    i.item_number = oi.item_number AND
    oi.order_number = o.order_number AND
    o.customer_number = c.customer_number;
```

The result, as it stands, is not quite what you are looking for, in that Kay Adams appears twice because she bought cookies twice. The solution is to insert the keyword `DISTINCT` into the `SELECT` clause, telling the SQL engine to return only one instance of each record, as shown in Figure 9-4.

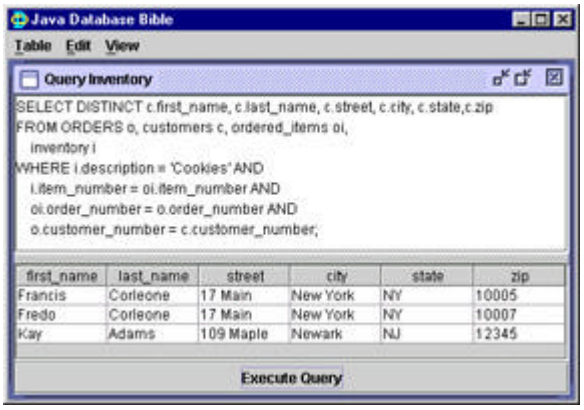


Figure 9-4: Using DISTINCT to eliminate duplicate records

Using Non-Equi-Joins

The Joins used up to this point have all been Equi-Joins, or Joins where the values of the keys used to make the join have been equal to each other. However, it seems reasonable that you should be able to do Non-Equi-Joins or Joins where the relationship is not equal. For example, since there are only two orders in the Orders Table used in the previous example, you can get the other order using the Non-Equi-Join. Here's an example:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer, oi.Qty,
    i.Name, i.Description, i.Cost * 1.6 AS Price_Each,
    i.Cost * 1.6 * oi.Qty AS Price
FROM Orders o, Customers c, Ordered_Items oi, Inventory i
WHERE o.Order_number = oi.Order_number AND
    c.Customer_Number = o.Customer_Number AND
    i.Item_Number = oi.Item_Number AND o.Order_number <> 2;
```

Using Outer Joins

The Joins discussed so far have been *Inner Joins*. An Inner Join is a Join between two tables. An inner Join includes only rows with matching rows in the both tables. A set oriented way of visualising Joins was shown in Figure 9-1. Another easy way to visualize this is by drawing a diagram like Figure 9-5, where the Customer_Number columns in the Customers and Orders Tables intersect in the shaded area to identify an Inner Join.

First_Name	MI	Last_Name	Customer Number	Order Number	Order_Dat	Ship_Date
Michael	A	Corleone	101	2	12/8/01	12/10/01
Fredo	X	Corleone	102	3	12/9/01	12/11/01
Sonny	A	Corleone	103	4	12/9/01	12/11/01
Francis	X	Corleone	104	5	12/10/01	12/12/01
Vito	G	Corleone	105			
Tom	B	Hagen	106			
Kay	K	Adams	107			
Francis	F	Coppola	108			
Mario	S	Puzo	109			

Figure 9-5: Tables joined on customer number

The two tables are shown in the rounded boxes, and the Joined fields are shaded.

Using an Inner Join, as shown in the last example, you can only list customers who have placed an order, so their customer numbers fall into the shaded area of Figure 9-5. If you want a list of all customers, together with the dates of any orders they have placed, you can't get there with an Inner Join.

An *Outer Join* can include not only records inside the union of the sets or tables, but records outside the union of the sets, as well. In other words, in addition to the set members that share customer numbers, you can get customers in the lower, or "Outer," part of the joined tables.

There are three types of Outer Joins:

- `LEFT OUTER JOIN (*=)`
- `RIGHT OUTER JOIN (=*)`
- `FULL OUTER JOIN`

The terms LEFT, RIGHT, and FULL describe which of the tables' unmatched columns to include in the Join relative to the order in which the tables appear in the `JOIN` command.

LEFT OUTER JOIN

The `LEFT OUTER JOIN` operator includes all rows from the left side of the Join, as shown in Figure 9-6.

Figure 9-6: Executing a LEFT OUTER JOIN

RIGHT OUTER JOIN

It is important to note that "left" and "right" are completely dependent on the order of the tables in the SQL statement, so you can turn this into a `RIGHT OUTER JOIN` by reversing the order of the tables in the `JOIN` command. Here's an example:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer,
    o.Order_Date
FROM ORDERS o RIGHT OUTER JOIN
    CUSTOMERS c ON c.customer_number = o.customer_number;
```

`OUTER JOIN` commands can also be written in shorthand similar to the form we use for our `INNER JOIN`. This is the form for the `LEFT OUTER JOIN`:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer,
    o.Order_Date
FROM CUSTOMERS c, ORDERS o
WHERE c.customer_number *= o.customer_number;
```

The form for the `RIGHT OUTER JOIN` follows:

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer,
    o.Order_Date
FROM ORDERS o, CUSTOMERS c
WHERE o.customer_number =* c.customer_number;
```

Note

In the shorthand version, the type of JOIN depends on both the order of the tables in the FROM clause, and the position of the asterisk in the *= operator.

FULL OUTER JOIN

A `FULL OUTER JOIN` includes all unmatched rows from both tables in the result. For example, to find any orders in the Orders Table with customer numbers that do not match any entries in our Customers Table, you can execute a Full Outer Join to show all the entries in both tables, as shown in Figure 9-7.

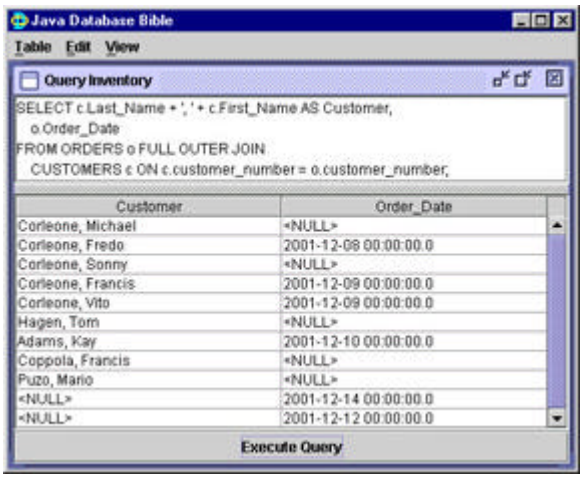


Figure 9-7: Full Outer Join

The preceding examples have illustrated the use of `JOINS` to find records from two tables with some degree of commonality. The next section discusses how to obtain result sets which specifically exclude those matching selected criteria.

Using NOT EXISTS

Now you know how to use `INNER JOINS` to find records from two tables with matching fields and how to use `OUTER JOINS` to find all records, matching or nonmatching. Next, consider the case where you want to find records from one table that don't have corresponding records in another.

Using the Customers and Orders Tables again, find all the customers who have not placed an order. The way to do this is to find customer records with customer numbers that do not exist in the Orders Table. Do this by using `NOT EXISTS`, as shown in Figure 9-8.

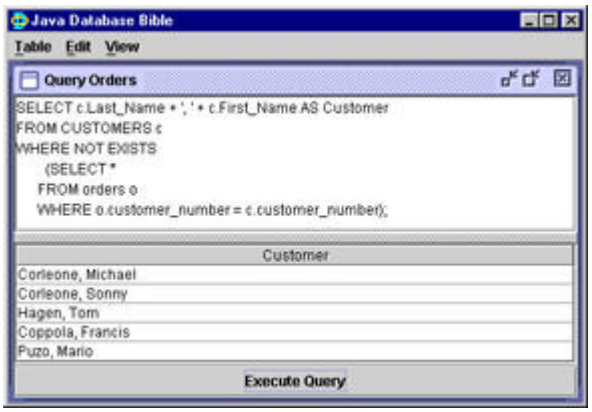


Figure 9-8: Using NOT EXISTS to find records in one table with no corresponding entry in another table.

```
SELECT c.Last_Name + ', ' + c.First_Name AS Customer

FROM CUSTOMERS c

WHERE NOT EXISTS

       (SELECT *

     FROM orders o

     WHERE o.customer_number = c.customer_number);
```

In addition to joining tables to each other, it is sometimes useful to join a table to itself. The next section discusses how and why you would perform a Self-Join on a table.

Using Self-Joins

A Self-Join is simply a normal SQL join that joins a table to itself. Use a Self-Join when rows in a table contain references to other rows in the same table. Here's an example of this situation in a table of employees, where each record contains a reference to the employee's supervisor by Employee_ID:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SUPERVISOR
100	Michael	Corleone	104
101	Fredo	Corleone	100
102	Sonny	Corleone	100
103	Francis	Corleone	100
104	Vito	Corleone	99
105	Tom	Hagen	100
106	Kay	Adams	100
107	Francis	Coppola	100
108	Mario	Puzo	100

Since the supervisor is also an employee, information about the supervisor is stored in the Employees Table, so you use a Self-Join to access it. Do this by using table-name aliases to give each reference to the table a separate name.

To get a list of employees and their supervisors, create a Self-Join by creating two separate references to the Employees Table, using two different aliases. An example is shown in Figure 9-9.

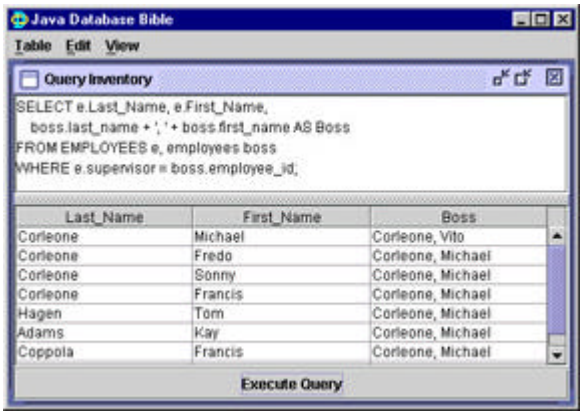


Figure 9-9: Using a Self-Join

The preceding SQL code is effectively creating what looks like two identical tables, e and boss, and joining them using an Inner Join, so that you can get employee information from one reference to the table and supervisor information from the other:

You can turn this into an Outer Self-Join very easily, as follows:

```
SELECT e.last_name, e.first_name,
```

```
    boss.last_name + ', ' + boss.first_name AS Boss
FROM EMPLOYEES e, employees boss
WHERE e.supervisor *= boss.employee_id;
```

This returns one additional row; the Employee_ID of Vito's supervisor does not appear in the Employees Table, so his boss is shown as <NULL> in the example of Figure 9-10.

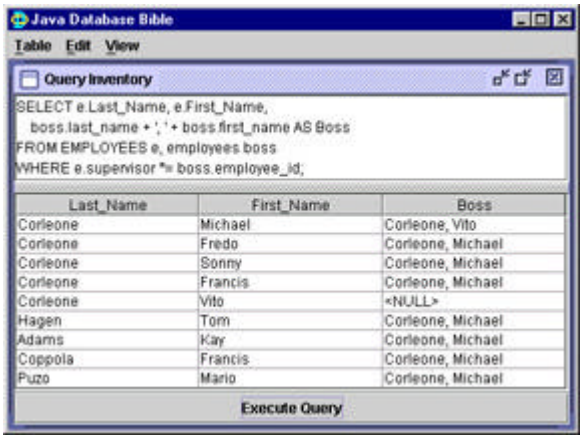


Figure 9-10: Using an Outer Self-Join

In addition to using joins, you can combine data from two separate sources using the UNION operator. The next section describes the UNION operator.

Using the UNION Operator to Combine Queries

Another way to combine data from two separate sources is to use the UNION operator. The default action of the UNION operator is to combine the results of two or more queries into a single query and to eliminate any duplicate rows. When ALL is used with UNION, duplicate rows are not eliminated.

In Figure 9-11, the first query returns the names and addresses of all the Corleones, and the second returns all customers in New Jersey. The UNION operator combines the results, removing the duplicate records that are generated for Corleones in New Jersey.

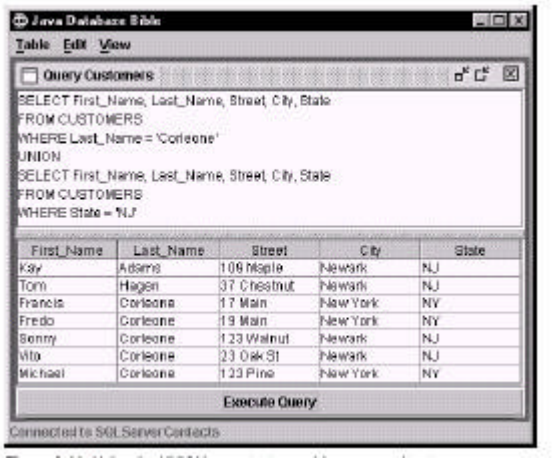


Figure 9-11: Using the UNION operator to combine two result sets

Understanding Cartesian Products

Cartesian Products, or cross products, are something you normally want to avoid. The Cartesian Product of a Join occurs when every record in one table is joined on every record of the other, so the Cartesian Product of two tables of 100 rows each is10,000 rows.

Cartesian Products are normally an error, caused by a bad or nonexistent WHERE clause. In the case of a small table like the ones in our examples, this is not a major problem, but on a large database, the time taken to generate cross products of thousands of rows can be significant.

You can use `ORDER BY` to sort the combined answer set by adding the `ORDER BY` clause after the last query. You do not have to use the same column in each query. Only the column counts and column types needs to match. If you create a `UNION` of two result sets with different columns, you have to apply the `ORDER BY` clause using the column number. An example of this usage is shown in Figure 9-12.

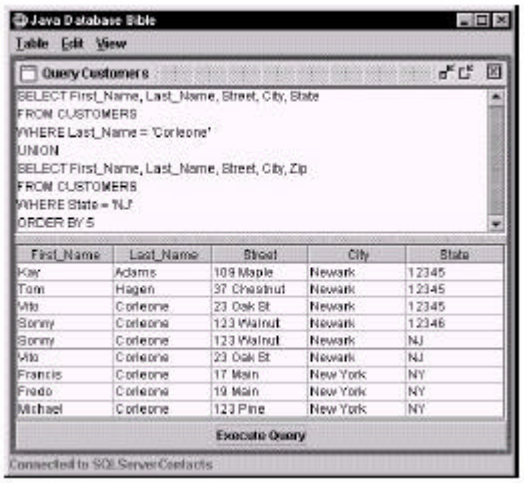


Figure 9-12: Using ORDER BY on a UNION

Two further set operators are supported by some SQL dialects. These are the EXCEPT operator, and the INTERSECT operator, which are discussed in the following paragraphs.

EXCEPT Operator

The `EXCEPT` operator creates a result set by including all rows returned by the first query but not returned by the second query. The default version eliminates all duplicate rows, but `EXCEPT ALL` does not.

INTERSECT Operator

The `INTERSECT` operator creates a result set by including only rows that exist in both queries and by eliminating all duplicate rows. When you use `ALL` with `INTERSECT`, the duplicate rows are not eliminated.

Summary

In this chapter, you learn about the following topics:

- `INNER JOINS` and `OUTER JOINS`
- `EQUI-JOINs`
- `NON-EQUI-JOINs`
- `OUTER JOINs`
- `LEFT OUTER JOINS`
- `RIGHT OUTER JOINS`
- `FULL OUTER JOINS`
- `SELF-JOINS`
- Cartesian Products
- The `UNION` operator

The next chapter discusses MetaData and moves on to combine the topics discussed thus far to build a complete client/server application.

Chapter 10: Building a Client/Server Application

In This Chapter

The aim of this chapter is to round out the discussion of the Java Database Connectivity (JDBC) core application programming interface (API). Also, this chapter extends the code examples using simple components in a client/server architecture. In addition, the chapter combines those examples to create a complete, general-purpose database-management console application. This application forms the basis of a generic toolkit for working with any data source from a flat file to a full object relational database.

In the process of creating this application, the capabilities of the MetaData objects in the core API are explored. The chapter also explains how to connect to different databases and use different drivers within a single application.

To add to the functionality of this database management application, the chapter discusses measuring and displaying the time taken to execute a query. The examples illustrate the significant difference in performance between a good commercial pure Java driver and the jdbc-odbc bridge provided by Sun as a basic implementation of JDBC.

Using Different Databases and Drivers

To demonstrate the flexibility of JDBC, you can create copies of the Contacts database under a variety of RDBMS systems and listed them in a `JComboBox`. The `JComboBox` is displayed in a `JOptionPane` used to select the database, as illustrated in Figure 10-1.

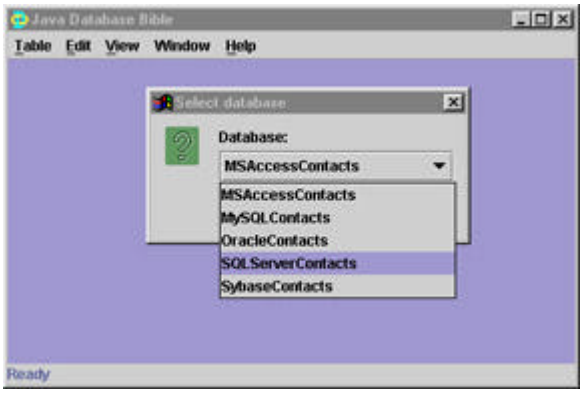


Figure 10-1: Selecting different databases using a JComboBox

With this new dialog box, the user can select any of a number of versions of the Contacts test database. Once the selection has been made, a second dialog box is displayed to enable the selection of a JDBC driver.

Although the use of other drivers is mentioned briefly at the beginning of Chapter 5, the examples in earlier chapters all use the JDBC-ODBC bridge, leaving the choice of database management system open. The reason for this is to get straight into the nuts and bolts of creating and working with a database.

As discussed in Chapter 4, the JDBC-ODBC bridge has one significant advantage when working with a variety of databases: it can be used with virtually any RDBMS, whereas most other drivers are database-system specific. On the other hand, it has the disadvantage of being less efficient than, for example, a pure Java driver optimized for a specific RDBMS (just how much less efficient is demonstrated by the query-timing code discussed later in this chapter).

The DriverManager can load a JDBC driver in two ways:

- During initialization, when the DriverManager loads drivers listed in the `"jdbc.drivers"` system property.
- Using `Class.forName()`, when a program can also explicitly load JDBC drivers at any time

When `getConnection()` is called, the DriverManager attempts to locate a suitable driver from amongst those loaded at initialization and those loaded explicitly. It does this by polling all registered drivers, passing the URL of the database to each driver's `acceptsURL()` method.

To illustrate explicit loading of a JDBC driver, a new `JOptionPane` with a `JComboBox` listing several different JDBC drivers has been added to the `DBManager` class. This makes it possible for the user to select the SQL Server version and use either the `JdbcOdbcDriver` or the Opta2000 pure Java driver.

Note

For brevity, the String arrays driving these `JOptionPanes` are restricted to only a few sample items.

In addition to these changes, a couple of new features have been included to the application to add the following functionality:

- **Window Menu** has been added, together with some supporting code that allows the user to perform such window-management tasks as cascading and tiling the JInternalFrames used to display result sets and other information.
- **Help Menu** has been added to allow the user to access information about the database management system and the JDBC driver in use.
- **StatusPanel** has been added to the bottom of the JFrame to support a message and a timer display showing the time required to execute a statement.
- **Code** has been added to get the system time before and after connecting to the database. The elapsed time in milliseconds is calculated from the difference between these times and is displayed on the status bar added to the bottom of the JFrame.

The code for the Window Menu and the Help Menu is similar to the menus shown in earlier chapters. Listing 10-1 shows the cascade and tile functions supported.

Listing 10-1: The Window Menu

```
package JavaDatabaseBible.part2;
import java.awt.*;
import javax.swing.*;

public class WindowMenu extends DBMenu{
  public WindowMenu(){
  setText("Window");
  setActionCommand("Window");
  setBorderPainted(false);
  add(new DBMenuItem("Cascade",'C',itemListener,true));
  add(new DBMenuItem("Tile horizontally",'H',itemListener,true));
  add(new DBMenuItem("Tile vertically",'V',itemListener,true));
  }
}
```

The window-management functions are implemented in the `DBManager` class through the `cascade()`, `tileVertically()`, and `tileHorizontally()` methods. The `selected()` method is used to identify the currently selected `JInternalFrame` in order to position it correctly.

The `StatusPanel` class is also very simple. It incorporates a couple of `JLabels` added to the `CENTER` and EAST areas of a `JPanel` with `BorderLayout`, as shown in Listing 10-2. The `StatusPanel` is

added to the SOUTH area of the main JFrame. JavaBean style-setter methods are used to set the messages the Status Panel displays.

Listing 10-2: Status Panel

```
package JavaDatabaseBible.part2;

import java.awt.*;
import javax.swing.*;

public class StatusPanel extends JPanel{
  JLabel msgLabel = new JLabel();
  JLabel timerLabel = new JLabel();
  public StatusPanel(){
    setLayout(new BorderLayout());
    add(msgLabel,BorderLayout.CENTER);
    add(timerLabel,BorderLayout.EAST);
  }
  public StatusPanel(String message){
    this();
    setMessage(message);
  }
  public void setMessage(String message){
    msgLabel.setText(message);
  }
  public void setTimerMsg(String message){
    timerLabel.setText(message);
  }
}
```

The Expanded DBManager Class

Since the changes to the DBManager class are fairly extensive, the whole class is shown in Listing 10-3, rather than showing the changes piecemeal. Comments have been added to identify the changes specific to this chapter.

Listing 10-3: The DBManager class

```
package JavaDatabaseBible.part2;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DBManager extends JFrame{
  JMenuBar menuBar = new JMenuBar();
```

```
JDesktopPane desktop = new JDesktopPane();
StatusPanel statusBar = new StatusPanel("Ready");
String database = null;
String jdbcDriver = null;
String tableName = null;
String menuSelection = null;
TableBuilderFrame tableMaker = null;
TableEditFrame tableEditor = null;
TableQueryFrame tableQuery = null;
DatabaseUtilities dbUtils = null;
InfoDialog infoDlg = null;

TableMenu tableMenu = new TableMenu();
EditMenu editMenu = new EditMenu();
ViewMenu viewMenu = new ViewMenu();
WindowMenu windowMenu = new WindowMenu();
HelpMenu helpMenu = new HelpMenu();

MenuListener menuListener = new MenuListener();

public DBManager(){
  setJMenuBar(menuBar);
  setTitle("Java Database Bible");
  setIconImage((new ImageIcon("od.gif")).getImage());
  getContentPane().setLayout(new BorderLayout());
  getContentPane().add(desktop,BorderLayout.CENTER);
  getContentPane().add(statusBar,BorderLayout.SOUTH);
  setSize(new Dimension(480,320));

  menuBar.add(tableMenu);
  tableMenu.setMenuListener(menuListener);

  menuBar.add(editMenu);
  editMenu.setMenuListener(menuListener);

  menuBar.add(viewMenu);
  viewMenu.setMenuListener(menuListener);

  menuBar.add(windowMenu);                  // added for Chapter 10
  windowMenu.setMenuListener(menuListener);

  menuBar.add(helpMenu);                    // added for Chapter 10
  helpMenu.setMenuListener(menuListener);
```

```
    setVisible(true);
  }

  private void displayTableBuilderFrame(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
                "Select table",JOptionPane.QUESTION_MESSAGE);
    tableMaker = new TableBuilderFrame(tableName);
    tableMaker.setCommandListener(new CommandListener());
    desktop.add(tableMaker);
    tableMaker.setSize(desktop.getSize());
    tableMaker.setVisible(true);
  }

  private void displayTableEditFrame(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
                "Select table",JOptionPane.QUESTION_MESSAGE);
    tableEditor = new TableEditFrame(tableName,dbUtils);
    desktop.add(tableEditor);
    tableEditor.setSize(desktop.getSize());
    tableEditor.setVisible(true);
  }

  private void displayTableQueryFrame(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
                "Select table",JOptionPane.QUESTION_MESSAGE);
    tableQuery = new TableQueryFrame(tableName,dbUtils);
    desktop.add(tableQuery);
    tableQuery.setSize(desktop.getSize());
    tableQuery.setVisible(true);
  }
  // added for Chapter 10
  private void displayInfoDialog(){
    infoDlg = new InfoDialog(dbUtils);
    Rectangle r = getBounds();
    infoDlg.setBounds(r.x+r.width-250,r.y+10,240,360);
    infoDlg.setVisible(true);
  }

  private void selectDatabase(){
    // revised for Chapter 10
    String[] databases = { "MSAccessContacts", "MySQLContacts",
                           "OracleContacts", "SQLServerContacts",
```

```
                        "SybaseContacts" };

database = (String)JOptionPane.showInputDialog(null, "Database:",
            "Select database", JOptionPane.QUESTION_MESSAGE,
            null, databases, databases[0]);

dbUtils = new DatabaseUtilities();

// added for Chapter 10
dbUtils.setJdbcDriverName(selectJDBCDriver());

if(database.equals("SQLServerContacts")&&
   jdbcDriver.equals("com.inet.tds.TdsDriver"))
   dbUtils.setDatabaseUrl("jdbc:inetdae7:localhost:1433");
dbUtils.setUserName("dba");
dbUtils.setPassword("sa");

if(!dbUtils.connectToDatabase(database)){
  statusBar.setMessage("Error connecting to "+database);
  return;
}
// added for Chapter 10
statusBar.setMessage("Retrieving MetaData from "+database);
statusBar.repaint();

System.out.println("Retrieving MetaData from "+database);
java.util.Date startTime = new java.util.Date();

MetaDataFrame dbTree = new MetaDataFrame(database,dbUtils);
java.util.Date endTime = new java.util.Date();
long elapsed = endTime.getTime() - startTime.getTime();
statusBar.setTimerMsg("Elapsed time = "+elapsed+" ms");

desktop.add(dbTree);
dbTree.setSize(desktop.getSize());
dbTree.setVisible(true);

tableMenu.enableMenuItem("New Table",true);
tableMenu.enableMenuItem("Drop Table",true);

editMenu.enableMenuItem("Insert",true);
editMenu.enableMenuItem("Update",true);
editMenu.enableMenuItem("Delete",true);
```

```
    viewMenu.enableMenuItem("ResultSet",true);

    helpMenu.enableMenuItem("Database Info",true);
  }
  // added for Chapter 10
  private String selectJDBCDriver(){
    String[] drivers = { "sun.jdbc.odbc.JdbcOdbcDriver",
                         "com.inet.tds.TdsDriver"};

    jdbcDriver = (String)JOptionPane.showInputDialog(null, "JDBCDriver:",
                "Select JDBC Driver", JOptionPane.QUESTION_MESSAGE,
                null, drivers, drivers[0]);
    return jdbcDriver;
  }

  private void executeSQLCommand(String SQLCommand){
    dbUtils.update(SQLCommand);
  }

  private void dropTable(){
    tableName = JOptionPane.showInputDialog(this,"Table:",
                "Select table",JOptionPane.QUESTION_MESSAGE);
  int option = JOptionPane.showConfirmDialog(null,
               "Dropping table "+tableName,
               "Database "+database,
               JOptionPane.OK_CANCEL_OPTION);
  if(option==0){
      executeSQLCommand("DROP TABLE "+tableName);
    }
  }
  // added for Chapter 10
  private int selected(){
    JInternalFrame[] jif = desktop.getAllFrames();
    for(int i=0;i<jif.length;i++){
      if(jif[i].isSelected())return i;
    }
    return 0;
  }
  // added for Chapter 10
  private void cascade(){
    JInternalFrame[] jif = desktop.getAllFrames();
    int j = selected();
```

```
      int nJifs = jif.length;
      j=(j<nJifs-1)?j+1:0;
      Dimension d = desktop.getSize();
      for(int i=0;i<nJifs;i++){
        jif[i].setBounds(
          new Rectangle(i*20,i*20,d.width-nJifs*20,d.height-nJifs*20));
        jif[i].toFront();
        j=(j<nJifs-1)?j+1:0;
      }
    }
    // added for Chapter 10
    private void tileVertically(){
      JInternalFrame[] jif = desktop.getAllFrames();
      int j = selected();
      int nJifs = jif.length;
      Dimension d = desktop.getSize();
      for(int i=0;i<nJifs;i++){
        jif[i].setBounds(new
Rectangle(i*d.width/nJifs,0,d.width/nJifs,d.height));
        jif[i].toFront();
        j=(j<nJifs-1)?j+1:0;
      }
    }
    // added for Chapter 10
    private void tileHorizontally(){
      JInternalFrame[] jif = desktop.getAllFrames();
      int j = selected();
      int nJifs = jif.length;
      Dimension d = desktop.getSize();
      for(int i=0;i<nJifs;i++){
        jif[i].setBounds(
          new Rectangle(0,i*d.height/nJifs,d.width,d.height/nJifs));
        jif[i].toFront();
        j=(j<nJifs-1)?j+1:0;
      }
    }

    class MenuListener implements ActionListener{
      public void actionPerformed(ActionEvent event){
        String menuSelection = event.getActionCommand();
        if(menuSelection.equals("Database")){
          selectDatabase();
        }else if(menuSelection.equals("New Table")){
```

```
        displayTableBuilderFrame();
      }else if(menuSelection.equals("Drop Table")){
        dropTable();
      }else if(menuSelection.equals("Insert")){
        displayTableEditFrame();
      }else if(menuSelection.equals("ResultSet")){
        displayTableQueryFrame();
      }else if(menuSelection.equals("Cascade")){// added for Chapter 10
        cascade();
      }else if(menuSelection.equals("Tile vertically")){
        tileVertically();
      }else if(menuSelection.equals("Tile horizontally")){
        tileHorizontally();
      }else if(menuSelection.equals("Database Info")){
        displayInfoDialog();
      }else if(menuSelection.equals("Exit")){
        System.exit(0);
      }
    }
  }

  class ExceptionListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      String exception = event.getActionCommand();
      JOptionPane.showMessageDialog(null, exception,
        "SQL Error", JOptionPane.ERROR_MESSAGE);
    }
  }

  class CommandListener implements ActionListener{
    public void actionPerformed(ActionEvent event){
      String SQLCommand = event.getActionCommand();
      executeSQLCommand(SQLCommand);
    }
  }
  public static void main(String args[]){
    DBManager dbm = new DBManager();
  }
}
```

The simplest way to illustrate the use of a JDBC application with different databases and different drivers is to move on to the next topic and to use the new version of the `DBManager` class as the basis of the examples.

The most noticeable effects of modifying the example to handle different RDBMS systems and different drivers are how minimal the required changes are and how pronounced a difference it makes to use the Opta2000 driver instead of the `jdbc:odbc` bridge in terms of speed.

Using DatabaseMetaData

The DatabaseMetaData interface provides the following types of information about the database:

- General information about the data source, including:
  - Database product name and version
  - Driver name
  - Database URL
- Feature support, such as:
  - SQL92 Support level
  - SQL keywords recognized
  - Transaction Isolation levels supported
  - Support of features such as batch updates
- Data-source limits including:
  - The maximum number of columns in a table
  - The maximum lengths of column and table names
- Information about the SQL objects the source contains, such as:
  - The types of tables in a catalog
  - The names of all tables of each type
  - Information about all columns in the tables

Many of the DatabaseMetaData methods return information in `ResultSets`, allowing you to use `ResultSet` methods such as `getString()` and `getInt()` to retrieve this information. If a given form of metadata is not available, these methods throw a SQLException. The next section illustrates how to retrieve information about the database.

Retrieving Information about the Database

Figure 10-2 illustrates the kind of information you can get about a database using the `DatabaseMetaData` object. The `JTree` displays the types of tables in the database, with the table names of each type of table displayed as child nodes of the table type. Tables can be expanded to display column names, and the columns themselves can be expanded to show information about the column.

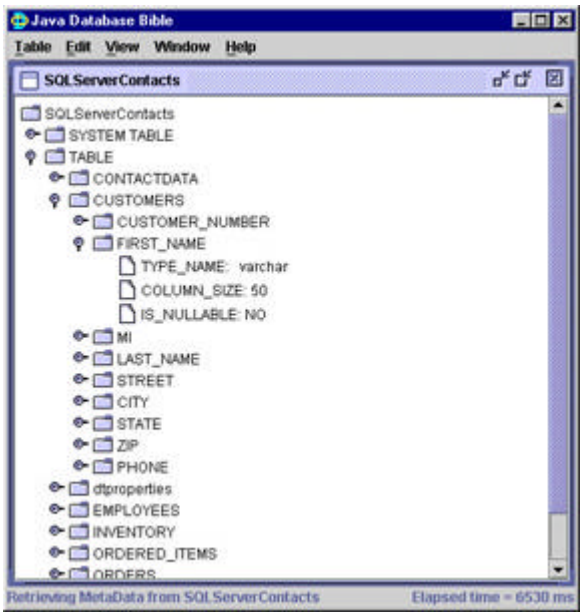


Figure 10-2: Tree view of tables in a database

Figure 10-2 also shows how long the application takes to get all the metadata for the display using the `sun.jdbc.odbc.JdbcOdbcDriver`. Contrast this elapsed time of nearly seven seconds with the elapsed time of just over two seconds for the Opta2000 driver, shown in Figure 10-3. Although the timing methodology used is by no means rigorous, the results speak for themselves.

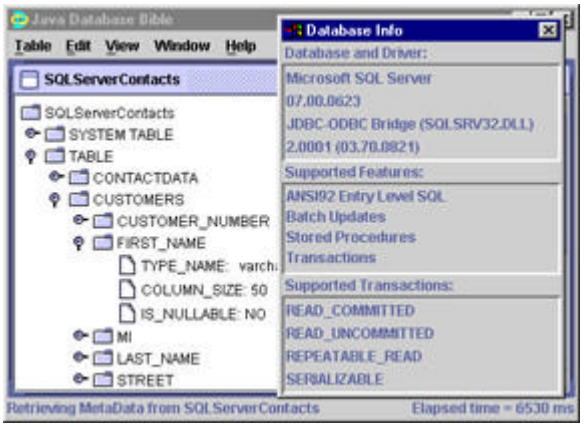


Figure 10-3: Additional DatabaseMetaData information

Many of the `DatabaseMetaData` methods take so-called "String pattern" arguments. These are arguments that may contain a mixture of Strings and wildcards. The wild cards conform to the normal wildcard rules for SQL Strings:

- "%" means match any substring of 0 or more characters.
- "_" means match any one character.

If a search-pattern argument is set to null, that argument's criteria will be ignored in the search.

Cross-Reference SQL escapes and wildcards are discussed in Chapter 3.

If a driver does not support a metadata method, a `SQLException` will normally be thrown. In the case of methods that return a ResultSet, either a ResultSet (which may be empty) is returned or a `SQLException` is thrown.

After connecting to the SQLServerContacts database, the `DatabaseMetaData` object is first queried for all table types, then for all tables within a type, then for columns within a table, and finally for information about the columns themselves. The results are used to populate the `JTree`.

A `DatabaseMetaData` object is created using the `Connection.getMetaData()` method. It is then used to get information about the database, as in the example shown in Listing 10-4, which gets the types of the tables in the database.

Listing 10-4: Retrieving table types

```
public Vector getTableTypes(){
  Vector typeVector = new Vector();

  try{
    Connection con = DriverManager.getConnection(url,userName,password);
    DatabaseMetaData dbmd = con.getMetaData();
    ResultSet rs = dbmd.getTableTypes();
    ResultSetMetaData md = rs.getMetaData();
    while(rs.next()){
      typeVector.addElement(rs.getString(1));
```

```
      }
    }
    catch(SQLException e){
      reportException(e);
    }
    return typeVector;
  }
```

The method `getTableTypes()` returns a ResultSet containing a single String column per row, identifying the table type. Typically, these types are as follows:

- TABLE
- VIEW
- SYSTEM TABLE

Using this table-type information, you can get the actual table names using the `getTables()` method. An example is shown in Listing 10-5.

Listing 10-5: Retrieving tables

```
  public Vector getTables(String[] types){
    Vector tableVector = new Vector();
    try{
      Connection con = DriverManager.getConnection(url,userName,password);
      DatabaseMetaData dbmd = con.getMetaData();
      ResultSet rs = dbmd.getTables(null,null,"%",types);
      ResultSetMetaData md = rs.getMetaData();
      int nColumns = md.getColumnCount();
      while(rs.next()){
        tableVector.addElement(rs.getString("TABLE_NAME"));
      }
    }
    catch(SQLException e){
      reportException(e);
    }
    return tableVector;
  }
```

The code to get the table names is similar to that used to get the table types. The most significant difference is in the argument list for the `getTables()` method. Since these types of arguments are fairly common when using metadata methods, it is worth discussing them in some detail.

The `getTables()` method takes these four arguments:

```
getTables(String catalog,
          String schemaPattern,
          String tableNamePattern,
          String[] types);
```

Here are explanations of each argument:

- `Catalog` — a pair of double quotes ("") retrieves tables without a catalog, and null retrieves all tables.
- `SchemaPattern` — a pair of double quotes ("") retrieves tables without a schema, and null retrieves all tables.
- `TableNamePattern` — This is a table-name pattern similar to the argument used with SQL "LIKE". The"%" matches any substring of 0 or more characters, and "_" matches any one character.
- `Types` — an array of table types to include; null returns all types.

The `getTables()` method returns a ResultSet containing descriptions of the tables available in a catalog. The result set contains the columns shown in Table 10-1.

Table 10-1: Columns Returned by getTables()

Column	Column Name	Type	Contents
1.	TABLE_CAT	String	table_catalog (may be null)
2.	TABLE_SCHEM	String	table_schema (may be null)
3.	TABLE_NAME	String	table_name
4.	TABLE_TYPE	String	table_type: "TABLE", "VIEW", "SYSTEM TABLE", etc.
5.	REMARKS	String	remarks explanatory comment on the table

Note

Some databases may not return information for all tables.

The `DatabaseMetaData` object also provides a mechanism to retrieve detailed information about the columns in a table through the use of the `getColumns()` method. Like the `getTables()` method, the `getColumns()` method returns a ResultSet. The method's argument list is also similar in that it takes a number of String patterns:

```
public ResultSet getColumns(String catalog,
                            String schemaPattern,
                            String tableNamePattern,
                            String columnNamePattern);
```

Here are explanations of each argument:

- `Catalog` — A pair of double quoutes ("") retrieves tables without a catalog, and null retrieves all tables.
- `SchemaPattern` — The double quotes ("") retrieve tables without a schema, and null retrieves all tables.
- `TableNamePattern` — This is a table name pattern similar to the argument used with SQL "LIKE". The "%" matches any substring of 0 or more characters; "_" matches any one character.
- `columnNamePattern`— This is a column name pattern similar to the argument used with SQL "LIKE". The "%" matches any substring of 0 or more characters; "_" matches any one character.

Using the table information that the code in Listing 10-5 returns, you can get column information using the `getColumns()` method. An example is shown in Listing 10-6.

Listing 10-6: Retrieving column data

```
public Vector getColumns(String tableName){
  Vector columns = new Vector();
  Hashtable columnData;
  try{
    Connection con = DriverManager.getConnection(url,userName,password);
```

```
      DatabaseMetaData dbmd = con.getMetaData();
      String catalog = con.getCatalog();
      ResultSet rs = dbmd.getColumns(catalog,"%",tableName,"%");
      ResultSetMetaData md = rs.getMetaData();
      int nColumns = md.getColumnCount();
      String value;
      while(rs.next()){
        columnData = new Hashtable();
        for(int i=1;i<=nColumns;i++){
          value = rs.getString(i);
          if(value==null)value="<NULL>";
          columnData.put(md.getColumnLabel(i),value);
        }
        columns.addElement(columnData);
      }
    }
    catch(SQLException e){
      reportException(e);
    }
    return columns;
  }
```

The code examples of Listings 10-5, 10-5, and 10-6 are additions to the `DatabaseUtilities` class. Further examples of the use of DatabaseMetaData are given later in this chapter.

Note

Many of the DatabaseMetaData methods have been added or modified in JDBC 2.0 and JDBC 3.0, so if your driver is not JDBC 2.0 or JDBC 3.0 compliant, a SQLException may be thrown by some DatabaseMetaData methods.

Displaying DatabaseMetaData in a JTree

The class required to display the table and column data retrieved from the `DatabaseMetaData` object is an extension of `JInternalFrame.` This is used to display a `JTree` in a `JScrollPane`, as shown in Listing 10-7.

Listing 10-7: Displaying DatabaseMetaData in a JTree

```
package JavaDatabaseBible.part2;

import java.awt.*;
import java.util.Hashtable;
import java.util.Vector;
import javax.swing.*;
import javax.swing.JTree;
import javax.swing.border.*;
import javax.swing.tree.*;
```

```
class MetaDataFrame extends JInternalFrame{

  protected JTree tree;
  protected JScrollPane JTreeScroller = new JScrollPane();
  protected DatabaseUtilities dbUtils;
  protected String dbName;
  protected String[] tableTypes;
  protected JPanel JTreePanel = new JPanel();

  public MetaDataFrame(String dbName, DatabaseUtilities dbUtils){
    setLocation(0,0);
    setClosable(true);
    setMaximizable(true);
    setIconifiable(true);
    setResizable(true);
    getContentPane().setLayout(new BorderLayout());
    this.dbName=dbName;
    this.dbUtils=dbUtils;
    setTitle(dbName);
    init();
    setVisible(true);
  }

  // initialise the JInternalFrame
  private void init(){
    JTreePanel.setLayout(new BorderLayout(0,0));
    JTreePanel.setBackground(Color.white);
    JTreeScroller.setOpaque(true);
    JTreePanel.add(JTreeScroller,BorderLayout.CENTER);
    DefaultTreeModel treeModel = createTreeModel(dbName);
    tree = new JTree(treeModel);
    tree.setBorder(new EmptyBorder(5,5,5,5));
    JTreeScroller.getViewport().add(tree);
    JTreePanel.setVisible(true);
    JTreeScroller.setVisible(true);
    tree.setRootVisible(true);
    tree.setVisible(true);
    getContentPane().add(JTreePanel,BorderLayout.CENTER);
  }

  // Create a TreeModel using DefaultMutableTreeNodes
  protected DefaultTreeModel createTreeModel(String dbName){
    DefaultMutableTreeNode treeRoot =  new DefaultMutableTreeNode(dbName);
```

```
    Vector tableTypes = dbUtils.getTableTypes();
    for(int i=0;i<tableTypes.size();i++){
      DefaultMutableTreeNode tableTypeNode =
        new DefaultMutableTreeNode((String)tableTypes.elementAt(i));
      treeRoot.add(tableTypeNode);
      String[] type = new String[]{(String)tableTypes.elementAt(i)};
      Vector tables = dbUtils.getTables(type);
      for(int j=0;j<tables.size();j++){
        DefaultMutableTreeNode tableNode =
          new DefaultMutableTreeNode(tables.elementAt(j));
        tableTypeNode.add(tableNode);
        Vector columns = dbUtils.getColumns((String)tables.elementAt(j));
        for(int k=0;k<columns.size();k++){
          Hashtable columnData = (Hashtable)columns.elementAt(k);
          DefaultMutableTreeNode columnNode =
            new DefaultMutableTreeNode(columnData.get("COLUMN_NAME"));
         columnNode.add(new DefaultMutableTreeNode("TYPE_NAME:
            "+columnData.get("TYPE_NAME")));
         columnNode.add(new DefaultMutableTreeNode("COLUMN_SIZE:
            "+columnData.get("COLUMN_SIZE")));
         columnNode.add(new DefaultMutableTreeNode("IS_NULLABLE:
            "+columnData.get("IS_NULLABLE")));
         tableNode.add(columnNode);
        }
      }
    }
   return new DefaultTreeModel(treeRoot);
 }
}
```

Most of the work in Listing 10-7 is done in the `createTreeModel()` method. This method first calls the `getTableTypes()` method shown in Listing 10-4 to get a vector of table-type names. These are used to create DefaultMutableTreeNodes attached to the root node representing the selected database.

For each table type, a vector of table names of that type is returned by the `getTables()` method shown in Listing 10-5. These are used to create DefaultMutableTreeNodes that are attached to the table-type nodes representing each of the tables.

Finally, for each table, a vector of Hashtables of column descriptors is obtained by calling the `getColumns()` method shown in Listing 10-6. This information is used to create the column node and column-information child nodes shown in Figure 10-2. Only a small amount of the available column information is used in this display for reasons of clarity. Additional fields that the `getColumns()` method makes available include those listed in Table 10-2.

Table 10-2: Column Information Provided by getColumns()

Column Name	Type	Meaning
TABLE_CAT	String	table catalog (may be null)

Table 10-2: Column Information Provided by getColumns()

Column Name	Type	Meaning
TABLE_SCHEM	String	table schema (may be null)
TABLE_NAME	String	table name
COLUMN_NAME	String	column name
DATA_TYPE	short	SQL type from java.sql.Types
TYPE_NAME	String	Data source dependent type name
COLUMN_SIZE	int	column size.
DECIMAL_DIGITS	int	the number of fractional digits
NUM_PREC_RADIX	int	Radix (typically either 10 or 2)
NULLABLE	int	▪ columnNoNulls - might not allow NULL values ▪ columnNullable - definitely allows NULL values ▪ columnNullableUnknown - nullability unknown
REMARKS	String	comment describing column (may be null)
COLUMN_DEF	String	default value (may be null)
CHAR_OCTET_LENGTH	int	the maximum number of bytes in the column
ORDINAL_POSITION	int	index of column in table (starting at 1)
IS_NULLABLE	String	▪ "NO" means column definitely does not allow NULLs. ▪ "YES" means the column might allow NULL values. ▪ An empty string means nullability unknown.

In addition to information about the structure of the database, you will frequently find it useful to know something about the capabilities of the RDBMS itself. The methods supported by the `DatabaseMetaData` object to provide this type of information are discussed in the next section.

Retrieving Information about RDBMS Functionality

In addition to describing the structure of the database, the `DatabaseMetaData` object provides methods to access to a great deal of general information about the RDBMS itself. Some of the information you can retrieve about the database-management system is illustrated in Figure 10-3.

The example shown in Figure 10-3 shows that the SQLServerContacts database is running under SQL Server 7, using the Opta2000 pure Java driver from i-net Software. Also listed are some of the features that this database configuration supports.

The elapsed time shown in the status bar is the time to access and display the tree view of the DatabaseMetaData, as shown in Figure 10-2. The difference between the elapsed time of just over two seconds using the Opta2000 driver and nearly seven seconds using the jdbc-odbc bridge illustrated in Figure 10-3 is significant. The code required to retrieve this information is shown in Listing 10-8.

Listing 10-8: Retrieving information about the RDBMS

```
package JavaDatabaseBible.part2;
```

```
import java.awt.*;
import java.util.Hashtable;
import java.util.Vector;
import javax.swing.*;
import javax.swing.JTree;
import javax.swing.border.*;
import javax.swing.tree.*;


public class InfoDialog extends JDialog{
  protected DatabaseUtilities dbUtils = null;
  protected JPanel dbInfoPanel = new JPanel();
  protected JPanel featuresPanel = new JPanel();
  protected JPanel topPanel = new JPanel(new BorderLayout());
  protected JPanel centerPanel = new JPanel(new BorderLayout());
  protected JPanel bottomPanel = new JPanel(new BorderLayout());

  public InfoDialog(DatabaseUtilities dbUtils){
    this.dbUtils=dbUtils;
    setTitle("Database Info");
    getContentPane().setLayout(new BorderLayout());

    String[] dbInfo = dbUtils.databaseInfo();
    dbInfoPanel.setLayout(new GridLayout(dbInfo.length,1,2,2));
    for(int i=0;i<dbInfo.length;i++){
      dbInfoPanel.add(new JLabel(dbInfo[i]));
    }
    dbInfoPanel.setBorder(new CompoundBorder(
                          new BevelBorder(BevelBorder.LOWERED),
                          new EmptyBorder(2,2,2,2)));
    topPanel.add(new JLabel(" Database and Driver:"),BorderLayout.NORTH);
    topPanel.add(dbInfoPanel,BorderLayout.CENTER);
    getContentPane().add(topPanel,BorderLayout.NORTH);

    String[] features = dbUtils.featuresSupported();
    featuresPanel.setLayout(new GridLayout(features.length,1,2,2));
    for(int i=0;i<features.length;i++){
      featuresPanel.add(new JLabel(features[i]));
    }
    featuresPanel.setBorder(new CompoundBorder(
                            new BevelBorder(BevelBorder.LOWERED),
                            new EmptyBorder(2,2,2,2)));
    centerPanel.add(new JLabel(" Supported Features:"),BorderLayout.NORTH);
```

```
    centerPanel.add(featuresPanel,BorderLayout.CENTER);
    getContentPane().add(centerPanel,BorderLayout.CENTER);
  }
}
```

Clearly, this example illustrates only a small percentage of the data available through the use of the `DatabaseMetaData` object. It is well worth referring to the Javadocs available on the Sun Web site at: `http://java.sun.com/j2se/1.4/docs/api/java/sql/package-summary.html`.

Note

A shorter link is `http://java.sun.com/docs/`. This takes you to he main Javadocs page, and you can navigate from there.

In addition to `DatabaseMetaData` methods, JDBC provides a large number of useful methods for accessing information about the `ResultSet` returned by a query. The next section discusses these methods.

Using ResultSetMetaData

The `ResultSetMetaData` object is similar to the `DatabaseMetaData` object, with the exception that it returns information specific to the columns in a `ResultSet`.

Information about the columns in a ResultSet is available by calling the `getMetaData()` method on the ResultSet. The `ResultSetMetaData` object returned gives the number, types, and properties of its `ResultSet` object's columns.

Table 10-3 shows some of the more commonly used methods of the `ResultSetMetaData` object.

Table 10-3: ResultSetMetaData Methods

ResultSetMetaData method	Description
`getColumnCount()`	Returns the number of columns in the ResultSet
`getColumnDisplaySize(int column)`	Returns the column's max width in chars
`getColumnLabel(int column)`	Returns the column title for use in displays
`getColumnName(int column)`	Returns the column name
`getColumnType(int column)`	Returns the column's SQL data-type index
`getColumnTypeName(int column)`	Returns the name of the column's SQL data type
`getPrecision(int column)`	Returns the number of decimal digits in the column
`getScale(int column)`	Returns the number of digits to the right of the decimal point
`getTableName(int column)`	Returns the table name
`isAutoIncrement(int column)`	Returns true if the column is autonumbered
`isCurrency(int column)`	Returns true if the column value is a currency value
`isNullable(int column)`	Returns true if the column value can be set to NULL

Listing 10-9 illustrates the use of the `ResultSetMetaData` methods `getColumnCount` and `getColumnLabel` in an example where the column names and column count are unknown.

Listing 10-9: Using ResultSetMetaData

```
  public void printResultSet(String query){
```

```
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection ("jdbc:odbc:Inventory");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      ResultSetMetaData md = rs.getMetaData();

      int nColumns = md.getColumnCount();
      for(int i=1;i<=nColumns;i++){
        System.out.print(md.getColumnLabel(i)+((i==nColumns)?"\n":"\t"));
      }
      while (rs.next()) {
        for(int i=1;i<=nColumns;i++){
          System.out.print(rs.getString(i)+((i==nColumns)?"\n":"\t"));
        }
      }
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
```

This example will print the ResultSet returned by a query to a file called "rs.txt". The command line to run the example is:

```
java printResultSet jdbc:odbc:Contacts "SELECT * FROM CONTACT_INFO"
```

The output is tab delimited, so that it can easily be imported into MSWord. The example first retrieves the column count for the ResultSet, then loops through the columns to get the column labels, which are printed as the first line. It then loops through all the rows, retrieving the data in an inner loop. Table 10-4 shows the ResultSet output by the command line shown above.

Table 10-4: Formatting a ResultSet using ResultSetMetaData

FIRST_NAME	MI	LAST_NAME	STREET	CITY	STATE	ZIP
Michael	A	Corleone	123 Pine	New York	NY	10006
Fredo	X	Corleone	17 Main	New York	NY	10007
Sonny	A	Corleone	123 Walnut	Newark	NJ	12346
Francis	X	Corleone	17 Main	New York	NY	10005
Vito	G	Corleone	23 Oak St	Newark	NJ	12345
Tom	B	Hagen	37 Chestnut	Newark	NJ	12345
Kay	K	Adams	109 Maple	Newark	NJ	12345
Francis	F	Coppola	123 Sunset	Hollywood	CA	23456
Mario	S	Puzo	124 Vine	Hollywood	CA	23456
Michael	J	Fox	109 Sepulveda	LA	CA	91234
James	A	Caan	113 Sunset	Hollywood	CA	92333

Summary

This chapter combines the examples in Chapters 5-9 to create the basis of a useful database-management tool and test platform. In the process, you learn about:

- Using DatabaseMetaData
- Using ResultSetMetaData
- Comparing the performance of different drivers

In Part 2 as a whole, you learn how to use the JDBC Core API to create, maintain, and query a database. You also gain hands-on experience in creating a practical client/server application

Part III explores the JDBC 2.0 Extension API in the context of a web application example. Web applications tend to be heavily database oriented, since they frequently involve a lot of form handling, and the need to upload and download large data items using streams and large data base objects.

Part III: A Three-Tier Web Site with JDBC

Chapter List

Part Overview

A significant part of Java's success has been its application to server-side programming. One of the most widespread applications of Java is the creation of dynamic Web sites using servlets, JSPs, and databases. This part discusses the JDBC Extension API in the context of developing a membership-based Web application.

The Web application employs a three-tier architecture built around an Apache/Tomcat-based Web server that implements the business logic in Servlets and Java Server Pages. The Web server uses JDBC to connect to a database server.

Since this part deals with Web applications, it includes an introduction to using Servlets and Java Server Pages. Basic handling of HTML forms is also discussed.

More advanced form handling using `PreparedStatements` and `CallableStatements` is discussed in a subsequent chapter. This chapter also discusses how to create stored procedures in SQL.

A relatively new feature in most databases is support for large objects such as images and entire documents. Examples of uploading and storing image files as binary large objects, and downloading them for display, are the subjects of another chapter.

In addition to discussing straightforward HTML applications, the possibilities of retrieving data as XML are discussed. Examples illustrate how to use the same XML document to create two completely different Web pages using an XSL transformation. This example also illustrates the use of scrollable ResultSets.

This part closes with a chapter on writing an e-mail application. The application uses JDBC and SQL with the JavaMail API to automate e-mail generation and to read and save e-mail to a database.

Chapter 11: Building a Membership Web Site

In This Chapter

An area in which Java and databases are used together very frequently is in creating dynamic Web sites. Part III of this book illustrates the use of the JDBC Extension API in the context of a membership Web site. The Web site is built around a membership database that incorporates a number of different tables. This chapter discusses the design of this database.

The application design uses a three-tier architecture built around an Apache/Tomcat-based Web server. Apache and Tomcat provide HTTP service and Servlet/Java Server Pages (JSP) support, respectively. Several alternative products can handle these tasks very adequately, but Apache and Tomcat have several advantages over most of the others, not the least of which is that they both run on all common platforms. Appendix B is a guide to downloading and installing Apache and Tomcat. Both are easy to install and run on Windows or Linux/Unix platforms.

Another important reason for selecting the Apache server to provide HTTP service and is that Apache is the most widely used server on the Internet at the present time, having been selected for over 60 percent of all web sites. This means, of course, that Apache is the server you are most likely to be using. Similarly, Tomcat was chosen as the Servlet and JSP engine since Sun selected Tomcat as the reference implementation for servlets and JSP applications, thus there is a high probability that you will use it at some time. A final advantage is that both are available for free download from `jakarta.org`.

Designing a Multi-Tier System

Partitioning a design into tiers allows you to apply various off-the-shelf technologies as appropriate for a given situation. For example, a browser displaying Web pages generated from JSP pages and servlets in the Web tier handles the client tier. This means that all you have to do is comply with the HTTP specifications and avoid any technologies that all the browsers you expect to encounter do not support.

Since the browser and the RDBMS are, essentially, off-the-shelf products with clearly defined interfaces, the following chapters concentrate on the business and presentation logic required to interface to them. The interface to the browser is handled through the Web server, which serves static Web pages and provides a front end for Tomcat. Tomcat is responsible for serving the dynamic Web content created in Java using servlets and JSP pages.

The structure of the three-tier system is shown in Figure 11-1. On the left is the client machine running a standard Web browser; in the center is the Web server; and on the right is the database server.

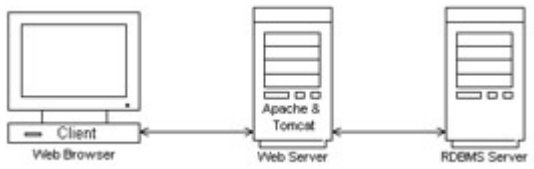


Figure 11-1: Three-tier Internet application

The business and data-presentation logic is handled using Java and JSPs in the Web-server tier. The database itself can use virtually any RDBMS. The examples in the following chapters are based on SQL Server and the Opta2000 drivers from Inet Software. This choice was made largely because the Opta2000 drivers are a good example of a family of pure Java drivers that support the JDBC Extension API, the primary topic of Part III of this book. Opta2000 drivers are available for most major databases.

Just as you will almost certainly have no trouble figuring out how to change parts of the sample code that refer to my user name, password, and server name, I am sure you will have no trouble figuring out how to switch to a different RDBMS using different drivers. The degree of difficulty involved in either case is similar.

The examples concentrate on the servlets and JSPs, and the JavaBeans encapsulating the business logic. These examples also illustrate various aspects of the JDBC Extension API.

Cross-Reference

One of the strengths of JDBC is that it is designed to plug and play with virtually any relational database management system with a minimum of effort. The use of different relational database management systems is discussed in Part II, with extensive examples in Chapter 10.

The first step in designing the Web site is to define the functionality of the site and to design the underlying database. Designing the database around the Web pages it supports makes the Java code simpler and faster to implement. The functional requirements of the membership Web site application are discussed in the next section.

Functional Requirements

The following chapters describe a membership web site that allows members to auction their vehicles over the Internet. The main reason for choosing this theme is to exploit the opportunities it provides to discuss the following important JDBC topics in the context of practical examples:

- HTML form handling with servlets, JSP, and JDBC
- Using scrollable `ResultSets` in a search engine
- Using updatable `ResultSets` to allow a member to call up and modify his or her profile
- Handling image upload, storage, and retrieval using HTML forms and blobs
- Using the JavaMail API with JDBC to send and receive e-mail

The use of XML and XSLT to create different Web pages from the same `ResultSet` is also discussed in the context of using updateable `ResultSets` to display data in one format and edit it in another format. The examples in Part IV discuss the use of XML with JDBC in more detail.

The sample application supports the functionality common to most commercial catalog sites as well as the normal features of a membership site. These include the following:

- Member login
- New member registration
- Member data entry
- Upload and storage of large objects such as images
- Site search
- Summary page display, with thumbnail photos
- Links from the summary pages to detail pages
- Automated email support

The best way to understand the logical structure of the Web site is to use a block diagram. The logical structure of the Web site discussed in Chapters 11 through 16 is illustrated in Figure 11-2.

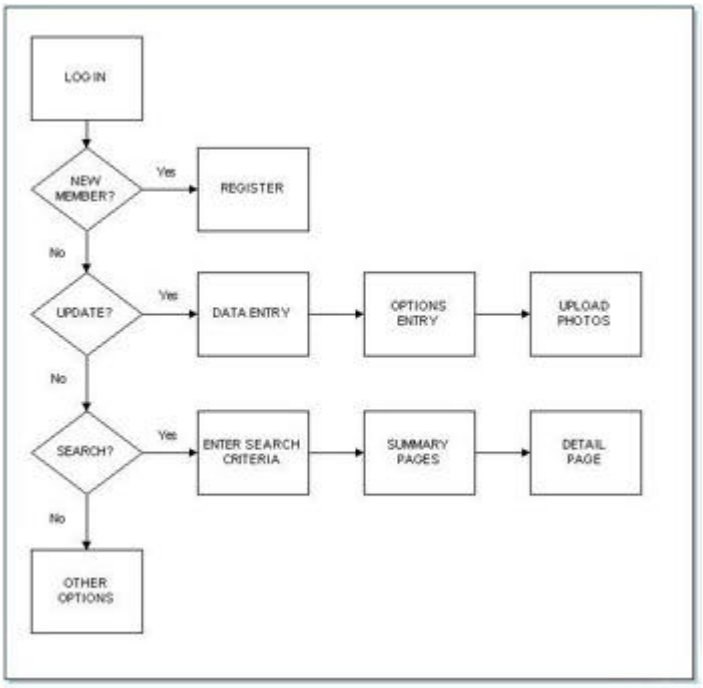


Figure 11-2: Structure of Web site developed in Chapters 11-16

The member login and registration process involves displaying HTML forms and processing their input. The examples use both servlet and Java Server Page approaches using JavaBeans to encapsulate logic functions. In addition to handling simple text-based forms, Chapter 14 shows you how to upload images from a browser page and store them in a database.

Having reviewed the functional requirements of the application, you are ready to design the database. The design of the database for this application is discussed in the next section.

Cross-Reference The theoretical aspects of database design, are discussed in Part I. It is particularly important to understand the use of primary anf foreign keys, as well as the Normal Forms. Both of these topics are discussed in Chapter 2.

Designing the Database

As a catalog site, the sample application has to support the basic functionality common to most commercial catalog sites. The primary functions supported include the following:

- Handling member logins
- Member registration
- Data entry
- Site search, with a summary display capability
- Detailed display of database items
- Database-driven e-mail using the JavaMail API

The examples don't get into secure sockets and payment handling because those topics are not really database related. The subject of this chapter is the overall design of the Web site and the underlying database.

Handling Member Logins

Users are first required to respond to a login-request form, with the usual user name and password combination. There are three possible outcomes to a login attempt:

- Successful login with the correct username and password, permitting site access
- Failed login attempt with a valid user name but an invalid password
- Failed login attempt with an invalid user name

For the purposes of this application, login with a valid user name and a bad password results in a prompt for an e-mail reminder, and completely erroneous logins lead the user along a registration trail. Although this is a simplistic approach, it serves to illustrate the technology and provides a starting point for a more complete solution.

The Login Table itself is very simple. It uses only the three columns illustrated in Table 11-1. The UserName column is the primary key and, as such, is indexed for speed of access. The price of fast access for returning users is that inserting new users is slower because of the need to build the index.

Table 11-1: Login Table

UserName	Password	MemberID
axman	hatchet	7
batman	robin	3
cat	balou	8
garfield	lasagna	1
snoopy	peanuts	2

The Password column is used simply for user validation, as shown in the examples in Chapter 12. The MemberID column, however, is the key to accessing all the other tables.

The importance of the MemberID field lies in the fact that it is the unique identifier used to access member-specific data from all the other tables. In most of the tables, MemberID is also the primary key, since each member has only one entry in most of the tables.

Some of the tables, however, have their own primary keys and use MemberID as a foreign key. For example, all member photos are stored as binary large objects (blobs) in the Photos Table. This table contains a unique PhotoID, which is the primary key, and the MemberID, which is a foreign key used to associate the photo with a specific member. The Photos Table also contains a column for the photos themselves, as well as a descriptor column used for selecting individual photos.

Member Registration

When a new member goes into the registration process, the system displays an HTML form for the member to complete. The data from the form is then saved to the Contact_Info table, shown in Table 11-2.

Table 11-2: Contact_Info Table

ID	FNAME	MI	LNAME	STREET	CITY	ST	ZIP	PHONE	EMAIL
1	Vito	A	Corleone	123 Main St	New York	NY	10002	212-555-0000	vito@home.com
2	Fred	A	Tagliatelle	123 Main	Phila.	PA	12345	123-456-7890	fat@cn.com
7	Al	X	Edwards	123 Pine	New York	NY	12345	123-456-1111	axman@abc.com

The Contact_Info Table is the only place in the database where the name and address information of the members is stored. The primary use of the Contact_Info table is for billing and administrative purposes.

The primary key for the Contact_Info Table is MemberID (shown in the tables as ID because of space limitations). Columns in the Contact_Info Table that require indexes are the following:

- ID — used extensively, any time you need data from the table.
- City, State, Zip — used in regional searches.

After completing the registration form, the member will be given the option of entering vehicle information for the auction part of the site. Vehicle data is stored primarily in the Product_Info and Options tables.

Data Entry

The vehicle data will be divided amongst a number of tables, both for convenience in data entry and to improve the efficiency of searches. The primary table will be the Product_Info table, which will contain such data as the make, model and year of the vehicle. Secondary tables will be used for less important data such as the optional accessories and photos of the vehicles.

The primary table: Product_Info

The Product_Info Table, shown in Table 11-3, is used in most of the searches, so it is important to ensure that it can be searched efficiently. This means that many of the columns will be indexed.

The primary key is Vehicle_ID. This key will also be used as the primary key of the Options table with which the Product_Info table will have a one-to-one relationship. Note the use of the Member_ID column as a foreign key linking the vehicle to its owner in the Contact_Info table.

Table 11-3: Product_Info Table

Vehicle_ID	Member_ID	Make	Body	Model	Year	Color
1000	1	Honda	Coupe	Civic	1996	Red
1001	1	Mitsubishi	SUV	Montero	2000	Green
1002	2	GM	Pickup	Sonoma	1999	Red

The Product_Info Table is updated using an HTML form, as shown in Figure 11-3. The data-entry form uses combo boxes extensively to minimize data-entry errors. It is particularly important to ensure that all terms that may be used in searches are input using combo boxes. The reason for this is purely practical: given the opportunity, a certain percentage of the users will enter data in the wrong place or in a format that makes it useless in a search, so free text fields are used only where no search capability is provided.

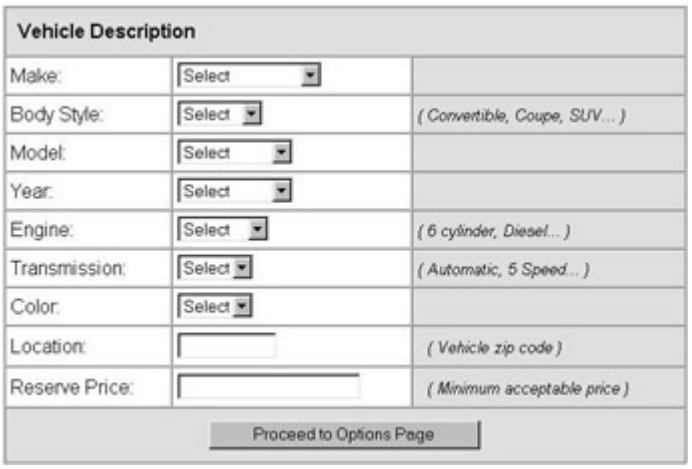


Figure 11-3: Data-entry form using combo-boxes to reduce data-entry errors

The Product_Info Table is worth looking at from the viewpoint of breaking a single table into two or more separate tables. In an application like this, there is a high probability that most searches will be conducted for vehicles of a specific type, such as pickups or SUVs. Breaking a large table into several separate, smaller tables organized by common search categories will obviously speed up searches.

Note that you can also enforce some restrictions on searches to improve response times. For example, dividing your database by regions is probably practical, since it is unlikely that members will really need to search beyond their local area. Using a combo box in a search form is an excellent way to do this.

Secondary tables populated using check boxes

Apart from the special tables used to store photographs and large blocks of free text, the remaining tables are all very similar, storing boolean variables to identify specific characteristics such as product options. These tables are intended to be easy to search.

In a larger application, where database items may be described by a large number of different characteristics, you may prefer to split the descriptions among a number of tables to simplify data entry. In this way, each table can map to a single HTML form. Dedicated JSP pages can handle the form data by using the same generic `SQLInsertBean` before forwarding the user to the next page.

The `SQLInsertBean` uses an enumeration to iterate through the http request parameters and create a SQL insert statement that saves the data. Listing 11-1 illustrates this technique.

Listing 11-1: Generic form handling using an enumeration

```
// use Enumeration to get param names and values from HTTP request

for(Enumeration e=request.getParameterNames();e.hasMoreElements();){
  pNames[i] = (String)e.nextElement();
  Values[i]  = request.getParameter(pNames[i]);
  ++i;
}

// create fieldNames and fieldValues Strings for SQL INSERT command
String fieldNames = "ID,";
String fieldValues = "'" + memberID + "',";

// append parameter names and values to fieldNames and fieldValues
for(int j=0;j<i;j++){
  if(!pNames[j].equals("DBName")&&
     !pNames[j].equals("TableName")&&
     !pNames[j].equals("SubmitButton")){
       fieldNames  += pNames[j] + ",";
       fieldValues += "'" + fixApostrophes(Values[j]) + "',";
  }
}

// strip trailing commas
fieldNames = fieldNames.substring(0,fieldNames.length()-1);
```

```
fieldValues = fieldValues.substring(0,fieldValues.length()-1);

// create SQL command
SQLCommand = "INSERT INTO " + tableName + " ("+fieldNames+") "+
             "VALUES ("+fieldValues+")";
```

Cross-Reference Chapter 12 explains how servlets and JSP pages work and how to use them to handle HTML forms.

This generic approach to handling the form data means that although the table structures have to track the HTML forms, the middle-tier code can be independent of both. This approach makes maintenance much easier, since you may find that you have to add new tables or modify existing tables.

Like the Product_Info Table, most of the tables are completed using data from forms designed to minimize data-entry errors. In this case, most of the entries are made using check boxes, as shown in Figure 11-4.

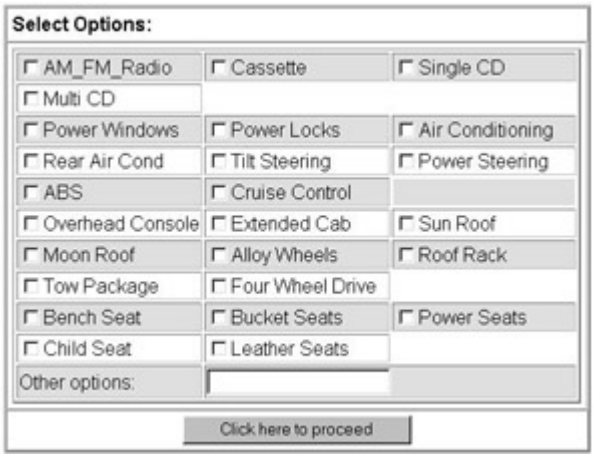


Figure 11-4: Data entry form using check boxes

As you can see, a single free-form text field supports the check boxes, as shown in Figure 11-4. Again, the rationale for this approach is to minimize data-entry errors. The check boxes map to boolean variables that are quick and easy to search. Assuming that most of the popular options are covered by the check boxes, the free-form text entries can simply be ignored for search purposes.

Table 11-4 shows a simplified subset of the table completed using the HTML form of Figure 11-5. The most significant column in Table 11-4 is the `List` column, which is used to provide a summary of the items in the table for display purposes. The data for this column is synthesized when the table is updated by creating a string from all the data-entry field names, plus the contents of the "Other options" field.

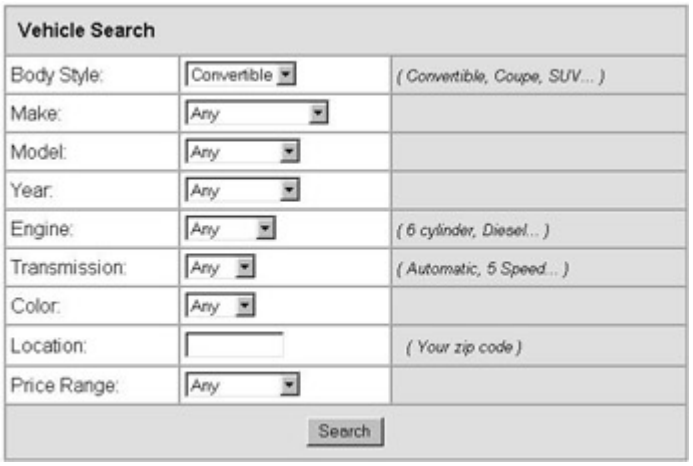


Figure 11-5: Database searches are performed using an HTML Search Form

Table 11-4: Part of Options Table

Tow_Bar	4WD	Other	List
0	0		AM/FM Radio, Cassette, Moon roof, Power windows
0	1	Entertainment center	AM/FM Radio, CD Changer, Moon roof
1	0		AM/FM Radio, CD, Power locks,Power windows

Tables used for photos and text objects

The tables discussed up to this point are structured so that they are easy to search. Support for easy searching has been carried through to the HTML forms used to populate the tables. The database includes the following additional tables that are not searchable:

- Photos, which contains member photos as blobs
- BodyText, which contains free-form text

These tables are also accessed using the MemberID. The Photos Table uses MemberID as a foreign key, because the primary key is the PhotoID. The Photos Table is interesting primarily because it stores the photos as blobs. These require special handling, since they are accessed as streams or byte arrays using pointers known in SQL terminology as *locators*. Uploading photos from a browser is also an interesting topic, since it involves special handling not included in the basic HTML form support that the servlet object provides.

Cross-Reference Chapter 14 explains how to use a servlet to upload images from a browser page and store them in a database as blobs.

Once the data has been stored in the database, it is accessible to members via a search form. The search capabilities of the Web site are discussed in the next section.

Searching the Database

Searches of the database are carried out using the search form illustrated in Figure 11-5. Notice the similarities between the search form and the data entry form of Figure 11-3.

The results of a search are presented in summary form, showing several database item summaries per page. Each summary item includes a thumbnail image that is downloaded from the database using a servlet. The general appearance of a summary is shown in Figure 11-6.

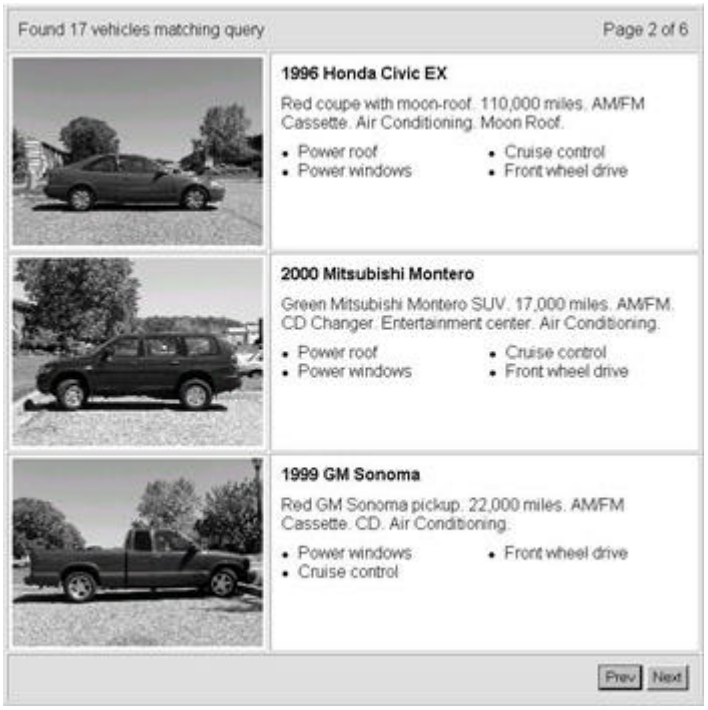


Figure 11-6: The Summary pages provide summaries of several of the items in the database.

From a summary page, the user is able to click the thumbnail image and select a more detailed page. The detailed page is actually retrieved as XML and processed with an XSL stylesheet on the server to create the detail page, a sample of which is shown in Figure 11-7.

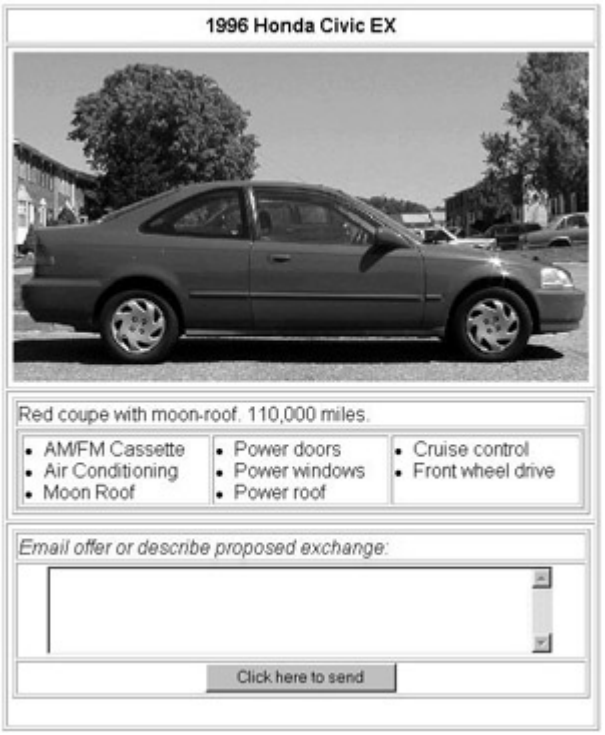


Figure 11-7: The detail page displays a larger image and additional information.

The XML approach to creating the detail page is selected primarily to illustrate the use of an updateable ResultSet, which is displayed either as the profile shown previously or as a preloaded XML form ready for editing. Generating the two completely different display formats from the same XML document is made possible using XSLT to transform the XML into different HTML documents.

Database-Driven E-mail

The detail page includes a text area that allows the user to send an e-mail to the owner of the vehicle. E-mails are handled through a JavaMail application, which obtains the sender and recipient information from the database and forwards the message.

The final chapter of Part III of this book illustrates the use of JDBC with the JavaMail API. The combination of JDBC and JavaMail lets you send e-mails to members automatically. It also allows you to receive e-mails and save them directly to a database.

Summary

This chapter provides an overview of the design of a three-tier, database-driven Web site application. The object of the chapter is to review practical aspects of the application in terms of the way the database tables relate to the pages the user views. In addition, the chapter looks at the following topics:

- Using primary and foreign keys
- Using indexes for better performance

Chapter 12 discusses Java servlets and JSP pages and how to use them to handle HTML forms. Subsequent chapters expand this base to discuss much of the JDBC Extension API.

Chapter 12: Using JDBC DataSources with Servlets and Java Server Pages

In This Chapter

Servlets and Java Server Pages (JSP) extend the power of Java technology to server-side applications. They are Java technology's answer to CGI programming, for they enable the developer to build dynamic Web pages combining user input with information from corporate data sources.

Server-side Java offers significant improvements in efficiency over the traditional Perl CGI, where a new process is started for each HTTP request, since the overhead of starting the process can dominate the execution time of the CGI program. With servlets and JSP applications, each request is handled by a lightweight Java thread, in a Java Virtual Machine that stays up all the time, rather than as a heavyweight operating system process.

Another major advantage of Java servlets and JSP pages is, of course, that they allow you to use a single development language across an entire application. You can write applications for an Apache server running on a Solaris platform but can do all your development and checkout under Linux or any other OS that supports Java.

This chapter provides a brief introduction to using servlets and JSP to create dynamic Web pages. These Web pages are driven by a membership database, accessed using the `DataSource` object.

Using JDBC DataSources

Database `Connections` obtained using the `DataSource` interface, introduced in the JDBC 2.0 Standard Extension API, offer the user considerably more capability than the basic `Connection` objects that the `DriverManager` provides; `DataSource` objects can support connection pooling and distributed transactions. These features make `DataSource` objects the preferred means of getting a `Connection` to any source of data. This source can be anything from a relational database to a spreadsheet or a file in tabular format.

There are three types of standard `DataSource` objects, each of which offers unique advantages:

- The basic `DataSource` that produces standard `Connection` objects just like those the `DriverManager` produces
- A `PooledDataSource` that supports connection pooling. Pooled connections are returned to a pool for reuse by another transaction.
- A `DistributedDataSource` that supports distributed transactions accessing two or more DBMS servers.

With connection pooling, connections can be used over and over again, avoiding the overhead of creating a new connection for every database access. Reusing connections in this way can improve performance dramatically, since the overhead involved in creating new connections is substantial.

Distributed transactions involve tables on more than one database server. When a `DataSource` is implemented to support distributed transactions, it is almost always implemented to produce connections that are pooled as well.

A `DataSource` object is normally registered with a JNDI naming service. JNDI naming services are analogous to a file directory that allows you to find and work with files by name. This means that an application can retrieve a `DataSource` object by name from the naming service in a manner independent of the system configuration.

Preparatory to discussing the use of JDBC `DataSource` objects in a Web application, the next section gives a brief introduction to Java servlets.

Using Servlets to Create Dynamic Web Pages

Servlets are Java classes that run in a servlet engine and receive and service client requests. Although they are not tied to a specific protocol, the most common use of servlets is to create dynamic Web pages. An online catalog is a classic example of a dynamic Web application. Requests from a client can be received by a servlet that gets the data from a database, formats it, and returns it to the client.

Note

The Tomcat servlet engine, from `http://jakarta.apache.org/tomcat/` is used in all the examples in this book. Tomcat was chosen because it is the servlet container used in the official reference implementation for the Java Servlet and Java Server Pages technologies. Commercial servlet containers such as JRun should work just as well.

Creating a Simple Servlet

Servlets are created by implementing `javax.servlet.Servlet.` All servlets implement this interface. Servlets are typically created by extending `javax.servlet.GenericServlet`, which implements the servlet interface, or by extending `javax.servlet.http.HttpServlet,` which is the base class for servlets that service HTTP requests.

The servlet interface defines so called life-cycle methods, which are called by the servlet engine to handle the major-life cycle tasks. These life-cycle tasks are initialization, client request service, destruction, and garbage collection.

Much of the work a servlet does is handled in the client request service methods. These are the two most important client request service methods of the `HttpServlet` class:

- `doGet`, which must be overridden to support HTTP GET requests
- `doPost`, which must be overridden to support HTTP POST requests

`GET` and `POST` are the CGI methods used to transfer request parameters to the server. The primary difference between the two is that the parameters in the GET request are appended to the host URL, whereas the parameters in the POST request are passed separately.

Another important reason for using the `POST` method is that it can transfer more data than the GET method . The maximum length of the parameters in a `GET` request is specified as 256 characters.

Typical uses for HTTP servlets include the following:

- Processing and/or storing data an HTML form submits
- Creating dynamic Web pages
- Managing state information for applications such as an online shopping cart

Servlets offer many advantages over traditional CGI scripts and are the backbone of today's application servers. In spite of their power, however, they are relatively easy to write and deploy, as the simple "Hello World" example of Listing 12-1 demonstrates.

Listing 12-1: A simple servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet{
  protected void doGet(HttpServletRequest req,HttpServletResponse resp)
  throws ServletException, IOException
  {
```

```
    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Hello Servlet</TITLE></HEAD>");
    out.println("<BODY><H1>Hello Servlet World</H1></BODY>");
    out.println("</HTML>");
    out.close();
  }
}
```

The next section expands on the basic example of Listing 12-1 to create a simple Login servlet.

Creating and Deploying a Login Servlet

As mentioned in Chapter 11, the examples in Part III of this book are geared toward building a simple membership Web site. The Web site in the examples is based on SQL Server, using the Opta2000 JDBC driver. The Opta2000 driver is an excellent example of an efficient, modern, pure Java driver. One reason for choosing the Opta2000 driver is to illustrate the use of a different driver, since most of the sample code in Part II uses the JDBC-ODBC bridge. A more practical consideration is that the JDBC-ODBC bridge is slow compared with Opta2000 and other commercial drivers. As I point out in Part II, and illustrate in Chapter 10, JDBC does such a great job of supporting different RDBMS systems and drivers that using a different driver or database involves only a couple of minor changes in the code.

The HTTP server used in the examples is Apache, currently the most widely used and one of the easiest to install. The servlet engine is Apache Tomcat; it has been chosen by Sun as the reference implementation, it works well, and both Apache and Tomcat are available as free downloads from the Apache Software Foundation at `http://www.apache.org/`. Like Tomcat, Apache can be installed on Linux, Windows, and most other major platforms.

Cross-Reference Installation and setup of Apache and Tomcat are covered in Appendix 2.

Implementing a Membership Web Site

The first step in implementing a membership Web site, obviously, is handling member logins. The Web-site design discussed in Chapter 11 calls for a dedicated table for user names and passwords. The design of this table is extremely simple, as shown in Table 12-1.

Table 12-1: Login Table Containing Usernames and Passwords

UserName	Password	MemberID
garfield	lasagna	1
snoopy	peanuts	2
batman	robin	3

When the table is created, the UserName column is defined as the primary key, because this column is used in a WHERE clause when a member logs in. The Password column is a simple VARCHAR field, used only for validation. The MemberID column is important because all the other tables containing member information use a MemberID column as their primary key to facilitate looking up member information in other tables. MemberID is defined with the IDENTITY constraint so that the DBMS automatically assigns a new, unique number to the field. The SQL `CREATE` statement used to create this table is shown below:

```
CREATE TABLE LOGIN(
 UserName VARCHAR(20) PRIMARY KEY,
```

```
Password VARCHAR(20) NOT NULL,
MemberID int IDENTITY);
```

Notice that the UserName column has been defined as the primary key but not as a clustered key, because the physical layout of a SQL Server database is ordered on the clustered key, if assigned. This means that if a new member is added with a clustered key value within the current range of clustered key values, as might easily be the case, the entire table will be reshuffled to reflect the change. This clearly has an adverse impact on performance if you sign up a lot of new members.

The importance of the MemberID column is easiest to understand when you consider a situation where you have to access member information from another table. When the user logs in, the first thing you do is look up his UserName and password in the Login Table. This lookup also returns the MemberID, which can be used to look up any other data you may need. Table 12-2 illustrates a member name and address table that can be indexed by MemberID for rapid access.

Table 12-2: Member Name and Address Table

MemberID	FNAME	LNAME	STREET	CITY	ST	ZIP	EMAIL
1	Giorgio	Corleone	123 Main St	NY	NY	10002	gcorleone@hotmail.com

Creating the Login Page

The user interface between the member and the database is based on the use of HTML forms. Members log in to the Web site by using a simple HTML form. The screen shot of Figure 12-1 shows the HTML form in an Opera browser window. The HTML for the login form is shown in Listing 12-2.

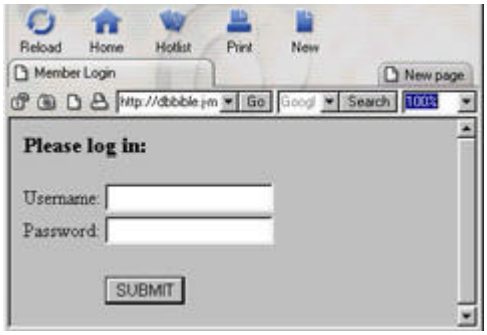


Figure 12-1: HTML login form displayed in the Opera browser

Listing 12-2: Using HTML to create a basic login form

```
<html>
<head>
<title>Member Login</title>
</head>
<body bgcolor="#c0c0c0">
<form method="POST" action="servlet/LoginServlet">
<table>
    <tr>
        <td colspan=2>
        <h3>Please log in:</h3>
        </td>
```

```
    </tr>
    <tr>
        <td>
        Username:
        </td>
        <td>
        <input type="text" name="username"><br>
        </td>
    </tr>
    <tr>
        <td>
        Password:
        </td>
        <td>
        <input type="password" name="password"><br>
        </td>
    </tr>
    <tr>
        <td colspan=2> </td>
    </tr>
    <tr>
        <td>
        <input type="submit" value="SUBMIT" name="submitButton">
        </td>
    </tr>
</table>
</form>
</body>
</html>
```

As you can see from Listing 12-2, the `action` method uses the `POST` method to call the LoginServlet. The two input fields UserName and Password are passed as parameters to the servlet. The `GET` method can work just as well and is, in fact, implemented in the servlet code. `POST` is normally preferred because it offers more flexibility. The code for the servlet itself is shown in the next section.

Creating the Servlet

The login servlet is not much more complex than the simple "Hello World" example of Listing 12-1. In this more practical example, the base class methods that need to be overridden are as follows:

`init()`

`doPost()`

`doGet()`

The code of Listing 12-3 shows the use of the `init()` method to load the JDBC driver. The `doGet()`and `doPost()` methods are overridden to handle the user request. The `writePage()` method simply exists to separate HTML output from JDBC code.

Listing 12-3: Login servlet

```
import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DataSourceLoginServlet extends HttpServlet{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  private Connection con = null;
  private DataSource ds = null;

  public void init(ServletConfig config) throws ServletException{
    super.init(config);
    try{
      Class.forName("com.inet.pool.PoolDriver").newInstance();
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );
      ds = tds;
    }
    catch(Exception e){
      System.err.println(e.getMessage());
    }
  }
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
  {
    doPost(request,response);
  }
  public void doPost(HttpServletRequest request, HttpServletResponse
response)
  throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter(response.getWriter());

    int id = -1;
```

```
    String memberPwd = null;
    String userName  = request.getParameter("username");
    String password  = request.getParameter("password");

    try {
    Connection con = ds.getConnection(dbUserName,dbPassword);

      Statement stmt;
      ResultSet rs = null;

      String SQLQuery = "SELECT * FROM LOGIN WHERE UserName =
'"+userName+"';";
      stmt = con.createStatement();
      rs   = stmt.executeQuery(SQLQuery);
      while(rs.next()){
        id = rs.getInt("MemberID");
        memberPwd = rs.getString("Password");
      }
      con.close();
    }catch(SQLException e){
      System.err.println(e.getMessage());
    }
    RequestDispatcher dispatcher = null;
    if(id==-1){
      dispatcher =
        getServletContext().getRequestDispatcher("/jdbc/NewMember.html");
    }else if(!memberPwd.equals(password)){
      dispatcher =
        getServletContext().getRequestDispatcher("/jdbc/BadPassword.html");
    }else{
      dispatcher =
        getServletContext().getRequestDispatcher("/jdbc/WelcomeBack.html");
    }
    dispatcher.forward(request, response);
  }
}
```

The login servlet works by getting the UserName and Password inputs from the `HttpServletRequest` object and plugging the UserName into a simple SQL query. The query returns matching rows from the LOGIN table. A series of simple checks on the returned values is used to create the appropriate response to the user.

JDBC initialization is performed in the `init()` method. In this example, the Opta2000 driver is being used, but you can substitute whichever driver you prefer by simply substituting the appropriate driver name and URL into the appropriate String variables.

Unlike the examples in Part II, which use the "classic" JDBC Core API approach of getting a `Connection` from the `DriverManager`, this servlet illustrates the preferred way of getting a connection, which is to use a `javax.sql.DataSource`. You can, of course, use the `DriverManager` instead.

Cross-Reference The `javax.sql.DataSource` is discussed in Chapter 4 and is used throughout the examples in Part III of this book.

Notice that, in addition to the `doPost()` method, the `doGet()` method has been minimally implemented with a simple call to `doPost()`. The main reason for doing this is to make it easier to check out the servlet from a browser by simply typing the entire `GET` string into the browser's address window. Here's an example:

```
http://jdbc.j-
machines.com/servlet/LoginServlet?UserName=OneFish&Password=TwoFish
```

The `ResultSet` the SQL query returns is used to determine whether a MemberID has been assigned to this UserName. If not, the servlet forwards the user to the new member sign up page. Similarly, if the UserName is recognized but the password is bad, the user is forwarded to a page that lets the user retry or request that the password be e-mailed. If the UserName and password are found in the database, the user is forwarded to the welcome page.

Forwarding is handled using a `javax.servlet.RequestDispatcher` object. To get a `RequestDispatcher`, use the `ServletContext` object's `getRequestDispatcher()` method, passing it the desired URL as an argument. Notice the format of this argument: a slash ("/") followed by the relative path, ending with the name of the resource. This is very important to remember, since, as soon as you call the servlet, you move from the HTTP server's environment, which may be a virtual host's root directory under Apache, to the servlet environment, which will probably be somewhere under Tomcat's root directory.

You should use forwarding when the servlet's job is done and the next page is logically decoupled from the servlet's functions to such an extent that another resource can handle it. Remember, however, that if you have already written any output from the servlet, using a `ServletOutputStream` or a `PrintWriter`, you can't use the `RequestDispatcher.forward` method; it will throw an `IllegalStateException`. In this case, you can use the `RequestDispatcher.include()` method instead.

Deployment

To deploy the login servlet, you have to put the class file into the appropriate directory. This is the usual path for a simple Tomcat installation:

```
TOMCAT/WEBAPPS/ROOT/WEB-INF/CLASSES
```

Tomcat maintains a configuration file that defines URL mappings, so that the `/servlet/` path is mapped to this directory. You can set up your own mappings as needed by editing this file.

To use the Opta2000 driver, you need to put the `Opta2000.jar` into a suitable directory and modify Tomcat's class path in the `tomcat.properties` file in the `Tomcat/conf` directory. In this example, the `.jar` file is saved in the `/lib` directory, and Tomcat's class path is modified by adding this line in the `tomcat.properties` file:

```
wrapper.classpath=lib/Opta2000.jar
```

To make the use of servlets even easier, Sun came up with the idea of Java Server Pages, or JSPs. A brief introduction to JSPs is given in the next section.

Using Java Server Pages

The login servlet example in Listing 12-3 shows how easy it is to write and deploy Java server-side applications using JDBC. There is also an even easier way — you can use Java Server Pages.

Java Server Pages provide a means of using Java code within an HTML page. Simply write the static HTML parts of the page in the normal way, and embed the Java code inside special tags. The Java is executed in the JSP engine, and the result is sent to the client as HTML.

Note

Java Server Pages are not limited to generating HTML. JSP technology is also a great way of generating XML, as you will see in Part IV.

You can use these four main types of elements in constructing a Java Server Page:

- Markup language elements, which, in the case of a Web page, are HTML
- Scripting elements, which let you specify a block of Java code
- JSP directives, which control the JSP structure and environment
- Actions, which let you specify execute commands such as loading a parameter

A special type of tag identifies the JSP-specific elements so that they are not confused with markup language tags. These tags take one of the two following forms:

- `<% %>`
- `<jsp: />`

Although Java Server Pages offer a lot of advantages over basic servlets, they actually build on servlet technology. The JSP engine compiles the JSP to a servlet the first time the page is requested, though there are various ways of forcing the compile to ensure that the first real user doesn't see any delay due to the translation. The simplest way to do this, of course, is to call the JSP page yourself to force a compile.

The easiest way to demonstrate the advantages of Java Server Pages is to rework the login example to use JSP. Since the JSP engine can serve static HTML as easily as dynamic HTML, even the login form can be turned into a JSP page. The login form shown in Listing 12-4 is basically the same form shown in Listing 12-1. All that is required to make it a JSP page is to save it where the JSP engine can find it and name it `LoginForm.jsp`.

Listing 12-4: A login form using JSP

```
<html>
<head>
<title>Member Login</title></head>
<body bgcolor="#c0c0c0">
<form method="POST" action="ProcessLogin.jsp">
<table>
    <tr>
        <td colspan=2><h3>Please log in:</h3></td>
    </tr>
    <tr>
        <td>Username: </td>
        <td><input type="text" name="username"></td>
    </tr>
    <tr>
        <td>Password:</td>
        <td><input type="password" name="password"></td>
    </tr>
    <tr><td colspan=2></td></tr>
```

```
    <tr>
        <td></td>
        <td><input type="submit" value="SUBMIT" name="submitButton"></td>
    </tr>
</form>
</body>
</html>
```

The only significant difference between `login.html` and `LoginForm.jsp` is that the action parameter has been changed to point to `ProcessLogin.jsp`. The UserName and Password parameters are passed to the new action method just as they are to LoginServlet.

A simple JSP page that picks up the HTTP request parameters and echoes them to the client is shown in Listing 12-5. The example illustrates how you can combine HTML and embedded Java in a single JSP page using a number of JSP-specific tags:

- `<% %>` delimiters for in-line Java scriptlets
- `<%=expression %>` output the evaluated value of the expression
- `<%@ page %>` directive defining page properties

Listing 12-5: Using a JSP page to display CGI parameters

```
<HTML>
<HEAD>
<TITLE>
Display Login Parameters
</TITLE>
</HEAD>
<BODY>
<%@ page language="java"%>
<%
    String userName=request.getParameter("username");
    String password=request.getParameter("password");
%>
User Name = <%=userName%><p/>
Password  = <%=password%><p/>
</BODY>
</HTML>
```


This example can be extended by including the SQL query code used in the JDBC servlet example of Listing 12-3. This approach combines both the JDBC code and the HTML generation, emulating the behavior of the original servlet example. However, writing the JSP page this way is messy, since it combines Java and HTML in a single page.

A much better way to structure the JSP page is to encapsulate the JDBC logic in a JavaBean, which acts as the model in a model-view-controller (MVC) structure. The view is provided by a JSP page, using the `<jsp:forward />` directive. The action method of the `LoginForm.jsp` calls a controller JSP, which loads the bean, passes it the request parameters, and forwards the user to the appropriate view JSP, depending on the results of the SQL the JavaBean executes.

Before implementing the MVC approach to handling the login form, it is worth reviewing the use of JavaBeans with JSP pages. The next few paragraphs give a brief overview.

Using JavaBeans with Java Server Pages

One of the most useful features of Java Server Pages is that JSP directly supports the use of JavaBeans through the `<jsp:useBean>` tag. Java Beans, as you probably know, are Java classes that can be loaded by name and otherwise conform to a specific set of rules. These rules include the following:

- Being a public class: public class `JavaBean`
- Having a public, no argument constructor: public `JavaBean ()`
- Using private data fields only: private `String` message
- Providing public, no-argument access methods for its private data fields:
  - public `getMessage()`
  - public `setMessage(String message)`
- Supporting *introspection* — the ability of an external class to query a bean for its behavior.

A big advantage of using JavaBeans in JSP applications is that they allow you to implement the logic of the JSP page as a separate Java class and "plug it in." This approach offers these significant advantages:

- Separation of content from logic
- Reusable plug-in components for common tasks

When they are used with Java Server Pages, JavaBeans have two primary functions. They can be used as logic blocks, where their primary purpose is to separate logic from display, and as storage classes, where their primary purpose is data storage.

Loading by Name: the <jsp:useBean> tag

The ability to load and execute a JavaBean by name is the real key to using JavaBeans as pluggable components. Since the JavaBean is linked into the JSP at runtime rather than at compile time, the JSP can be edited and updated separately from the business logic in the JavaBean.

To call a Java Bean from a JSP, simply write this:

```
<jsp:useBean id="TestBean" class="java_database_bible.TestBean"/>
```

The `<jsp:useBean>` element has a number of attributes. Of these, the most commonly used are the following:

- `id="beanInstanceName". The id` attribute assigns a local name to the JavaBean for references within the JSP page. The `id` is case sensitive and must be used consistently throughout the scope of the bean.
- `scope="page | request | session | application".` The `scope` attribute defines the scope in which the bean exists and the in which variable named in `id` is available. The default value is page.
- `class="package.class".` The `class` attribute defines the JavaBean class to load if the JavaBean with the specified `id` has not already been loaded in the defined scope. The new bean is instantiated using the new keyword and the class constructor.

Note

The `<jsp:useBean>` tag first looks for the bean instance with the specified name and instantiates a new one only if it cannot find the bean instance within the specified scope. If the bean has already been created by another `<jsp:useBean>` element, the value of `id` must match the value of id used in the original `<jsp:useBean>` element.

Scope

Once a JavaBean has been loaded, it can be accessed from various parts of your application, depending on its *scope*. In other words, the scope of a JavaBean defines that part of your application that can access the bean. The default value is `page scope`. The meanings of the different scopes are as follows:

- **page scope** — The bean is accessible within the JSP page with the <jsp:useBean> element or any of the page's static include files until the page sends a response to the client or forwards a request to another file.
- **request scope** — The bean is accessible from any JSP page processing the same request until a JSP page sends a response to the client or forwards the request to another file.
- **session scope** — The bean is accessible from any JSP page in the same session as the JSP page that creates the bean. The bean exists across the entire session, and any page that participates in the session can use it. The page in which you create the bean must have a <%@ page %> directive with session=true.
- **application** — The bean is accessible from any JSP page in the same application as the JSP page that creates the bean. The bean exists across an entire JSP application, and any page in the application can use the bean.

Caution

When using the <jsp:useBean> tag, the closing "/" at the end of the tag is very important. If you forget the "/", the tag will work, but it will work unpredictably. For example, I have spent hours trying to find what I thought was a scope problem before noticing that I had lost the closing slash in one of a number of JSP pages using the bean.

Properties: the <jsp:getProperty> and <jsp:setProperty> tags

JavaBean properties are private data fields that can be accessed through predefined "accessor" methods or *getter* and *setter* methods. Property getter and setter method names follow specific rules called *design patterns*. By using these design pattern-based method names, JSP pages can access a JavaBean's properties through these directives:

```
<jsp:setProperty name="TestBean" property="service" value="login"/>
<jsp:setProperty name="TestBean" property="username"

value='<%=request.getParameter("username")%>'/>
<jsp:setProperty name="TestBean" property="password"/>
<jsp:getProperty name="TestBean" property="message"/>
```

Notice the different ways in which you can set the value of a JavaBean parameter. The first example shows how you can set the parameter using a static value. The second shows the use of an evaluation expression to set the value. In the third case, the value is set implicitly to the value of the same name in the CGI query parameters. A fourth variant is discussed later in this chapter, under the heading "Introspection."

Using JavaBeans, you can get away from using in-line Java code completely. Simply code the logic as a JavaBean, and load the JavaBean by name, using the `jsp:getProperty` and `jsp:setProperty` tags to access its properties.

Using <jsp:setProperty> for initialization

Recall that the JSP engine only loads a new instance of the bean if it can't find an existing instance in scope. This means you can use a bean as a storage container that keeps track of the application's data anywhere within its scope. This raises the question of how the bean is initialized. The answer is to nest the initialization inside the <jsp:useBean> element itself, as shown here:

```
<jsp:useBean id="TestBean" class="JavaDatabaseBible.TestBean"/>
 <jsp:setProperty name="TestBean" property="message" value="goodbye"/>
</jsp:useBean>
```

Any <jsp:setProperty> elements nested within the <jsp:useBean> element are executed only when the bean is first loaded and run. These nested elements are not executed if the bean is found within the current scope. The idea here is that the first time the bean is used, it is initialized; in subsequent references; however, it is assumed that you actually want the data stored in the bean by earlier references.

Introspection

Introspection refers to the ability to look inside a JavaBean and identify the methods available to the user. For example, introspection allows the JSP engine to look at the JavaBean properties that can be set from the JSP page. This means that the JSP engine can handle most of the details of setting properties automatically, as shown here:

```
<jsp:setProperty name=" TestBean" property="*"/>
```

Using the "*" wildcard as the value of the property attribute in a `<jsp:setProperty>` tag tells the JSP engine to use introspection to identify the JavaBean's properties and match them by name with the parameters the form passes. This approach is illustrated in Listing 12-6.

Listing 12-6: Using a JSP with the <jsp:useBean/> tag

```
<html>
<head>
<title>ParameterTestBean</title>
</head>
<body>
<%@ page language="java"%>
<jsp:useBean id="ParameterTestBean"
class="JavaDatabaseBible.ch12.ParameterTestBean"/>
<jsp:setProperty name="ParameterTestBean" property="*"/>
User Name:
<jsp:getProperty name="ParameterTestBean" property="username"/><p/>
Password:
<jsp:getProperty name="ParameterTestBean" property="password"/>.
</body>
</html>
```

Much nicer, isn't it? Of course, if you don't have a one-to-one match between the parameter names and the bean properties, you can always resort to mapping them by hand. Incidentally, this technique also works for saving values from radio buttons, where the JavaBean property is set to the selected radio-button value.

You can test the use of JSP and JavaBeans using the simple JavaBean of Listing 12-7. This is the JavaBean originally used with the JSP code of Listing 12-6.

Listing 12-7: Simple JavaBean illustrating getter and setter methods

```
package JavaDatabaseBible.ch12;

public class ParameterTestBean extends java.lang.Object{
  protected String username;
  protected String password;

 public ParameterTestBean(){
 }
 public void setUsername(String username){
```

```
  this.username = username;
 }
 public void setPassword(String password){
  this.password = password;
 }
 public String getUsername(){
  return username;
 }
 public String getPassword(){
  return password;
 }
}
```

If you are working with checkboxes, where you specify the same name, but different values for each checkbox, the only difference is that you must specify the JavaBean property as a String array. Unfortunately, you can't get String-array values as easily. The solution in this case is to use a scriptlet, as shown here:

```
<%
String[] checkBoxes = formHandler.getCheckBoxes();
for(int i=0;i< checkBoxes.length;i++){
  if(i>0)out.print(",");
  out.print(" "+ checkBoxes [i]);
}
%>
```

Using built-in JSP objects in a JavaBean

The JSP API provides access to a range of useful information about the client and about the JSP's context through a set of built-in, implicit objects. The javax.servlet.jsp.PageContext object is the general point of access for most of the built-in JSP objects. These objects are as follows:

- request
- response
- out
- session
- application
- pageContext
- page
- exception

request

The request object encapsulates the current request from the browser. The servlet container creates a `ServletRequest` object and passes it to the servlet's service method. A `ServletRequest` object provides such data as parameter name and values, attributes, and an input stream. Useful request object methods include the following:

- `getQueryString()`
- `getHeader(String headerName)`
- `getCookies()`

response

The `ServletResponse` object is designed to assist a servlet in sending a response to the client. The servlet container creates a `ServletResponse` object and passes it to the servlet's service method.

The `ServletResponse` object allows you to set response parameters such as content type. You can also get an OutputStream from the response object for binary writes.

out

The out object is the instance of JspWriter used to write output to the client. The ability to access the JspWriter directly allows you to send output directly from a scriptlet.

session

The session object is an instance of HttpSession. It encapsulates session information in the form of objects that can be written and read by beans or JSP pages within the session scope.

application

The application object is an instance of the ServletContext object.

pageContext

The pageContext object is the general point of access for most of the built-in objects; for example, to get the session object, you can call:

```
HttpSession session = pageContext.getSession();
```

page

The page object is a reference to the current page.

exception

The exception object is used by an error page to access the exception that causes the error page to be displayed.

Automatic Type Conversion

Clearly, when you create a JavaBean, you may want to use property variables of various types, such as `integers` and `doubles`. The values of the request parameters sent from the client to the server are always of type `String`. These values are converted to other data types automatically, using the appropriate `valueOf(String)` expression.

```
valueOf(String)
```

JSP's automatic type conversion uses the methods listed in Table 12-2 to perform conversions.

Table 12-2: Automatic Type Conversions Supported by JSP

Data Type	Conversion Method
boolean or Boolean	java.lang.Boolean.valueOf(String)
byte or Byte	java.lang.Byte.valueOf(String)
char or Character,	java.lang.Character.valueOf(String)
int or Integer	java.lang.Integer.valueOf(String)
double or Double	java.lang.Double.valueOf(String)
integer or Integer	java.lang.Integer.valueOf(String)
float or Float	java.lang.Float.valueOf(String)
long or Long	java.lang.Long.valueOf(String)

Creating and Deploying a JDBC LoginBean

The simple example of Listings 12-6 and 12-7 provide the basics of a JSP-based and JavaBean-based login form handler. Using the `LoginForm.jsp` of Listing 12-4, the user enters his or her name and password and clicks the Submit button to call `ProcessLogin.jsp`.

`ProcessLogin.jsp` is an extended version of the JSP page illustrated in Listing 12-6. These are the main differences:

- `ProcessLogin.jsp` has no HTML content. It acts as a pure controller.
- A `<jsp:forward />` tag is used to display the view portion of the MVC structure.

Like the JSP page illustrated in Listing 12-6, the `ProcessLogin.jsp` relies on a JavaBean to handle the business logic. In this instance, the bean incorporates the JDBC code illustrated in the servlet example of Listing 12-3.

Since the functionality of the `ProcessLogin.jsp` page is reduced to launching the JavaBean and interpreting the response, the resulting MVC controller is simple and easily understood. All it does is accept the form inputs, pass them to the LoginBean, and select one of three pages to forward the user to, depending on his or her login status. The resulting JSP code is shown in Listing 12-8.

Listing 12-8: ProcessLogin.jsp

```
<%@ page language="java"%>
<jsp:useBean id="LoginBean" class="JavaDatabaseBible.ch12.LoginBean"/>
<jsp:setProperty name="LoginBean" property="*"/>
<%
String status = LoginBean.validate();
String nextPage = "MemberWelcome.jsp";
if(status.equals("New Member")) nextPage = "NewMemberForm.jsp";
if(status.equals("Bad Password")) nextPage = "BadPasswordForm.jsp";
%>
<jsp:forward page="<%=nextPage%>"/>
```

Like the servlet example of Listing 12-3, the JSP version uses page forwarding. In Java Server Pages, this function is implemented by the `<jsp:forward/>` tag.

The LoginBean is also relatively simple. Unlike the servlet, which incorporates a certain amount of HTML generation and other overhead, the LoginBean is simply a logic block. Setup is handled when the bean is instantiated, and the JSP page sets username and password. Listing 12-9 shows the simplicity of the LoginBean.

Listing 12-9: LoginBean

```
package JavaDatabaseBible.ch12;

import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginBean extends java.lang.Object{
```

```
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  private Connection con = null;
  protected String username;
  protected String password;

  public LoginBean(){
  }
  public void setUsername(String username){
    this.username = username;
  }
  public void setPassword(String password){
    this.password = password;
  }
  public String getUsername(){
    return username;
  }
  public String getPassword(){
    return password;
  }
  public String validate(){
    int id = -1;
    String memberPwd = null;

    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);
      Statement stmt;
      ResultSet rs = null;

      String SQLQuery = "SELECT * FROM LOGIN WHERE UserName = '" +
username + "';"; // for clarity
      stmt = con.createStatement();
      rs   = stmt.executeQuery(SQLQuery);
      while(rs.next()){
```

```
        id = rs.getInt("MemberID");
        memberPwd = rs.getString("Password");
      }
      con.close();
stmt.close();

    }catch(ClassNotFoundException e1){
      System.err.println(e1.getMessage());
    }catch(SQLException e2){
      System.err.println(e2.getMessage());
    }
    if(id==-1){
      return "New Member";
    }else if(!memberPwd.equals(password)){
      return "Bad Password";
    }else{
      return "Member#"+id;
    }
  }
}
```

Note

In order to use the "*" wildcard to set JavaBean properties from a JSP page, the property names must match the variable names used in the HTML form. Property names are case sensitive.

The LoginBean of Listing 12-9 returns a String indicating the user's login status. The three possible return values are as follows:

- "New Member"
- "Bad Password"
- "Member#nnn"

The JSP page deals with each of these possible return values by forwarding the user to the appropriate JSP page. In the case of a registered member, the user is simply forwarded to the welcome page of the main site. Someone logging on with an unknown user name is considered a new member and is forwarded to the Member Registration trail, discussed in the next chapter. If the user name is recognized, but the password is not, the user is offered the option of retrying, signing on as a new member, or having the correct password sent to his or her e-mail address.

Summary

This chapter gives you a look at the use of the JDBC `DataSource` object as a means of obtaining database connections. Other topics covered include the following:

- Creating Dynamic Web Pages
- Using the `HttpServlet` object
- Implementing a Membership Web Site
- Using Java Server Pages
- Using JavaBeans with Java Server Pages

Chapter 13 covers creating and populating the basic membership database tables. The examples are based on the use of `PreparedStatements` and `CallableStatements`.

Chapter 13: Using PreparedStatements and CallableStatements

In This Chapter

All of the discussions and examples up to this point have been about how to use the JDBC API to execute SQL statements. The subject of what actually happens when the SQL statement is passed to the DBMS has not been considered. The purpose of this chapter is to address two significant ways in which you can improve the performance of a Java database application by improving the execution performance of your SQL statements.

One of the main drawbacks of using the basic `java.sql.Statement` is that every time the basic `Statement` object is executed, the SQL command is passed to the RDBMS, where it has to be parsed and compiled before it can be executed. Most versions of SQL allow the user to define stored procedures, which are, in effect, precompiled SQL statements or groups of statements. Stored procedures, being precompiled, execute faster and more efficiently than statements that have to be parsed and compiled each time they are used.

To eliminate the overhead of repeated parsing and compilation of the SQL command, JDBC provides the user with two ways of using precompiled SQL statements: the `PreparedStatement` object and the `CallableStatement` object. Using `PreparedStatements` and `CallableStatements` greatly increases the efficiency of an application when a specific SQL command is executed frequently or repeatedly, as is the case when handling forms for a Web site.

The three different flavors of the `Statement` object are intended to be used in very different situations. The first situation arises when you want to execute a statement just once. This is the ideal place to use a basic java.sql.Statement. If you want to execute a SQL command repeatedly in a loop, and then discard it, the best approach is to use a `PreparedStatement`, which is parsed, compiled and cached temporarily by the RDBMS. Finally, if you have a statement or group of statements you want to execute frequently, the `CallableStatement` is ideal, since it is compiled and stored permanently in the RDBMS to be called by name when needed.

Creating and Using a PreparedStatement

The main difference between a basic `Statement` object and a `PreparedStatement` object is that when the `PreparedStatement` is used, the SQL command is sent to the DBMS when the `PreparedStatement` is created, so that it can be precompiled and saved in a cache. This means that when the `PreparedStatement` is executed, the database management system can run the `PreparedStatement`'s SQL statement without having to compile it first.

Using `PreparedStatements` improves efficiency; when you execute the `PreparedStatement`, it is once again parsed, but no recompile occurs. Instead, the precompiled statement is found in the cache and is reused. For an application that requires the repeated execution of a SQL command in a loop, the use of `PreparedStatements` can greatly improve the performance of the database.

`PreparedStatement` objects can be used for SQL statements with no parameters or for SQL statements that take parameters. `PreparedStatements` can contain placeholders for variables known as IN parameters, which are set using setter methods. The JDBC `PreparedStatement` provides `setter` methods for all SQL data types.

Creating a PreparedStatement Object

`PreparedStatements`, like `Statements`, are created using a `Connection`. For example, you can easily replace the `Statement` object in the LoginBean developed in Listing 12-9 with a `PreparedStatement`, as shown in Listing 13-1.

Listing 13-1: Using a PreparedStatement

```
Class.forName("com.inet.pool.PoolDriver");
com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
tds.setServerName( "JUPITER" );
tds.setDatabaseName( "MEMBERS" );
tds.setUser( dbUserName );
tds.setPassword( dbPassword );

DataSource ds = tds;
Connection con = ds.getConnection(dbUserName,dbPassword);

String SQLQuery = "SELECT * FROM LOGIN WHERE UserName = ?;";
PreparedStatement pstmt = con.prepareStatement(SQLQuery);
pstmt.setString(1, username);

ResultSet rs = pstmt.executeQuery();
while(rs.next()){
  id = rs.getInt("MemberID");
  memberPwd = rs.getString("Password");
}
con.close();
```

The main difference between the `PreparedStatement` used in this example and the `Statement` object used in Chapter 12 lies in the form of the SQL command. In this example, a "?" is used as a place holder for the variable UserName, which is set using the `pstmt.setString()` method. There are corresponding setter methods in the `PreparedStatement` for all SQL data types.

You need to supply values to be used in place of all placeholders before you can execute a `PreparedStatement`. Once a `PreparedStatement` parameter has been set to a given value, it retains that value until it is reset to another value or until the method `clearParameters` is called.

Using PreparedStatement in a Loop

The real efficiency gain in using `PreparedStatement` objects occurs when you use them repeatedly (for example, when you need to execute a SQL command in a loop). If you need to use the same SQL command frequently from different instances of the Java class, a better alternative is the use of a `CallableStatement`.

An example of using a `PreparedStatement` in a loop is shown in Listing 13-2. A simple `for` loop sets the parameters of the `PreparedStatement` from the Orders array. The data is then inserted into the Ordered_Items Table, which is similar to the table of the same name used in the examples of Part II.

Listing 13-2: Using a PreparedStatement in a loop

```
package JavaDatabaseBible.ch13;

import java.sql.*;
import javax.sql.*;
```

```
public class PStatement {
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  public static void main(String args[]){
    int[][] Orders = {{1001,327,2},
                      {1001,412,1},
                      {1001,906,5},
                      {1002,111,7},
                      {1002,112,19}};

    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      String SQLCmd =
      "INSERT INTO ORDERED_ITEMS (ORDER_NUMBER,ITEM_NUMBER,QTY)
VALUES(?,?,?)";
      PreparedStatement pstmt = con.prepareStatement(SQLCmd);

      for(int i=0;i<5;i++){
        pstmt.setInt(1, Orders[i][0]);
        pstmt.setInt(2, Orders[i][1]);
        pstmt.setInt(3, Orders[i][2]);
        pstmt.executeUpdate();
      }
      con.close();

    }catch(ClassNotFoundException e1){
      System.err.println(e1.getMessage());
    }catch(SQLException e2){
      System.err.println(e2.getMessage());
    }
 }
}
```

The appearance of the Ordered_Items Table after executing the loop is shown in Table 13-1. Note that the ID column uses the auto increment data type, so it is not specifically updated by the Java example.

Table 13-1: Ordered_Items Table

ID	Order_Number	Item_Number	Qty
1	1001	327	2
2	1001	412	1
3	1001	906	5
4	1002	111	7
5	1002	112	19

Values Returned by PreparedStatements

The kinds of values a `PreparedStatement` can return are exactly the same as for a basic `Statement`. Although the `executeQuery()` method used in Listing 13-1 returns a `ResultSet` object containing the results of the query, the return value for the `executeUpdate()` method in Listing 13-2 is an `int` that indicates how many rows are updated in the table. For example, you can wrap the `pstmt.executeUpdate()of` Listing 13-2 in the following `System.out.println()` method call:

```
System.out.println(pstmt.executeUpdate());
```

This results in a series of ones being printed to the console. When the `executeUpdate()`method is used to execute a DDL statement, such as `CREATE TABLE`, it returns a zero.

This section illustrated the most efficient way to execute a SQL command repeatedly in a loop when you have no expectation of using the command again later. If you have a statement or group of statements you want to execute frequently, the `CallableStatement` is a better solution, as illustrated in the next section.

Creating and Using a CallableStatement

The `CallableStatement` object allows you to call a database stored procedure from a Java application. The `CallableStatement` object is similar to the `PreparedStatement`, which it extends; but whereas a `PreparedStatement` actually contains the SQL command, a `CallableStatement` object contains a call to a procedure stored in the database; it does not contain the stored procedure itself. For an application such as a Web site, with a large number of users executing the same SQL statements repeatedly, the use of `CallableStatement` can greatly improve the performance of the database.

Since `CallableStatement` extends `PreparedStatement` , a `CallableStatement` object can take input parameters like a `PreparedStatement` object can. A `CallableStatement` can also take output parameters or parameters that are for both input and output.

The input parameters are defined in the `SQL CREATE PROCEDURE` statement, using syntax of this form:

```
@param_name type [(size)]
```

The @ symbol preceding the parameter name identifies the name as a parameter to the SQL engine. The type and size fields correspond to the normal SQL data type fields used in creating a table.

Creating a Stored Procedure

Reverting to the membership Web site example, the first step a user takes is to log in. A basic login form is developed in Chapter 12. In the event that the username and password is not recognized, the user is given the opportunity to register as a new member.

One of the first tables to be updated for a new member is Contact_Info. This occurs by having the new member complete a basic new-member registration form. After completing the member registration form, applicants for membership are forwarded to a series of additional forms. These are used to complete the member's entries in the Contact_Info and Member_Profile Tables.

The Contact_Info Table contains such data as the member's real name and address and his or her e-mail address. The layout of the Contact_Info Table is shown in Table 13-2.

Table 13-2: Contact_Info Table

ID	FNAME	MI	LNAME	STREET	CITY	ST	ZIP	EMAIL
1	Giorgio	A	Corleone	123 Main St	New York	NY	10002	gac@cn.com

Since updating the Contact_Info Table is a process that is repeated frequently, it is a good choice for implementation using a stored procedure. The `CallableStatement` object will be used to execute the stored procedure. Listing 13-3 illustrates the creation of a simple stored procedure to populate this table.

Listing 13-3: Creating a stored procedure

```
package JavaDatabaseBible.ch13;

import java.sql.*;
import javax.sql.*;

public class CreateCallableStmt{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  public static void main(String args[]){
    String createProc = "CREATE PROCEDURE INSERT_CONTACT_INFO "+
                        "@ID INT, @FName VARCHAR(20), @MI CHAR(1), "+
                        "@LName VARCHAR(30),@Street VARCHAR(50), "+
                        "@City VARCHAR(30), @ST CHAR(2), "+
                        "@ZIP VARCHAR(10), @Phone VARCHAR(20), "+
                        "@Email VARCHAR(50) "+
                        "AS INSERT INTO CONTACT_INFO "+
                        "(ID, FName, MI, LName, Street, City, ST, ZIP, "+
                        "Phone, Email) "+
                        "VALUES "+
                        "(@ID, @FName, @MI, @LName, @Street, @City, "+
                        " @ST, @ZIP, @Phone, @Email);";

    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
```

```
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      Statement stmt = con.createStatement();
      stmt.executeUpdate(createProc);
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
  }
}
```

Calling a Stored Procedure

Stored procedures are called using a simple escape syntax defined by enclosing the call in curly braces. When the driver encounters the curly braces, it will translate the command they enclose into the native SQL used by the database to call the stored procedure. This syntax has two forms: one that has a result parameter and one that does not. Here's an example:

```
{?= call <procedure-name>[<arg1>,<arg2>, ...]}
{call <procedure-name>[<arg1>,<arg2>, ...]}
```

Question marks (?) serve as placeholders for parameters defined in the stored procedure using the `@Name` convention as shown in the example. `IN` parameter values are set using the set methods inherited from `PreparedStatement`.

When calling a stored procedure, a `CallableStatement` object is created using the `Connection` method `prepareCall()`. The argument of the `prepareCall()` method is the escape String, with question marks as place holders for each of the input parameters. Values for each of these parameters are assigned using the setter methods of the `CallableStatement` object; then the `CallableStatement` is executed. The following code fragment shows how the stored procedure created in Listing 13-3 can be called:

```
String[] newMember   = {"Fred","A","Tagliatelle","123 Ziti",
                        "Penne","PA","12345","123-456-7890","fat@pasta.com"};
CallableStatement cs =
             con.prepareCall("{call
INSERT_CONTACT_INFO(?,?,?,?,?,?,?,?,?,?)}");

cs.setInt(1,2);
for(int i=0;i<newMember.length;i++){
  cs.setString(i+2,newMember[i]);
}
System.out.println(cs.executeUpdate()+" row updated");
```

Calling stored procedures that return ResultSets is just as easy. For example, a simple stored procedure, GET_LOGIN_FOR_USER, which gets the login data for a given UserName, can be defined as follows:

```
CREATE PROCEDURE GET_LOGIN_FOR_USER @USERNAME VARCHAR(20)
 AS SELECT *
   FROM LOGIN
   WHERE USERNAME = @USERNAME;
```

The example in Listing 13-4 shows how to call the stored procedure GET_LOGIN_FOR_USER.

Listing 13-4: Calling a stored procedure that returns a ResultSet

```
package JavaDatabaseBible.ch13;

import java.sql.*;
import javax.sql.*;

public class CallableGetLogin{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  public static void main(String args[]){
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

       CallableStatement cs = con.prepareCall("{call GET_LOGIN_FOR_USER(?)}");
      cs.setString(1,"garfield");
      ResultSet rs = cs.executeQuery();
      ResultSetMetaData md = rs.getMetaData();

      while(rs.next()){
        for(int i=1;i<=md.getColumnCount();i++){
          System.out.print(md.getColumnLabel(i)+"\t=\t");
          if(md.getColumnType(i)==java.sql.Types.INTEGER)
            System.out.println(rs.getInt(i));
          else
            System.out.println(rs.getString(i));
```

```
        }
      }
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
  }
}
```

Note that the sequence of events is the same as in the previous example:

1. The `CallableStatement` is created using `prepareCall()`.
2. The `CallableStatement`'s parameters are set.
3. The `CallableStatement` is executed, in this instance returning a `ResultSet`.

In this example, a simple type check is performed on the returned values to ensure that the right getter method is used to retrieve the data.

Using a StoredProcedure from a JSP Bean

Now that the stored procedure has been created, it is called from a JavaBean instantiated from a JSP page. The JSP page itself is called from the action method of the HTML form displayed as part of the member-registration process. The form itself is shown in Figure 13-1.

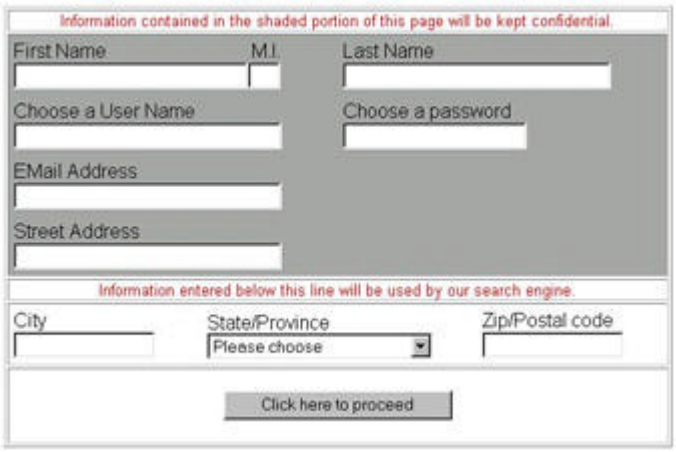


Figure 13-1: A basic name and address form used to provide data for the Contact_info Table

The HTML to create this form is shown in Listing 13-5. Note that a simple validation script has been included to ensure that at least some data is entered for each of the important fields. This form is saved as `NewMemberForm.jsp` and is called from `theProcessLogin.jsp` form handler shown in Listing 12-8.

Listing 13-5: Registration form NewMemberForm.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
```

```
<TITLE>
Member Registration
</TITLE>

<SCRIPT language=JavaScript1.1 >
function validate(form){
  if( form.elements["firstName"].value==" "||
      form.elements["lastName"].value=="" ||
      form.elements["email"].value=="" ||
      form.elements["city"].value==""  ||
      form.elements["state"].value=="?"||
      form.elements["zip"].value==""   ){
      alert("Please enter first name, last name, email, city, state and
zip.");
      return false;
  }  return true;
}
</SCRIPT>

<META content="text/html; charset=windows-1252" http-equiv=Content-Type>
</HEAD>
<BODY bgColor=#ffffff>
<BASEFONT face=Arial size=3>

<FORM action=ProcessNAForm.jsp method=POST target=_self
      onSubmit="return validate(this);">
<TABLE cellPadding=0 BORDER=0>
  <TR>
  <TD>
  <TABLE cellPadding=0 BORDER=1>
  <TR>
    <TD valign=center>
      </P>
      <SMALL>
      <CENTER>
        <FONT color=#ff0000>
          Information contained in the shaded portion of this page will be
kept
          confidential.
        </FONT>
      </CENTER>
      <SMALL>
    </TD>
  </TR>
```

```
  <TR>
    <TD>
      <TABLE cellPadding=4 cellSpacing=0 BGCOLOR="#AAAAAA" WIDTH="100%">
        <TBODY>
          <TR>
            <TD vAlign=bottom>
              <TABLE  cellSpacing=0 cellPadding=0 Border=0>
                <TR>
                  <TD>
                    First Name<BR><INPUT maxLength=30 name=firstName
size=26>
                  </TD>
                  <TD vAlign=bottom align=right>
                    M.I.<BR><INPUT maxLength=1 name=mI size=2>
                  </TD>
                </TR>
              </TABLE>
            </TD>
            <TD vAlign=bottom>
              Last Name<BR><INPUT maxLength=30 name=lastName size=30>
            </TD>
          </TR>
          <TR>
            <TD vAlign=bottom>
              Choose a User Name<BR><INPUT maxLength=30 name=username
size=30>
            </TD>
            <TD height=43  colspan=2 vAlign=bottom>
              Choose a password<BR><INPUT name=password size=20>
            </TD>
          </TR>
          <TR>
            <TD vAlign=bottom>
              EMail Address<BR><INPUT maxLength=30 name=email size=30>
            </TD>
          </TR>
          <TR>
            <TD height=43 vAlign=bottom>
              Street Address<BR><INPUT name=Street size=30>
            </TD>
          </TR>
        </TBODY>
      </TABLE>
    </TD>
```

```
  </TR>
  <TR>
    <TD valign=center>
      </P>
      <SMALL>
      <CENTER>
        <FONT color=#ff0000>
          Information entered below this line will be used by our search
engine.
        </FONT>
      </CENTER>
      <SMALL>
    </TD>
  </TR>
  <TR>
    <TD>
      <TABLE cellPadding=4 cellSpacing=0>
        <TBODY>
          <TR>
            <TD height=43 vAlign=bottom width=163>
                City<BR><INPUT name=City size=15>
            </TD>
            <TD height=49 vAlign=bottom width=229>
              State/Province<BR>
              <SELECT name=State size=1>
                <OPTION selected value="?">Please choose</OPTION>
                <OPTION value=aa>I live outside US or Canada</OPTION>
                <OPTION value=AB>Alberta</OPTION>
                <OPTION value=AK>Alaska</OPTION>
                <OPTION value=AL>Alabama</OPTION>
                <OPTION value=AR>Arkansas</OPTION>
                <OPTION value=AZ>Arizona</OPTION>
                <OPTION value=BC>British Columbia</OPTION>
                <OPTION value=CA>California</OPTION>
                <OPTION value=CO>Colorado</OPTION>
                <OPTION value=CT>Connecticut</OPTION>
                <OPTION value=DC>District of Columbia</OPTION>
                <OPTION value=DE>Delaware</OPTION>
                <OPTION value=FL>Florida</OPTION>
                <OPTION value=GA>Georgia</OPTION>
                <OPTION value=HI>Hawaii</OPTION>
                <OPTION value=IA>Iowa</OPTION>
                <OPTION value=ID>Idaho</OPTION>
```

```
<OPTION value=IL>Illinois</OPTION>
<OPTION value=IN>Indiana</OPTION>
<OPTION value=KS>Kansas</OPTION>
<OPTION value=KY>Kentucky</OPTION>
<OPTION value=LA>Louisiana</OPTION>
<OPTION value=MA>Massachusetts</OPTION>
<OPTION value=MB>Manitoba</OPTION>
<OPTION value=MD>Maryland</OPTION>
<OPTION value=ME>Maine</OPTION>
<OPTION value=MI>Michigan</OPTION>
<OPTION value=MN>Minnesota</OPTION>
<OPTION value=MO>Missouri</OPTION>
<OPTION value=MS>Mississippi</OPTION>
<OPTION value=MT>Montana</OPTION>
<OPTION value=NB>New Brunswick</OPTION>
<OPTION value=NC>North Carolina</OPTION>
<OPTION value=ND>North Dakota</OPTION>
<OPTION value=NE>Nebraska</OPTION>
<OPTION value=NF>Newfoundland</OPTION>
<OPTION value=NH>New Hampshire</OPTION>
<OPTION value=NJ>New Jersey</OPTION>
<OPTION value=NM>New Mexico</OPTION>
<OPTION value=NS>Nova Scotia</OPTION>
<OPTION value=NT>Northwest Territories</OPTION>
<OPTION value=NV>Nevada</OPTION>
<OPTION value=NY>New York</OPTION>
<OPTION value=OH>Ohio</OPTION>
<OPTION value=OK>Oklahoma</OPTION>
<OPTION value=ON>Ontario</OPTION>
<OPTION value=OR>Oregon</OPTION>
<OPTION value=PA>Pennsylvania</OPTION>
<OPTION value=PE>Prince Edward Island</OPTION>
<OPTION value=QC>Quebec</OPTION>
<OPTION value=RI>Rhode Island</OPTION>
<OPTION value=SC>South Carolina</OPTION>
<OPTION value=SD>South Dakota</OPTION>
<OPTION value=SK>Saskatchewan</OPTION>
<OPTION value=TN>Tennessee</OPTION>
<OPTION value=TX>Texas</OPTION>
<OPTION value=UT>Utah</OPTION>
<OPTION value=VA>Virginia</OPTION>
<OPTION value=VT>Vermont</OPTION>
<OPTION value=WA>Washington</OPTION>
```

```
                <OPTION value=WI>Wisconsin</OPTION>
                <OPTION value=WV>West Virginia</OPTION>
                <OPTION value=WY>Wyoming</OPTION>
                <OPTION value=YK>Yukon</OPTION>
              </SELECT>
            </TD>
            <TD height=49 vAlign=bottom width=158>
              Zip/Postal code<BR><INPUT name=Zip size=15>
            </TD>
          </TR>
        </TBODY>
      </TABLE>
    </TD>
  </TR>
  <TR>
    <TD align=center>
      <BR/>
      <INPUT name=SubmitButton type=SUBMIT value="Click here to
proceed">
      <BR/>
      <P/>
    </TD>
  </TR>
  </TABLE>
  </TD>
  </TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

After local validation by the JavaScript, the form data is passed to the JSP page `ProcessNAForm.jsp`, which uses the `ProcessNABean` to insert the form data into the database. `ProcessNAForm.jsp` is a simple example of a JSP form handler. It loads the `ProcessNABean` and sets its properties using the wild card property setter that relies on introspection to set all the properties of the JavaBean from the form data. When the `insertData()` method is called, `ProcessNABean` returns a `boolean` which is used to set the `String nextPage` to the appropriate handler. Finally, the `<jsp:forward>` tag is used to forward the user to the appropriate page. Listing 13-6 shows the JSP page.

Listing 13-6: ProcessNAForm.jsp

```
<%@ page language="java"%>
<jsp:useBean id="ProcessNABean"
class="JavaDatabaseBible.ch13.ProcessNABean" scope="session"/>
<jsp:setProperty name="ProcessNABean" property="*"/>
<%
```

```
String nextPage = "MemberWelcome.jsp";
if(ProcessNABean.insertData()){
    nextPage = "MemberProfile.jsp";
}else{
    nextPage = "NewMemberForm.jsp";
}
%>
<jsp:forward page="<%=nextPage%>"/>
```

Operation of the ProcessNABean

The first part of the `ProcessNABean` is the collection of getter and setter methods required to access the bean's parameters. These must be supplied for the bean introspection that the JSP engine requires to work properly.

The real work is done in the `insertData()` method. The `ProcessNABean` makes extensive use of a `CallableStatement` object, `cs`. First it calls the stored procedure `GET_LOGIN_FOR_USER` to validate the username against the Login table. If the username is already in use, the boolean flag `username_selection_ok` is set to false so that the JSP page can notify the user that he or she needs to select a different username.

Once the user has selected a valid, unique username, the `CallableStatement` object is used to call the stored procedure `SET_LOGIN_FOR_USER` to update the Login table with the new username and password. The stored procedure `SET_LOGIN_FOR_USER` is defined as follows:

```
CREATE PROCEDURE SET_LOGIN_FOR_USER
  @USERNAME VARCHAR(20),
  @PASSWORD VARCHAR(20)
  AS
  INSERT INTO LOGIN (USERNAME, PASSWORD)
         VALUES (@USERNAME, @PASSWORD);
```

The stored procedure `GET_LOGIN_FOR_USER` is then called again to get the auto generated MemberID assigned to this user. A more elegant way to do this is to use the `getGeneratedKeys()` method defined in JDBC 3.0 for the `Statement` object as shown here:

```
if(cs.executeUpdate()!=1)ok = false;
Result rs = cs.getGeneratedKeys();
```

Cross-Reference

The use of the JDBC 3.0 extension method `Statement.getGeneratedKeys()` is discussed in Chapter 4.

Finally, the stored procedure INSERT_CONTACT_INFO is called to insert the member data stored in the `ProcessNABean`.

The code for the `ProcessNABean` is shown in Listing 13-7.

Listing 13-7: Calling a stored procedure from a JavaBean

```
package JavaDatabaseBible.ch13;

import java.sql.*;
import javax.sql.*;
```

```
public class ProcessNABean extends java.lang.Object{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  protected String firstName;
  protected String lastName;
  protected char   mi;
  protected String street;
  protected String city;
  protected String state;
  protected String zip;
  protected String phone;
  protected String email;
  protected String username;
  protected String password;

  public ProcessNABean(){
  }
  public void setUsername(String username){
    this.username = username;
  }
  public void setPassword(String password){
    this.password = password;
  }
  public void setFirstName(String firstName){
    this.firstName = firstName;
  }
  public void setLastName(String lastName){
    this.lastName = lastName;
  }
  public void setMi(char mi){
    this.mi= mi;
  }
  public void setStreet(String street){
    this.street = street;
  }
  public void setCity(String city){
    this.city = city;
  }
  public void setState(String state){
    this.state = state;
  }
```

```
public void setZip(String zip){
  this.zip = zip;
}
public void setPhone(String phone){
  this.phone = phone;
}
public void setEmail(String email){
  this.email = email;
}
public String getUsername(){
  return username;
}
public String getPassword(){
  return password;
}
public String getFirstName(){
  return firstName;
}
public String getLastName(){
  return lastName;
}
public char getMi(){
  return mi;
}
public String getStreet(){
  return street;
}
public String getCity(){
  return city;
}
public String getState(){
  return state;
}
public String getZip(){
  return zip;
}
public String getPhone(){
  return phone;
}
public String getEmail(){
  return email;
}
public boolean insertData(){
```

```
    boolean username_selection_ok = true;
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      CallableStatement cs = con.prepareCall("{call
GET_LOGIN_FOR_USER(?)}");
      cs.setString(1,username);
      ResultSet rs = cs.executeQuery();
      ResultSetMetaData md = rs.getMetaData();

      int id = -1;
      while(rs.next()){
        id = rs.getInt("MemberID");
      }

      if(id>=0){
        System.out.println(id+": "+username+"; "+password);
        username_selection_ok = false;
      }else{

        cs = con.prepareCall("{call SET_LOGIN_FOR_USER(?,?)}");

        cs.setString(1,username);
        cs.setString(2,password);

        if(cs.executeUpdate()!=1) username_selection_ok = false;

        cs = con.prepareCall("{call GET_LOGIN_FOR_USER(?)}");
        cs.setString(1,username);
        rs = cs.executeQuery();
        while(rs.next()){
          id = rs.getInt("MemberID");
        }

        cs = con.prepareCall("{call
```

```
INSERT_CONTACT_INFO(?,?,?,?,?,?,?,?,?,?)}");

        cs.setInt(1,id);
        cs.setString(2,firstName);
        cs.setString(3,String.valueOf(mi));
        cs.setString(4,lastName);
        cs.setString(5,street);
        cs.setString(6,city);
        cs.setString(7,state);
        cs.setString(8,zip);
        cs.setString(9,"<NULL>");
        cs.setString(10,email);
        if(cs.executeUpdate()!=1) username_selection_ok = false;
      }

    }catch(ClassNotFoundException e1){
      System.err.println(e1.getMessage());
    }catch(SQLException e2){
      System.err.println(e2.getMessage());
    }
    return username_selection_ok;
 }
}
```

Error Handling

Recall that the `ProcessNABean` notifies the `ProcessNAForm.jsp` page that the user needs to select a different username by setting the boolean flag `username_selection_ok` to false. This lets the `ProcessNAForm.jsp` know that a problem has arisen, so it then sends the user back to the form so he or she can select a new username and password.

As it stands, the form is cleared when redisplayed. This is virtually guaranteed to ensure that the user gets fed up and surfs on. The way to avoid this is to fill in the fields the user has already completed and to present a message telling him or her what to do next.

One of the primary uses of JavaBeans in JSP applications is data storage. Since all the form data has already been inserted into the `ProcessNABean`, completing the form for the user requires only the addition of this line:

```
<jsp:useBean id="ProcessNABean".../>
```

Also, include these few extra lines of code to set the properties:

```
First Name<BR><INPUT maxLength=30 name=firstName
  value='<jsp:getProperty name="ProcessNABean" property="firstName"/>'
size=26>
```

A partial listing of the modified form is shown in Listing 13-8.

Listing 13-8: ProcessNAForm.jsp modified for use as an error page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>
Member Registration
</TITLE>

<SCRIPT language=JavaScript1.1 >
function validate(form){
  if( form.elements["firstName"].value==" "||
      form.elements["lastName"].value=="" ||
      form.elements["email"].value=="" ||
      form.elements["city"].value==""  ||
      form.elements["state"].value=="?"||
      form.elements["zip"].value==""   ){
      alert("Please enter first name, last name, email, city, state and
zip.");
      return false;
  }
  return true;
}
</SCRIPT>

<META content="text/html; charset=windows-1252" http-equiv=Content-Type>
</HEAD>
<BODY bgColor=#ffffff>
<BASEFONT face=Arial size=3>
<%@ page session="true" %>
<jsp:useBean id="ProcessNABean"
class="JavaDatabaseBible.ch13.ProcessNABean" scope="session"/>

<FORM action=ProcessNAForm.jsp method=POST target=_self
      onSubmit="return validate(this);">
<TABLE cellPadding=0 BORDER=0>
  <TR>
  <TD>
  <TABLE cellPadding=0 BORDER=1>
  <TR>
    <TD valign=center>
      </P>
      <SMALL>
      <CENTER>
        <FONT color=#ff0000>
```

```
          Information contained in the shaded portion of this page will
be kept
          confidential
        </FONT>
      </CENTER>
      <SMALL>
    </TD>
  </TR>
  <TR>
    <TD>
      <TABLE cellPadding=4 cellSpacing=0 Border=0 BGCOLOR="#AAAAAA"
             WIDTH="100%">
        <TBODY>
          <TR>
            <TD vAlign=bottom>
              <TABLE  cellSpacing=0 cellPadding=0 Border=0>
                <TR>
                  <TD>
                  First Name<BR><INPUT maxLength=30 name=firstName
                  value=
                  '<jsp:getProperty name="ProcessNABean"
property="firstName"/>'
                  size=26>
                  </TD>
                  <TD vAlign=bottom align=right>
                  M.I.<BR><INPUT maxLength=1 name=mi value=
                  '<jsp:getProperty name="ProcessNABean"
property="mi"/>'size=2>
                  </TD>
                </TR>
              </TABLE>
            </TD>
            <TD vAlign=bottom>
            Last Name<BR><INPUT maxLength=30 name=lastName
            value='<jsp:getProperty name="ProcessNABean"
property="lastName"/>
            'size=30>
            </TD>
          </TR>
          <TR>
            <TD vAlign=bottom>
              Choose a User Name<BR><INPUT maxLength=30 name=username
              value=
```

```
              '<jsp:getProperty name="ProcessNABean"
property="username"/>'
              size=30>
            </TD>
            <TD height=43  colspan=2 vAlign=bottom>
              Choose a password<BR><INPUT name=password size=20>
            </TD>
          </TR>
          <TR>
            <TD vAlign=bottom>
              EMail Address<BR><INPUT maxLength=30 name=email
              value='<jsp:getProperty name="ProcessNABean"
property="email"/>'
              size=30>
            </TD>
          </TR>
          <TR>
            <TD height=43 vAlign=bottom>
              Street Address<BR><INPUT name=street
              value=
              '<jsp:getProperty name="ProcessNABean"
property="street"/>'
              size=30>
            </TD>
          </TR>
        </TBODY>
      </TABLE>
    </TD>
  </TR>
```

Note

Listing 13-8 is only partial, showing the basic workings of the JSP page. Substitute this into Listing 13-5, and implement the additional lines for the remaining properties to create a complete form.

Figure 13-2 shows the use of the original form as a means of providing interactive feedback to the user. This kind of user feedback is important in terms of ensuring that a user will take the trouble to complete a form instead of simply surfing on.

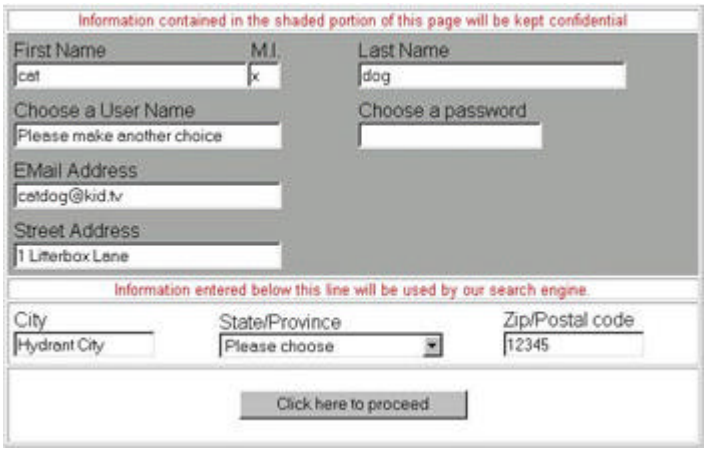


Figure 13-2: Member-registration form with user data restored and error message displayed for user name

Using Stored Procedures with Input and Output Parameters

In addition to supplying input parameters to a stored procedure, you can get output parameters from a stored procedure. If you decide to use an output parameter, it must be registered as an `OUT` parameter using the `CallableStatement.registerOutParameter()` method before the execute method is called. Here's an example:

```
cstmt.registerOutParameter(1, java.sql.Types.VARCHAR);
```

`OUT` parameter values can be retrieved after execution using get methods appropriate to the data types of the values. Because of limitations some relational database management systems impose, all of the results the execution generates of a `CallableStatement` object should be retrieved before `OUT` parameters are retrieved.

Listing 13-9 gives an example of a simple stored procedure that checks a user name and password against the database, returning the `String` "PASS" if a match is found or "FAIL" otherwise.

Listing 13-9: Using an output parameter with a stored procedure

```
CREATE PROCEDURE CHECK_USER_NAME
 @UserName varchar(30),
 @Password varchar(20),
 @PassFail varchar(20) OUTPUT
As
IF EXISTS(Select * From Login
Where UserName = @UserName
 And
Password = @Password)
 SELECT @PassFail = 'PASS'
else
 SELECT @PassFail = 'FAIL';
```

Note

Stored procedures can contain more than one SQL statement, in which case they produce multiple results, and the `execute` method should be used. In cases where a `CallableStatement` object returns multiple `ResultSet` objects, all of the results should be retrieved using the method `getMoreResults` before `OUT` parameters are retrieved.

Listing 13-10 provides an example of using the simple stored procedure of Listing 13-9. Notice the call to the `registerOutParameter()` method prior to calling the `CallableStatement's getString` method to retrieve the output parameter.

Listing 13-10: Getting an output parameter from a stored procedure

```
package JavaDatabaseBible.ch13;

import java.sql.*;
import javax.sql.*;

public class CheckPassword{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  public static void main(String args[]){
    int id = -1;
    String password = null;
    String username = "";
    if(args.length>0)username = args[0];
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      CallableStatement cs = con.prepareCall("{call
CHECK_USER_NAME(?,?,?)}");
      cs.setString(1,"garfield");
      cs.setString(2,"lasagna");
      cs.registerOutParameter(3, java.sql.Types.VARCHAR);
      cs.executeUpdate();
      System.out.println(cs.getString(3));
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
```

```
    }
  }
}
```

Summary

This chapter discusses improving the efficiency of JDBC-based applications by comparing and contrasting the three variations on the `java.sql.Statement` object:

- `java.sql.Statement`, which performs in line execution of a SQL command. This approach is ideal for one-shot execution of a single command, since it involves minimum overhead.
- `java.sql.PreparedStatement`, which offers a means of precompiling SQL commands. This approach is best for executing a command in a loop, since the `PreparedStatement` passes the SQL command to the SQL engine where it is parsed, compiled and cached for efficiency and speed of execution. There is a slight overhead incurred in the precompilation and caching process.
- `java.sql.CallableStatement`, which allows you to call SQL stored procedures. This approach takes advantage of SQL's ability to precompile and store procedures which can subsequently be executed by name.

Now you know all about inserting basic data types into a database from an HTML form. Chapter 14 discusses inserting and retrieving large objects, such as images and word-processor documents, as blobs and clobs.

Chapter 14: Using Blobs and Clobs to Manage Images and Documents

In This Chapter

Traditionally, relational database management systems have been designed around the need to handle simple traditional data types such as bytes, integers, floats, and Strings. The evolution of computer hardware and software has introduced both the need and the capability to store much larger data objects, such as images and even video clips, economically and efficiently.

Until recently, these larger data objects have been stored in traditional file systems, resulting in significant loss of efficiency whenever very large numbers of such objects were involved. The designers of relational database management systems have responded by providing support for the management and storage of these large objects within the database itself.

This chapter discusses the use of relational databases to store and retrieve large objects in various ways. Examples include the use of servlets to upload images to a database, and to retrieve them for display in a browser.

Large Objects

Support for *large objects (LOBs)* is an important feature of modern object relational databases. The SQL3 standard defines a number of new data types for managing large objects. These data types are supported by the JDBC extension API. The new SQL3 large object data types supported by the JDBC 2.0 extension include the following:

- `ARRAY` — which can store an array as a column value
- `BLOB` (binary large object) — which can store large amounts of data as raw bytes
- `CLOB` (character large object) — which can store large amounts of character data
- Structured types
- References to structured types

Caution

Different RDBMS systems use different internal types to manage large objects, so refer to your documentation to find out which data types to use for large-object storage.

JDBC 2.0 defines a set of interfaces that map SQL3 types. Table 14-1 shows the type mappings and the retrieval, storage, and update methods for the different large object types.

Table 14-1: SQL3 Large Object Data Types

SQL3 type	Java Interface	get	set	Update
BLOB	java.sql.Blob	getBlob	setBlob	updateBlob
CLOB	java.sql.Clob	getClob	setClob	updateClob
ARRAY	java.sql.Array	getArray	setArray	updateArray
SQL Structured type	java.sql.Struct	getObject	setObject	updateObject
REF to Structured Type	java.sql.Ref	getObject	setObject	updateObject

Note

At the time of this writing, the update methods are scheduled for future release. Until then, you can use the method updateObject, which works just as well.

Large-object support is the database community's response to evolving requirements to manage nontraditional data types, such as images, as well as more traditional data types, such as prices, dates, and quantities. The traditional data types are relatively simple and typically require anywhere from a handful of bytes for an integer value to perhaps a few tens of bytes for a name or address. Relational

database management systems have been optimized to handle rows containing relatively small numbers of these types of data fields.

Many modern applications require the management of much larger data objects, from images, which may require tens of kilobytes of storage, to video clips, which may run into the hundreds of megabytes. The earliest approach to handling large objects was to store them as files in the underlying operating system, using the database to store only the file path and letting the application code manage the file. Today, many enterprise RDBMS systems support large objects directly as special data types, albeit with certain restrictions on using them in queries.

Since large objects are, by definition, large, they are managed using SQL locators. Conceptually, a locator is similar to a C or C++ pointer which contains the location of an object rather than the object itself. RDBMS systems use locators to manage large objects because handling them *in-line* destroys the optimization that RDBMS systems perform to map data objects to physical-storage devices such as disk sectors.

An important feature of `ARRAY`s, `BLOB`s, and `CLOB`s, is that, since they are accessed using locators, you can manipulate them without having to copy all the data from the server to the client machine. In fact, when you query a database for a large object, the locator, rather than the actual object, is returned in the ResultSet. Using pointers in this way is more efficient than moving large quantities of data around the system for each column, so this feature can improve performance dramatically. As a JDBC developer, you won't have to deal with locators, but it is useful to understand the concept so you can see why the various large-object manipulation methods work the way they do.

Once you have the locator, you must specifically ask for the large-object data. This process is known as *materializing* the data. For example, to retrieve an image stored as a `BLOB`, you can materialize it either as a byte array, using `Blob.getBytes()`, or as an `InputStream`, using `Blob.getBinaryStream()`.

Although this chapter focuses on the use of Blobs and Clobs, you can see from Table 14-1 that large-object support works consistently for all of these data types. Once you understand how to handle one, you understand them all.

Using Blobs to Store Binary Data

Blobs provide a means of storing and managing large quantities of binary data. Typical examples of large binary data objects are audio and video clips and image files. Blobs are particularly useful in Web applications for storing images. JDBC support for Blobs is provided by the `Blob Interface`, which defines these access methods:

- `public InputStream getBinaryStream()`
- `public byte[] getBytes(long position, int length)`

In addition, the `Blob` interface defines the utility methods `length()` and `position()`, which return the number of bytes in the `Blob` and the offset to a contained byte array or `Blob`. The `ResultSet` method `getBlob()` is used to retrieve the locator of a `Blob` from a `ResultSet`, while the method `setBlob()` in the `PreparedStatement` interface can be used to set a `Blob`. In practice, a more common way to write a Blob to a database table is to use `PreparedStatement.setBinaryStream()` to transfer data directly from an `InputStream` to the RDBMS system. An example of this approach is shown in Listing 14-1.

Listing 14-1: Inserting a Blob into a table

```
package JavaDatabaseBible.ch14;

import java.io.*;
import java.sql.*;
import javax.sql.*;
```

```
public class BlobSaver{
  private static String dbUserName = "jod";
  private static String dbPassword = "jod";

  public static void main(String args[]){
    BlobSaver blobber = new BlobSaver();
    blobber.saveImage(1,"Witch","Witch.gif");
  }

  public void saveImage(int imageID,String description,String filename){
    String cmd =
      "INSERT INTO Photos (ImageID,Description,Image) VALUES(?,?,?)";
    File imgFile = new File(filename);
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "MARS" );
      tds.setDatabaseName( "CONTACTS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      PreparedStatement pstmt = con.prepareStatement(cmd);

      pstmt.setInt(1, imageID);
      pstmt.setString(2, description);
      pstmt.setBinaryStream(3, new FileInputStream(filename),
                            (int)imgFile.length());
      pstmt.executeUpdate();
      con.close();
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
    catch(FileNotFoundException e){
      e.printStackTrace();
    }
```

```
  }
}
```

As you can see from the listing, the method `PreparedStatement.setBinaryStream()` is very bit as easy to use as any of the other set parameter methods. You simply use the `setStream()` methods just like `setInt()` or `setString()`.

Note

The Blob interface makes no attempt to check whether the Blob contains an image or an audio clip or whatever. Essentially, the Blob is defined as a means of storing large chunks of binary data; what you do with the data is up to you.

Using Clobs to Store Text Data

Clobs are similar to Blobs in that they are designed for the storage and management of large data objects; but in the case of Clobs, these are defined as text objects. The primary difference between Clobs and Blobs is that the Clob interface supports character-oriented access methods such as the following:

- `public InputStream getAsciiStream()`
- `public Reader getCharacterStream()`
- `public String getSubString(long pos, int length)`

Like the `Blob`, the `Clob` has the utility methods `length()` and `position()`, which return the number of characters in the `Clob` and the offset to a contained search String or an included `Clob`.

Note

Unlike normal String methods, `getSubString()` starts counting from 1 rather than from 0; to return the entire clob as a `String`, use `getSubString(1, clob.length())`.

The `ResultSet` method `getClob()` can be used to retrieve the locator of a `Clob` from a `ResultSet`, and the method `setClob()` in the `PreparedStatement` interface can be used to set a `Clob`. As in the case of a `Blob`, a more common way to write a Clob to a database table is to use a `setStream()`method (in this case, the ones listed here):

- `setAsciiStream()`
- `setUnicodeStream()`
- `setCharacterStream()`

Using one of the `setStream()` methods lets you transfer data directly from an `InputStream` to the RDBMS system. Listing 14-2 illustrates the use of a `FileReader` and the `setCharacterStream()` method.

Listing 14-2: Saving a Clob to an RDBMS using a FileReader

```
public void saveDocument(int memberID,String title,String filename){
  String cmd =
           "INSERT INTO Documents "+
           "(MemberID,Title,Document) VALUES(?,?,?)";
  File doc = new File(filename);
  System.out.println(filename+" - "+doc.length());
  try {
    Class.forName("com.inet.pool.PoolDriver");
   com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
   tds.setServerName( "MARS" );
   tds.setDatabaseName( "CONTACTS" );
```

```
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      PreparedStatement pstmt = con.prepareStatement(cmd);

      pstmt.setInt(1, memberID);
      pstmt.setString(2, title);
      pstmt.setCharacterStream(3, new FileReader(doc),
                                  (int)doc.length());
      pstmt.executeUpdate();

      con.close();
     }
     catch(ClassNotFoundException e){
     e.printStackTrace();
     }
     catch(SQLException e){
       e.printStackTrace();
     }    catch(FileNotFoundException e){
       e.printStackTrace();
     }
   }
```

Uploading Images and Documents from a Browser

A common requirement in Web applications is to upload images and documents from a client machine over the Internet. Uploading files using an HTML form is part of the HTML standard and is supported by all major browsers. However, in spite of being a standard capability, HTML file upload isn't very well documented elsewhere, so it is worth reviewing how to create a servlet to handle uploads.

HTML file uploads use the multipart message format defined by the Multipurpose Internet Mail Extensions (MIME) standard, sending each field of the form as a separate MIME part. The main points to notice about creating the HTML upload form are as follows:

- The "method" attribute of the FORM is set to "post".
- The attribute "enctype = multipart/form-data" is added to the FORM element.
- An INPUT element with the type "file" is used to specifiy the file to upload.

When the form is set up like this, the browser creates a file select control that lets you select the file to upload. Listing 14-3 shows an example of a simple HTML upload form.

Listing 14-3: HTML file-upload form

```
<HTML>
<BODY>
  <FORM action="servlet/BlobUploadServlet"
```

```
        enctype="multipart/form-data" method="post">
  <INPUT type="hidden" name="ID" value="1">
  <TABLE BORDER=1>
    <TR>
      <TD ALIGN="center">
      Filename: <INPUT type="file" name="submit-file" size="40">
      </TD>
    </TR>
    <TR>
      <TD ALIGN="center">
      <center>
      <INPUT type="submit" value="Send">
      <INPUT type="reset">
      </center>
      </TD>
    </TR>
    </TABLE>
  </FORM>
</BODY>
</HTML>
```

This form contains a hidden field, with the member ID field set by the JSP page or servlet that creates the form. The form also contains a file select field. The servlet shown in Listing 14-4 echoes the upload back to the browser, so you can look at the upload format.

Listing 14-4: Blob upload test servlet

```
import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BlobTestServlet extends HttpServlet{
  public void doPost( HttpServletRequest request, HttpServletResponse
response )
  throws ServletException, IOException{
    ServletOutputStream out = response.getOutputStream();
    BufferedInputStream in  = new
BufferedInputStream(request.getInputStream());
    out.println(request.getHeader("content-type"));
    int c = -1;
    while ( (c=in.read()) >= 0 )out.write( c );
    out.close();
  }
}
```

If you select a GIF file for the first file, the data stream that this forms will look something like Listing 14-5. The listing has been edited to remove most of the bytes representing the GIF image file.

Listing 14-5: Edited view of the multipart data stream

```
multipart/form-data; boundary=---------------------------7d21e01ffec
-----------------------------7d21e01ffec
Content-Disposition: form-data; name="ID"


1
-----------------------------7d21e01ffec
Content-Disposition: form-data; name="submit-file";
filename="C:\Clipart\Test.gif"
Content-Type: image/gif


GIF89a_ _ ñ—....................................._#_6  1¢•6•°U  ;
-----------------------------7d21e01ffec--
```

One way to parse a data stream in multipart MIME format is to use the JavaMail API. However, a simpler approach is to parse the data stream yourself. This approach will be demonstrated by developing a `BlobUploadServlet` illustrates the basics of parsing a multipart MIME document.

The MIME parts are separated by boundaries, which are unique lines of text defined in the header and guaranteed not to occur inside any MIME part. Each MIME part is made of a header section, a blank line, and the body or payload.

The header section contains several headers defining the content and format of the body area. Headers have a colon separated name/value pair and, optionally, several parameters separated by semicolons. The parameters are similar to HTML attributes, with a name = value pair.

The MIME boundary is specified in the `Content-Type` header. In the Blob upload servlet, the `getBoundary()` method parses out the boundary substring, prepends `CRLF` and two hyphens, and returns the boundary as a String. This `read()` method, which is used to retrieve the payload Blob, uses this boundary string.

The `read()` method creates a `PushbackInputStream` from the `ServletInputStream` and returns input characters from the stream. If it encounters a boundary, it discards it, returning a flag to indicate that a boundary has been reached. Since all normal characters are positive integers, a –1 is returned when a boundary is encountered (unless it is the final boundary, in which case a –2 is returned).

The header area of each part, which, as you recall, corresponds to a field in the HTML form, contains a `Content-Disposition` header, with the value "form-data". This `Content-Disposition` header contains the attribute "name" with the name of the field specified in the HTML form as its value. If the field type is "file", the header will also contain the attribute "filename", with the name of the file being uploaded.

The headers are parsed by the `parseHeader()` method, which returns a `Hashtable` of header parameters. These are merged into the parameter `Hashtable`, since parameters such as member id are in a different header from the file name.

Th `BlobUploadServlet` has been written to output header information to the `ServletOutputStream`, so you can see the results of parsing the `ServletInputStream`. Listing 14-6 shows the servlet output.

Listing 14-6: Ouput of the BlobUploadServlet

```
boundary =

-----------------------------7d2104226b0

Content-Disposition: form-data; name="ID"

Content-Disposition: form-data; name="submit-file"; filename="C:\JDBC
Bible\Projects\Ch14\bather.jpg"

  ID = 101
  filename = C:\JDBC Bible\Projects\Ch14\bather.jpg
  name = submit-file
  Content-Disposition = form-data
  Content-Type = image/pjpeg
  ...saving payload
```

The servlet is designed specifically to handle Blob uploads, but it can obviously be modified to handle Clobs with minimal effort. You can use the `Content-Type` parameter to determine the uploaded file type and select the appropriate JDBC methods when saving the data. If the uploaded file is an image, the `Content-Type` parameter will be `image/pjpeg` or `image/gif`, and so on. Similarly, if you upload a text file, the `Content-Type` will be set automatically to `text/plain`, and MSWord documents will have their `Content-Type` set to `application/msword`, and so on.

The method `savePayload()` parses the Blob to a byte array and saves it to the DBMS table in the method `saveBlob()`. The `saveBlob()` method uses the member id retrieved from a preceding header and saved in the params `Hashtable` as one of the inputs to the `PreparedStatement` used to save the Blob to the database table. The Blob upload servlet is shown in Listing 14-7.

Listing 14-7: Uploading images using a Blob upload servlet

```
import java.io.*;
import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BlobUploadServlet extends HttpServlet{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  private static final char CR = 13;
  private static final char LF = 10;
```

```
protected String boundary = null;
protected Hashtable params = new Hashtable();

public void doPost( HttpServletRequest request,

 HttpServletResponse response )
  throws ServletException, IOException{
  ServletOutputStream out = response.getOutputStream();
  ServletInputStream  in  = request.getInputStream();
  BufferedInputStream bin  = new BufferedInputStream(in);
  boundary = getBoundary(request.getHeader("content-type"));

  out.println("<html><body><pre>");
  out.println("boundary =\n"+boundary);
  out.println();

  byte[] bytes = new byte[128];
  in.readLine(bytes,0,bytes.length);
  String line = new String(bytes);
  Hashtable header = null;
  while(in.readLine(bytes,0,bytes.length)>=0){
    line = new String(bytes);
    if(line.startsWith("Content-Disposition:")){
      out.println(line);
      header = parseHeader(line);
      updateParams(header);
    }else if(line.startsWith("Content-Type:")){
      params.put("Content-Type",
                  line.substring("Content-Type:".length()).trim());
    }else{
      if(header!=null&&bytes[0]==13){
        if(header.containsKey("filename")){
          displayParams(out);
          out.println("  ...saving payload");
          savePayload(params,bin);
          header = null;
        }else{
          String name = (String)header.get("name");
          String value = getParameter(in).trim();
          params.put(name,value);
        }
      }if(line.indexOf(boundary)>=0)out.println(line);
    }
```

```
      bytes = new byte[128];
    }
    out.println("</pre></body></html>");
    out.close();
  }
  private void displayParams(ServletOutputStream out)
  throws java.io.IOException{
    for (Enumeration e = params.keys();e.hasMoreElements();) {
      String key = (String)e.nextElement();
      out.println("  "+key+" = "+params.get(key));
    }
  }
  private void updateParams(Hashtable header){
    for (Enumeration e = header.keys();e.hasMoreElements();) {
      String key = (String)e.nextElement();
      params.put(key,header.get(key));
    }
  }
  private String getParameter(ServletInputStream  in)
  throws java.io.IOException{
    byte[] bytes = new byte[128];
    in.readLine(bytes,0,bytes.length);
    return new String(bytes);
  }
  private String getBoundary(String contentType){
    int bStart = contentType.indexOf("boundary=")+"boundary=".length();
    return "" + CR + LF + "--" + contentType.substring(bStart);
  }
  private void savePayload(Hashtable params,BufferedInputStream is)
  throws java.io.IOException{
    int c;
    PushbackInputStream input = new PushbackInputStream(is,128);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    while ( (c=read(input,boundary)) >= 0 )out.write( c );
    int id = Integer.parseInt((String)params.get("ID"));
    saveBlob(id,(String)params.get("filename"),out.toByteArray());
    out.close();
  }

  private int read( PushbackInputStream input, String boundary )
  throws IOException
  {
    StringBuffer buffer = new StringBuffer();
```

```
    int index = -1;
    int c;

    do {
      c = input.read();
      buffer.append( (char)c );
      index++;
    }while ( (buffer.length() < boundary.length()) &&
             (c == boundary.charAt(index)) );

    if ( c == boundary.charAt(index) ){
      int type = -1;
      if ( input.read() == '-' )
        type = -2;
      while ( input.read() != LF );
      return type;
    }

    while ( index >= 0 ){
      input.unread( buffer.charAt(index));
      index--;
    }
    return input.read();
  }
  private Hashtable parseHeader(String line){
    Hashtable header = new Hashtable();
    String token = null;
    StringTokenizer st = new StringTokenizer(line,";");
    while(st.hasMoreTokens()){
      token = ((String)st.nextToken()).trim();
      String key = "";
      String val = "";
      int eq = token.indexOf("=");
      if(eq <0) eq = token.indexOf(":");
      if(eq >0){
        key = token.substring(0,eq).trim();
        val = token.substring(eq+1);
        val = val.replace('"',' ');
        val = val.trim();
        header.put(key,val);
      }
    }
    return header;
```

```
  }
  public void saveBlob(int memberID,String description,byte[] out){
    String cmd =
      "INSERT INTO Photos (MemberID,Description,Image) VALUES(?,?,?)";
    System.out.println(cmd);
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      PreparedStatement pstmt = con.prepareStatement(cmd);
      pstmt.setInt(1, memberID);
      pstmt.setString(2, description);
      pstmt.setBytes(3, out);
      System.out.println(pstmt.executeUpdate());

      con.close();
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
  }
}
```

A Servlet for Downloading Large Objects from a DBMS

The conventional way of incorporating images or other large objects in a Web page is to provide a link to a disk file and to rely on the operating system to find the file. This works just fine when you have only a few image files, but in a membership Web site with tens or hundreds of thousands of members, each of whom may have several photos on file, search times become significant. One way around this is to design a directory tree, containing hundreds of subdirectories arranged in some logical manner so that you can navigate rapidly to the right subdirectory.

Letting your DBMS do the work is a much more elegant and attractive way to find the image files. A big advantage of object relational database management Systems, after all, is that they are designed specifically for this kind of thing.

The servlet of Listing 14-8 shows how you can retrieve an image from a DBMS as a Blob and write it as a byte array to the ServletOutputStream. The servlet also retrieves text as a Clob.

Caution

It is important when downloading non-html data to set the correct content type in the response object. Some browsers are more sensitive to this than others.

Listing 14-8: A servlet that retrieves large objects

```
package JavaDatabaseBible.ch14;

import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LobServlet extends HttpServlet{
  private String dbUserName = "sa";
  private String dbPassword = "dba";
  protected void doGet(HttpServletRequest request,HttpServletResponse
response)
  throws ServletException, IOException
  {
    ServletOutputStream out = response.getOutputStream();
    String dataType = request.getParameter("type");
    int memberID = Integer.parseInt(request.getParameter("id"));
    if(dataType.equalsIgnoreCase("blob")){
      response.setContentType("image/jpeg");
      out.write(getBlob(memberID));
    }else if(dataType.equalsIgnoreCase("clob")){
      response.setContentType("text/html");
      out.write(getClob(memberID));
    }
    out.flush();
    out.close();
  }
  public byte[] getBlob(int memberID){
    String query = "SELECT Image FROM Photos WHERE MemberID = ?";
    Blob blob = null;
    byte[] bytes = null;
    String description = "";
    try {
    Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
```

```
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      PreparedStatement pstmt = con.prepareStatement(query);
      pstmt.setInt(1, memberID);

      ResultSet rs = pstmt.executeQuery();
      ResultSetMetaData md = rs.getMetaData();
      while (rs.next()) {
        blob = rs.getBlob(1);
      }
      bytes = blob.getBytes( 1, (int)(blob.length()));
      con.close();
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
    return bytes;
  }
  public byte[] getClob(int memberID){
    String query = "SELECT Document FROM Documents WHERE MemberID = ?";
    Clob clob = null;
    String text = null;
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      PreparedStatement pstmt = con.prepareStatement(query);
      pstmt.setInt(1, memberID);
```

```
      ResultSet rs = pstmt.executeQuery();
      ResultSetMetaData md = rs.getMetaData();
      while (rs.next()) {
        clob = rs.getClob(1);
      }
      text = clob.getSubString(1,((int)clob.length()));
      con.close();
    }
    catch(ClassNotFoundException e){
      e.printStackTrace();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
    byte[] bytes = null;
    if(text!=null)bytes = text.getBytes();
    return bytes;
  }
}
```

The large object servlet can be used to drive a Web page by combining images saved as Blobs and text or HTML saved as Clobs. Listing 14-9 offers a simple illustration.

Listing 14-9: Creating a Blob-based and Clob-based Web page using frames

```
<html>
<head>
<title>Byron</title>
</head>
<frameset cols="50%,*">
<frame src="http://localhost/servlet/LobServlet?type=blob&id=1">
<frame src="http://localhost/servlet/LobServlet?type=clob&id=1">
</frameset>
<body>
</body>
</html>
```

In Figure 14-1, the image is a jpeg, previously stored using ID = 1, and the text is stored in HTML form as a Clob using the same ID. The HTML frame set simply serves to format the page, and the content is entirely database driven. This is a simple way to drive catalog pages and similar formats.

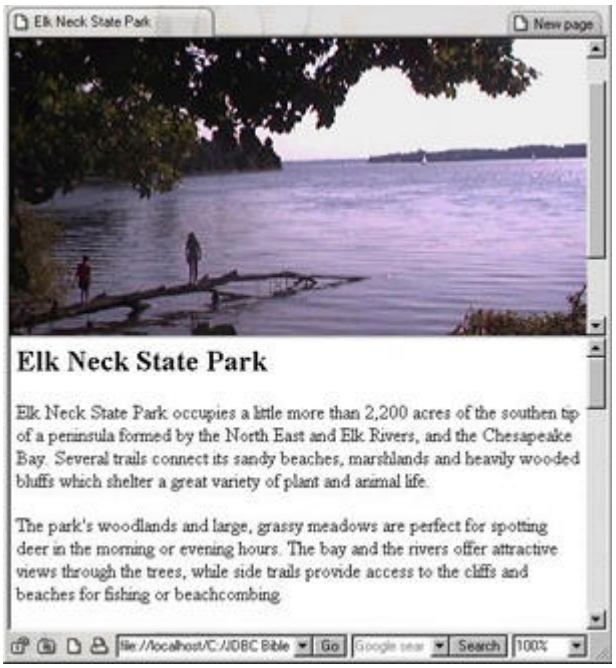


Figure 14-1: Blob-based and Clob-based Web page using frames

Summary

In this chapter, you learned about handling large data objects. Specifically, you were introduced to these topics:

- Saving and retrieving images and other binary data as Blobs
- Saving and retrieving documents and other text data as Clobs
- Handling HTML image uploads using servlets
- Creating Blob-based and Clob-based Web pages using a large object retrieval servlet.

In Chapter 15, you learn about retrieving data from a DBMS using scrollable `ResultSets`.

Chapter 15: Using JSPs, XSL, and Scrollable ResultSets to Display Data

In This Chapter

One of the limitations of the JDBC `ResultSet` is that the user is restricted to scrolling forwards through the rows. The JDBC 2.0 API adds the ability to define a `ResultSet` as scrollable so you can move the cursor in either direction or to a particular row.

This enhancement is particularly when you need to add a graphical user interface to the `ResultSet`. The ability to move through a `ResultSet` in only one direction would be very restrictive.

Scrollable ResultSets

In the `ResultSet` object defined in the JDBC Core API, the only way to scroll through the rows was to use the `next()` method, which moves the cursor forward to the next row. One of the features added in the JDBC 2.0 API is the ability to define a `ResultSet` as scrollable. Unlike the basic `ResultSet`, which only lets you move the cursor forward, the scrollable `ResultSet` lets you move the cursor in either direction or to a particular row. In addition, the scrollable `ResultSet` lets you get the cursor position.

Creating a Scrollable ResultSet

The type of `ResultSet` a `java.sql.Statement` object returns is defined when the `Statement` is created by the `Connection.createStatement` method. There are two forms of the `Connection.createStatement` method.

This basic version of `createStatement()`gets you a nonscrollable default `ResultSet`:

```
public Statement createStatement()
```

The second variant allows you to create scrollable and updateable `ResultSets`, as shown here:

```
public Statement createStatement(int rsType, int
rsConcurrency)
```

The first argument, `rsType`, must be one of the three following constants added to the `ResultSet` interface to indicate the type of a `ResultSet` object:

- `TYPE_FORWARD_ONLY`
- `TYPE_SCROLL_INSENSITIVE`
- `TYPE_SCROLL_SENSITIVE`

If you want a scrollable `ResultSet` object, you must specify either `TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`. A `ResultSet` defined using `TYPE_SCROLL_INSENSITIVE` *does not* reflect changes made while it is still open. A `TYPE_SCROLL_SENSITIVE ResultSet` *does* reflect changes made while it is still open. Of course, you can always see changes, regardless of the type of `ResultSet` by closing the `ResultSet` and then reopening it.

If you specify `TYPE_FORWARD_ONLY`, you will get a nonscrollable result set, where the cursor moves forward only. If you also specify `CONCUR_READ_ONLY` for the second argument, you will get the default `ResultSet` identical to the `ResultSet` created with the no argument variant.

The second argument must be one of the two following `ResultSet` constants for specifying whether a `ResultSet` is read-only or updateable:

- `CONCUR_READ_ONLY`
- `CONCUR_UPDATABLE`.

Note

If you specify a `ResultSet` type, you must also specify whether the `ResultSet` is read-only or updateable.

You can check the type of `ResultSet` you have using the `ResultSet.getType()` method, as shown here:

```
if(rs.getType()==ResultSet.TYPE_FORWARD_ONLY)
 System.out.println("FORWARD_ONLY");
else
 System.out.println("SCROLLABLE");
```

Moving the Cursor in a Scrollable ResultSet

Once you have a scrollable `ResultSet` object, you can move the cursor both backward and forward in the `ResultSet` by using these methods:

- `ResultSet.next()`, which moves the cursor forwards to the next row
- `ResultSet.previous()`, which moves the cursor back one row

Both methods return `false` when the cursor goes beyond the result set, so you can easily use these methods in a `while` loop.

In addition to using the `next()` and `previous()` methods to scroll forward and backward, you can move the cursor to a designated row using these methods:

- `first()`, which moves the cursor to the first row
- `last()`, which moves the cursor to the last row
- `beforeFirst()`, which moves the cursor to a point just before the first row
- `afterLast()`, which moves the cursor to a point just after the last row
- `absolute(int rowNumber)`, which moves the cursor to the specified row
- `relative(int rowNumber)`, which moves the cursor the specified number of rows

The method `absolute(int rowNumber)` moves the cursor to the row number indicated in the argument. If the number is positive, the cursor moves to the given row number from the beginning. If the number is negative, the cursor moves to the given row number from the end, so `absolute(1)` moves the cursor to the first row, and `absolute(-1)` moves it to the last row.

The method `relative(int rowNumber)` lets you specify how many rows to move from the current row and in which direction to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows. The effect of the first four of these is apparent from the method names.

Note

As with the default `ResultSet` that is not scrollable, a scrollable `ResultSet's` cursor is initially positioned before the first row.

Using Scrollable ResultSets to Create a Search Page

In the course of the last couple of chapters, you have learned how to use servlets and JSP pages to handle HTML forms and to save form data, images, and documents to a database. This chapter concentrates on retrieving data from the database and presenting it as a Web page.

Chapter 13 illustrates how to create and handle HTML forms using JSP pages. The chapter goes on to develop examples showing how to handle a simple registration form. This chapter extends these concepts to a database driven web site which members can use to buy and sell vehicles.

The Web site features a search capability, allowing members to enter search criteria and scroll through a formatted `ResultSet`. Clicking a selection takes the user to a detail page displaying more information about an item in the database

The starting point for a search is another form in which the user sets up his or her search criteria. A simple search form is illustrated in Figure 15-1. The search criteria this form defines are collected using a JSP page and a JavaBean. The search criteria are passed as inputs to a SQL stored procedure of the form shown in Listing 15-1.

Figure 15-1: Search form

Listing 15-1: SQL stored procedure to return matching database items

```
CREATE PROCEDURE SEARCH @BODY VARCHAR(50),
@ZIP VARCHAR(10), @MAKE VARCHAR(50),
@MODEL VARCHAR(50), @ENGINE VARCHAR(50),
@TRANSMISSION VARCHAR(50), @PRICE INT, @YEAR1 INT,
@YEAR2 INT AS SELECT TOP 50 *
             FROM VEHICLES
             WHERE BODY LIKE @BODY AND
                  ZIP LIKE @ZIP AND MAKE LIKE @MAKE AND
                  MODEL LIKE @MODEL AND
                  ENGINE LIKE @ENGINE AND
                  TRANSMISSION LIKE @TRANSMISSION AND
                  PRICE <= @PRICE AND YEAR BETWEEN
                  @YEAR1 AND @YEAR2;
```

The stored procedure uses the `LIKE` comparator so that wild cards can be used. This approach allows a great deal of flexibility in searching the database. The HTML snippet below shows how the `SELECT` element is defined to return the wild card character '%' when the `OPTION` Any is selected:

```
<TR>
  <TD>Make: </TD>
  <TD>
    <SELECT name=Make size=1>
      <OPTION VALUE="%" SELECTED>Any</OPTION>
```

```
      <OPTION VALUE="Acura">Acura</OPTION>
      <OPTION VALUE="Audi">Audi</OPTION>
      <OPTION VALUE="BMW">BMW</OPTION>
    </SELECT>
  </TD>
</TR>
```

Notice also the use of the `TOP 50` clause in the stored procedure of Listing 15-1. The `TOP 50` clause is used to limit the number of hits the search returns. If a large database is searched for common criteria, you will get a lot of hits, which means big `ResultSets` using lots of memory. In all probability, your users won't scroll through more than a few pages before tightening up the search criteria, so giving them a huge `ResultSet` is a waste of resources.

If you return the `ResultSet` to a JavaBean, and make it scrollable, it will be easy to create pages of, say, five database items per page that the user can scroll. You can always offer the user the option of requesting additional blocks of 50 results based on the original search criteria by ordering the search on the primary key and specifying that subsequent `ResultSets` have higher primary key values.

The search form shown in Figure 15-1 calls a simple JSP page, `ProcessSearchForm.jsp`, which uses a JavaBean to handle the query and return the results. As you can see from Listing 15-2, the JSP page is very simple. It loads the bean, set its properties, and calls the `SearchFormBean.getMatches()` method, which executes the query. It then forwards the user to `SearchFormResultsPage.jsp`, which displays the search results.

Listing 15-2: JSP page that loads a JavaBean to query the database

```
<%@ page language="java"%>
<jsp:useBean id="SearchFormBean"
class="JavaDatabaseBible.ch15.SearchFormBean" scope="session"/>
<jsp:setProperty name="SearchFormBean" property="*"/>
<%SearchFormBean.getMatches();%>
<jsp:forward page="SearchFormResultsPage.jsp"/>
```

The `SearchFormBean` itself is shown in Listing 15-3.

Listing 15-3: JavaBean to handle database query from a JSP page

```
package JavaDatabaseBible.ch15;

import java.sql.*;
import javax.sql.*;

public class SearchFormBean extends java.lang.Object{
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";


  protected int price;
  protected int year;
```

```
  protected String id;
  protected String make;
  protected String model;
  protected String color;
  protected String body;
  protected String engine;
  protected String transmission;
  protected String zip;

  protected int index = 0;
  protected int pageSize = 5;
  protected int rowCount = 0;
 protected ResultSet rs = null;

  public SearchFormBean(){
  }
  public void setYear(int year){
    this.year = year;
  }
  public void setMake(String make){
    this.make = make;
  }
  public void setZip(String zip){
    this.zip = zip;
  }
  public void setModel(String model){
    this.model = model;
  }
  public void setColor(String color){
    this.color = color;
  }
  public void setBody(String body){
    this.body = body;
  }
  public void setEngine(String engine){
    this.engine = engine;
  }
  public void setTransmission(String transmission){
    this.transmission = transmission;
  }
  public void setPrice(int price){
    this.price = price;
  }
```

```
public String getId(){
  return id;
}
public String getMake(){
  return make;
}
public String getZip(){
  return zip;
}
public String getModel(){
  return model;
}
public String getColor(){
  return color;
}
public String getBody(){
  return body;
}
public String getEngine(){
  return engine;
}
public String getTransmission(){
  return transmission;
}
public int getPrice(){
  return price;
}
public int getYear(){
  return year;
}
public int getIndex(){
  return index;
}
public int getRowCount(){
  return rowCount;
}
public String getPage(){
  return ""+(index/pageSize+1)+" of "+(rowCount/pageSize+1);
}
public boolean pageForward(){
  boolean validRow = false;
  if(index<0||index+pageSize>rowCount){
    index=0;
```

```
    }else{
      index += pageSize;
    }
    try {
      validRow = rs.absolute(index+1);
    }catch(SQLException e){
      System.err.println(e.getMessage());
    }
    return validRow;
  }
  public boolean pageBack(){
    boolean validRow = false;
    if(index<pageSize){
      index=rowCount/pageSize*pageSize;
    }else if(index>=pageSize){
      index -= pageSize;
    }
    try {
      validRow = rs.absolute(index);
    }catch(SQLException e){
      System.err.println(e.getMessage());
    }
    return validRow;
  }
  public boolean selectRow(int row){
    boolean validRow = false;
    try {
      validRow = rs.absolute(index+1);
      if(validRow){
        if(row > 0)validRow = rs.relative(row);
        if(rs.getRow()<0)validRow=false;

        if(validRow){
          id = rs.getString("ID");
          year = rs.getInt("year");
          make = rs.getString("make");
          zip = rs.getString("zip");
          model = rs.getString("model");
          body = rs.getString("body");
          engine = rs.getString("engine");
          transmission = rs.getString("transmission");
          price = rs.getInt("price");
        }
```

```
      }
    }catch(SQLException e){
      System.err.println(e.getMessage());
    }
    return validRow;
  }
  /*
  getMatches uses the stored procedure SEARCH:

  CREATE PROCEDURE SEARCH @BODY VARCHAR(50),
  @ZIP VARCHAR(10), @MAKE VARCHAR(50),
  @MODEL VARCHAR(50), @ENGINE VARCHAR(50),
  @TRANSMISSION VARCHAR(50), @PRICE INT, @YEAR1 INT,
  @YEAR2 INT AS SELECT TOP 50 *
             FROM VEHICLES
             WHERE BODY LIKE @BODY AND
                  ZIP LIKE @ZIP AND MAKE LIKE @MAKE AND
                  MODEL LIKE @MODEL AND
                  ENGINE LIKE @ENGINE AND
                  TRANSMISSION LIKE @TRANSMISSION AND
                  PRICE <= @PRICE AND YEAR >= @YEAR;
  */
  public int getMatches(){
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "VEHICLES" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      // clean up parameters from free format text fields for
CallableStatement
      if(model==null)model="%";
      if(zip==null)zip="%";

      CallableStatement cs = con.prepareCall("{call
SEARCH(?,?,?,?,?,?,?,?)}");
      cs.setString(1,body);
      cs.setString(2,zip);
```

```
      cs.setString(3,make);
      cs.setString(4,model);
      cs.setString(5,engine);
      cs.setString(6,transmission);
      cs.setInt   (7,price);
      cs.setInt   (8,year);

      rs = cs.executeQuery();
      rs.last();
      rowCount = rs.getRow();
      con.close();
    }catch(ClassNotFoundException e1){
      System.err.println(e1.getMessage());
    }catch(SQLException e2){
      System.err.println(e2.getMessage());
    }
    return rowCount;
  }
}
```

In addition to the normal getter and setter methods for its properties, the JavaBean has a number of methods to retrieve the data and to navigate around the scrollable `ResultSet`. These methods include:

- `getMatches()`. This method sets the parameters of the SQL stored procedure and calls it to get the scrollable `ResultSet`. It then gets the row count by navigating to the end of the `ResultSet` and getting the row number using `getRow()`. The `ResultSet` is now stored in the JavaBean for access by the other methods.
- `selectRow(int row)`. The `selectRow()` method moves the cursor to the selected row. The `row` argument refers to the row number within the displayed JSP page, and the integer `index` provides the offset to the row corresponding to the beginning of the current JSP page. The `selectRow()` method navigates to the desired row using `ResultSet.absolute(index)` to move the cursor to the row corresponding to the first row in the displayed JSP page. It then uses `ResultSet.relative(row)` to move to the row corresponding to the relative row within the displayed JSP page. Finally, it sets the JavaBean's properties by getting the appropriate data from the `ResultSet`.
- `pageForward()`. The `pageForward()` method moves the row index to the row corresponding to the top of the next page.
- `pageBack()`. The `pageBack()` method moves the row index to the row corresponding to the top of the previous page.
- `getRowCount()`. The `getRowCount()` method returns the `rowCount`.
- `getPage()`. The `getPage()` method returns a String representation of the current page number in the form "Page 1 of n".

The only slightly tricky logic involved in Listing 15-3 is in the area of moving the cursor. It is important to remember that absolute row numbers start at 1 and that relative row numbers can never be zero.

The JSP page used to display the search results is kept separate from the JSP page that instantiates the bean and executes the query. It includes two HTML form elements containing the page buttons and calls separate JSP pages to handle navigation through the `ResultSet`. The code for the display page is shown in Listing 15-4.

Listing 15-4: Search-results page JSP

```
<%@ page language="java"%>
<jsp:useBean id="SearchFormBean"
class="JavaDatabaseBible.ch15.SearchFormBean" scope="session"/>
<html>
<head>
<title>Summary</title>
</head>
<body bgcolor="#ffffff">
<BASEFONT FACE="Arial">

<TABLE BORDER="2">
    <TR BGCOLOR="#E0E0E0">
    <TD COLSPAN="2">

    <!-- header -->
        <TABLE WIDTH=100%>
        <TR BGCOLOR="#E0E0E0">
        <TD>
        Found <%=SearchFormBean.getRowCount()%> vehicles matching query.
        </TD>
        <TD ALIGN="RIGHT">
        Page <%=SearchFormBean.getPage()%>
        </TD>
        </TR>
        </TABLE>
    </TD>
    </TR>

<!-- results -->

<%
if(SearchFormBean.getRowCount()>0){
    for(int i=0;i<3;i++){
        if(SearchFormBean.selectRow(i)){
    %>
        <TR>
        <TD>
     <A HREF="GetDetailPage.jsp?memberId=<%=SearchFormBean.getId()%>">
     <img src="http://192.168.0.2/servlet/LobServlet?type=blob
          &id=<%=(i+5230001)%>&description=Thumbnail">
     </A>
        </TD>
        <TD>
```

```
        <TABLE CELLPADDING = 4 width = 100%>
        <TR>
        <TD>
        <%=SearchFormBean.getYear()%> <%=SearchFormBean.getMake()%>
        <%=SearchFormBean.getModel()%>, <%=SearchFormBean.getBody()%>.
        <%=SearchFormBean.getEngine()%>,
<%=SearchFormBean.getTransmission()%>.
        </TD>
        </TR>
        <TR>
        <TD>
        Asking $<%=SearchFormBean.getPrice()%>.
        Location ( zip code ): <%=SearchFormBean.getZip()%>.
        </TD>
        </TR>
        </TABLE>

        </TD>
        </TR>
        <%
        }
    }
}
%>

<!-- footer -->
<TR BGCOLOR="#E0E0E0">
<TD COLSPAN="2">

    <TABLE WIDTH=100%>
    <TR BGCOLOR="#E0E0E0">
    <TD WIDTH="60%">
    </TD>
    <TD>
    <form METHOD="POST" ACTION="SearchFormPageBack.jsp" target= "_self">
    <input type="submit" value="Prev Page"></td>
    </form>
    </TD>
    <TD>
    <form METHOD="POST" ACTION="SearchFormPageForward.jsp" target= "_self">
    <input type="submit" value="Next Page"></td>
    </form>
```

```
    </TD>
    <TD ALIGN="RIGHT">
    </TD>
    </TR>
    </TABLE>

</TD>
</TR>
</TABLE>

</TABLE>
</body>
</html>
```

The JSP pages that support the Prev Page and Next Page buttons are as simple as the basic `ProcessSearchForm` JSP page. `SearchFormPageForward.jsp` calls the bean's `PageForward()` method to increment the page index variable by an amount equal to the page size. Here's an example:

```
<%@ page language="java"%>
<jsp:useBean id="SearchFormBean"
class="JavaDatabaseBible.ch15.SearchFormBean" scope="session"/>
<%=SearchFormBean.pageForward()%>
<jsp:forward page="SearchFormResultsPage.jsp"/>
```

`SearchFormPageBack.jsp` calls the `SearchFormBean.pageBack()` method to decrement the index by the page size. Both methods wrap the index to handle transitions through the beginning and end of the `ResultSet` as shown here:

```
<%@ page language="java"%>
<jsp:useBean id="SearchFormBean"
class="JavaDatabaseBible.ch15.SearchFormBean" scope="session"/>
<%=SearchFormBean.pageBack()%>
<jsp:forward page="SearchFormResultsPage.jsp"/>
```

The page that `SearchFormResultsPage.jsp` creates is shown in Figure 15-2.

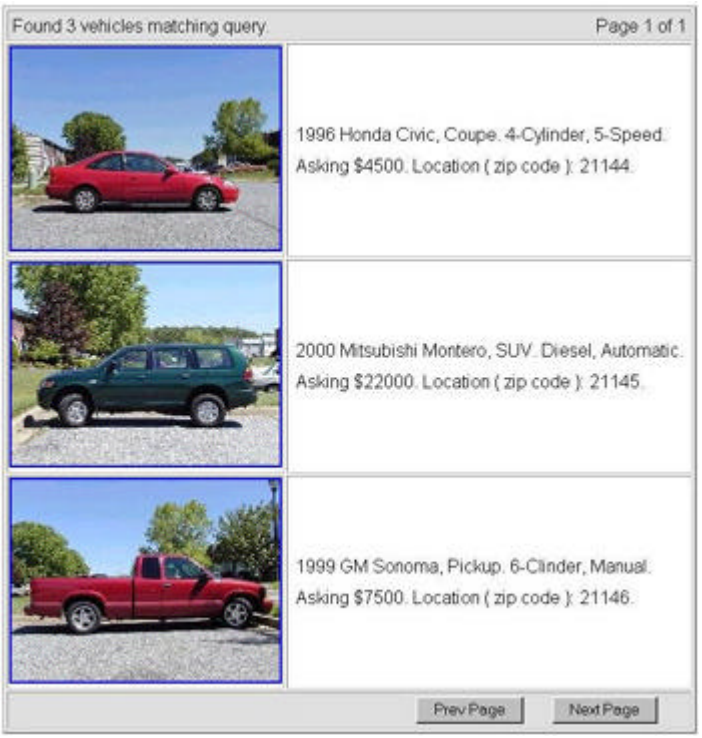


Figure 15-2: Search-results page

As you may have noticed in Listing 15-4, the images are obtained from a servlet that returns Blobs from the Photos Table in the database. This servlet is derived from the Large Object servlet example of Chapter 14, Listing 14-8.

If you refer to Listing 15-4, you will notice that the images can be clicked to access a detail page for the vehicle. The next section discusses the creation of both the detail page and an edit page from the same XML formatted result set using an XSL transform

Using XSL to Create a Web Page from a SQL Query

The previous example shows how to create a Web page using JSP to manage its formatting. This approach has the disadvantage that you have to understand how to use Java Server Pages to modify your display format. Another way to manage database-driven Web page formatting is to use an XSL stylesheet to transform XML data into HTML. A simple change to the XSLT transforms the same page of XML into a completely different HTML page.

How XSLT Works

Extensible Stylesheet Language (XSL) provides the user with a means of transforming XML documents from one form to another. In practice, this means you can retrieve information from a database as basic, content-oriented XML and use an XSL stylesheet to convert it into a human-readable document.

XSL actually combines two major components: a transformation language and a formatting language. Each of these is an XML dialect. The transformation language, XSLT, is used to define rules for the conversion of one XML document into another, and the formatting component deals with formatting the output.

From the viewpoint of generating HTML, the important component is the XSL Transformation Language (XSLT). To perform a transformation, an XSLT processor reads both an XML document and an XSLT stylesheet and outputs a new XML document.

Applying an XSL transform in Java is very simple, as you can see from the example in Listing 15-5. The xalan library methods do most of the work.

Listing 15-5: Applying an XSL transform

```
import org.xml.sax.SAXException;
import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTResultTarget;
import org.apache.xalan.xslt.XSLTProcessor;

/**
 * Sample code to apply a stylesheet to an xml document to create an
HTML page.
*/

public class SimpleXSLTransform
{
  public static void main(String[] args)throws org.xml.sax.SAXException
  {
    XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

    processor.process(new XSLTInputSource("MemberInfo.xml"),
                      new XSLTInputSource("MemberInfo.xsl"),
                      new XSLTResultTarget("MemberInfo.html"));
  }
}
```

XSL stylesheets can be used either on the server side or on the client side. In practice, server-side transforms work better; different browsers implement different subsets of the specification, so results are unpredictable. The main drawback of using XSL transforms on the server is that they tend to be resource intensive. It is worth experimenting with different XSL transform libraries, as some perform transforms very much faster than others.

Retrieving Data from a Database as an XML Document

Of course, before you can transform it, you need to get your `ResultSet` and turn it into XML. The data set used to create the search-results page of Figure 15-2 is derived from a single table. To create a more detailed page, information must be combined from several different tables.

The tables accessed for the detail page include the Vehicles table, which holds the basic information about each vehicle, and the Options table, which contains information about accessories and options. As discussed in Chapter 11, these tables are set up so that they correlate well to the forms used for adding vehicles to the database, as well as being more convenient for searches.

A significant aspect of the way the Options table is designed is that it contains a number of columns representing check box selections with YES/NO values, as well as a single text entry labeled "Other" on the HTML form. When the form data is saved to the table, the data from all of these HTML form inputs is combined into a column labeled LIST. Designing the table this way makes it easy to search for specific YES/NO attributes without incurring extra overhead creating a text summary of the attributes.

Cross-Reference The design and layout of the member database used in the Web-applications part of this book is discussed in Chapter 11.

As shown in the SQL snippet of Listing 15-6, the `ResultSet` for the detail page is based on the LIST columns from the attribute tables. These columns are combined with information from the individual columns of the Vehicle table. Listing 15-6 shows the SQL code required to create the stored procedure `GET_DEATIL_PAGE`, which the JavaBean used to create the XML document will call.

Listing 15-6: Stored procedure for detail page

```
CREATE PROCEDURE GET_DETAIL_PAGE @id int
AS SELECT v.*, o.list OPTIONS
   FROM Vehicles v, Options o
   WHERE o.id = v.id AND
         v.id = @id;
```

Listing 15-7 shows a JavaBean that calls the stored procedure of Listing 15-6 and formats the `ResultSet` as XML. A `ResultSetMetaData` object is used to get the column names that are used as tag names for the XML elements. The column data Strings are appended to the XML elements as text nodes.

Listing 15-7: JavaBean that returns a ResultSet as XML

```
package JavaDatabaseBible.ch15;

import java.io.*;
import java.sql.*;
import javax.sql.*;

public class DetailPageXMLBean{
  protected static String dbUserName = "sa";
  protected static String dbPassword = "dba";
  protected String xmlHeader = "<?xml version=\"1.0\"?>";

  protected int id;

  public DetailPageXMLBean(){
  }
  public void setId(int id){
    this.id=id;
  }
  public String getXmlString(){
    String xml = new String(getVehicleData());
    return xml.trim();
  }
  public byte[] getVehicleData(){
    String rootTag = "VehicleData";
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    try {
```

```
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "VEHICLES" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);

      Statement stmt = con.createStatement();

      CallableStatement cs = con.prepareCall("{call GET_DETAIL_PAGE
(?)}");

      cs.setInt(1,id);
      ResultSet rs = cs.executeQuery();
      ResultSetMetaData md = rs.getMetaData();

      os.write(xmlHeader.getBytes());
      os.write(("<"+rootTag+">").getBytes());

      String xml = "";
      int columns = md.getColumnCount();
      rs.next();
      for(int i=1;i<=columns;i++){
        if(md.getColumnType(i)==Types.VARCHAR){
          xml="<"+md.getColumnLabel(i)+">"+
               rs.getString(i)+
              "</"+md.getColumnLabel(i)+">";
          os.write(xml.getBytes());
        }else if(md.getColumnType(i)==Types.INTEGER){
          xml="<"+md.getColumnLabel(i)+">"+
               rs.getInt(i)+
              "</"+md.getColumnLabel(i)+">";
          os.write(xml.getBytes());
        }
      }
      os.write(("</"+rootTag+">").getBytes());
    }catch(Exception e){
      e.printStackTrace();
    }
    return os.toByteArray();
```

```
  }
  public static void main(String args[]){
    File f = new File("Detail.xml");
    int id = 1000;
    DetailPageXMLBean xmlBean = new DetailPageXMLBean();
    xmlBean.setId(id);
    try {
      FileOutputStream fos = new FileOutputStream(f);
      fos.write(xmlBean.getVehicleData());
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Although this is a simple approach to generating XML, it saves the overhead of building a DOM object and serializing it. The `getXmlString()` method is simply a convenience for use in the simple JSP page shown in Listing 15-8. Similarly, the `main` method is included to let you check out the XML by dumping it to a file.

Cross-Reference The advantages of the DOM-based approach to handling XML are discussed in Part IV, which deals with XML and Java databases.

The resulting XML can be displayed in a browser using the simple JSP page shown in Listing 15-8. Notice the use of the `contentType` attribute in the `<% @ page %>` directive. This attribute is required so that the browser can recognize the data as XML and display it accordingly.

Listing 15-8: JSP page using a JavaBean to display a ResultSet as XML

```
<%@ page language="java" contentType="text/xml"%>
<jsp:useBean id="DetailPageXMLBean"
  class="JavaDatabaseBible.ch15.DetailPageXMLBean" scope="session"/>
<jsp:setProperty name="DetailPageXMLBean" property="*"/>
<%=DetailPageXMLBean.getXmlString()%>
```

The resulting XML is shown in Listing 15-9. Although the structure shown here is very simple, with no nested elements, everything discussed in the examples applies equally to more complicated XML documents.

Listing 15-9: ResultSet formatted as XML

```
<?xml version="1.0"?>
    <VehicleData>
        <ID>1000</ID>
        <Make>Honda</Make>
        <Body>Coupe</Body>
        <Model>Civic</Model>
        <Year>1996</Year>
        <Color>Red</Color>
```

```
        <Zip>21144</Zip>
        <Engine>4-Cylinder</Engine>
        <Transmission>5-Speed</Transmission>
        <Price>4500</Price>
        <OPTIONS>
            AM_FM_Radio,
            Cassette,
            Power Windows,
            Power Locks,
            Air Conditioning,
            Tilt Steering,
            Power Steering,
            ABS,
            Moon Roof,
            Bucket Seats
            </OPTIONS>
</VehicleData>
```

Transforming the XML Using an XSL Stylesheet

Stylesheets are valid XML documents that contain a set of XSL commands used to transform a document. XSL stylesheets start with the `xsl:stylesheet` declaration, which forms the root node of the XML document. The stylesheet declaration consists of a version and namespace. The namespace declares the stylesheet tag prefix and the URL of tag definitions, as shown here:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
</xsl:stylesheet>
```

The namespace prefix `xsl:` is used in the body of the XSL document to identify XSL processing statements. Tags that are not prefixed with `xsl:` are simply copied to the output without being processed, so you can embed HTML tags in the XSL stylesheet, and they will be sent to the output stream unchanged.

XSLT is unlike conventional programming languages such as Java because XSLT is a rule-based, declarative language. XSL rules define templates that specify how an XML document should be processed. These template rules can be defined in any order.

XSL templates are used to select XML elements for processing using the match operator. Here's a typical example of the use of the `xsl:template` tag:

```
<xsl:template match="VehicleData">
    ...
</xsl:template>
```

The argument of the match operator is defined using an XPath expression. XPath simply defines the path to a child node in much the same way as a file path defines a path to a file. For example, to select an entire document for processing, you can match the root node, using match="/". Alternatively, you can match the document element tag, in this case 'VehicleData'.

Caution

The difference between matching the root node, defined with "/", and matching the document node, defined in this case with "`VehicleData`", is that the XPath to a child node is different in each case. For example, if you match the root node, using

<xsl:template match="/">, the XPath from the root node to the ID node in Listing 15-8 is "MemberInfo/ID". On the other hand, if you use <xsl:template match="MemberInfo"> to match the document node, the XPath is simply "ID".

The only other XSL used in this example is the xsl:value-of expression. This expression returns the value of the node selected using the XPath expression defined in the select attribute. For example, to get the vehicle's color, use the following xsl:value-of expression:

```
<xsl:value-of select="Color"/>
```

This returns the value "Red" from the corresponding node in the XML document of Listing 15-8:

```
<Color>Red</Color>
```

In addition to selecting values from XML documents, XSLT allows you to use your XML data in calculations or to create and manipulate strings. An example of string manipulation to create an image URL is shown in the following code snippet:

```
<xsl:variable name="imageUrl"
    select="string('http://192.168.0.2/servlet/BlobServlet?id=')"/>
<xsl:variable name="id" select="ID"/>
<img>
    <xsl:attribute name="src">
        <xsl:value-of select="concat($imageUrl,$id)"/>
    </xsl:attribute>
</img>
```

This code snippet illustrates how you can define a String variable and concatenate the value of an XML element to the string to create a URL. This URL is then set as the value of the src attribute of an HTML img tag. The complete stylesheet is shown in Listing 15-9. Note how the stylesheet freely combines XSL and HTML tags as required.

Listing 15-9: XSL stylesheet

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="html"/>
<xsl:preserve-space elements="*"/>
<xsl:template match="VehicleData">

<HTML>
<HEAD>
<TITLE>Detail Page</TITLE>
<BASEFONT FACE="Arial"/>
</HEAD>
<BODY>
<P/>
<TABLE BORDER="1" WIDTH="480" CELLPADDING="4">
<TR>
<TD ALIGN="CENTER" VALIGN="TOP">
<xsl:variable name="imageUrl"
```

```
select="string('http://192.168.0.2/servlet/BlobServlet?id=')"/>
<xsl:variable name="id" select="ID" />
<img>
    <xsl:attribute name="src">
        <xsl:value-of select="concat($imageUrl,$id)"/>
    </xsl:attribute>
</img>
</TD>
<TD>
<xsl:value-of select="Color"/>
<xsl:text> </xsl:text>
<xsl:value-of select="Year"/>
<xsl:text> </xsl:text>
<xsl:value-of select="Make"/>
<xsl:text> </xsl:text>
<xsl:value-of select="Model"/>.
<P/>
<xsl:value-of select="Engine"/>, <xsl:value-of select="Transmission"/>
Transmission.
<P/>
<xsl:value-of select="OPTIONS"/>
<P/>
$<xsl:value-of select="Price"/>.
<P/>
Vehicle is located in Zip code: <xsl:value-of select="Zip"/>.
</TD>
</TR>
<TR>
<TD COLSPAN="2">
<FORM method="post" action="/jsp/ProcessMessageForm.jsp" target="_self"
id="form1" name="form1">
<INPUT type="hidden">
<xsl:variable name="id" select="ID" />
<xsl:attribute name="memberId">
    <xsl:value-of select="$id"/>
</xsl:attribute>
</INPUT>
<TABLE BORDER="1">
<TR>
<TD>
<FONT COLOR="blue">
<EM>Contact the seller:</EM>
</FONT>
```

```
</TD>
</TR>
<TR>
<TD>For more information, or to arrange to see the vehicle, send a
message to the seller:</TD>
</TR>
<TR>
<TD ALIGN="CENTER">
<textarea name="Message" cols="48" rows="4" />
</TD>
</TR>
<TR>
<TD ALIGN="CENTER">
<input type="submit" value="Click here to send" />
</TD>
</TR>
</TABLE>
</FORM>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Applying an XSL Transform in a JSP Page

To apply the XSL stylesheet on the server side, you need to create a JSP page. Listing 15-10 illustrates the use of a second JavaBean to transform the XML document produced by the JavaBean of Listing 15-7. The only property the JSP page expects is the member id.

Listing 15-10: Applying an XSL stylesheet in a JSP page

```
<%@ page language="java"%>
<%@ page language="java"%>
<jsp:useBean id="DetailPageXMLBean"
     class="JavaDatabaseBible.ch15.DetailPageXMLBean"/>
<jsp:useBean id="DetailPageTransformBean"
     class="JavaDatabaseBible.ch15.DetailPageTransformBean"/>
<jsp:setProperty name="DetailPageXMLBean" property="*"/>
<%
DetailPageTransformBean.setXslFileName("DetailPage.xsl");
%>
<%=new String(
```

```
DetailPageTransformBean.applyTransform(DetailPageXMLBean.getVehicleData
()))%>
```

The JavaBean required to apply the XSL transform is shown in Listing 15-11. For ease of checkout, a main method is included.

Caution

The only deployment problem you are likely to encounter is that the path for the stylesheet defaults to Tomcat's/bin directory unless you specify the path fully.

Listing 15-11: XSL transform bean

```
package JavaDatabaseBible.ch15;

import java.io.*;
import org.xml.sax.SAXException;
import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTResultTarget;
import org.apache.xalan.xslt.XSLTProcessor;

public class DetailPageTransformBean{
  private String xslFileName = null;
  private byte[] xmlSource = null;
  private ByteArrayInputStream xmlInputStream = null;

  public DetailPageTransformBean(){
  }

  public void setXmlSource(byte[] xmlSource){
    this.xmlSource=xmlSource;
    xmlInputStream = new ByteArrayInputStream(xmlSource);
  }

  public void setXslFileName(String xslFileName){
    this.xslFileName=xslFileName;
    File f = new File(xslFileName);
    if(!f.exists())System.out.println("Cannot find file: "+xslFileName);
  }

 public byte[] applyTransform(byte[] xmlSource){
   setXmlSource(xmlSource);
   return applyTransform();
 }

 public byte[] applyTransform(){
```

```
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

    try{
      XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

      processor.process(new XSLTInputSource(xmlInputStream),
                        new XSLTInputSource(xslFileName),
                        new XSLTResultTarget(outputStream));
    }catch(Exception e){
      System.err.println(e);
    }
    return outputStream.toByteArray();
 }

  public static void main(String args[]){
    File f = new File("Detail.html");
    int id = 1000;
    DetailPageXMLBean xmlBean = new DetailPageXMLBean();
    DetailPageTransformBean transformBean = new
DetailPageTransformBean();
    xmlBean.setId(id);
    transformBean.setXslFileName("DetailPage.xsl");
    try {
      FileOutputStream fos = new FileOutputStream(f);
      fos.write(transformBean.applyTransform(xmlBean.getVehicleData()));
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Assuming you have deployed everything correctly, you should see a Web page that looks like Figure 15-3 when you call the JSP page. The simplest way to do this for checkout purposes is with a simple HTML form, like the following:

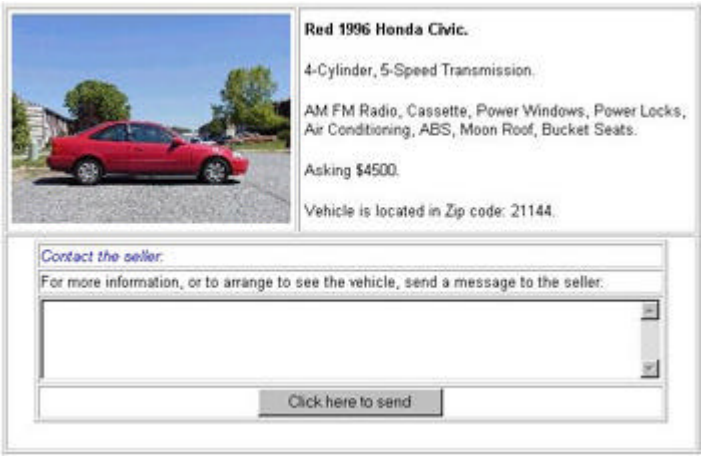


Figure 15-3: Web page created by applying an XSL transform to an XML document built from a ResultSet

```
<html>

<head><title>Get Web Page</title></head>

<body>

<form METHOD="POST" ACTION="GetDetailPage.jsp" target= "_self">

    <table><tr><td align="center" colspan="3">

        <input type="text" name="id">

    </td></tr>

    <tr><td align="center" colspan="3">

        <input type="submit" value="Show Web Page">

    </td></tr></table>

</form>

</body>

</html>
```

To call up the detail page in a practical application, make a simple modification to the JSP page of Listing 15-4, so that when the user clicks the thumbnail image in the search form, he or she is forwarded to the detail page.

To forward the user to the detail page in response to a mouse click on the thumbnail image, all you need to do is wrap the thumbnail image in an HTML anchor element as shown here:

```
    <TR>
    <TD>
    <A HREF="GetDetailPage.jsp?memberId=<%=SearchFormBean.getId()%>">
    <img src="http://192.168.0.2/servlet/LobServlet?type=
              blob&id=<%=(i+1)%>&description=Thumbnail">
    </A>
    </TD>
```

Note the use of the <%=SearchFormBean.getId()%> tag to supply the member id to the JSP page.

Using an Updatable ResultSet with an XSL Stylesheet

In addition to using the ResultSet for display purposes, it is very useful to be able to update the database using the ResultSet itself. Updatable ResultSets offer just this capability, in that they can be updated directly. In other words, you can make updates to the values in the ResultSet itself, and these changes are reflected in the database.

The XML-based and XSLT-based approach to creating a Web page lends itself well to use with updatable ResultSets. One of the advantages of XS L is that you can create a completely different Web page from the same XML by simply applying a different stylesheet. To illustrate this capability, try applying the stylesheet shown in Listing 15-12 to the original XML of Listing 15-9.

Listing 15-12: Creating a different Web page from the same XML

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">

<xsl:output method="html"/>

<xsl:template match="VehicleData">
<HTML>
<HEAD>
 <TITLE>
 Edit Detail Page
 </TITLE>
</HEAD>
<BASEFONT FACE="Arial"/>
<BODY>
<FORM method="post" action="ProcessVehicleUpdateForm.jsp">

<TABLE BORDER="1" CELLPADDING="4">
<TR>
<TD>Color</TD>
<TD>
<INPUT type="text" name="color">
<xsl:attribute name="value">
    <xsl:value-of select="Color"/>
</xsl:attribute>
</INPUT>
</TD>
</TR>
<TR>
<TD>Year</TD>
<TD>
```

```
<INPUT type="text" name="year">
<xsl:attribute name="value">
    <xsl:value-of select="Year"/>
</xsl:attribute>
</INPUT>
</TD>
</TR>
<TR>
<TD>Make</TD>
<TD>
<INPUT type="text" name="make">
<xsl:attribute name="value">
    <xsl:value-of select="Make"/>
</xsl:attribute>
</INPUT>
</TD>
</TR>
<TR>
<TD>Model</TD>
<TD>
<INPUT type="text" name="model">
<xsl:attribute name="value">
    <xsl:value-of select="Model"/>
</xsl:attribute>
</INPUT>
</TD>
</TR>
<TR>
<TD COLSPAN="2">
<INPUT type="submit" value="CLICK HERE TO SUBMIT CHANGES"/>
</TD>
</TR>
</TABLE>

</FORM>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

This stylesheet generates the form shown in Figure 15-4. It uses the same `<xsl:value-of>` tags to get the data from the XML file, but this time it wraps them in an HTML form, using the `<xsl:attribute>` tag, so that rather than just displaying the vehicle data, the stylesheet uses the data to preload a form.

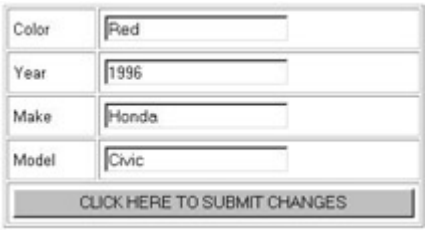


Figure 15-4: Form generated from the XML of Listing 15-9 using the stylesheet of listing 15-12

Caution

Bean properties are case sensitive. Use lowercase for the property names.

Note that this example only displays a small number of elements from the underlying XML. Obviously, you can create a single large form to display the entire document for editing, or you can create a series of smaller forms like the example in Figure 15-4, processing them sequentially.

The JSP page required to handle the update is shown in Listing 15-13. It simply passes the attributes picked up from the form to the `UpdateXMLBean` and calls the bean's `updateVehicleData()` method. On completion, the user is forwarded to the detail Web page to view the results of the change.

Listing 15-13: JSP to process the database update form

```
<%@ page language="java" contentType="text/html"%>
<jsp:useBean id="UpdateXMLBean"
class="JavaDatabaseBible.ch15.UpdateXMLBean" scope="session"/>
<jsp:setProperty name="UpdateXMLBean" property="*"/>
<%UpdateXMLBean.updateVehicleData();%>
<%
String id = UpdateXMLBean.getVehicleId();
String nextPage = "GetDetailPage.jsp?DetailId="+id;
%>
<jsp:forward page="<%=nextPage%>"/>
```

The JavaBean that updates the vehicle data is shown in Listing 15-14. This JavaBean is similar to the code of Listing 15-7, with the exception that it creates an updatable ResultSet and includes a method to perform the update.

Listing 15-14: Updatable ResultSet bean

```
package JavaDatabaseBible.ch15;

import java.io.*;
import java.sql.*;
import javax.sql.*;

public class MemberUpdateXMLBean{
  protected static String dbUserName = "sa";
  protected static String dbPassword = "dba";
  protected String xmlHeader = "<?xml version=\"1.0\"?>";
```

```
protected String memberId;
protected String eyecolor;
protected String haircolor;
protected String build;
protected String height;

protected Connection con;
protected Statement stmt;
protected ResultSet rs;
protected ResultSetMetaData md;

public MemberUpdateXMLBean(){
}
public void setMemberId(String memberId){
  this.memberId=memberId;
}
public String getMemberId(){
  return memberId;
}
public void setEyecolor(String eyecolor){
  this.eyecolor=eyecolor;
}
public void setHaircolor(String haircolor){
  this.haircolor=haircolor;
}
public void setBuild(String build){
  this.build=build;
}
public void setHeight(String height){
  this.height=height;
}
public String getMemberXmlString(){
  String xml = new String(getMemberData());
  return xml.trim();
}
public String updateMemberData(){
  String status = "Update successful";
  System.out.println("ResultSet = "+rs);
  try {
    if(rs.getConcurrency()==ResultSet.CONCUR_UPDATABLE){
      System.out.println("UPDATABLE");
      int nColumns = md.getColumnCount();
      rs.updateString("eyecolor", eyecolor);
```

```
        rs.updateString("haircolor", haircolor);
        rs.updateRow();
      }
      else{
        System.out.println("READ_ONLY");
        status = "Update failed";
      }
    }catch(Exception e){
      e.printStackTrace();
    }
    return status;
  }
  public byte[] getMemberData(){
    String rootTag = "MemberInfo";
    String SQLQuery = "SELECT i.*, "+
                      "m.list MUSIC, g.list GOING_OUT, f.list FOODS, "+
                      "ph.list SPORTS, p.list PERSONALITY, a.list
ACTIVITIES  "+
                      "FROM personalinfo i, MUSIC m, goingout g, foods
f, "+
                      "physicalactivities ph, personality p, activities
a  "+
                      "WHERE m.id = g.id AND m.id = f.id AND m.id =
ph.id AND "+
                      "m.id = p.id AND m.id = a.id AND m.id = i.id AND
"+
                      "m.id = '"+memberId+"';";

    ByteArrayOutputStream os = new ByteArrayOutputStream();
    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;

      con = ds.getConnection(dbUserName,dbPassword);

      stmt = con.createStatement(
                       ResultSet.TYPE_SCROLL_SENSITIVE,
```

```
                        ResultSet.CONCUR_UPDATABLE);

      rs = stmt.executeQuery(SQLQuery);
      md = rs.getMetaData();
      if(rs.getConcurrency()==ResultSet.CONCUR_UPDATABLE){
        System.out.println("UPDATABLE");
      }else{
        System.out.println("READ_ONLY");
      }
      os.write(xmlHeader.getBytes());
      os.write(("<"+rootTag+">").getBytes());

      String xml = "";
      int columns = md.getColumnCount();
      rs.next();
      for(int i=1;i<=columns;i++){
        if(md.getColumnType(i)==Types.VARCHAR){
          xml="<"+md.getColumnLabel(i)+">"+
               rs.getString(i)+"</"+md.getColumnLabel(i)+">";
          os.write(xml.getBytes());
        }
      }
      os.write(("</"+rootTag+">").getBytes());

    }catch(Exception e){
      e.printStackTrace();
    }
    return os.toByteArray();
  }
}
```

To create an updatable `ResultSet` object, you need to call the `createStatement` method with these `ResultSet` constants:

- `TYPE_SCROLL_SENSITIVE`
- `CONCUR_UPDATABLE`

The `Statement` object that is created produces an updatable `ResultSet` object when it executes a query. Once you have an updatable `ResultSet` object, you can insert a new row, delete an existing row, or modify one or more column values.

Here are several considerations to bear in mind when using updatable ResultSets:

- An updatable `ResultSet` object does not necessarily have to be scrollable.
- An updatable `ResultSet` must generally specify the primary key as one of the columns selected.
- Requesting a ResultSet be updatable does not guarantee that the ResultSet you get will actually be updatable. Drivers that do not support updatable ResultSets return one that is read-only.
- If the driver does not support the definition of `UpdatableResultSet`, the `Statement` object may throw a SQL "Optional feature not implemented" exception.

Caution

Since requesting an `UpdatableResultSet` does not guarantee that you will actually get one, depending on the driver in use, you should check whether the `ResultSet` is updatable using `ResultSet.getConcurrency()`.

Note that setter methods are shown for the properties in the HTML form. If you don't include a setter method for each form property, Tomcat will give you a "method not found" error message.

Notice that I leave in a couple of `System.out.println()` statements. Before assuming that the update works correctly, it is important to ensure that you actually have an updatable `ResultSet`. Drivers that do not support updatable `ResultSets` go through the motions, but they return a `READ_ONLY ResultSet`.

Cross-Reference

Chapter 4 discusses the uses of updatable `ResultSets` at greater length, although the examples are not as detailed as the JavaBean example given here.

Summary

In this chapter, you learned how to combine ResultSets, Java Server Pages, and XML and XSL to create database-driven Web pages. Specifically, you learned the following:

- Using scrollable ResultSets to search a Web site
- Creating an XML document from a ResultSet
- Applying XSL stylesheets to create different Web pages from a single XML document
- Using an updatable ResultSet with an HTML form to update a database record

Chapter 16 explains how to use the JavaMail API with JDBC to send e-mail from a database and to receive and store e-mail to a database.

Chapter 16: Using the JavaMail API with JDBC

In This Chapter

In the earlier days of the Internet, e-mail mainly consisted of text messages that were handled using simple Java applications. As the popularity of e-mail has grown, its capabilities have expanded dramatically to the point where most e-mails these days are sent in both text and HTML formats and can include a wide range of different content types. The JavaMail API has been developed to simplify the task of handling these more complex e-mail messages using Java. This chapter gives a brief overview of the JavaMail API and illustrates the use of JDBC and JavaMail to send and receive e-mail.

Using E-mail Protocols

The backbone of e-mail is a network of interconnected Simple Mail Transfer Protocol (SMPT) servers, which store and forward e-mails. To send an e-mail, you connect to your local SMTP server and send the e-mail using SMTP. The e-mail is then forwarded to the recipient's server and held in the recipient's e-mail folder. The recipient later retrieves the e-mail, usually using the Post Office Protocol (POP). As the complexity of e-mail messages has grown, so has the need to manage the different data types contained in e-mail messages. This has led to the development of the Multipurpose Internet Mail Extensions.

Multipurpose Internet Mail Extensions (MIME)

The Multipurpose Internet Mail Extensions (MIME) define the content of e-mail messages, attachments, and so on. The MIME data type is defined in a Content-Type header field, used to specify the type and subtype of data in the body of a message. These are the common MIME types:

- "text" — used to represent standard text content
- "multipart" — used to combine several body parts, possibly of differing types, into a single message
- "application" — used to transmit application data or binary data
- "image" — used for transmitting still-image (picture) data
- "audio" — used for transmitting audio or voice data
- "video" — used for transmitting video or moving-image data

As a user of the JavaMail API, you are able to retrieve the MIME type from the header and to use it in deciding how to process the message.

Simple Mail Transfer Protocol (SMTP)

The SMTP is used for sending e-mail to an SMTP server, which your Internet Service Provider (ISP) usually manages. That SMTP server relays the e-mail message on to the recipient's SMTP server, where it is held in an e-mail store for the recipient's retrieval. The SMTP is defined in RFC 821, available at http://www.faqs.org/rfcs/rfc821.html.

Post Office Protocol (POP)

Since the current revision is version of POP is 3, the protocol is also known as POP3. Supporting a single mailbox for each user, POP3 is the most widely used way to download e-mail. POP3 is defined in RFC 1939, available at http://www.faqs.org/rfcs/rfc1939.html.

Note

POP3 supports only basic storage and download of e-mails. Features such as tracking new e-mails are handled by clients such as Eudora.

The next section explains how the JavaMail API works, and how to use it.

Using the JavaMail API

The best way to handle e-mail is to use the JavaMail API. The JavaMail API makes handling e-mail very straightforward, as it is designed to provide a protocol-independent means of sending and receiving messages. You can download the JavaMail API from Sun. In addition, you need to download the Java Activation Framework (JAF), which provides the basic MIME-type support used in most e-mail applications.

The first step in sending an e-mail using JavaMail is to get a JavaMail `Session`. Within the context of the `Session`, you create a new `Message` object, set its properties, and send it. These are the core JavaMail API classes needed to do perform these tasks:

- `Session` — defines a basic mail session
- `Message` — in most cases you will use `javax.mail.internet.MimeMessage`.
- `Address` — normally, you use the `javax.mail.internet.InternetAddress`.
- `Transport` — performs the protocol-specific tasks involved in sending the message
- `Store` — to receive e-mail messages, you first connect to a Mail Store.
- `Folder` — a Mail Store contains folders of messages that can be downloaded and read.

The `session, message, address` and `transport` classes of the core JavaMail API are explained below, and are illustrated in the first example, which shows you how to send an e-mail message. The remaining classes are used when receiving e-mails messages, and are explained and illustrated in the second example which illustrates how to receive e-mail messages.

The `Session` object defines a basic mail session. It uses a `java.util.Properties` object to hold application-level information such as the mail server, username, and password. In most cases, you can just use the shared session, even if you are working with multiple-user mailboxes.

The `Message` object represents the e-mail message. Properties of the `Message` object include the subject, the content, and the addresses of the sender and the recipient. A Mime Message is an e-mail message that understands different MIME types and headers.

E-mail addresses are implemented using the `Address` object. Normally, you use the `javax.mail.internet.InternetAddress` class. The `Address` object has constructors that let you set just an e-mail address or set an e-mail address and the name of the sender or recipient.

Note

The JavaMail API does not check the contents of an `Address` object, so unless your mail server prevents you, there is nothing stopping you from sending a message that appears to be from anyone.

The `Transport` object handles the protocol-specific language for sending the message (usually SMTP). You can use the default version of the class by calling the static `send()` method, or you can get a specific instance from the session. Here's an example:

```
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message, message.getAllRecipients());
transport.close();
```

Note

The basic send() mechanism makes a separate connection to the server for each method call. When you need to send multiple messages, it is better to get a specific instance of `Transport`; this keeps the connection with the mail server active between messages.

The first example explains how to use these classes to send an e-mail message. Receiving e-mail messages is covered in the second example.

Using JavaMail with JDBC to Send an E-mail Message

Apart from the endless stream of advertising messages one seems to receive, one of the most common uses of database-driven e-mail is to help a user who has forgotten his or her password. This is a fairly simple application, and a JSP page and JavaBean can handle it easily.

To send an e-mail to a member who has lost his or her password, it is necessary to query the ContactInfo Table to retrieve the member's e-mail address. The query also retrieves the password from the Login Table. This is the SQL query required to retrieve the member's password and e-mail address from the Login and ContactInfo Tables:

```
SELECT l.password, c.email
FROM LOGIN l, CONTACTINFO c
WHERE l.username = 'garfield' AND
      l.MemberID = c.MemberID;
```

For the purposes of this example, all that is required is a simple message containing the member's password, together with a small amount of explanatory text, as the content. The member's e-mail address, retrieved from the ContactInfo Table in the database, is plugged in to the `Message` object's recipient property.

For simplicity, this example is developed in the context of a JSP application, using the member login database developed in Chapter 12. The login JSP page can be set to redirect a user who fails the login check to a JSP page that uses the `SendMailBean` developed in the example that follows.

Cross-Reference The use of JSP pages and JavaBeans in database-driven Internet applications is discussed in Chapter 12.

Using a JSP Page and JavaMail to Send E-mails

The basic `SendMailBean` is similar to the `LoginBean` example in Listing 12-9, with the addition of the JavaMail component. The example uses a `DataSource` object to connect to the database and retrieve the member's password and e-mail address. If the e-mail address is not null, the `emailPassword()` method is called to send the password to the user.

The inner workings of the JavaMail-based `emailPassword()` method are simple. The first step is to get the System `properties` object and insert the name of the e-mail host serving the account to be used to send the e-mail:

```
props.put("mail.smtp.host", host);
```

The next step is to get the `Session` object, which provides the context in which the e-mail is sent:

```
Session session = Session.getDefaultInstance(props, null);
```

Once you have a `Session` object, use it to create a `MimeMessage` object as shown here:

```
MimeMessage message = new MimeMessage(session);
```

All that remains now is to set the properties of the `Message` object and to send it. In addition to the `Message.Recipient.TO` recipient type, the `Message` object defines these types:

- `Message.Recipient.CC`
- `Message.Recipient.BCC`

```
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,new InternetAddress(email));


message.setSubject("Password Reminder");


message.setText("Hi "+memberName+",Your password is: "+password);
```

Finally, call `Transport.send()` with the populated `Message` object as the argument, as shown here:

```
Transport.send(message);
```

The details of the `SendMailBean` are shown in Listing 16-1.

Listing 16-1: Sending e-mail by using the JavaMail API and JDBC

```
package JavaDatabaseBible.ch12;

import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.sql.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SendMailBean {
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  private Connection con = null;
  protected String username;

  public SendMailBean(){
  }
  public void setUsername(String username){
    this.username = username;
  }
  public String getUsername(){
    return username;
  }
  public String getPasswordAndEmailAddress(){
    String password = null;
    String email = null;

    try {
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "JUPITER" );
      tds.setDatabaseName( "MEMBERS" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );
```

```
      DataSource ds = tds;
      Connection con = ds.getConnection(dbUserName,dbPassword);
      Statement stmt;
      ResultSet rs = null;

      String SQLQuery = "SELECT l.Password, c.Email FROM LOGIN l, "+
                        "CONTACTINFO c WHERE l.MemberID = c.MemberID "+
                        "AND UserName = '"+username+"';";
      stmt = con.createStatement();
      rs   = stmt.executeQuery(SQLQuery);
      while(rs.next()){
        password = rs.getString("Password");
        email = rs.getString("Email");
      }
      con.close();

    }catch(ClassNotFoundException e1){
      System.err.println(e1.getMessage());
    }catch(SQLException e2){
      System.err.println(e2.getMessage());
    }

    if(email==null){
      return "Bad Email";
    }else{
      emailPassword(email,username,password);
      return "OK";
    }
 }

  public void emailPassword(String email,String memberName,String
password){
    String host = "mail";
    String from = "webmaster@j-machines.com";

    Properties props = System.getProperties();

    props.put("mail.smtp.host", host);

    // Get session
    Session session = Session.getDefaultInstance(props, null);

    // Define message
```

```
    MimeMessage message = new MimeMessage(session);

    try{
      // Set the sender and recipient addresses
      message.setFrom(new InternetAddress(from));
      message.addRecipient(Message.RecipientType.TO,new
InternetAddress(email));

      // Set the subject
      message.setSubject("Password Reminder");

      // Set the message content
      message.setText("Hi "+memberName+",\nYour password is: "+
                           password+"\nregards - "+from);

      // Send it
      Transport.send(message);
    }catch(AddressException ae){
    }catch(MessagingException me){
    }
  }
}
```

The SQL query in this example is fairly efficient, since the Login Table is indexed by UserName and the ContactInfo Table is indexed by MemberID. However, you can make the query faster by forwarding the MemberID from the previous page, so that you need only query the ContactInfo Table.

A JSP Page for Use with the SendMailBean

Using the SendMailBean requires a JSP page to set the UserName argument and to call the method `SendMailBean.getPasswordAndEmail()` to query the database and send the e-mail. The JSP page required to use the SendMailBean looks something like the example in Listing 16-2.

Listing 16-2: A JSP page for use with the SendMailBean

```
<html>
<head>
<title>Email Password</title>
</head>
<body>
<%@ page language="java"%>
<jsp:useBean id="SendMailBean" scope="application"
class="JavaDatabaseBible.ch12.SendMailBean"/>
<jsp:setProperty name="SendMailBean" property="*"/>
<%
    String userName = request.getParameter("username");
```

```
    String emailStatus = SendMailBean.getPasswordAndEmail();
    if(emailStatus.equals("OK")){
%>
Hi <%=userName%>,<br>
Your password is being emailed to the address we have on file.
<%
    }else{
%>
Sorry, <%=userName%>,<br>
Your email address is not on file.
<%
    }
%>
</body>
</html>
```

Deployment

To deploy JavaBeans for use with JSP pages, put the class files for the beans into the appropriate directory. For a simple Tomcat installation, the usual path is as follows:

```
TOMCAT/WEBAPPS/ROOT/WEB-INF/CLASSES
```

Recall that servlet deployment requires you to put any jar files you need into a suitable directory and to modify Tomcat's class path in the tomcat.properties file in the `Tomcat/conf` directory. In this example, the jar file is saved in the `/lib` directory, and Tomcat's class path is modified by adding the following lines to the `tomcat.properties` file:

```
wrapper.classpath=lib/jdbc2_0-stdext.jar
wrapper.classpath=lib/activation.jar
wrapper.classpath=lib/mail.jar
wrapper.classpath=lib/Opta2000.jar
```

The `store` and `folder` classes of the core JavaMail are explained and illustrated in the next example which illustrates how to receive e-mail messages.

Receiving E-mail Using the JavaMail API

Receiving e-mail with the JavaMail API is only a little more complicated than sending e-mail. In addition to the JavaMail objects used to send an e-mail, receiving e-mails involves the use of the `Store` and `Folder` objects. The following sequence of events is similar to sending an e-mail:

1. Get the default e-mail session.
2. Get the POP3 message store object.
3. Connect to the store, using the server name, mail-user name, and password.
4. Get the default folder.
5. Get the `INBOX`.
6. Open the `INBOX` and read the messages.

The process is started in much the same way as sending a message, but, after getting the session, you connect to a Store instead of a Transport. Here's an example:

```
Store store = session.getStore("pop3");
store.connect(host, username, password);
```

After connecting to the Store, get a folder and open it. Using POP3, the only folder available is the `INBOX`. Once the folder is open, you can read messages from it, as shown here:

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
```

The `folder.getMessages()` method uses *lazy data retrieval*. In other words, the message content is only downloaded when specifically requested. You can get the content of a message with `getContent()` or write the content to a stream with `writeTo()`. The `getContent()` method only gets the message content, and `writeTo()` output includes headers. Here's an example:

```
System.out.println(((MimeMessage)message).getContent());
```

Notice that the `INBOX` is opened in `READ_ONLY` mode. Write access can be used to mark messages as received or to delete them from the server. Listing 16-3 illustrates how easy it is to receive e-mail messages using the JavaMail API.

Listing 16-3: Reading e-mail using JavaMail and saving it to a database

```
import javax.mail.*;
import javax.mail.internet.*;

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

public class JavaMailReceiver{
  static String server="mail.home.com";
  static String username="user";
  static String password="password";

  static MailSaver db = new MailSaver();

  public static void main(String args[]){
    try {
      receive(server, username, password);
    }catch (Exception e){
      System.err.println(e);
    }
    System.exit(0);
  }

  public static void receive(String server,
                             String username, String password){
    Store store=null;
    Folder folder=null;
```

```
    try{
      // -- Get the default session --
      Properties props = System.getProperties();
      Session session = Session.getDefaultInstance(props, null);

      // -- Get a POP3 message store, and connect to it --
      store = session.getStore("pop3");
      store.connect(server, username, password);

      // -- Get the default folder --
      folder = store.getDefaultFolder();
      if (folder == null) throw new Exception("No default folder");

      // -- Get its INBOX --
      folder = folder.getFolder("INBOX");
      if (folder == null) throw new Exception("No POP3 INBOX");

      // -- Open the folder for read only --
      folder.open(Folder.READ_ONLY);

      // -- Get the message wrappers and process them --
      Message[] msgs = folder.getMessages();
      int msgNum = msgs.length;
      while(processMessage(msgs[--msgNum]));
    }
    catch (Exception e){
      e.printStackTrace();
    }
    finally{
      try{
        if (folder!=null) folder.close(false);
        if (store!=null) store.close();
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }

  public static boolean processMessage(Message message){
    Calendar today = Calendar.getInstance();
    try{
```

```
      // Get the header information
      String subject = message.getSubject();
      String dateString = "unknown date";
      String to
            =
 ((InternetAddress)message.getAllRecipients()[0]).getPersonal();
      String toEmail
            =
 ((InternetAddress)message.getAllRecipients()[0]).getAddress();
      String from  =
((InternetAddress)message.getFrom()[0]).getPersonal();
      String email =
((InternetAddress)message.getFrom()[0]).getAddress();

      if (to==null)  to   = toEmail;
      if (from==null from = email;

      java.util.Date date=message.getSentDate();

      Calendar mDate = Calendar.getInstance();
      if(date!=null){
        dateString = date.toString();
        mDate.setTime(date);
        if(mDate.get(Calendar.DAY_OF_MONTH)<
           today.get(Calendar.DAY_OF_MONTH)-3)return false;
      }

      System.out.println("DATE: "+dateString);
      System.out.println("TO:   "+to+" <"+toEmail +">");
      System.out.println("FROM: "+from+" <"+email +">");
      System.out.println("SUBJECT: "+subject);

      // -- Get the message --
      Part messagePart=message;
      Object content=messagePart.getContent();

      if (content instanceof Multipart){
        for(int i=0;i<((Multipart)content).getCount();i++){
          messagePart=((Multipart)content).getBodyPart(i);
          String contentType=messagePart.getContentType();
          if (contentType.startsWith("text/plain") ||
              contentType.startsWith("text/html")){
            String msg = readMsg(messagePart);
```

```
            db.saveEmail(dateString,from,email,subject,contentType,msg);
          }
        }
      }else{
        String contentType=messagePart.getContentType();
        if (contentType.startsWith("text/plain") ||
            contentType.startsWith("text/html")){
          String msg = readMsg(messagePart);
          db.saveEmail(dateString,from,email,subject,contentType,msg);
        }
      }
    }
    catch (Exception ex){
      ex.printStackTrace();
    }
    return true;
  }
  private static String readMsg(Part messagePart){
    String message = "";
    try{
      String contentType=messagePart.getContentType();
      if (contentType.startsWith("text/plain")||
          contentType.startsWith("text/html")){
        InputStream is = messagePart.getInputStream();
        BufferedReader reader = new BufferedReader(new
InputStreamReader(is));
        String line = reader.readLine();
        while(line!=null){
          message = message + line;
          line = reader.readLine();
        }
      }
    }catch(Exception e){
      System.err.println(e);
    }
    return message;
  }
}

class MailSaver{
  private static String dbUserName = "dbUser";
  private static String dbPassword = "dbPwd";
  Connection con = null;
```

```
  public MailSaver(){
    try{
      Class.forName("com.inet.pool.PoolDriver");
      com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
      tds.setServerName( "MARS" );
      tds.setDatabaseName( "EMAIL" );
      tds.setUser( dbUserName );
      tds.setPassword( dbPassword );

      DataSource ds = tds;
      con = ds.getConnection(dbUserName,dbPassword);
    }
    catch(Exception e){
      System.err.println("SQL Exception registering driver");
    }
  }

  public void saveEmail(String date,String sender,String senderEmail,
                        String subject,String mimeType,String msg){
    String cmd = "INSERT INTO EMAIL "+

"(MsgDate,Sender,SenderEmail,Subject,ContentType,Message) "+
                 "VALUES(?,?,?,?,?,?)";
    try {
      PreparedStatement pstmt = con.prepareStatement(cmd);

      pstmt.setString(1, date);
      pstmt.setString(2, sender);
      pstmt.setString(3, senderEmail);
      pstmt.setString(4, subject);
      pstmt.setString(5, mimeType);
      pstmt.setString(6, msg);
      pstmt.executeUpdate();
    }
    catch(SQLException e){
      e.printStackTrace();
    }
  }
}
```

The example of Listing 16-3 is a straightforward JavaMail application. It logs on to the SMTP server and uses the `getMessages()` method to get all the messages on the server, as shown here:

```
Message[] msgs = folder.getMessages();
```

```
int msgNum = msgs.length;
while(processMessage(msgs[--msgNum]));
```

The `processMessage()` method does the actual message processing. Note that messages are processed in reverse order until the `processMessage()` method returns `false`. I do this for the eminently practical reason that I am more interested in recent messages and do not want to loop through the hundreds of messages on my e-mail server.

The `processMessage()` method parses out the date, subject, and sender information for display to the console. It then checks the date against today's date, returning `false` if the message is more than three days old.

If the MIME content type of the message is text or HTML, the message is saved as a Clob to a simple E-mail Table. The table includes columns for the following items:

- Date
- Sender
- Sender E-mail Address
- Subject
- Mime Type
- Content

Note that the individual parts of a single message are saved as separate rows, so a message may be saved over more than one row. Using an automatically incremented message id in the table helps identify message parts separately, as does the Mime-type field.

Summary

This chapter provided an overview of the JavaMail API. Specific topics discussed were:

- Developing e-mail applications using the JavaMail API
- Sending e-mails from database-driven applications by using JavaMail
- Receiving e-mails by using the JavaMail API and saving them to a database

This chapter concludes Part III, in which the JDBC Extension API has been discussed in the context of Web applications. Part IV focuses on using Java databases with XML.

Part IV: Using Databases, JDBC,and XML

Chapter List

Part Overview

XML is a text-based markup language that is fast becoming a standard for data management and interchange, both within an application and between applications on and off the Web. Although at first glance an XML document looks much like an HTML Web page, there are significant differences between the two. The foremost of these can be summed up as follows:

- HTML is primarily used to mark up text and other data with formatting information.
- XML is primarily used to structure data, either for transport or for an application's local use.

In other words, HTML documents are primarily document-centric (that is, they are designed primarily for human consumption). XML documents, on the other hand, are primarily data-centric (that is, they are primarily intended for machine use where some degree of human readability is desirable).

Web pages, obviously, are typical examples of document-centric applications. Examples of typical data-centric XML applications include:

- Messaging between applications via the SOAP protocol. This is primarily used by the new Web-services paradigm.
- Remote procedure calls over HTTP using XML-RPC
- Data transport, such as the delivery of stock quotes or news headlines over the Internet
- Initialization functions that used to be handled by `.ini` or `.properties` files.Tomcat's `web.xml` is a good example of an initialization file implemented with XML.
- Scripting in such applications as the build language ANT. The ANT build file is XML based.

Data-centric documents are also typically characterized by a regular structure, frequently because they are machine generated. Document-centric material is frequently less regularly structured, as humans generate it. The content of data-centric documents frequently either originates in a database, in which case the XML document is used to publish it, or is intended to be stored in a database, in which case the XML is used to transport it there.

In some instances, an XML document, being a data repository, can be a database in itself. For example, the contact lists on my Linux-based PDA are saved as XML documents.

The chapters in Part IV discuss working with databases and XML. Chapter 17 reviews retrieving data from a table and formatting the `ResultSets` as XML, as well as fetching XML data from the Internet and saving the data to a table. The JDBC `RowSet` is discussed in Chapter 18, and Chapter 19 goes on to look at creating a simple JDBC driver that allows you to access XML documents using SQL.

Note

This distinctions drawn between HTML and XML deliberately overlook the fact that well-formed HTML is a particular application of XML. In terms of common usage, the distinction is valid.

Chapter 17: The XML Document Object Model and JDBC

In This Chapter

XML is fast becoming a universal standard for exchanging data between applications. Today, many major organizations are using XML in the daily course of business. The International Press Telecommunications Council, for example, has defined an XML DTD to simplify news distribution and publishing. Even local phone companies are using XML as the basis of a computerized order placement and billing system.

This means that XML processing is steadily becoming more and more important to Java programmers. Typically XML documents are used as a means of transferring database records between businesses. This chapter starts with a brief introduction to XML, and then goes on to discuss how to generate XML documents from a SQL query, and how to populate a database from an XML document.

XML versus HTML

The eXtensible Markup Language (XML) is a text-based markup language that is fast becoming a standard for data interchange both on the Web and between applications. XML is similar to the HyperText Markup Language (HTML) in that it uses tags enclosed in angle brackets (<>) to identify data, as shown in Listing 17-1.

Listing 17-1: XML example

```
<?xml version="1.0"? >
 <CONTACT_INFO>
  <FIRST_NAME>Vito</FIRST_NAME>
  <LAST_NAME>Corleone</LAST_NAME>
  <STREET>123 Main</STREET>
  <CITY>New York</ CITY >
  <STATE>NY</STATE>
  <ZIP>12345</ZIP>
 </CONTACT_INFO>
```

Unlike HTML tags, XML tags identify and describe data rather than specifying how to display it. The tags used to identify and describe data are application dependent, so you can use any tags you like, as long as they follow the rules the W3C XML standard defines.

A major difference between XML and HTML is that an XML document must always be well formed. Among other things, this means that every tag has a closing tag. For example, in HTML, you frequently see dangling paragraph <P> and break
 tags. These are illegal in XML, which requires that a closing tag be provided, either by using the form <P></P> or <P/>.

Another frequent usage in HTML, but illegal in XML, is incorrect nesting. Most browsers can handle HTML with elements closed in arbitrary order, as in the following example, which combines incorrectly closed elements with elements that simply aren't closed at all. This example displays just fine in a browser but is unreadable as XML:

```
<HTML>
 <BODY>
  <CENTER><FONT FACE="Arial">
    Hello World
```

```
</CENTER></FONT>
```

Caution

XML documents must always be well formed. Most browsers can be used to check that HTML documents are well formed, as can XML tools such as XML-Spy.

From a programming viewpoint, XML can be handled either as a character stream, or as an object. In a stream based parser, XML elements are identified sequentially and used to trigger an event driven processor. When the Document Object Model is used, the entire document is parsed into a document object, in which the various elements and attributes can be accessed by name and path in much the same way as files are accessed in a directory tree. The next section discusses the Document Object Model.

XML and the Document Object Model

The Document Object Model (DOM) represents an XML document as a tree. The document element is the top level of the tree. The document element has a number of child nodes that represent the branches of the tree. Figure 17-1 shows an XML document displayed as a tree.

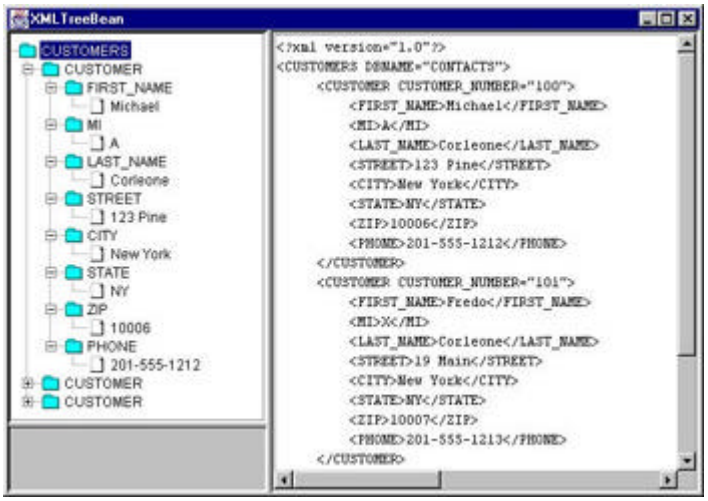


Figure 17-1: XML document displayed as a tree

The most basic component of the DOM is the Node interface. Every constituent component in a DOM representation of an XML document is a node. The Node interface is extended by other interfaces such as the Element interface and the Document interface. In the example shown in Figure 17-1, the document element is the CUSTOMERS element, highlighted in the JTree representation. You can see in the text representation on the right-hand side that the CUSTOMERS element contains the attribute node `DBNAME= "CONTACTS".`

CUSTOMERS is an element node, as are CUSTOMER, FIRST_NAME, and so on. The JTree represents elements as folders. Within the element nodes are additional element nodes, as well as text nodes, represented by a document icon, with the actual text printed next to the icon.

Representing the XML document as a tree structure of Java objects allows the Java programmer to create, access, and modify XML documents and their contents through one of a number of widely available APIs. Before discussing the DOM, it is appropriate to review the structure of an XML document.

The XML Header

An XML file must always start with a declaration that identifies the document as an XML. The minimal header looks like this:

```
<?xml version="1.0"?>
```

The declaration may contain additional information identifying the character set or encoding the presence or absence of additional reference documents such as a document type definition and other related information:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

In this example, the header contains the following information:

- The XML version number is 1.0
- The character encoding is ISO-8859-1, the HTTP default character encoding
- The documentis a standalone document, or one which requires no supporting documents such as an external Document Type Definition

Everything that comes after the XML header constitutes the document's content.

Note

The XML version attribute is required. An XML parser will report an error if no version attribute is supplied.

Tags and Attributes

The tags in the example of Listing 17-1 identify the content as a whole, as well as the individual elements: the contact's first name, last name, street, city, and zip. These data elements are contained in a hierarchical structure, defined by nesting them inside the `<CONTACT_INFO>` tag. The capability of one tag to contain others permits XML to represent hierarchical data structures

The format of an XML document on the printed page is largely a matter of convenience. As is the case with HTML, whitespace is not considered significant.

In addition to the tag name, XML tags can contain attributes within the tag's angle brackets. Attributes, as in HTML, are generally used to provide additional information about an element. A good example of a tag with attributes is the HTML `<FONT>` tag shown here, which contains attributes describing the font face, size, and color:

```
<FONT FACE="Arial" SIZE="3" COLOR="#0000FF">Hello World</FONT>
```

As in HTML, attributes are defined as key = value pairs, separated by spaces. Unlike HTML, however, XML requires that attribute values be quoted, separated only by whitespace. In other words, the FONT tag shown above complies with the requirements for defining xml attributes, while the example below, which works fine as HTML, is invalid as XML:

```
<FONT FACE=Arial SIZE=3 COLOR=#0000FF>Hello World</FONT>
```

Since you can design a data structure like <message> equally well using either attributes or tags, it can take a considerable amount of thought to figure out which design is best for your purposes. The last part of this tutorial, "Designing an XML Data Structure," includes ideas to help you decide when to use attributes and when to use tags.

Elements and Nodes

The DOM represents an XML document as a tree structure, where each node contains one of the components of the XML document. Using DOM methods, you can create and remove nodes, change their contents, and traverse the node hierarchy.

The DOM defines a number of different types of nodes in the `org.w3c.dom.Node` interface. The most commonly used of these are summarized in Table 17-1.

Table 17-1: org.w3c.dom Interface Node

org.w3c.dom Node_Type	Application	Example
ATTRIBUTE_NODE	Attribute	`key="value"`
COMMENT_NODE	Comment	`<-- This is a comment --`

Table 17-1: org.w3c.dom Interface Node

org.w3c.dom Node_Type	Application	Example
		>
DOCUMENT_NODE	Document	The enclosing Document
ELEMENT_NODE	Element	<NAME>...</NAME>
TEXT_NODE	Body Text within element	Hello world

The next section shows how to use some of the widely available DOM parsing tools to process XML documents.

Using a Java XML API — Xerces and JDOM

A number of tools are available for working with XML in Java. Among the most widely used are the Xerces package, available for download from `apache.org.`, and JDOM, available from `jdom.org`.

Xerces includes fully-validating parsers, implementing the W3C XML and DOM (Level 1 and 2) standards, as well as the de facto SAX (version 2) standard. Xerces is a large, comprehensive implementation of the full standard.

JDOM is a totally Java-oriented approach to working with XML. It seeks to provide a robust, light-weight means of reading and writing XML data. It is certainly rather more intuitive to work with than the Xerces API, but the differences are minor.

Having worked extensively with both, as well as with my own light-weight, custom XML API, I have selected Xerces for the examples in the book for several reasons:

- Xerces is intended as a complete implementation of the DOM.
- The sample code is built around Xbeans, from `Xbeans.org`. The origianl Xbean code uses Xerces.
- The Xerces jar contains everything you need to implement all the examples.

Having said all that, I should point out that translating the examples from one API to the other is relatively simple, since the sample code uses only a small part of the API. The following code snippet shows the creation of an XML document using the Xerces API. This code is taken from the SystemTime bean example that you'll find later in this chapter under "Using XBeans as Pluggable XML Processing Blocks."

```
Document doc = new DocumentImpl();
Element root = (Element) doc.createElement("SYSTEMDATE");
doc.appendChild (root);


Element year = (Element) doc.createElement("YEAR");
root.appendChild(year);
year.appendChild(doc.createTextNode(""+calendar.get(Calendar.YEAR)));
```

The following JDOM equivalent is similar, but quite a bit simpler than the Xerces example, since it is specifically designed to be a Java-oriented way of working with XML:

```
Element root = new Element("SYSTEMDATE");
Document doc = new Document(root);


Element year = new Element("YEAR");
root.addContent(year);
year.setText(""+calendar.get(Calendar.YEAR));
```

One of the main advantages of using the DOM approach to handling XML documents is that once you have parsed the document into a DOM object, you can access any element as required. This feature is illustrated in the next section on using Xbeans to process XML documents.

Using Xbeans as Pluggable XML Processing Blocks

One of the most straightforward ways to work with XML documents in Java is to use Xbeans. Xbeans are the brainchild of Bruce Martin, whose work lies at the core of `Xbeans.org`, an open-source project, where you can read Bruce's excellent white paper "Creating Distributed Applications Using Xbeans." You can also download the basic Xbean interface library from this site.

Essentially, Xbeans are pluggable XML-processing blocks. They are intended to be connected in chains, where each Xbean performs one logical step in processing an XML document. The Xbean then passes the document to the next Xbean in the chain as the event object in a bean event, as illustrated in Figure 17-2.

Figure 17-2: Xbean connectivity

Connectivity is the primary key to the Xbean concept. Each Xbean in a chain performs one processing step; then it passes the processed XML document to the next bean for further processing. This approach makes it easy to break a project down into simple, frequently repeated operations, each of which can be implemented as a reusable component.

Xbean connectivity is implemented using the delegation event model to communicate with other Xbeans. The delegation event model is implemented using the `DOMEvent` interface, which defines a DOM document as the event object. By firing a `DOMEvent`, the event source Xbean passes an XML document as the `DOMEvent` object to the `EventListener` bean.

The Xbean connectivity model is defined by two interfaces included in the package `org.Xbeans`:

- `DOMSource` defines two methods, `setDOMListener()` and `getDOMListener()`, for setting and getting the Xbean that receives the output.
- `DOMListener` defines a single method, `documentReady(DOMEvent e)`, which is called by the event source Xbean to pass the XML document encapsulated in the `DOMEvent`.

Conventionally, the `documentReady()` method calls a `processDocument(DOMEvent e)` method, which processes the XML document and returns it. If there is another Xbean in the chain, the processed document is wrapped in a new `DOMEvent` and passed on to the next Xbean in the chain.

In practice, the simplest way to use Xbeans is to create a base class with an empty `processDocument()` method. Then extend the Xbean base class as required, overriding the `processDocument()` method to implement the desired functionality. Listing 17-2 illustrates the Xbean base class.

Listing 17-2: XBean base class

```
package JavaDatabaseBible.ch17.Xbeans;

import org.Xbeans.*;
import org.w3c.dom.Document;

/**
```

```
* XBean base class implements org.Xbeans interfaces.
*
* extend XBean to create useful Xbeans.
*/
public class XBean implements org.Xbeans.DOMListener,
org.Xbeans.DOMSource{
  protected DOMListener DOMListener;
  protected Document processedXmlDoc = null;

  public XBean(){
  }
  public void setDOMListener(DOMListener newDomListener) {
    DOMListener = newDomListener;
  }

  public DOMListener getDOMListener(){
    return DOMListener;
  }

  public void documentReady(DOMEvent evt) throws XbeansException {
    processedXmlDoc = processDocument(evt.getDocument());
    if(DOMListener!=null)
      DOMListener.documentReady(new DOMEvent(this, processedXmlDoc));
  }

  public void processDocument() throws XbeansException {
  }

  public Document processDocument(Document doc) throws XbeansException {
    return doc;
  }
}
```

Xbeans are normally used in chains, which implicitly have a beginning and an end. An extremely useful ending Xbean, which extends the basic XBean of Listing 17-1, is the `SerializerBean` shown in Listing 17-3. This bean simply outputs a document to a stream, defaulting to `System.out`.

Listing 17-3: SerializerBean

```
package JavaDatabaseBible.ch17.Xbeans;

import java.io.*;
import org.Xbeans.*;
import org.w3c.dom.Document;
import org.apache.xml.serialize.OutputFormat;
```

```
import org.apache.xml.serialize.XMLSerializer;

/**
* Serialize a document to a stream or writer
*/

public class SerializerBean extends JavaDatabaseBible.ch17.Xbeans.XBean
{
  protected OutputStream os = System.out;
  protected Writer writer = null;
  protected XMLSerializer serializer;

  public SerializerBean(){
  }

  public void setOutputStream(OutputStream os){
  this.os = os;
  }

  public void setWriter(Writer writer){
    this.writer = writer;
  }

  public Document processDocument(Document doc) {
    OutputFormat fmt = new OutputFormat("xml",null,true);
    if(writer!=null){
      serializer = new XMLSerializer(writer,fmt);
    }
    else{
      serializer = new XMLSerializer(os,fmt);
    }

    if(doc!=null){
      try{
        serializer.asDOMSerializer().serialize(doc);
      }
      catch (Exception e){
       e.printStackTrace();
      }
    }
    return doc;
  }
}
```

An Xbean chain starts with an Xbean that either reads an existing document from a stream or creates one from some data source. Listing 17-4 illustrates a simple Xbean that creates an XML document from the system clock.

Listing 17-4: SystemTime bean

```
package JavaDatabaseBible.ch17.Xbeans;

import java.io.*;
import java.util.Calendar;
import java.util.GregorianCalendar;
import org.Xbeans.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.apache.xerces.dom.DocumentImpl;
/**
* Create a w3c.dom.Document containg system time and date.
*/
public class SystemTimeBean extends JavaDatabaseBible.ch17.Xbeans.XBean{

  protected String textFileName = "";

  public SystemTimeBean() {
  }

  public void processDocument() throws XbeansException{
    GregorianCalendar calendar = new GregorianCalendar();
    try{
      Document doc = new DocumentImpl();

      Element root = (Element) doc.createElement("SYSTEMDATE");
      doc.appendChild (root);

      Element year = (Element) doc.createElement("YEAR");
      root.appendChild(year);
      year.appendChild(doc.createTextNode(""+calendar.get(Calendar.YEAR)));

      Element month = (Element) doc.createElement("MONTH");
      root.appendChild(month);
      int mon = calendar.get(Calendar.MONTH)+1;
      month.appendChild(doc.createTextNode(String.valueOf(mon)));

      Element day = (Element) doc.createElement("DAY");
      root.appendChild(day);
```

```
      int mDay = calendar.get(Calendar.DAY_OF_MONTH);
      day.appendChild(doc.createTextNode(String.valueOf(mDay)));

      Element dayOfWeek = (Element) doc.createElement("DAY_OF_WEEK");
      root.appendChild(dayOfWeek);
      int wDay = calendar.get(Calendar.DAY_OF_WEEK);
      dayOfWeek.appendChild(doc.createTextNode(String.valueOf(wDay)));

      Element hour = (Element) doc.createElement("HOUR");
      root.appendChild(hour);
      int h = calendar.get(Calendar.HOUR_OF_DAY);
      hour.appendChild(doc.createTextNode(String.valueOf(h)));

      Element min = (Element) doc.createElement("MINUTE");
      root.appendChild(min);
      int m = calendar.get(Calendar.MINUTE);
      min.appendChild(doc.createTextNode(String.valueOf(m)));

      DOMListener.documentReady(new DOMEvent(this,doc));
    }
    catch (Exception e){
      throw(new XbeansException("","SystemTimeBean",
                                e.toString(),e.getMessage()));
    }
  }
}
```

Now you have enough Xbeans to create a chain and experiment with the technology. Listing 17-5 is a simple test program that illustrates how to instantiate and interconnect the Xbeans.

Listing 17-5: Using Xbeans to create an output of an XML document

```
import java.io.*;
import java.beans.Beans;
import JavaDatabaseBible.ch17.Xbeans.*;

public class SysTimeBeanTest {
  static public void main(String args[]) {

    try{
      SystemTimeBean timeBean = (SystemTimeBean)Beans.instantiate(null,

"JavaDatabaseBible.ch17.Xbeans.SystemTimeBean");
      SerializerBean serializer =
(SerializerBean)Beans.instantiate(null,
```

```
"JavaDatabaseBible.ch17.Xbeans.SerializerBean");
      timeBean.setDOMListener(serializer);
      serializer.setOutputStream(new FileOutputStream("TimeStamp.xml"));
      timeBean.processDocument();
    }catch(Exception e){
      System.err.println(e);
    }
  }
}
```

As you can see from the example in Listing 17-5, using Xbeans to create and process xml documents is a simple, two-step process:

1. Instantiate the Xbeans and set any required properties,
2. Call the `processDocument()` method of the first Xbean in the chain, the `SystemTimeBean`.

The sequence of events that follows is as follows. The `SystemTimeBean` first creates a new XML document using the following code:

```
Document doc = new DocumentImpl();
```

It then creates a root element and appends it to the document, as shown here:

```
Element root = (Element) doc.createElement("SYSTEMDATE");
doc.appendChild (root);
```

Then it creates elements for year, month, day, and so on, appending them to the root. Here's an example:

```
Element year = (Element) doc.createElement("YEAR");
root.appendChild(year);
year.appendChild(doc.createTextNode(""+calendar.get(Calendar.YEAR)));
```

Finally, it calls the `documentReady()` method of its registered listener, the `SerializerBean`, passing it the new XML document, as shown here:

```
DOMListener.documentReady(new DOMEvent(this,doc));
```

The `SerializerBean`, in turn, calls its own `processDocument()` method to serialize the document to the stream defined in its `outputStream` property. The resulting XML is shown in Listing 17-6.

Listing 17-6: XML TimeStamp generated and serialized using Xbeans

```
<?xml version="1.0"?>
<SYSTEMDATE>
    <YEAR>2002</YEAR>
    <MONTH>3</MONTH>
    <DAY>17</DAY>
    <DAY_OF_WEEK>1</DAY_OF_WEEK>
    <HOUR>15</HOUR>
    <MINUTE>43</MINUTE>
</SYSTEMDATE>
```

A much more common application for Xbeans is in converting JDBC ResultSets to XML documents, as illustrated in the next section.

Creating XML Documents by Querying a Database

Creating XML documents from a database is every bit as simple as creating them from a source such as the system clock. If you refer to the XML document shown in Figure 17-1, you will notice that it bears a structural resemblance to a DBMS Table. This is not surprising, since document is derived from one.

There are two basic ways to create XML documents. The first is illustrated in Chapter 15, where a JSP is used to extract data from a database and output it as XML in much the same way as JSPs are used to output HTML. Although this approach produces perfectly valid XML, another, more flexible approach is to create a DOM representation of the document, which can be processed as desired before serialization.

An Xbean designed to create DOM documents using SQL queries is shown in Listing 17-7. The `processDocument()` method of this Xbean first creates a DOM document with a root element that identifies the table name and the database in use. It then calls the method `appendDataNodes()`, which executes the SQL query and appends an element for each row in the `ResultSet`. The Customer Number is inserted into this element as an attribute. Nested in each row are elements with tag names set to the column name and containing the column data as a text node. This example happens to use the jdbc-odbc bridge driver, but you can easily substitute the DataSource code from any of the examples in Part III of this book.

Listing 17-7: Creating an XML document using a SQL query

```
package JavaDatabaseBible.ch17.Xbeans;

import java.io.*;
import java.sql.*;
import org.Xbeans.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.apache.xerces.dom.DocumentImpl;

/**
* Query a database and return the ResultSet as an XML document
*/
public class SQLQueryBean extends XBean{
  private String databaseName = "";
  private String tableName = "";
  private String SQLQuery = null;

  Document document;

  public SQLQueryBean(){
  }

  public void setDatabaseName(String databaseName){
    this.databaseName = databaseName;
```

```
}

public void setTableName(String tableName){
  this.tableName = tableName;
}

public void setSQLQuery(String SQLQuery){
  this.SQLQuery = SQLQuery;
}

public void processDocument() throws XbeansException{
  try{
    document = new DocumentImpl();
    if(databaseName.length()>0&&tableName.length()>0){
      String d = databaseName.toUpperCase();
      String t = tableName.toUpperCase();
      Element root = (Element)document.createElement(t);
      document.appendChild (root);
      root.setAttribute("DBNAME",d);
      appendDataNodes();
    }
    else{
      throw new XbeansException("SQLQueryBean",
                null,"DBName/TableName Undefined",null);
    }
    DOMEvent domEvt = new DOMEvent(this,document);
    DOMListener.documentReady(domEvt);
  }
  catch (Exception e){
    e.printStackTrace();
  }
}

public void appendDataNodes() {
  String url = "jdbc:odbc:"+databaseName;

  if(SQLQuery==null)SQLQuery="SELECT * FROM "+tableName;

  try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con = DriverManager.getConnection(url,"user","pwd");
    Statement stmt = con.createStatement();
    ResultSet rs   = stmt.executeQuery(SQLQuery);
```

```
      ResultSetMetaData md   = rs.getMetaData();

      int nColumns = md.getColumnCount();
      Element root = document.getDocumentElement();

      while(rs.next()){
        Element record=(Element)document.createElement("CUSTOMER");
        root.appendChild (record);

        for(int i=1;i<=nColumns;i++){
          String fName = md.getColumnLabel(i);
          String data = rs.getString(i);
          if(fName.equals("CUSTOMER_NUMBER")){
            record.setAttribute("CUSTOMER_NUMBER",String.valueOf(data));
          }else{
            Element fld = (Element)document.createElement(fName);
            record.appendChild(fld);
            fld.appendChild(document.createTextNode(data));
          }
        }
      }
      con.close();
    }
    catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

The `SQLQueryBeanTest` code shown in Listing 17-8 is nothing more than a modified version of Listing 17-5, with the `SQLQueryBean` plugged in place of the `SystemTimeBean`. This also serves to illustrate how simple it is to use Xbeans to process XML data.

Listing 17-8: Using the SQLQueryBean

```
package JavaDatabaseBible.ch17.Xbeans;

import java.io.*;
import java.beans.Beans;
import JavaDatabaseBible.ch17.Xbeans.*;

public class SQLQueryBeanTest {
  static public void main(String args[]) {
    String databaseName = "SQLServerContacts";
    String tableName = "CUSTOMERS";
```

```
    String SQLQuery = "SELECT * FROM CUSTOMERS WHERE STATE = 'NY'";
    try{
      SQLQueryBean queryBean = (SQLQueryBean)Beans.instantiate(null,

"JavaDatabaseBible.ch17.Xbeans.SQLQueryBean");
      SerializerBean serializer =
(SerializerBean)Beans.instantiate(null,

"JavaDatabaseBible.ch17.Xbeans.SerializerBean");

      queryBean.setDatabaseName(databaseName);
      queryBean.setTableName(tableName);
      queryBean.setSQLQuery(SQLQuery);

      queryBean.setDOMListener(serializer);
      serializer.setOutputStream(new FileOutputStream("Customers.xml"));
      queryBean.processDocument();
    }catch(Exception e){
      System.err.println(e);
    }
  }
}
```

The output from this example is shown in Listing 17-9. You should be able to recognize it as the XML document shown in Figure 17-1, which is created using an Xbean designed to output a JTree.

Listing 17-9: DOM document serialized from the Customer Table

```
<?xml version="1.0"?>
<CUSTOMERS DBNAME="CONTACTS">
    <CUSTOMER CUSTOMER_NUMBER="100">
        <FIRST_NAME>Michael</FIRST_NAME>
        <MI>A</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>123 Pine</STREET>
        <CITY>New York</CITY>
        <STATE>NY        </STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1212</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="101">
        <FIRST_NAME>Fredo</FIRST_NAME>
        <MI>X</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>19 Main</STREET>
```

```
        <CITY>New York</CITY>
        <STATE>NY        </STATE>
        <ZIP>10007</ZIP>
        <PHONE>201-555-1213</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="103">
        <FIRST_NAME>Francis</FIRST_NAME>
        <MI>X</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>17 Main</STREET>
        <CITY>New York</CITY>
        <STATE>NY        </STATE>
        <ZIP>10005</ZIP>
        <PHONE>201-555-1215</PHONE>
    </CUSTOMER>
</CUSTOMERS>
```

Just as you can generate XML from a database, you can also populate a database using an XML data feed. The Internet offers many sources of XML data which can be used in this way. The next section shows the use of an Internet data source to populate a database.

Populating a Database Using XML Data Sources

XML's rise to prominence as a B2B solution has led to the introduction of a number of XML-based services accessible over the Internet. An excellent example of such a service is the XML-based news service on the demonstration portal maintained by `Moreover.com`.

Listing 17-10 illustrates the format of the top stories headline link page available at: `http://www.moreover.com/cgi-local/page?o=xml&query=top+stories.`

The XML document contains the root tag `<moreovernews>`, which, in turn, contains a number of `<article>` elements. Each of these elements contains 11 child elements. Together, the `<article>` elements and their nested child elements can be envisaged as the rows of a database table with 11 columns corresponding to the 11 child elements.

Listing 17-10: XML top stories headline format from Moreover.com

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE moreovernews SYSTEM
"http://p.moreover.com/xml_dtds/moreovernews.dtd">
   <!-- by using this feed you have read and agree to our terms and
conditions
   at http://w.moreover.com/site/about/termsandconditions.html
   If the presence of this comment has caused an error in your parser
you may
   use the older uncommented version by using &amp;o=xml_1 or +xml_1 in
the URL.
   Using the xml_1 version still means that you have read and agree to
```

```
our terms
   and conditions above -->
   <moreovernews>
      <article id="_34226715">
         <url>http://c.moreover.com/click/here.pl?x34226715</url>
         <headline_text>
           Assistant fire chief ascends to departments top role
         </headline_text>
         <source>Springfield News-Leader</source>
         <media_type>text</media_type>
         <cluster>moreover...</cluster>
         <tagline></tagline>
         <document_url>
           http://www.springfieldnews-leader.com/news/
         </document_url>
         <harvest_time>Mar 20 2002 2:30AM</harvest_time>
         <access_registration></access_registration>
         <access_status></access_status>
      </article>
      ...
   </moreovernews>
```

The `SQLInsertBean` used to insert the content of an XML document into a database table is an extension of the basic Xbean base class of Listing 17-2. The `processDocument()` method first calls the `prepareStatement()` method, which creates a `PreparedStatement` to handle the `SQL INSERT` command. The `PreparedStatement` is simply a SQL INSERT command with 11 placeholders, one for each data node in the article element (10 child elements, plus the id attribute).

Cross-Reference

`PreparedStatements` reduce the processing overhead of compiling a SQL statement when it is to be used repetitivley. The advantages of using the `PreparedStatement` object when performing multiple repetitions of a SQL command is discussed in Chapter 4, with a brief example in Chapter 13.

The `getValues()` method is passed an `<article>` element, which it parses to retrieve the id attribute and the child elements. These are returned in a `String` array, which is passed to the `insertHeadline()` method. Note that a Java `null` is specifically inserted into the array for empty child elements. These nulls are converted automatically to `SQL NULL`s on insertion.

The `insertHeadline()` method sets the parameters of the `PreparedStatement` from the `String` array and then calls the `PreparedStatement`'s `executeUpdate()` method to insert the data from the XML document.

After looping through all the <article> elements, the `Connection` object's `close()` method is called to close the connection to the `DataSource`. If you fail to close the connection, the garbage collector will close it for you. Listing 17-11 illustrates the use of a `PreparedStatement` object to insert news headlines into the database.

Listing 17-11: SQLInsertBean

```
package JavaDatabaseBible.ch17.Xbeans;

import java.io.*;
import java.sql.*;
import javax.sql.*;

import org.Xbeans.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
* Parse an xml document and insert into a database table using a
* PreparedStatement
*/
public class SQLInsertBean extends JavaDatabaseBible.ch17.Xbeans.XBean{

  static String moreoverUrl =
         "http://www.moreover.com/cgi-local/page?o=xml&query=top+stories";
  private static String dbUserName = "sa";
  private static String dbPassword = "dba";

  Connection con = null;
  PreparedStatement pstmt = null;

  public SQLInsertBean(){
  }

  public Document processDocument(Document doc) throws XbeansException {
    prepareStatement();
    Element root = doc.getDocumentElement();
    NodeList articles = root.getElementsByTagName("article");
    for(int i=0;i<articles.getLength();i++){
      Element article = (Element)articles.item(i);
      insertHeadline(getValues(article));
    }
    closeConnection();
    return doc;
  }

  private String[] getValues(Element article){
    String[] values = new String[11];
```

```
      values[0] = article.getAttribute("id");
      NodeList dataNodes = article.getChildNodes();
      for(int i=0,j=1;i<dataNodes.getLength();i++){
        Node dataNode = dataNodes.item(i);
        if(dataNode.getNodeType()==Node.ELEMENT_NODE){
          Node textNode = ((Element)dataNode).getFirstChild();
          if(textNode!=null)
            values[j++] = textNode.getNodeValue();
          else
            values[j++] = null;
        }
      }
      return values;
    }

    private void prepareStatement(){
      try {
        Class.forName("com.inet.pool.PoolDriver");
        com.inet.tds.TdsDataSource tds = new com.inet.tds.TdsDataSource();
        tds.setServerName( "MARS" );
        tds.setDatabaseName( "MOREOVERNEWS" );
        tds.setUser( dbUserName );
        tds.setPassword( dbPassword );

        DataSource ds = tds;
        con = ds.getConnection(dbUserName,dbPassword);

        String SQLCmd = "INSERT INTO HEADLINES
  VALUES(?,?,?,?,?,?,?,?,?,?,?);";
        pstmt = con.prepareStatement(SQLCmd);
      }catch(ClassNotFoundException e){
        System.err.println(e.getMessage());
      }catch(SQLException e){
        System.err.println(e.getMessage());
      }
    }

    private void closeConnection(){
      try {
        con.close();
      }catch(SQLException e){
        System.err.println(e.getMessage());
      }
```

```
  }

  private int insertHeadline(String[] values){
    int rowsInserted = -1;
    try {
      for(int i=0;i<values.length;i++){
        pstmt.setString(i+1, fixApostrophes(values[i]));
      }
      rowsInserted = pstmt.executeUpdate();
    }catch(SQLException e){
      System.err.println(e.getMessage());
    }
    return rowsInserted;
  }

  private String fixApostrophes(String in) {
    if(in!=null){
      int n=0;
      while((n=in.indexOf("'",n))>=0){
        in = in.substring(0,n)+"'"+in.substring(n);
        n+=2;
      }
    }
    return in;
  }

  public static void main(String[] args){
    try{
      DOMParserBean parser = new DOMParserBean();
      SQLInsertBean insertBean = new SQLInsertBean();
      SerializerBean serializer = new SerializerBean();

      parser.setUrlString(moreoverUrl);
      parser.setDOMListener(insertBean);
      //insertBean.setDOMListener(serializer);

      parser.processDocument();
    }catch(Exception e){
      System.err.println("Exception in SQLInsertBeanTest");
    }
  }
}
```

The `DOMParserBean` referenced in the example of Listing 17-11 is shown in Listing 17-12. Again, this class is a simple extension of the `XBean` base class. Its `getXml()` method returns a byte array containing an XML document read from a file or a URL or simply passed as a `String`. The `processDocument()` method uses a Xerces `DOMParser` to convert the byte array to a DOM representation of the document. The DOM document is then sent to the `DOMListener` defined in the `XBean` base class, using a `DOMEvent`.

Listing 17-12: DOMParserBean

```
package JavaDatabaseBible.ch17.Xbeans;

import java.io.*;
import java.net.*;
import org.Xbeans.*;
import org.w3c.dom.Document;
import org.xml.sax.InputSource;
import org.apache.xerces.parsers.DOMParser;

public class DOMParserBean extends JavaDatabaseBible.ch17.Xbeans.XBean{
  protected String UrlString = null;
  protected String XmlString = null;
  protected String XmlFileName = null;

  public DOMParserBean(){
  }

  public void setXmlString(String XmlString){
    this.XmlString = XmlString;
  }

  public void setUrlString(String UrlString){
    this.UrlString = UrlString;
  }

  public void setXmlFileName(String XmlFileName){
    this.XmlFileName = XmlFileName;
  }

  public void processDocument(){
    try{
      byte[] xml = getXml();
      if(xml.length > 0){
        ByteArrayInputStream x = new ByteArrayInputStream(xml);
        DOMParser p = new DOMParser();
        InputSource s = new InputSource(x);
```

```
        p.parse(s);
        Document doc = p.getDocument();
        DOMEvent e = new DOMEvent(this,doc);
        DOMListener.documentReady(e);
      }
    }
    catch (Exception e){
      e.printStackTrace();
    }
  }
  private byte[] getXml() {
    byte[] xml = new byte[4096];
    if(XmlFileName!=null){
      File f = new File (XmlFileName);
      xml = new byte[(int)f.length()];
      try {
        FileInputStream is = new FileInputStream(f);
        is.read(xml,0,(int)f.length());
      } catch (Exception e) {
        e.printStackTrace();
      }
    }else if(UrlString!=null){
      URL fileURL = null;
      String buffer = "";
      try {
        fileURL = new URL(UrlString);
        InputStream inputStream = fileURL.openStream();

        int byteCount;
        while((byteCount = inputStream.read(xml))>0){
          buffer += new String(xml,0,byteCount);
        }
        xml = buffer.getBytes();
      }
      catch( Exception e ){
        e.printStackTrace();
      }
    }else{
      if(XmlString!=null) xml = XmlString.getBytes();
    }
    return xml;
  }
}
```

The combination of XML as a transport mechanism and relational databases as a storage medium will become more and more common as Web-based architectures proliferate. The two technologies are complimentary, each offering capabilities not found in the other.

Summary

In this chapter, you learn how to use the DOM to work with XML and JDBC. The main topics covered include the following:

- Using DOM parsers to parse XML documents
- Creating and processing XML documents by using Xbeans
- Creating XML documents by querying a database
- Populating tables from XML data sources

Chapter 18 shows you how to apply this knowledge in a real-world XML application. Also, it discusses JDBC RowSets.

Chapter 18: Using Rowsets to Display Data

In This Chapter

RowSets add significant new capabilities to JDBC by adding JavaBeans support to the JDBC API. Rowsets make it easy to send tabular data over a network. They can also be used as a wrapper, providing scrollable ResultSets or updatable ResultSets when the underlying JDBC driver does not support them.

This chapter discusses RowSets, comparing them with the ResultSets of the JDBC core API and illustrating the features of the different types of RowSets.

Understanding RowSets

A RowSet is an object that contains a set of rows from a ResultSet or some other source of tabular data, like a file or spreadsheet. The RowSet object is an extension of ResultSet, with the added benefit of incorporating JavaBeans support. The RowSet object is supported by the RowSetMetaData interface, which extends the ResultSetMetaData interface.

A RowSet differs significantly from a ResultSet in that it provides a set of JavaBeans properties to connect to a JDBC data source and to read data from the data source for making connections, executing commands, and reading and writing data to and from the data source. These properties include the following:

- `rowSet.setUrl( url );`
- `rowSet.setUsername( login );`
- `rowSet.setPassword( password );`
- `rowSet.getConnection();`
- `rowSet.setCommand("SELECT * FROM sysusers");`
- `rowSet.execute();`

Since RowSets are JavaBeans, notice that they follow the JavaBeans model for setting and getting properties such as the Username and Password. They also follow the JavaBeans API to handle events such as changes in a column value. Being JavaBeans, RowSets use the Java event model to notify listeners when the RowSet is changed.

Rowsets make it easy to send tabular data over a network. They can also be used as a wrapper, providing scrollable ResultSets or updatable ResultSets when the underlying JDBC driver does not support them.

There are two main types of RowSets — connected and disconnected. A connected RowSet, like a ResultSet, maintains a connection to a data source for as long as the RowSet is in use. A disconnected RowSet gets a connection to a data source to load data or to propagate changes back to the data source, but most of the time it does not have a connection open.

While it is disconnected, a RowSet does not need a JDBC driver or the full JDBC API, so its footprint is very small. Since it is not continually connected to its data source, a disconnected RowSet stores its data in memory. It also maintains metadata about the columns it contains and information about its internal state.

Creating and Using a RowSet

The simplest way to explain how a RowSet works is to use an example. Listing 18-1 illustrates the use of a JdbcRowSet to retrieve some names and e-mail addresses from the Contacts Table created as part of the LEDES database in Chapter 2.

The first thing you will notice is that the entire example centers on the methods of the `RowSet`. If you are working with a `ResultSet`, you will have to create and work with the following objects:

- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.ResultSet`

When using a `RowSet` such as the one in Listing 18-1, set the required properties of the `RowSet` itself. Then use the `RowSet.execute()` method to execute the SQL command. The `JdbcRowSet` is implemented as a wrapper around a `ResultSet` object that makes it possible to use the ResultSet as a JavaBeans component. Because a `JdbcRowSet` is a connected `RowSet`, continually maintaining its connection to the database using a JDBC driver, it effectively makes the driver a JavaBeans component.

Listing 18-1: Using a RowSet

```
package JavaDatabaseBible.ch18;

import java.sql.*;
import com.inet.tds.JDBCRowSet;

public class JDBCRowSetExample{

  public static void main(String[] argv){
    String url      = "jdbc:inetdae7:localhost:1433?database=LEDES";
    String login    = "jod";
    String password = "jod";

    try {
      Class.forName("com.inet.tds.TdsDriver").newInstance();

      JDBCRowSet rowSet = new JDBCRowSet();

      //set url,login and password;
      rowSet.setUrl( url );
      rowSet.setUsername( login );
      rowSet.setPassword( password );


      //get the driver version
      DatabaseMetaData dbmd = rowSet.getConnection().getMetaData();
      System.out.println("Driver Name:\t"    + dbmd.getDriverName());
      System.out.println("Driver Version:\t" + dbmd.getDriverVersion());

      //set the sql command
      rowSet.setCommand("SELECT ID,FName,LName,EMail FROM CONTACTS");

      //execute the command
```

```
      rowSet.execute();

      // read the data and put it to the console
      while (rowSet.next()){
        for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
          System.out.print( rowSet.getObject(j)+"\t");
        }
        System.out.println();
      }
      rowSet.close();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

The results of the `JDBCRowSetExample` of Listing 18-1 returns are shown in Table 18-1.

Table 18-1: Results the JDBCRowSetExample Returns

ID	FName	LName	EMail
2001	Oliver	Dewey	o.dewey@dsh.com
2002	Ichabod	Cheatham	i.cheatham@dsh.com
2003	Anne	Howe	a.howe@dsh.com

Making a RowSet Scrollable and Updatable

Among the enhancements of the JDBC Extension API was the ability to make `ResultSets` scrollable and updatable. The same capabilities can be added to a `RowSet` by setting the appropriate properties. Listing 18-2 shows how simply the `RowSet` created in Listing 18-1 can be made scrollable.

Listing 18-2: Making a RowSet scrollable

```
package JavaDatabaseBible.ch18;

import java.sql.*;
import com.inet.tds.JDBCRowSet;

public class JDBCRowSetExample{

  public static void main(String[] argv){
    String url      = "jdbc:inetdae7:localhost:1433?database=LEDES";
    String login    = "jod";
    String password = "jod";

    try {
```

```
      Class.forName("com.inet.tds.TdsDriver").newInstance();

      JDBCRowSet rowSet = new JDBCRowSet();

      //set url,login and password;
      rowSet.setUrl( url );
      rowSet.setUsername( login );
      rowSet.setPassword( password );

      //make the rowset scrollable
      rowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);

      //set the sql command
      rowSet.setCommand("SELECT ID,FName,LName,EMail FROM CONTACTS");

      //execute the command
      rowSet.execute();

      // read the data and put it to the console
      while (rowSet.next()){
        for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
          System.out.print( rowSet.getObject(j)+"\t");
        }
        System.out.println();
      }
      while (rowSet.previous()){
        for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
          System.out.print( rowSet.getObject(j)+"\t");
        }
        System.out.println();
      }
      rowSet.close();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

The output of the example in Listing 18-2 is the same as the output of Listing 18-1, except that the results are printed again in reverse order using the `RowSet.previous()` method to step backwards through the rows. You can use the other cursor-control and scrolling methods inherited from the `ResultSet` in exactly the same way.

In much the same way, you can very easily make a `RowSet` updatable. An updatable `RowSet` allows you to make updates to the values in the `RowSet` itself. These changes are reflected in the database

when the `RowSet.updateRow()` method is called. A `RowSet` is made updatable by setting its concurrency property to `ResultSet.CONCUR_UPDATABLE`. Once you have an updatable `RowSet`, you can insert a new row, delete an existing row, or modify one or more column values.

Since requesting an updatable `RowSet` does not guarantee that you will actually get one, you should check whether the `RowSet` is updatable by using `RowSet.getConcurrency()`. Listing 18-3 illustrates the use of the `rowSet.setConcurrency(ResultSet.CONCUR_UPDATABLE)` method to make a `RowSet` updatable and the use of `RowSet.getConcurrency()` to ensure that the `RowSet` actually is updatable.

Listing 18-3: Making a RowSet updatable

```
package JavaDatabaseBible.ch18;

import java.sql.*;
import com.inet.tds.JDBCRowSet;

public class JDBCUpdatableRowSet{

  public static void main(String[] argv){
    String url      = "jdbc:inetdae7:localhost:1433?database=LEDES";
    String login    = "jod";
    String password = "jod";

    try {
      Class.forName("com.inet.tds.TdsDriver").newInstance();

      JDBCRowSet rowSet = new JDBCRowSet();

      //set url,login and password;
      rowSet.setUrl( url );
      rowSet.setUsername( login );
      rowSet.setPassword( password );

      //make the rowset scrollable and updatable
      rowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
      rowSet.setConcurrency(ResultSet.CONCUR_UPDATABLE);

      //set the sql command
      rowSet.setCommand(
        "SELECT ID,FName,LName,EMail "+
        "FROM CONTACTS WHERE FName = 'Ichabod'");

      //execute the command
      rowSet.execute();
```

```
      // check whether the RowSet can be updated
      if(rowSet.getConcurrency()==ResultSet.CONCUR_UPDATABLE)
        System.out.println("Rowset is UPDATABLE");
      else
        System.out.println("Rowset is READ_ONLY");

      // update the record and output it to the console
      while (rowSet.next()){
        rowSet.updateString("FName", "Igor");
        rowSet.updateRow();
        for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
          System.out.print( rowSet.getObject(j)+"\t");
        }
        System.out.println();
      }
      rowSet.close();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Updating a RowSet

As you can see from the code in Listing 18-3, the use of an updatable `RowSet` is considerably simpler than using the SQL UPDATE command with a conventional `Statement.executeUpdate()`.This is particularly true when you consider that updates made to an updatable `RowSet` always affect the current row, so there is no need to find the row to update. Of course, this does mean that you must make sure you have moved the cursor to the correct row prior to making an update.

`RowSet` updates use the update methods inherited from the `ResultSet`. Most of the `ResultSet.update` methods take two parameters: the column to update and the new value to put in that column. The column may be specified using either the column name or the column number.

Table 18-2 summarizes the update methods for the `ResultSet`, showing only the variant using column name as the specifier for reasons of space.

Table 18-2: ResultSet Update Methods

Data Type	Method
BigDecimal	updateBigDecimal(String columnName, BigDecimal x)
boolean	updateBoolean(String columnName, boolean x)
byte	updateByte(String columnName, byte x)
byte[]	updateBytes(String columnName, byte[] x)
double	updateDouble(String columnName, double x)
float	updateFloat(String columnName, float x)
int	updateInt(String columnName, int x)

Table 18-2: ResultSet Update Methods

Data Type	Method
java.io.InputStream	updateAsciiStream(String columnName, InputStream x, int length)
java.io.InputStream	updateUnicodeStream(String columnName, InputStream x, int length)
java.io.InputStream	updateBinaryStream(String columnName, InputStream x, int length)
java.sql.Date	updateDate(String columnName, Date x)
java.sql.Time	updateTime(String columnName, Time x)
java.sql.Timestamp	updateTimestamp(String columnName, Timestamp x)
long	updateLong(String columnName, long x)
Object	updateObject(String columnName, Object x)
Object	updateObject(String columnName, Object x, int scale)
short	updateShort(String columnName, short x)
String	updateString(String columnName, String x)
NULL	updateNull(String columnName)

Caution

After updating a column value, you must call the `updateRow()` method to make a permanent change in the database before moving the cursor, since changes made using the update methods do not take effect until `updateRow()` is called. If you move the cursor to another row before calling `updateRow()`, the updates will be lost, and the row will revert to its previous column values.

Inserting a New Row

In addition to supporting updates, an updatable `RowSet` supports the insertion and deletion of entire rows. The updatable `RowSet` object inherits from `ResultSet` the *insert row*, which is, in effect, a dedicated row buffer in which you can build a new row.

The new row is created in a manner very similar to the row updates discussed earlier. The following steps are involved:

1. Move the cursor to the insert row by calling the method `moveToInsertRow()`.
2. Set a new value for each column in the row using the appropriate update method.
3. Call the method `insertRow` to insert the new row into the `ResultSet` and, simultaneously, into the database.

Listing 18-4 demonstrates the use of the updatable `RowSet` to insert a new row into a database.

Listing 18-4: Inserting a new row in an updatable RowSet

```
package JavaDatabaseBible.ch18;

import java.sql.*;
import com.inet.tds.JDBCRowSet;

public class JDBCUpdatableRowSetInsert{

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

```
String url      = "jdbc:inetdae7:localhost:1433?database=LEDES";
String login    = "jod";
String password = "jod";

try {
  Class.forName("com.inet.tds.TdsDriver").newInstance();

  JDBCRowSet rowSet = new JDBCRowSet();

  //set url,login and password;
  rowSet.setUrl( url );
  rowSet.setUsername( login );
  rowSet.setPassword( password );

  //make the rowset scrollable and updatable
  rowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
  rowSet.setConcurrency(ResultSet.CONCUR_UPDATABLE);

  //set the sql command
  rowSet.setCommand("SELECT * FROM CONTACTS");

  //execute the command
  rowSet.execute();

  if(rowSet.getConcurrency()==ResultSet.CONCUR_UPDATABLE)
    System.out.println("Rowset is UPDATABLE");
  else
    System.out.println("Rowset is READ_ONLY");

  // move to the Insert Row
  rowSet.moveToInsertRow();

  // update the fields of the Insert Row
  rowSet.updateInt("company_id", 1050);
  rowSet.updateInt("address_info_id", 1004);
  rowSet.updateString("FName", "Nigel");
  rowSet.updateString("LName", "Thornebury");
  rowSet.updateString("phone", "555-456-0123");
  rowSet.updateString("fax",   "555-456-0129");

  // insert the Insert Row into the table
  rowSet.insertRow();
```

```
      // output all rows to the console
      rowSet.beforeFirst();
      while (rowSet.next()){
        for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
          System.out.print( rowSet.getObject(j)+"\t");
        }
        System.out.println();
      }
      rowSet.close();

    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

If you insert a row without supplying a value for every column in the row, the default value for the column will be used if there is one. Otherwise, if the column accepts SQL NULL values, a `NULL` will be inserted. Failing either of those, a `SQLException` will be thrown.

Caution

A `SQLException` will be thrown if a required table column is missing in the updatable `RowSet`, so the query used to get the updatable `RowSet` object should generally select all columns.

Deleting a Row

Deleting a row in an updatable `RowSet` is very simple. All you have to do is move the cursor to the row you want to delete and call the method `deleteRow()`. The example in the following code snippet shows how to delete the third row in a `ResultSet` by moving the cursor to the third row and using the `deleteRow()` method:

```
rowSet.absolute(3);
rowSet.deleteRow();
```

Seeing Changes Made to an Updatable RowSet

Changes made to an updatable `RowSet` are not necessarily visible, either to the `RowSet` itself or to other open transactions. An application can determine if the changes a `ResultSet` makes are visible to the `ResultSet` itself by calling the appropriate `DatabaseMetaData` methods.

One way to get the most recent data from a table is to use the method `refreshRow()`, which gets the latest values for a row straight from the database. This is done by positioning the cursor to the desired row and calling `refreshRow()`, as shown here:

```
rs.absolute(3);
rs.refreshRow();
```

Note

The `RowSet` should be `TYPE_SCROLL_SENSITIVE`; otherwise, `refreshRow()` does nothing.

RowSet Events

A RowSetEvent is generated when something significant happens in a RowSet, such as a change in a column value. Being JavaBeans, RowSets can use the Java event model to notify listeners when the RowSet is changed.

These are the RowSetListener methods:

- rowSetChanged — Called when the rowset is changed, for example, when a SQL command is executed
- rowChanged – Called when a row is inserted, updated, or deleted
- cursorMoved — Called when a rowset's cursor is moved

The example of Listing 18-5 illustrates the use of RowSet events to monitor changes to a RowSet. In this example, a RowSetListener is used to report the insertion of a new row in the Contacts Table. Note the use of the method RowSet.moveToCurrentRow() in the RowSetListener. This method is used to return to the current row when the cursor is on the insertRow. This allows the RowSetListener to report the contents of the row just added.

Listing 18-5: Using RowSet events

```
package JavaDatabaseBible.ch18;

import java.sql.*;
import javax.sql.*;
import com.inet.tds.JDBCRowSet;

public class JDBCUpdatableRowSetInsert{

  public static void main(String[] argv){
    String url      = "jdbc:inetdae7:localhost:1433?database=LEDES";
    String login    = "jod";
    String password = "jod";

    try {
      Class.forName("com.inet.tds.TdsDriver").newInstance();

      JDBCRowSet rowSet = new JDBCRowSet();

      //set url,login and password;
      rowSet.setUrl( url );
      rowSet.setUsername( login );
      rowSet.setPassword( password );

      //make the rowset scrollable and updatable
      rowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
      rowSet.setConcurrency(ResultSet.CONCUR_UPDATABLE);

      // add a RowSetListener
      rowSet.addRowSetListener(new RowSetChangeListener());
```

```
      //set the sql command
      rowSet.setCommand("SELECT * FROM CONTACTS");

      //execute the command
      rowSet.execute();

      // read the data and put it to the console
      rowSet.moveToInsertRow();

      rowSet.updateInt("company_id", 1050);
      rowSet.updateInt("address_info_id", 1004);
      rowSet.updateString("FName", "Nigel");
      rowSet.updateString("LName", "Thornebury");
      rowSet.updateString("phone", "555-456-0123");
      rowSet.updateString("fax",   "555-456-0129");

      rowSet.insertRow();

      //close the RowSet
      rowSet.close();

    } catch(Exception e) {
    }
  }
}
class RowSetChangeListener implements RowSetListener{
  public void rowSetChanged(RowSetEvent event){
  }
  public void rowChanged(RowSetEvent event){
    RowSet rowSet = (RowSet)event.getSource();
    try{
      rowSet.moveToCurrentRow();
      for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
        System.out.print( rowSet.getObject(j)+"\t");
      }
      System.out.println();
    }catch(Exception e){
    }
  }
  public void cursorMoved(RowSetEvent event){
    RowSet rowSet = (RowSet)event.getSource();
    try{
```

```
      System.out.println("cursor moved to row "+rowSet.getRow());
    }catch(Exception e){
    }
  }
}
```

One of the neatest features of `RowSets` is that, in addition to using them like `ResultSets`, as discussed in this section, you can also use them as data containers disconnected from the database. The use of disconnected `RowSets` is discussed in the next section.

Disconnected RowSets

As mentioned at the beginning of this chapter, there are two main types of `RowSets`: connected and disconnected. Connected `RowSets` maintain a connection for as long as the `RowSet` is in use. Disconnected `RowSets` get a connection to a data source as needed.

Sun released a number of `RowSet` implementations as an early access release. These implementations include the following:

- `CachedRowSet`
- `JdbcRowSet`
- `WebRowSet`

The `CachedRowSet` is a disconnected `RowSet`, intended for use as a means of caching a `ResultSet` object's rows in memory, so it doesn't require the continuous use of a database connection. All `CachedRowSets` are scrollable and updatable, and, just like any other JavaBean, they can be serialized. This provides a means of serializing `ResultSets` and sending them to remote clients to be updated and sent back to the server.

Being disconnected means that a `CachedRowSet` connects to its data source only while it is reading data to load rows and while it is sending changes back to its underlying database. The rest of the time, it is disconnected, even while changes are being made to it. In effect, a `CachedRowSet` object can be thought of as simply a disconnected set of rows cached in a JavaBean.

Being thin and serializable, a `CachedRowSet` can easily be sent across a wire, and it is well suited to sending data to a thin client, such as a PDA, because, while a `CachedRowSet` object is disconnected, it can be much leaner than a `ResultSet` object with the same data. The `CachedRowSet` class provides a means of working with the rows of a ResultSet without the overhead associated with using the full JDBC API.

Updating a `CachedRowSet` object is similar to updating a `JDBCRowSet`, but because the rowset is not connected to its data source while it is being updated, one extra step is required to make a change in the underlying data source. After calling the method `updateRow()` or `insertRow()`, a `CachedRowSet` object must also call the method `acceptChanges()`in order to make a connection and write the updates to the data source.

The Sun implementation of the `CachedRowSet` also requires that you specifically set the name of the table you are working with. If you fail to do this, any attempts to update the table will throw a SQL exception.

Note

After making changes to a `CachedRowSet` using `updateRow()` or `insertRow()`, you must also call `acceptChanges()`to make a connection and write the updates to the data source. When you are changing or inserting several rows, you need only call `acceptChanges()` once after all calls to `updateRow()` and `insertRow()` have been made.

Using a CachedRowSet with a PDA

Since `CachedRowSets` are JavaBeans, they can be serialized just like any other JavaBean This makes them very useful when working with a remote client, such as a PDA.

You may recall that the database design examples in Chapter 2 revolve around a database for managing and invoicing projects or, as lawyers prefer to call them, "matters" for a law firm. A part of this example included the creation of a number of tables to handle contact information. It would obviously be useful for the employees of the firm to have a copy of the contact names and addresses in their PDAs. An elegant way to do this is through the use of a `CachedRowSet`, since a `CachedRowSet` only needs to connect to its data source while it is reading or updating data.

The first thing to consider in creating the contact list is the SQL query required to build the `RowSet`. The contact information required for the `RowSet` is distributed across the three following tables because of the nature of the application:

- The client is frequently a corporation, represented by several individuals.
- A corporation may operate out of a number of different locations.
- Each individual may have a different phone number, mail drop, or cell phone, but all may have the same mailing address.

Since one of the principles of database design is to avoid storing the same item of information in two or more places, this means that the information about a client has to be divided among a number of different tables. The structure and relationships of these tables is shown in Figure 18-1.

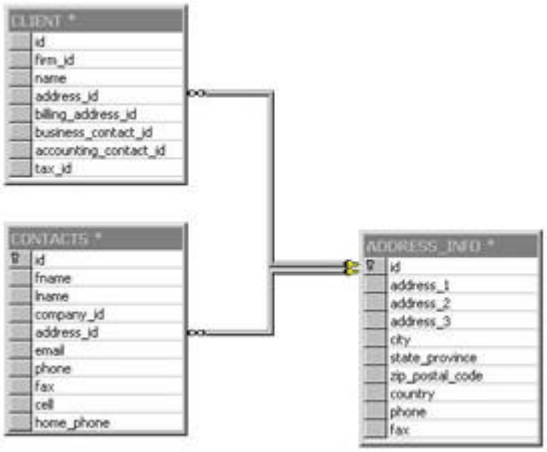


Figure 18-1: Tables containg contact information

Server-side code

The best way to retrieve the required contact information is to define a SQL stored procedure called `GET_CONTACT_LIST` and to call it to get the data. As you can see from Listing 18-6, the stored procedure is relatively simple.

Listing 18-6: Stored procedure to retrieve contact data

```
CREATE PROCEDURE GET_CONTACT_LIST AS
SELECT c.fName, c.lName, f.name AS firm, a.address_1 as street, a.city,
a.state_province AS state, a.zip_postal_code AS zip, c.phone
FROM CONTACTS c, Address_info a, clients f
WHERE f.address_id = a.id AND c.company_id = f.id
```

Table 18-3 shows the `ResultSet` returned by the query. Obviously, you can include the contact's cell phone number, e-mail address, and so on.

Table 18-3: Contact List RowSet

fName	lName	firm	street	city	state	zip	phone
Oliver	Dewey	Dewey,Cheatham and Howe	123 Penny Lane	New York	NY	10006	555-123-4567
Ichabod	Cheatham	Dewey,Cheatham and Howe	123 Penny Lane	New York	NY	10006	555-123-4568
Anne	Howe	Dewey,Cheatham and Howe	123 Penny Lane	New York	NY	10006	555-123-4569
Michael	West	Acme Insurance	211 Elm St	New York	NY	10007	555-213-2346
James	Nateland	Acme Insurance	211 Elm St	New York	NY	10007	555-213-2347
Bob	Guppy	Nigel Watson and Sons	17 Main St	New York	NY	10007	555-213-1114
Nigel	Watson	Nigel Watson and Sons	17 Main St	New York	NY	10007	555-213-1115
Seamus	Maloney	Maloney's Pizza Pub, Inc	211 Pine St	New York	NY	10007	555-233-3335

To retrieve the data as a `CachedRowSet`, create a `CachedRowSet` object and use it to execute the stored procedure. The example shown in Listing 18-7 uses the `CachedRowSet` implementation from the Sun rowset jar file, loading it using the `jdbc:odbc` bridge. As you can see, the code is very similar to the `JDBCRowSet` examples, with the exception that rather than displaying the `CachedRowSet`, the `serializeRows()` method serializes the entire `CachedRowSet` bean to a file.

Listing 18-7: Executing a SQL query in a CachedRowSet

```
package JavaDatabaseBible.ch18;

import java.io.*;
import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

public class CachedRowSetSerializer{
  String fName = "ContactRowSet.ser";
  public static void main(String[] argv){
    CachedRowSetSerializer crs = new CachedRowSetSerializer();
    crs.serializeRows();
```

```
  }
  public CachedRowSetSerializer(){
  }
  public void serializeRows(){
    String url      = "jdbc:odbc:LEDES";
    String login    = "jod";
    String password = "jod";
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      CachedRowSet rowSet = new CachedRowSet();

      //set url,login and password;
      rowSet.setUrl( url );
      rowSet.setUsername( login );
      rowSet.setPassword( password );

      //make the rowset scrollable and updatable
      rowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
      rowSet.setConcurrency(ResultSet.CONCUR_UPDATABLE);

      //set the sql command
      rowSet.setCommand("GET_CONTACT_LIST");

      //execute the command
      rowSet.execute();

      FileOutputStream fOut = new FileOutputStream(fName);
      ObjectOutput out = new ObjectOutputStream(fOut);
      out.writeObject(rowSet);
      out.flush();
      out.close();

      //close the RowSet
      rowSet.close();
    }catch(Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```


Client-side code

The code at the client side is even simpler. The serialized `CachedRowSet` bean is deserialized, and the `RowSets` are output to the console, as shown in Listing 18-8. A practical application would probably use the `CachedRowSet` bean as a component driving a simple GUI.

Listing 18-8: Using a CachedRowSet

```
package JavaDatabaseBible.ch18;

import java.io.*;
import sun.jdbc.rowset.*;

public class CachedRowSetDeserializer{
  public static void main(String[] argv){
    CachedRowSetDeserializer crd = new CachedRowSetDeserializer();
    crd.deserializeRows(argv[0]);
  }

  public CachedRowSetDeserializer(){
  }

  public void deserializeRows(String fName){
    try {
      FileInputStream fIn = new FileInputStream(fName);
      ObjectInputStream in = new ObjectInputStream(fIn);
      CachedRowSet rowSet = (CachedRowSet)in.readObject();

      while(rowSet.next()){
        for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
          System.out.print( rowSet.getObject(j)+"\t");
        }
        System.out.println();
      }
      rowSet.close();

    }catch(Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Despite its name, the distinguishing feature of the WebRowSet is that it is designed to serialize and deserialize itself in XML. The details of how it works are discussed in the next section.

Generating XML from a RowSet

The designers of the `RowSet` object realized that `RowSets` had the potential to be very useful in XML applications. One of the sample implementations in Sun's rowset jar is the `WebRowSet`. The `WebRowSet` is an extension of the `CachedRowSet` designed to serialize and deserialize a `RowSet` in XML format. The class stores an `XmlReader` object that it uses to read a `RowSet` in XML format and an `XmlWriter` object that it uses to write a `RowSet` in XML format.

Chapter 2 discusses the design of a database intended to implement an XML-based billing standard known as LEDES 2000 — the Legal Electronic Data Exchange Standard. Since the database structure is quite complex, it is not discussed here. Suffice it to say that the query required to retrieve information about billable items to be inserted in an invoice is defined in the stored procedure called `Itemise_Fees`, shown in Listing 18-9.

Listing 18-9: Stored procedure to retrieve billable item data

```
CREATE PROCEDURE ITEMISE_FEES @Matter_Id INT AS
SELECT bi.date AS charge_date, bi.tk_id,
    bi.description AS charge_desc, bi.task_code AS acca_task,
    bi.activity_code AS acca_activity, rc.value AS charge_type,
    bi.units, fc.rate, bi.units * fc.rate AS base_amount,
    dc.value AS discount_type, br.discount AS discount_percent,
    (bi.units * fc.rate) * (1 - br.discount / 100)
    AS total_amount
FROM Billings b, Billable_Items bi, Billing_Rates br,
    Discount_Codes dc, Timekeeper tk, Fee_Codes fc,
    Rate_Codes rc
WHERE b.matter_id = bi.matter_id AND
    dc.code = br.discount_type AND bi.rate_code = br.id AND
    fc.code = tk.rate_code AND tk.id = bi.tk_id AND
    rc.code = bi.rate_code AND bi.invoice_number IS NULL AND
    b.status = 1 AND bi.matter_id = @Matter_Id;
```

Listing 18-10 illustrates the use of the stored procedure to retrieve billable items for Matter_Id 10001. As you can see, the XML is written to a file called `fees.xml`.

Listing 18-10: Writing XML with a WebRowSet

```
package JavaDatabaseBible.ch18;

import java.io.*;
import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

public class WebRowSetExample{

  public static void main(String[] argv){
    String url      = "jdbc:odbc:LEDES";
    String login    = "jod";
    String password = "jod";

    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
      WebRowSet rowSet = new WebRowSet();

      //set url,login and password;
      rowSet.setUrl( url );
      rowSet.setUsername( login );
      rowSet.setPassword( password );

      //set the sql command
      rowSet.setCommand("Itemise_Fees 10001;");

      //execute the command
      rowSet.execute();

      // write the RowSet as XML
      FileWriter xmlFileWriter = new FileWriter("fees.xml");
      rowSet.writeXml(xmlFileWriter);

      //close the RowSet
      rowSet.close();

    } catch(Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Listing 18-11 shows the format of the XML generated by the `WebRowSet`. Since the `WebRowSet` is actually serialized in XML format, it contains not only the row data but all the associated metadata required to reconstruct the `RowSet` in its entirety.

Listing 18-11: XML generated by WebRowSet

```
<?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>Itemise_Fees 10001;</command>
    <concurrency>1008</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:odbc:LEDES</url>
  </properties>
  <metadata>
    <column-count>12</column-count>
    <column-definition>
      <column-index>1</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>0</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
      <column-display-size>23</column-display-size>
      <column-label>charge_date</column-label>
      <column-name>charge_date</column-name>
      <schema-name></schema-name>
      <column-precision>23</column-precision>
      <column-scale>3</column-scale>
      <table-name></table-name>
      <catalog-name></catalog-name>
      <column-type>93</column-type>
      <column-type-name>datetime</column-type-name>
    </column-definition>
    <-- the next 11 column-definition elements have been removed -->
   </metadata>
  <data>
    <row>
      <col>1018929600000</col>
      <col>1001</col>
      <col>Replace File</col>
      <col>L140</col>
      <col>A110</col>
```

```
      <col>F</col>
      <col>0.1</col>
      <col>400.0</col>
      <col>40.0</col>
      <col>percent</col>
      <col>12.5</col>
      <col>35.0</col>
    </row>
  </data>
</RowSet>
```

The XML file generated by the `WebRowSet` object is divided into the three following main sections:

- <property> — This section contains all the property data associated with the `WebRowSet` bean.
- <metadata> — This section contains a metadata element describing each of the columns.
- <data> — This section contains the actual data.

Note that only one column-definition element is shown in Listing 18-11, since all 12 are very similar.

As it stands, this XML output does not meet the requirements of the LEDES 2000 specification. There are three obvious ways to deal with this:

- Apply an XSL transform to generate the desired XML from the `WebRowSet` XML.
- Write a custom XmlWriter to generate the desired XML directly.
- Generate an XML document directly from a `CachedRowSet`.

The approach used for this example is to generate the XML directly from a `CachedRowSet`. The reason for this is that the `WebRowSet` XML, designed to serialize the entire bean, contains far more data than is needed for the application. Moreover, the data is organized in such a way as to make reconstitution of the `RowSet` easy rather than to make it suitable for this application.

Using an XSL transform is a very heavyweight solution. Similarly, writing a custom XmlWriter is much more complex than just writing the required XML for the application; unless you design it to write out all the data required to serialize the bean and support it with a corresponding XmlReader, it negates the advantages of XML serialization without giving you any obvious benefits.

Since the XML is intended for transmission to the client as a file, it is generated by writing strings to an `OutputStream`, as shown in Listing 18-12. If you intend to do any additional processing, you can use a Xerces `Document` object and build it as a DOM.

Listing 18-12: Generating XML using a CachedRowSet

```
package JavaDatabaseBible.ch18;

import java.io.*;
import sun.jdbc.rowset.*;

public class CachedRowSetToXML{

  public static void main(String[] argv){
    String url      = "jdbc:odbc:LEDES";
    String login    = "jod";
    String password = "jod";
```

```
      String fName    = "fees.xml";
      try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        CachedRowSet rowSet = new CachedRowSet();

        PrintWriter out = new PrintWriter(new FileOutputStream(fName));

        //set url,login and password;
        rowSet.setUrl( url );
        rowSet.setUsername( login );
        rowSet.setPassword( password );

        //set the sql command
        rowSet.setCommand("Itemise_Fees 10001;");

        //execute the command
        rowSet.execute();
        out.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
        out.println("  <matter>");

        // read the data and put it to the console
        while(rowSet.next()){
          out.println("    <fee>");
          for(int j=1; j<=rowSet.getMetaData().getColumnCount(); j++){
            out.print("     <"+
                      rowSet.getMetaData().getColumnLabel(j)+">");
            out.print( rowSet.getObject(j));
            out.println( "</"+rowSet.getMetaData().getColumnLabel(j)+">");
          }
          out.println("    </fee>");
        }
        out.println("  </matter>");

        //close the RowSet
        rowSet.close();
        out.close();
      } catch(Exception e) {
       e.printStackTrace();
      }
     }
  }
```

The XML generated by this example is much closer to the LEDES 2000 specification, as you can see from Listing 18-13. It still needs a little cleanup, particularly in terms of tidying up the date String. The rest of the LEDES 2000 invoice document can be generated in much the same way.

Listing 18-13: XML invoice elements

```
<?xml version="1.0" encoding="UTF-8"?>
  <matter>
    <fee>
      <charge_date>2002-04-16 00:00:00.0</charge_date>
      <tk_id>1001</tk_id>
      <charge_desc>Replace File</charge_desc>
      <acca_task>L140</acca_task>
      <acca_activity>A110</acca_activity>
      <charge_type>F</charge_type>
      <units>0.1</units>
      <rate>400.0</rate>
      <base_amount>40.0</base_amount>
      <discount_type>percent</discount_type>
      <discount_percent>12.5</discount_percent>
      <total_amount>35.0</total_amount>
    </fee>
    <fee>
      <charge_date>2002-04-12 00:00:00.0</charge_date>
      <tk_id>1000</tk_id>
      <charge_desc>Review File</charge_desc>
      <acca_task>L110</acca_task>
      <acca_activity>A101</acca_activity>
      <charge_type>U</charge_type>
      <units>1.5</units>
      <rate>600.0</rate>
      <base_amount>900.0</base_amount>
      <discount_type>percent</discount_type>
      <discount_percent>15.0</discount_percent>
      <total_amount>765.0</total_amount>
    </fee>
    <fee>
      <charge_date>2002-04-15 00:00:00.0</charge_date>
      <tk_id>1000</tk_id>
      <charge_desc>Analyse Case</charge_desc>
      <acca_task>L120</acca_task>
      <acca_activity>A104</acca_activity>
      <charge_type>U</charge_type>
      <units>1.0</units>
```

```
      <rate>600.0</rate>
      <base_amount>600.0</base_amount>
      <discount_type>percent</discount_type>
      <discount_percent>15.0</discount_percent>
      <total_amount>510.0</total_amount>
    </fee>
    <fee>
      <charge_date>2002-04-15 00:00:00.0</charge_date>
      <tk_id>1000</tk_id>
      <charge_desc>Consult Expert</charge_desc>
      <acca_task>L130</acca_task>
      <acca_activity>A108</acca_activity>
      <charge_type>U</charge_type>
      <units>2.0</units>
      <rate>600.0</rate>
      <base_amount>1200.0</base_amount>
      <discount_type>percent</discount_type>
      <discount_percent>15.0</discount_percent>
      <total_amount>1020.0</total_amount>
    </fee>
  </matter>
```

Summary

This chapter compared `RowSets` with `ResultSets` and discussed some of the advantages of disconnected `RowSets`. Among the topics discussed are the following:

- Using `RowSets` to add functionality to `ResultSets`
- Using disconnected `RowSets` to transfer data between devices
- Generating XML from a `RowSet`

Chapter 19 shows you how to create a simple JDBC driver for XML documents. This driver allows you to create and query XML documents using SQL.

Chapter 19: Accessing XML Documents Using SQL

In This Chapter

XML (or eXtensible Markup Language) has become increasingly popular for a variety of applications ranging from platform-independent data transfer, exemplified by the legal invoicing example illustrated in Chapters 11 and 18, to use in configuration files such as the web.xml file used by the Tomcat server. XML documents are in many ways similar to the HTML documents familiar from Web applications.

The primary difference between XML and HTML is that XML documents are based on user-defined tags, whereas HTML tags are predefined for use by the browser. An important secondary difference is that XML documents must be well formed in order to be machine readable.

To be well formed, a document must follow a few simple rules. The most important of these are that all tags must be properly closed, and that when tags are nested they must be nested correctly. A properly closed tag is a tag that either has a closing tag after its contents, or is self-closing. The following code snippet shows examples of two properly closed tags:

```
<text>Some text</text>
<element attrib="value"/>
```

Proper nesting requires that nested tags be closed in the opposite order to the order in which they were opened. In the example below, the `nested` element is nested inside the `tag` element, and is closed before the `tag` element is closed:

```
<tag>
  <nested>
  </nested>
</tag>
```

These rules are similar to the rules a programmer is used to following when using braces or parentheses. However, it is important to realize that unlike HTML, which lets you get away with breaking these rules, the XML parser requires that the rules be obeyed.

By enforcing the basic rules of well-formed documents, XML defines a structure which can be parsed very easily with no knowledge of the content of a document. HTML parsers, on the other hand, can handle ill formed documents because a knowledge of the meanings of the HTML tags is built into the parser.

Because XML documents are well formed, they can have an inherently tabular structure, which makes them ideal for representing data tables. This chapter explores the design of a simple JDBC driver that exploits this structure to use XML documents as the data storage element of a simple database.

Reasons for Accessing XML Documents with SQL

Although the primary use of XML is to provide a platform-independent way to structure data for transfer between applications, an important secondary use of XML is for local data storage. Common examples include the following:

- XML as a replacement for properties files or INI files
- XML as a replacement for comma-delimited CSV files in text databases
- XML as a small, downloadable database for the delivery of stock quotes or news headlines

In some instances, an XML document, being a data repository, can be a database in itself. For example, the contact lists on my Linux-based PDA are saved as XML documents.

Since the data in an XML file is stored in two different node types, there are two obvious ways to set up a database using an XML file:

- Store each record as an element, with the field data in attributes.

- Store each record as an element, with field data in child elements.

The advantage of using attributes is that the XML file is shorter, since the attribute name occurs only once. If you store the data in an element, the name occurs twice: once in opening the element and once in closing it. The attribute-based approach is shown here:

```
<CUSTOMER FIRST_NAME=" Michael" MI="A" LAST_NAME="Corleone" STREET="123
Pine"/>
```

The alternative approach, which uses child elements for each data item, is more verbose but more structured, as shown in Listing 19-1.

Listing 19-1: Customer data record in XML

```
<?xml version="1.0"?>
<CUSTOMERS>
    <CUSTOMER CUSTOMER_NUMBER="100">
        <FIRST_NAME>Michael</FIRST_NAME>
        <MI>A</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>123 Pine</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1212</PHONE>
    </CUSTOMER>
</CUSTOMERS>
```


Clearly, as exemplified by the `CUSTOMER_NUMBER` field in Listing 19-1, you can also use a combination of these two approaches. There is no "best" way. My Linux PDA uses the attribute-oriented approach, presumably to save space. Most XML documents used as `INI` files use the element-based approach, presumably for readability.

The JDBC driver described in this chapter supports the insertion of data as an attribute by defining a custom data type: `ATTRIBUTE`. Other data types are always inserted as child elements. In practice, there can really only be one other type: `VARCHAR`, since all data in an XML document is represented as a `String`. The details of the JDBC driver are discussed in the next section.

Building a JDBC-accessible XML DBMS

There are two main components required to build an XML database system incorporating a JDBC programming interface: JDBC driver classes and the SQL engine.

In designing the JDBC API, Sun foresaw the need for implementations of a subset intended for lightweight databases that would not provide full support for the API and SQL 92 Entry Level. The method `jdbcCompliant()` is defined in the `java.sql.Driver` interface, specifically to indicate compliance or noncompliance with the standard.

Although building a highly efficient, fully compliant JDBC driver is a significant undertaking, implementing a useful subset is a much simpler task.

The Implementation Base Classes

JDBC was designed to be a rich API. In other words, there are dedicated methods to handle anything you might want to do. These methods are specified in a set of interface classes. To create a JDBC driver that can be registered with the `DriverManager`, all of these interface methods must be implemented. This is done using implementation base classes.

The implementation base classes implement all the methods defined in the interfaces in a minimal form. These methods simply throw an exception when called. The JDBC driver classes are simply extensions of the implementation base classes, which override the methods required to get the job done. The first few lines of one of these implementation classes is shown in Listing 19-2 to give you the idea.

Listing 19-2: Typical implementation base class

```
package JavaDatabaseBible.ch19.JDBCImpl;

import java.sql.*;

public class JDBCStatementImpl implements java.sql.Statement {

  public JDBCStatementImpl() {
  }

  public void setFetchSize(int fetchSize) throws SQLException {
    throw new SQLException("not supported");
  }

  public int getFetchSize() throws SQLException {
    throw new SQLException("not supported");
  }
```

If you don't feel like typing literally hundreds of methods that only throw exceptions, you can download the implementation classes from the Web site. Alternatively, if your application doesn't need to register the driver with the `java.sql.DriverManager`, you can simply remove the `extends` clause from the class definitions.

Our SQL engine is also limited, but it, too, is expandable. The SQL engine handles the basic, generic parsing of the SQL commands themselves. This code is applicable to any SQL application. Our application implements only a subset of possible commands, allowing us to process common queries as well as document creation and updating.

Our XML document handlers extend the basic SQL engine classes to provide XML-specific data access and update capabilities. Obviously, if you want to create your own storage-medium handlers, perhaps for use with arrays rather than as XML documents, you can simply plug them in place of ours.

Implementing the JDBC Classes

The inner workings of JDBC have been discussed at some length in earlier chapters — in particular, in Chapter 4. The following brief overview discusses how the classes work together.

The XMLDriver class

The function of the `DriverManager` is to provide basic services for managing JDBC drivers. Drivers can be loaded either during initialization or on request, using `Class.forName()`. All drivers contain a static initializer, as specified in Sun's JDBC API guide, that creates an instance of the driver and

registers the newly created instance with the `DriverManager`. Listing 19-3 shows how simple the `XMLDriver` is.

Listing 19-3: XMLDriver class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.sql.*;
import java.util.Properties;
import JavaDatabaseBible.ch19.JDBCImpl.JDBCDriverImpl;

public class XMLDriver extends JDBCDriverImpl{
  protected XMLConnection con;

  static {
    try {
      java.sql.DriverManager.registerDriver(new XMLDriver());
    }catch(SQLException e) {
      System.err.println(e);
    }
  }

  public XMLDriver() {
  }

  public boolean acceptsURL(String url) throws SQLException {
    return url.endsWith(".xml");
  }

  public Connection connect(String url, Properties info)
                            throws SQLException {
    con = new XMLConnection(url);
    return con;
  }
}
```

In addition to loading JDBC drivers, the `DriverManager` attempts to locate a suitable driver for the specified URL and returns a connection to the appropriate driver. It does this by polling the registered drivers' `acceptsURL(String url)` methods.

The `DriverManager` is also responsible for getting a `java.sql.Connection` to the database. It does this by calling the driver's `connect()` method, passing the `URL` for the database. The driver then creates a `Connection` object and returns it to the `DriverManager`.

The XMLConnection class

A `java.sql.Connection` represents a session with a specific database or, in this case, with a specific XML document. The XML document is defined by the URL passed to the `Connection` object. The `Connection` object now attempts to connect to the URL. Depending on the URL protocol, this may be done in one of the following ways:

- If the URL protocol indicates that the document is a file, the `Connection` attempts to open the file.
- If the URL protocol indicates an HTTP connection, the `Connection` attempts to connect to the URL and to open the XML document that way.

Once a connection to an existing file or to an HTTP data source has been established, the XML document is parsed to a DOM document. In the case of a file URL where the file does not exist, a new DOM document is created. Listing 19-4 illustrates the `XMLConnection` class.

Listing 19-4: XMLConnection class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.io.*;
import java.net.*;
import java.sql.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.xml.sax.InputSource;
import org.apache.xerces.parsers.DOMParser;
import org.apache.xerces.dom.DocumentImpl;
import JavaDatabaseBible.ch19.JDBCImpl.JDBCConnectionImpl;

/**
* Set up interface to xml document & create XMLStatement on request
*/

public class XMLConnection extends JDBCConnectionImpl{
  private URL url;
  private Document xmlDoc;

  public XMLConnection() throws SQLException{
  }

  public XMLConnection(InputSource xml) throws SQLException{
    try{
      DOMParser p = new DOMParser();
      p.parse(xml);
      xmlDoc = p.getDocument();
    }
    catch (Exception e){
      System.err.println(e);
    }
```

```
}

public XMLConnection(URL url) throws SQLException{
  this.url = url;
  xmlDoc = setXmlDoc(url);
}

public XMLConnection(String UrlString) throws SQLException{
  try{
    xmlDoc = setXmlDoc(new URL(UrlString));
  }
  catch (Exception e){
    System.err.println(e);
  }
}

private Document setXmlDoc(URL url){
  Document xmlDoc = null;

  // if the URL points to a file, and the file does not exist,
  // create a new DOM document and return
  if(url.getProtocol().equalsIgnoreCase("file")){
    File f = new File(url.getFile());

    if(!f.exists()){
      String rootTag = f.getName();
      rootTag =
        rootTag.substring(rootTag.lastIndexOf("/")+1,
                          rootTag.indexOf("."));
      return createXmlDoc(rootTag);
    }
  }
  // otherwise parse the file to the DOM document
  try{
    InputStream s = url.openStream();
    if(s!=null){
      DOMParser p = new DOMParser();
      p.parse(new InputSource(s));
      xmlDoc = p.getDocument();
    }
  }catch (Exception e){
    System.err.println(".."+e);
  }
```

```
    return xmlDoc;
  }

  // create an XML Document with the specified root tag
  private Document createXmlDoc(String rootTag){
    xmlDoc = new DocumentImpl();
    Element root = (Element)xmlDoc.createElement(rootTag);
    xmlDoc.appendChild (root);
    return xmlDoc;
  }

  // methods to return an XMLStatement
  public Statement createStatement(){
    return new XMLStatement(xmlDoc);
  }

  public XMLStatement createStatement(URL url){
    return new XMLStatement(xmlDoc);
  }
}
```

In addition to connecting to a data source, the `Connection` object is also responsible for returning a `Statement` object when its `createStatement()` method is called. The `createStatement()` method creates a new `Statement` object, passing it the DOM document contained in the `Connection`.

The XMLStatement class

The `java.sql.Statement` object acts as a top-level command interpreter for `execute()` and `executeQuery()` methods. In this implementation, the simpler `CREATE` and `INSERT` commands are handled locally, and queries are handled by an `XMLQuery` object.

The primary methods of the `Statement` object are:

- `public ResultSet executeQuery(String sqlQuery) - executeQuery()` creates a new `XMLQuery` object, using it to process the SQL query. The `XMLQuery` object is illustrated in Listing 19-10.
- `public int executeUpdate(String sqlString) – executeUpdate()` creates a new `XMLCommand` object, passing it to either the createTable() method or the insert() method.
- `private boolean createTable(XMLCommand sql) – createTable()` uses the `splitColumns()` method of `XMLCommand` to return the column list as a `Vector` of columns, which it then uses to create the table. The table is defined in the Vectors `columnNameVector` and `columnTypeVector`.
- `private boolean insert(XMLCommand sql) - insert()` uses `columnNameVector` and `columnTypeVector` to create an XML element for each data field, nesting them inside a row element representing the inserted row.

Note the reference to a custom data type: `ATTRIBUTE`, which specifies that the data be added as an attribute. The `XMLStatement` object handles `SQL INSERT` commands in one of two ways. If the data type is `ATTRIBUTE`, the data String is inserted into the XML element as an attribute. Otherwise, it is appended as an element. Listing 19-5 shows the code for the `XMLStatement` object.

Listing 19-5: XMLStatement class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.util.Vector;
import java.sql.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;
import org.xml.sax.InputSource;
import JavaDatabaseBible.ch19.JDBCImpl.JDBCStatementImpl;

public class XMLStatement extends JDBCStatementImpl{
  private InputSource xml;
  private Document xmlDoc;
  private Vector columnNameVector = new Vector();
  private Vector columnTypeVector = new Vector();

  public XMLStatement(){
  }

  public XMLStatement(InputSource xml){
    this.xml = xml;
  }

  public XMLStatement(Document xmlDoc){
    this.xmlDoc = xmlDoc;
  }

  public ResultSet executeQuery(String sqlQuery) throws SQLException{
    XMLQuery query = new XMLQuery(sqlQuery);
    return query.processDoc(xmlDoc);
  }

  public int executeUpdate(String sqlString){
    XMLCommand sql = new XMLCommand(sqlString);
    if(sql.cmd.equals("CREATE")){
      createTable(sql);
    }
    if(sql.cmd.equals("INSERT")){
      insert(sql);
    }
```

```
    return 0;
  }

  private boolean createTable(XMLCommand sql){
    Vector columnVector = sql.splitColumns(sql.columns);
    for(int i=0;i<columnVector.size();i++){
      String columnDef = ((String)columnVector.elementAt(i)).trim();
      int space = columnDef.indexOf(" ");
      if(space>=0){
        String colName = columnDef.substring(0,space);
        String colType = columnDef.substring(space+1);
        columnNameVector.addElement(colName);
        columnTypeVector.addElement(colType);
      }
    }
    return true;
  }

  private void initColumnData(XMLCommand sql){
    NodeList records = xmlDoc.getElementsByTagName(sql.tableName);
    Element record = (Element)records.item(0);
    NamedNodeMap attribs = record.getAttributes();
    for(int i=0;i<attribs.getLength();i++){
      Node n = attribs.item(i);
      if(n.getNodeType()==Node.ATTRIBUTE_NODE){
        columnNameVector.addElement(n.getNodeName());
        columnTypeVector.addElement("ATTRIBUTE");
      }
    }
    NodeList fields = record.getChildNodes();
    for(int i=0;i<fields.getLength();i++){
      Node n = fields.item(i);
      if(n.getNodeType()==Node.ELEMENT_NODE){
        Element field = (Element)n;
        columnNameVector.addElement(field.getTagName());
        columnTypeVector.addElement("VARCHAR");
      }
    }
  }

  private boolean insert(XMLCommand sql){
    if(columnNameVector.isEmpty())initColumnData(sql);
    Vector data = sql.splitValues(sql.values);
```

```
    try{
      Element root = xmlDoc.getDocumentElement();
      Element row = (Element)xmlDoc.createElement(sql.tableName);
      root.appendChild(row);
      for(int i=0;i<data.size();i++){
        String cName = (String)columnNameVector.elementAt(i);
        String cType = (String)columnTypeVector.elementAt(i);
        String cData = (String)data.elementAt(i);
        if(cData.startsWith("'")&&cData.endsWith("'")){
          if(cData.length()>1){
            cData=cData.substring(1,cData.length()-1);
          }
        }
        if(cType.equals("ATTRIBUTE")){
          row.setAttribute(cName,cData);
        }else{
          Element column = xmlDoc.createElement(cName);
          row.appendChild(column);
          column.appendChild(xmlDoc.createTextNode(cData));
        }
      }
    }catch(Exception e){
      System.err.println("Insert error: "+e);
    }
    return true;
  }

  public Document getXmlDocument(){
    return xmlDoc;
  }
}
```

The XMLQuery object returns an `XMLResultSet`, which implements `java.sql.ResultSet`. The `ResultSet` is a container for the data the query returns. Since this application deals with XML, the `ResultSet` is maintained as a DOM document containing the specific elements and child elements requested.

The XMLResultSet

The most important methods of `XMLResultSet` are `next()` and `getString()`. The `next()` method uses the `NodeList` rows to iterate through the nodes making up the `ResultSet`. When `next()` is first called, it initializes the `NodeList` from the `ResultSet` document. It then maintains a cursor pointing to the current row in the `int rowIndex`.

The `getString()` method shown in Listing 19-6 is implemented using only the column-name variant. There are two reasons for this:

- Since `XMLResultSet` is intended to support data stored either as attributes or as elements; position is meaningless.
- XML processors frequently reorder child nodes; once again, position is rendered meaningless.

Notice how `getString()` first looks for an attribute node and then looks for a matching element. It is usually quicker to retrieve an attribute.

Listing 19-6: The XMLResultSet class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.io.*;
import java.sql.*;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.apache.xerces.dom.DocumentImpl;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.XMLSerializer;
import JavaDatabaseBible.ch19.JDBCImpl.JDBCResultSetImpl;

/**
* XMLResultSet is based on a document which is built by
* XMLStatement.executeQuery().
* This class provides the tools to traverse the document
* and return nodes by row & column.
*/
public class XMLResultSet extends JDBCResultSetImpl {
  private int rowIndex = -1;
  private int rowCount = 0;
  public  Document xmlDoc;
  private Element root = null;
  private Element currentRow = null;
  private NodeList rows = null;

  public XMLResultSet() throws SQLException {
    xmlDoc = new DocumentImpl();
    root = (Element)xmlDoc.createElement("RESULTSET");
    xmlDoc.appendChild (root);
  }

  private void initialise(){
    if(rows==null){
      root = xmlDoc.getDocumentElement();
```

```
      rows = root.getChildNodes();
      rowCount = rows.getLength();
    }
  }

  public boolean next(){
    if(rows==null){
      initialise();
    }
    if(++rowIndex == rowCount){
      return false;
    }else{
      currentRow = (Element)rows.item(rowIndex);
      return true;
    }
  }

  // scrollable ResultSet methods
  public boolean previous(){
    if(rows==null){
      initialise();
      rowIndex = rowCount;
    }
    if(--rowIndex < 0){
      return false;
    }else{
     currentRow = (Element)rows.item(rowIndex);
      return true;
    }
  }

  public boolean absolute(int row) throws SQLException {
    if(row == 0)throw new SQLException("invalid row number");
    boolean onValidRow = true;

    if(rows==null){
      initialise();
    }
    if(row > 0)rowIndex = row - 1;
    else if(row < 0){
      rowIndex = rowCount + row;
    }
    if(rowIndex<-1){
```

```
      rowIndex=-1;
      onValidRow = false;
    }
    if(row > rowCount){
      rowIndex = rowCount;
      onValidRow = false;
    }
   currentRow = (Element)rows.item(rowIndex);
    return onValidRow;
  }

  public boolean relative(int row) throws SQLException {
    boolean onValidRow = true;
    if(rows==null){
      initialise();
    }
    return absolute(row + rowIndex + 1);
  }

  public void beforeFirst() throws SQLException {
    if(rows==null){
      initialise();
    }
    rowIndex = -1;
  }

  public void afterLast() throws SQLException {
    if(rows==null){
      initialise();
    }
    rowIndex = rowCount;
  }

  public boolean first() throws SQLException {
    if(rows==null){
      initialise();
    }
    return absolute(1);
  }

  public boolean last() throws SQLException {
    if(rows==null){
      initialise();
```

```
    }
    return absolute(-1);
  }

  // first look for an attribute matching the columnName,
  // then look for a child element
  public String getString(String columnName)throws SQLException{
    if(currentRow==null)
       throw(new SQLException("Invalid row: "+currentRow));
    String value = currentRow.getAttribute(columnName);
    if(value.length()>0){
      return value;
    }else{
      NodeList cols = currentRow.getElementsByTagName(columnName);
      if(cols.getLength()>0){
        Node column = cols.item(0);
        NodeList children = column.getChildNodes();
        for(int i=0;i<children.getLength();i++){
          if(children.item(i).getNodeType()==Node.TEXT_NODE){
            return (String)children.item(i).getNodeValue();
          }
        }
      }
    }
    return null;
  }

  public ResultSetMetaData getMetaData() throws SQLException {
    return new XMLResultSetMetaData(this);
  }

  // utility method for serializing the result set document
  public void serializeAsFile(String fileName){
    try {
     OutputFormat fmt = new OutputFormat("xml",null,true);
      XMLSerializer serializer =
                    new XMLSerializer(new FileWriter(fileName),fmt);
      serializer.asDOMSerializer().serialize(xmlDoc);
    }
    catch (Exception e){
     e.printStackTrace();
    }
  }
```

```
}
```

Implementing a scrollable ResultSet

As you can see from the code in Listing 19-6, it is relatively easy to implement the necessary methods to create a scrollable `ResultSet`. All that is required to convert the `XMLResultSet` to a scrollable `ResultSet` is the addition of the following methods:

- `previous()`, which moves the cursor back one row at a time
- `first()`, which moves the cursor to the first row
- `last()`, which moves the cursor to the last row
- `beforeFirst()`, which moves the cursor to a point just before the first row
- `afterLast()`, which moves the cursor to a point just after the last row
- `absolute(int rowNumber)`, which moves the cursor to the specified row
- `relative(int rowNumber)`, which moves the cursor the specified number of rows

These methods are pretty self-explanatory. The `absolute()` method moves the cursor to the row number indicated in the argument. If the number is positive, the cursor moves to the given row number from the beginning. If the number is negative, the cursor moves to the given row number from the end, so `absolute(1)` moves the cursor to the first row, and `absolute(-1)` moves it to the last row.

The method `relative(int rowNumber)` lets you specify how many rows to move from the current row and in which direction to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows.

The `absolute()` method is used in Listing 19-7 to handle all the other cursor movements by passing it the computed value of the target row. For example, the method `first()` is implemented by calling the `absolute()` method with the argument 1. Similarly, the `relative()` method is implemented by computing the target row and passing it in a call to `absolute()`.

Cross-Reference Scrollable `ResultSets` are described in more detail in Chapter 4. Chapter 15 gives an example of the use of scrollable `ResultSets`.

Listing 19-7: Scrollable ResultSet methods

```
// scrollable ResultSet methods
public boolean previous(){
  if(rows==null){
    initialise();
    rowIndex = rowCount;
  }
  if(--rowIndex < 0){
    return false;
  }else{
   currentRow = (Element)rows.item(rowIndex);
    return true;
  }
}

public boolean absolute(int row) throws SQLException {
  if(row == 0)throw new SQLException("invalid row number");
  boolean onValidRow = true;
```

```
    if(rows==null){
      initialise();
    }
    if(row > 0)rowIndex = row - 1;
    else if(row < 0){
      rowIndex = rowCount + row;
    }
    if(rowIndex<-1){
      rowIndex=-1;
      onValidRow = false;
    }
    if(row > rowCount){
      rowIndex = rowCount;
      onValidRow = false;
    }
   currentRow = (Element)rows.item(rowIndex);
    return onValidRow;
  }

  public boolean relative(int row) throws SQLException {
    boolean onValidRow = true;
    if(rows==null){
      initialise();
    }
    return absolute(row + rowIndex + 1);
  }

  public void beforeFirst() throws SQLException {
    if(rows==null){
      initialise();
    }
    rowIndex = -1;
  }

  public void afterLast() throws SQLException {
    if(rows==null){
      initialise();
    }
    rowIndex = rowCount;
  }

  public boolean first() throws SQLException {
```

```
    if(rows==null){
      initialise();
    }
    return absolute(1);
  }

  public boolean last() throws SQLException {
    if(rows==null){
      initialise();
    }
    return absolute(-1);
  }
```

A more complete implementation of the scrollable `ResultSet` requires additional methods to report the cursor position. It also requires a mechanism to handle requests for different `ResultSet` typessuch as `TYPE_FORWARD_ONLY` or `TYPE_SCROLL_SENSITIVE`.

The XMLResultSetMetaData class

`XMLResultSet` is supported by `XMLResultSetMetaData`, an implementation of the interface `ResultSetMetaData`. This provides such utility information as column counts, column names, column data types, and so on. The code is shown in Listing 19-8.

Listing 19-8: XMLResultSetMetaData class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.sql.*;
import java.util.Vector;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import JavaDatabaseBible.ch19.JDBCImpl.JDBCResultSetMetaDataImpl;

public class XMLResultSetMetaData extends JDBCResultSetMetaDataImpl{
  private XMLResultSet rs;
  private NodeList rows = null;
  private NodeList cols = null;
  private Vector columnNameVector = new Vector();

  public XMLResultSetMetaData(XMLResultSet rs){
    this.rs = rs;
    Element root = rs.xmlDoc.getDocumentElement();
    rows = root.getChildNodes();
    Element currentRow = (Element)rows.item(0);
```

```
    NodeList children = currentRow.getChildNodes();
    for(int i=0;i<children.getLength();i++){
      if(children.item(i).getNodeType()==Node.ELEMENT_NODE){

columnNameVector.addElement(((Element)children.item(i)).getTagName());
      }
    }
  }

  public int getColumnCount()throws java.sql.SQLException{
    return columnNameVector.size();
  }

  public String getColumnLabel(int column)throws java.sql.SQLException{
    return (String)columnNameVector.elementAt(column);
  }

  public String getColumnName(int column)throws java.sql.SQLException{
    return (String)columnNameVector.elementAt(column);
  }

  public int getColumnType(int column)throws java.sql.SQLException{
    return java.sql.Types.VARCHAR;
  }

  public String getColumnTypeName(int column)throws java.sql.SQLException{
    return "VARCHAR";
  }

  public String getTableName(int column)throws java.sql.SQLException{

    return ((Element)rows.item(0)).getTagName();
  }
}
```

Implementing the SQL Engine

The SQL engine handles the parsing of the SQL commands into their various components, as well as performing more detailed processing such as evaluating the logic of compound `WHERE` clauses. This application implements only a tiny subset of the SQL-92 command set, but these are enough to process common queries as well as to create and update XML documents.

The query subset is limited to handling the types of queries listed in Table 19-1:

Table 19-1: Supported Query Operators

Function	Operator	Comment

Table 19-1: Supported Query Operators

Function	Operator	Comment
Equality	=	String.equals()
Inequality	<>	
Comparison	LIKE	Supports the % wildcard character
Negation	NOT	Supports all the above

The base class used to process SQL commands is `XMLCommand`. This class is not XML specific, despite its name. XML-specific functionality is handled by extending the class. That way, the class can be used as the basis of any other SQL engine you may wish to build.

The XMLCommand class

One of the key methods of the `XMLCommand` class is the `parseSQLCmd()` method, which is used to split the command into its main components clauses. These include the command itself, the table name, the column names, and the `WHERE` clause. Additional utility methods such as `splitFields()`split these clauses into vectors of subclauses for further processing. Listing 19-9 shows the `XMLCommand` class.

Listing 19-9: XMLCommand class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.sql.*;
import java.net.*;
import java.util.*;

/**
* XMLCommand is an implementation independent SQL command preprocessor
*/
public class XMLCommand {
  protected String SQLString;
  protected String cmd = null;
  protected String tableName = null;
  protected String columns = null;
  protected String values = null;
  protected String fields = null;
  protected String where = null;
  protected String orderBy = null;

  public XMLCommand() {
  }

  public XMLCommand(String SQLString) {
    this.SQLString = SQLString.toUpperCase().trim();
```

```
    parseSQLCmd(SQLString);
  }

  protected void parseSQLCmd(String SQLCmd){
    cmd = SQLCmd.substring(0,SQLCmd.indexOf(" "));
    tableName = getTableName(SQLCmd);

    int tNameEnds    = SQLCmd.indexOf(tableName) + tableName.length();
    int columnsEnd   = SQLCmd.indexOf(" VALUES");
    int valuesIndex  = SQLCmd.indexOf(" VALUES");
    int fromIndex    = SQLCmd.indexOf(" FROM ");
    int whereIndex   = SQLCmd.indexOf(" WHERE ");
    int orderIndex   = SQLCmd.indexOf(" ORDER ");
    int orderByIndex = SQLCmd.indexOf(" BY ",orderIndex);

    if(whereIndex>-1) whereIndex  += " VALUES".length();
    if(valuesIndex>-1)valuesIndex += " VALUES".length();

    if(cmd.equals("CREATE")){
      columns = SQLCmd.substring(tNameEnds).trim();
    }
    else if(cmd.equals("INSERT")){
      columns = SQLCmd.substring(tNameEnds,columnsEnd).trim();
      values  = SQLCmd.substring(valuesIndex).trim();
    }
    else if(cmd.equals("SELECT")){
      fields = SQLCmd.substring("SELECT".length(),fromIndex).trim();
      if(whereIndex>-1){
        if(orderIndex>-1){
          where = SQLCmd.substring(whereIndex,orderIndex);
        }else{
          where = SQLCmd.substring(whereIndex);
        }
        where = where.trim();
      }

      if(orderIndex>-1){
        orderBy  = SQLCmd.substring(orderByIndex).trim();
      }
    }
  }

  private String getTableName(String SQLCmd){
```

```
    String tableName = null;
    if(SQLCmd.startsWith("SELECT")){
      tableName = wordAfter(SQLCmd, "FROM");
    }
    else if(SQLCmd.startsWith("INSERT")){
      tableName = wordAfter(SQLCmd, "INTO");
    }
    else if(SQLCmd.startsWith("UPDATE")){
      tableName = wordAfter(SQLCmd, "UPDATE");
    }
    else if(SQLCmd.startsWith("DELETE")){
      tableName = wordAfter(SQLCmd, "FROM");
    }
    else if(SQLCmd.startsWith("CREATE")){
      tableName = wordAfter(SQLCmd, "TABLE");
    }
    return tableName;
  }

  protected Vector splitFields(String fields){
    Vector fieldVector = new Vector();
    fields = fields.trim();
    if(fields.startsWith("("))fields = fields.substring(1);
    if(fields.endsWith(")"))
       fields = fields.substring(0,fields.length()-1);

    int comma = fields.indexOf(",");
    while(comma >= 0){
      String field = fields.substring(0,comma).trim();
      fieldVector.addElement(field);
      fields = fields.substring(comma+1).trim();
      comma = fields.indexOf(",");
    }
    fieldVector.addElement(fields.trim());
    return fieldVector;
  }

  protected Vector splitColumns(String columns){
    return splitFields(columns);
  }

  protected Vector splitValues(String values){
    return splitFields(values);
```

```
  }

  protected String wordAfter(String SQLCmd, String after){
    String word = SQLCmd.substring(SQLCmd.indexOf(after)+
                                   after.length()).trim();
    if(word.indexOf(" ")>-1)word = word.substring(0,word.indexOf(" "));
    return word.trim();
  }
}
```

The XMLQuery class

The `XMLQuery` class extends the basic `XMLCommand` class. An `XMLQuery` object is created by the `XMLStatement` when the `Statement.executeQuery()` method is called. In its constructor, `XMLQuery` calls the `parseSQLCmd()` method of its base class. Listing 19-10 shows the `XMLQuery` class.

Listing 19-10: XMLQuery class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.io.*;
import java.util.StringTokenizer;
import java.util.Vector;
import java.sql.SQLException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.xml.sax.InputSource;
import org.apache.xerces.parsers.DOMParser;

/**
* XMLQuery extends XMLCommand to provide XL specific query processing
*/
public class XMLQuery extends XMLCommand{
  private Document xmlDoc;

  public XMLQuery(String SQLString) {
    this.SQLString = SQLString.toUpperCase().trim();
    parseSQLCmd(SQLString);
  }

  // process xml doc and build ResultSet doc
  public XMLResultSet processDoc(Document xmlDoc) throws SQLException{
```

```
    this.xmlDoc = xmlDoc;
    XMLResultSet resultSet = new XMLResultSet();
    NodeList records = xmlDoc.getElementsByTagName(this.tableName);

    if(where==null){
      for(int i=0;i<records.getLength();i++){
        Element record = (Element)records.item(i);
        Node importedNode = resultSet.xmlDoc.importNode(record,true);
        if(!fields.equals("*"))pruneFields(importedNode);
        resultSet.xmlDoc.getDocumentElement().appendChild(importedNode);
      }
    }else{
      Vector whereClauses = splitWhereClause(where);
      XMLWhereEvaluator evaluator = new XMLWhereEvaluator(whereClauses);

      for(int i=0;i<records.getLength();i++){
        Element record = (Element)records.item(i);
        if(evaluator.testRecord(record)){
          Node importedNode = resultSet.xmlDoc.importNode(record,true);
          if(!fields.equals("*"))pruneFields(importedNode);
          resultSet.xmlDoc.getDocumentElement().appendChild(importedNode);
        }
      }
    }
     return resultSet;
  }

  // split the WHERE clause into a Vector of individual tests
  protected Vector splitWhereClause(String whereClause){
    Vector where = new Vector();

    String subTest = "";
    String token = "";
    StringTokenizer st = new StringTokenizer(whereClause," ()",true);
    while (st.hasMoreTokens()) {
      token = st.nextToken();
      if(token.equals("AND")||token.equals("OR")||
         token.equals("(")||token.equals(")")){
        subTest = subTest.trim();
        if(subTest.length()>0)where.addElement(subTest);
        where.addElement(token);
        subTest="";
      }else{
```

```
        subTest += token;
      }
    }
    if(subTest.trim().length()>0){
      where.addElement(subTest.trim());
    }
    return where;
  }

  // prune to include only selected fields
  private void pruneFields(Node record){
    Vector fieldClauses = splitFields(fields);
    NodeList nodes = record.getChildNodes();
    for(int i=0;i<nodes.getLength();i++){
      Node n = nodes.item(i);
      if(n.getNodeType()==Node.ELEMENT_NODE){
        String tagName = ((Element)n).getTagName();
        if(!fieldClauses.contains(tagName))record.removeChild(n);
      }
    }
  }
}
```

After creating the `XMLQuery` object, the `XMLStatement` calls `XMLQuery`'s `processDocument()` method, passing it a reference to the document being queried. `XMLQuery.processDocument()`handles the actual processing of the query. It does this by first creating an `XMLResultSet` and then retrieving the XML elements corresponding to the table and evaluating them against the `WHERE` clause.

Since the database is contained in an XML document, the `XMLResultSet` is also returned as an XML document. XML elements that match the `WHERE` clause are imported into the newly created document and pruned of element nodes that are not itemized in the column list of the SQL query. Attribute nodes, on the other hand, are returned without pruning in this implementation, though, of course, you can easily change this if you wish.

The final step is to append the selected and pruned node to the root element of the `XMLResultSet`. Once the entire `XMLResultSet` has been created, it is returned in the normal way.

The XMLWhereEvaluator class

The SQL query engine itself is implemented in the `XMLWhereEvaluator` class, shown later in this section. The protected element record contains the record currently being tested, and the vector `testVector` contains the Strings representing the individual subtests. For example, a SQL query might look like this:

```
"SELECT * FROM CUSTOMER
WHERE ( FIRST_NAME LIKE 'M%' OR CUSTOMER_NUMBER = '102' )"
```

The `WHERE` clause is split into subtests as follows:

```
(
```

```
FIRST_NAME LIKE 'M%'
OR
CUSTOMER_NUMBER = '102'
)
```

The `XMLWhereEvaluator` evaluates the test vector against each row element in the XML table document to create a result String. The result String is created from the test vector by appending parens and boolean operators such as `AND` and `OR` directly to the result String and by evaluating the subtests containing operators. Evaluation of the subtests occurs by calling the appropriate test method for the operator in the subtest. Only two test methods are implemented here, though implementing additional tests is quite simple:

- `isLike()`, which parses out the '%' wildcard and performs the appropriate String comparison
- `isEqual()`, which simply compares the Strings

These test methods return a boolean result. Negation is handled by set a boolean flag used to toggle the true/false value the test returns. For example, if the `WHERE` clause is evaluated against a row containing this element:

```
<FIRST_NAME>Michael</FIRST_NAME>
```

The returned result String will be this:

```
( true OR false )
```

This result String, which is in `infix` notation, is passed to the method `evaluate(String infix)`. This method uses a simple two-stack approach to evaluate the result String and to return a boolean result for the overall test, as shown in Listing 19-11.

Listing 19-11: XMLWhereEvaluator class

```
package JavaDatabaseBible.ch19.JDBCforXML;

import java.io.*;
import java.util.*;
import java.sql.SQLException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.xml.sax.InputSource;
import org.apache.xerces.parsers.DOMParser;

public class XMLWhereEvaluator{
  Element record = null;
  Vector testVector = null;

  public XMLWhereEvaluator(Vector testVector){
   this.testVector = testVector;
  }
```

```
public boolean testRecord(Element record){
  String test;
  String results = "";
  for(int i=0;i<testVector.size();i++){
    test = (String)testVector.elementAt(i);
    if(test.equals("OR")||test.equals("AND")||
       test.equals("(")||test.equals(")")){
      results += " "+test;
    }else {
      if(testWhereClause(record, test))results += " true";
      else results += " false";
    }
  }
  return evaluate(results.trim());
}

// test individual where clauses
private boolean testWhereClause(Element record, String whereClause){
  boolean not = false;
  boolean retval = false;
  String fieldName =
         whereClause.substring(0,whereClause.indexOf(" ")).trim();
  whereClause      = whereClause.substring(fieldName.length()).trim();
  String test      =
         whereClause.substring(0,whereClause.indexOf(" ")).trim();
  if(test.equals("NOT")){
    not = true;
    whereClause = whereClause.substring(test.length()).trim();
    test = whereClause.substring(0,whereClause.indexOf(" ")).trim();
  }
  String operand = whereClause.substring(test.length()).trim();

  operand = operand.replace('\'',' ').trim();
  String nodeValue = record.getAttribute(fieldName);
  if(nodeValue.length()==0){
    NodeList fields = record.getElementsByTagName(fieldName);
    Element field = (Element)fields.item(0);
    nodeValue = field.getFirstChild().getNodeValue();
  }
  if(test.equals("LIKE")){
    retval = isLike(operand,nodeValue);
  }
  if(test.equals("=")){
```

```
      retval = isEqual(operand,nodeValue);
    }
    if(test.equals("<>")){
      not = true;
      retval = isEqual(operand,nodeValue);
    }
    if(not)retval = !retval;
    return retval;
  }

  private boolean isEqual(String operand,String nodeValue){
    boolean retval = false;
    operand = operand.trim();
    if(nodeValue.equals(operand))retval = true;
    return retval;
  }

  private boolean isLike(String operand,String nodeValue){
    boolean retval = false;
    if(operand.startsWith("%")){
      if(operand.endsWith("%")){
        operand = operand.replace('%',' ').trim();
        if(nodeValue.indexOf(operand)>-1)retval = true;
      }else{
        operand = operand.replace('%',' ').trim();
        if(nodeValue.endsWith(operand))retval = true;
      }
    }else if(operand.endsWith("%")){
      operand = operand.replace('%',' ').trim();
      if(nodeValue.startsWith(operand))retval = true;
    }else{
      operand = operand.trim();
      if(nodeValue.equals(operand))retval = true;
    }
    return retval;
  }

  protected boolean evaluate(String infix){
    int parens = 0;
    Stack ops = new Stack();
    Stack args = new Stack();
    infix = infix.trim();
```

```
    StringTokenizer st = new StringTokenizer(infix," ()",true);
    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      if(!token.equals(" ")){
        if(token.equals("AND")||token.equals("OR")){
        if(ops.size()>parens)evaluate(ops,args);
          ops.push(token);
        }else if(token.equals("(")){
          if(args.size()>0)++parens;
        }else if(token.equals(")")){
          --parens;
        }else {
          args.push(token);
        }
      }
    }
    while(!ops.empty()){
      evaluate(ops,args);
    }
    String result = (String)args.pop();
    return (result.equals("true"))?true:false;
  }
  private void evaluate(Stack ops,Stack args){
    boolean a = (((String)args.pop()).equals("true"))?true:false;
    boolean b = (((String)args.pop()).equals("true"))?true:false;
    boolean c = false;
    String  o = (String)ops.pop();
    if(o.equals("AND"))c = a & b;
    if(o.equals("OR")) c = a | b;
    args.push(c?"true":"false");
  }
}
```

Testing the JDBC/XML Database

You can check out the JDBC/XML database using code similar to any of the `DriverManager`-based examples in earlier chapters. Listing 19-12 shows a typical example using all the features of a scrollable `ResultSet` implemented by adding the optional scrollable `ResultSet` methods of Listing 19-7 to the basic `XMLResultSet` of Listing 19-6.

Listing 19-12: JDBC/XML database test code

```
import java.io.*;
import java.net.*;
import java.sql.*;
```

```
import java.util.*;
import org.w3c.dom.Document;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.XMLSerializer;

public class XMLDBTest{
  static String UrlString = "file:///c:/projects/CustomerDB.xml";

  static String SQLQuery =
  "SELECT * FROM CUSTOMER WHERE "+
  "( FIRST_NAME LIKE 'M%' OR CUSTOMER_NUMBER = '102' ) "+
  "AND LAST_NAME = 'Corleone'";

  static String[] SQLCmd =
  { "INSERT INTO CUSTOMER VALUES('101','Vito',  'Q','Corleone","+
    "'137 Main', 'New York','NY','10006','201-555-1213')",
    "INSERT INTO CUSTOMER VALUES('102','James', 'J','Corleone","+
    "'123 Pine', 'New York','NY','10006','201-555-1214')",
    "INSERT INTO CUSTOMER VALUES('103','Raquel','S','Corleone","+
    "'123 Pine', 'New York','NY','10006','201-555-1215')",
    "INSERT INTO CUSTOMER VALUES('104','James', 'J','Witherspoon","+
    "'17 Oak','New York','NY','10006','201-555-1216')",
    "INSERT INTO CUSTOMER VALUES('105','Fred',  'Q','Bloggs","+
    "  '22 Walnut','New York','NY','10006','201-555-1217')"};

  public String cNum   = null;
  public String fName  = null;
  public String lName  = null;
  public String street = null;
  public String city   = null;
  public String state  = null;
  public String zip    = null;
  public String phone  = null;

  Document xmlDoc = null;

  public XMLDBTest(){
    try{
      Class.forName("JavaDatabaseBible.ch19.JDBCforXML.XMLDriver");
    }
    catch (Exception e){
      System.out.println(e);
    }
```

```
  }

  public static void main(String args[]){
    XMLQueryTest test = new XMLQueryTest();
    serializeDocumentAsFile(test.createTable(),UrlString);
    serializeDocumentAsFile(test.updateTable(SQLCmd),UrlString);
    serializeDocumentAsFile(test.queryTable(SQLQuery),
                                "file:///c:/projects/ResultSet.xml");
  }
  public Document createTable(){
    try{
      Connection con = DriverManager.getConnection(UrlString);
      Statement stmt = con.createStatement();

      stmt.executeUpdate("CREATE TABLE CUSTOMER "+
                         "(CUSTOMER_NUMBER ATTRIBUTE, "+
                         "FIRST_NAME VARCHAR(30), "+
                         "MI         VARCHAR(30), "+
                         "LAST_NAME  VARCHAR(30), "+
                         "STREET     VARCHAR(30), "+
                         "CITY       VARCHAR(30), "+
                         "STATE      VARCHAR(30), "+
                         "ZIP        VARCHAR(30), "+
                         "PHONE      VARCHAR(30))");

      stmt.executeUpdate("INSERT INTO CUSTOMER VALUES("+
                         "'100','Michael','A','Corleone',"+
                         "'123 Pine','New York','NY','10006','201-555-
1212'");
      xmlDoc =

((JavaDatabaseBible.ch19.JDBCforXML.XMLStatement)stmt).getXmlDocument();
    }
    catch (Exception e){
      System.out.println(e);
    }
    return xmlDoc;
  }
  public Document updateTable(String[] SQLCmd){
    try{
      Connection con = DriverManager.getConnection(UrlString);

      Statement stmt = con.createStatement();
      for(int i=0;i<SQLCmd.length;i++){
```

```
        stmt.executeUpdate(SQLCmd[i]);
      }
      xmlDoc =

((JavaDatabaseBible.ch19.JDBCforXML.XMLStatement)stmt).getXmlDocument();
    }
    catch (Exception e){
      System.out.println(e);
    }
    return xmlDoc;
  }
  public Document queryTable(String SQLQuery){
    ResultSet rs = null;
    try{
      Connection con = DriverManager.getConnection(UrlString);
      Statement stmt = con.createStatement();

      rs = stmt.executeQuery(SQLQuery);
      while(rs.next()){
        getRowData(rs);
      }
      while(rs.previous()){
        getRowData(rs);
      }
      rs.first();
      getRowData(rs);

      rs.last();
      getRowData(rs);

      rs.absolute(2);
      getRowData(rs);

      rs.relative(-1);
      getRowData(rs);
    }
    catch (Exception e){
      System.out.println(e);
    }
    return ((JavaDatabaseBible.ch19.JDBCforXML.XMLResultSet)rs).xmlDoc;
  }
  private void getRowData(ResultSet rs){
    try {
```

```
      cNum   = rs.getString("CUSTOMER_NUMBER");
      fName  = rs.getString("FIRST_NAME");
      lName  = rs.getString("LAST_NAME");
      street = rs.getString("STREET");
      city   = rs.getString("CITY");
      state  = rs.getString("STATE");
      zip    = rs.getString("ZIP");
      phone  = rs.getString("PHONE");
    }catch (Exception e){
      System.out.println(e);
    }
  }
  public static void serializeDocumentAsFile(Document xmlDoc,String
UrlString){
   String fileName = "XMLOut.xml";
    try {
     URL url = new URL(UrlString);
     if(url.getProtocol().equals("file")){
        fileName = url.getFile().substring(1);
     }
     OutputFormat fmt = new OutputFormat("xml",null,true);
      XMLSerializer serializer =
                    new XMLSerializer(new FileWriter(fileName),fmt);
      serializer.asDOMSerializer().serialize(xmlDoc);
    }
    catch (Exception e){
     e.printStackTrace();
    }
  }
}
```

This test code creates the XML database shown in Listing 19-13. The `createTable()` method creates the XML document and inserts the first record. Calling the `updateTable()` method results in the insertion of the other records.

Listing 19-13: XML database created using XMLDBTest class

```
<?xml version="1.0"?>
<CustomerDB>
    <CUSTOMER CUSTOMER_NUMBER="100">
        <FIRST_NAME>Michael</FIRST_NAME>
        <MI>A</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>123 Pine</STREET>
```

```
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1212</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="101">
        <FIRST_NAME>Vito</FIRST_NAME>
        <MI>Q</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>137 Main</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1213</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="102">
        <FIRST_NAME>James</FIRST_NAME>
        <MI>J</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>123 Pine</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1214</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="103">
        <FIRST_NAME>Raquel</FIRST_NAME>
        <MI>S</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>123 Pine</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1215</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="104">
        <FIRST_NAME>James</FIRST_NAME>
        <MI>J</MI>
        <LAST_NAME>Witherspoon</LAST_NAME>
        <STREET>17 Oak</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
```

```
        <PHONE>201-555-1216</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="105">
        <FIRST_NAME>Fred</FIRST_NAME>
        <MI>Q</MI>
        <LAST_NAME>Bloggs</LAST_NAME>
        <STREET>22 Walnut</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1217</PHONE>
    </CUSTOMER>
</CustomerDB>
```

Note that although the `SQL CREATE` command specifies type `VARCHAR (30)` for most of the fields, this type specification defaults to String. The reason for this is that all data is stored as a String, and the only significance attached to data type is to check for the custom type `ATTRIBUTE`, which is used to denote that the field should be added to the row element as an attribute.

Note also that the XML document must be saved after each update. The XML database actually exists as a DOM document in memory, so it must be serialized after changes are made.

Tests are carried out using a variety of different queries. These queries include the following:

```
SELECT * FROM CUSTOMER
SELECT * FROM CUSTOMER WHERE FIRST_NAME LIKE 'M%'
SELECT * FROM CUSTOMER WHERE FIRST_NAME NOT LIKE 'M%'
SELECT * FROM CUSTOMER WHERE FIRST_NAME NOT = 'Michael'
SELECT * FROM CUSTOMER WHERE FIRST_NAME <> 'Michael'
SELECT * FROM CUSTOMER WHERE FIRST_NAME LIKE 'M%' OR FIRST_NAME LIKE 'F%'
SELECT * FROM CUSTOMER WHERE (FIRST_NAME LIKE 'M%' OR FIRST_NAME LIKE 'V%')
SELECT * FROM CUSTOMER WHERE ( FIRST_NAME LIKE 'M%' OR CUSTOMER_NUMBER =
'102' )
```

In addition to supporting the `ResultSet.getString()` method used to set the `String` variables in Listing 19-12, the `XMLResultSet` can also be retrieved as an XML document. Listing 19-14 shows the `XMLResultSet` generated by running this query:

```
SELECT * FROM CUSTOMER WHERE
( FIRST_NAME LIKE 'M%' OR CUSTOMER_NUMBER = '102' ) AND LAST_NAME =
'Corleone'
```

Listing 19-14: XMLResultSet

```
<?xml version="1.0"?>
<RESULTSET>
    <CUSTOMER CUSTOMER_NUMBER="100">
        <FIRST_NAME>Michael</FIRST_NAME>
        <MI>A</MI>
        <LAST_NAME>Corleone</LAST_NAME>
```

```
        <STREET>123 Pine</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1212</PHONE>
    </CUSTOMER>
    <CUSTOMER CUSTOMER_NUMBER="102">
        <FIRST_NAME>James</FIRST_NAME>
        <MI>J</MI>
        <LAST_NAME>Corleone</LAST_NAME>
        <STREET>123 Pine</STREET>
        <CITY>New York</CITY>
        <STATE>NY</STATE>
        <ZIP>10006</ZIP>
        <PHONE>201-555-1214</PHONE>
    </CUSTOMER>
</RESULTSET>
```

The advantage of returning the entire `XMLResultSet` as an XML document is that many applications are designed to work with XML. In this form, the `XMLResultSet` can be transferred between applications or manipulated using an XSL transform.

Since the target database is defined by a URL, you are not restricted to using local XML files as databases. Try substituting the URL `http://www.moreover.com/cgi-local/page?o=xml&query=top+stories`.

Cross-Reference Chapter 17 discusses working with XML sources over the Internet.The examples are based on accessing the `http://www.moreover.com/` Web site.

Summary

In this chapter, you learn to create a JDBC driver and a simple SQL engine. The examples can be expanded and modified to form the basis of any custom application requiring a JDBC API. The main topics covered included the following:

- Detailed operation of a JDBC driver
- A simple, String-oriented SQL query engine
- Examples of working with XML documents

This chapter ends Part IV. Part V explores persistence in the context of Enterprise Java Beans and JDO.

Part V: EJBs, Databases, and Persistence

Chapter List

Part Overview

Part V is a discussion of the use databases in the context of J2EE applications using Enterprise JavaBeans. The first chapter gives a brief overview of Enterprise JavaBeans, including descriptions of:

- The three types of Enterprise JavaBeans: session beans, entity beans, and message-driven beans.
- Activation and passivation
- Bean-managed persistence and container-managed persistence
- Enterprise JavaBean transactions

After reading this chapter, you should have a good understanding of Enterprise JavaBeans and of the ways they interact with databases.

Subsequent chapters discuss bean-managed persistence and container-managed persistence, with extensive examples. They include sections on the use of JDBC and SQL in bean-managed persistence and of the Enterprise JavaBean query language (EJBQL).

The final chapter in Part V covers Java data objects and transparent persistence. This is a new technology that handles persistence in a manner that is completely transparent to the developer.

Chapter 20: Enterprise JavaBeans

In This Chapter

This chapter gives a brief overview of Enterprise JavaBeans (EJBs). The features and purposes of three types of EJBs are discussed. The fundamentals of transaction and persistence management are reviewed. After reading this chapter, you should have a good understanding of Enterprise JavaBeans (EJBs) and of the ways they interact with databases.

Enterprise JavaBeans Overview

The Enterprise JavaBeans Specification defines EJBs as follows: "Enterprise JavaBeans is an architecture for component-based distributed computing. EJBs are components of the distributed transaction-oriented enterprise applications." In a nutshell, EJBs are server-side components that encapsulate the business logic of an application. The business logic is the code that fulfills the purpose of the application. For example, in an online shopping application, the EJBs might implement the business logic in methods called `searchCatalog` and `checkOut`. By invoking these methods, remote clients can access the online shopping services the application provides.

An EJB typically communicates with Enterprise Information Systems (EIS) such as databases and legacy systems and other EJBs. At the same time, different types of clients access EJBs requesting services. The clients can be other EJBs, Web applications, servlets, or application clients.

At runtime, an EJB resides in an EJB container. An EJB container provides the deployment and runtime environment for EJBs, including services such as security, transaction, deployment, concurrency management, and instance life-cycle management. The process of installing an EJB in an EJB container is called *EJB deployment*. EJB containers are typically part of an application server. EJBs by nature are portable components; therefore, the application assembler can build new applications from existing beans with minimum effort. These applications can run on any J2EE-compliant application servers.

EJBs are designed to simplify the development of large, distributed applications. Because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, not the bean developer, is responsible for system-level services such as transaction management and security authorization. Furthermore, since the application's business logic is contained in EJBs instead of in clients, client developers can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, clients are thinner. This is particularly beneficial for clients that run on small devices such as cell phones or PDAs.

EJBs are especially suitable for applications that have the following requirements and characteristics:

- **Scalablability.** To accommodate a growing number of users, one may need to distribute an application's components across multiple machines. Not only can the EJBs of an application run on different machines, but their location remains transparent to clients.
- **Transactions-oriented.** EJBs support transactions through container services, the mechanisms that manage the concurrent access of shared objects and ensure data integrity.
- **Multiple types of clients.** With just a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

The EJB 2.0 Specification specifies the three following types of EJBs:

- Session beans
- Entity beans
- Message-driven beans

The features, as well as the appropriate uses of each type of EJB, are discussed in more details in the following sections.

Session Beans

A *session bean* represents a single client inside the J2EE server and performs tasks on behalf of the client. This type of bean manages sessions (or conversations between the client and the server) on behalf of the client. A typical session is transient, and its state is usually not persistent. An example of a session is tracking your courier package using a Web-based status-query application. If, for some reason, the Web server dies or the session times out, the session terminates, and the user is required to start a new session. Most online transactions are session oriented, with the user initiating a session performing a set of actions and then terminating a session. Hence, a session bean generally stores its state in transient variables.

Not all sessions are conversational. Some sessions involve only one interaction between the client and server. For example, getting a stock quote does not need the multiple invocations of the service the stock-quote server provides. These sessions are *stateless*, and their management can be significantly simplified. To address these different scenarios, the EJB specification specifies two types of session beans: stateful and session.

In general, the use a session bean is appropriate if the following circumstances hold:
- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period (perhaps a few hours).

Once the session bean is chosen, we still need to decide which one to use, stateless or stateful, based on whether a conversational state needs to be held in the session bean.

Stateless Session Beans

Stateless session beans are components that implement a single-use service. That service can be invoked many times, but since the component does not maintain any state, the effect is that each invocation provides a single use. In a lot of ways, stateless session beans provide a *reusable single-use service*.

Although a stateless session bean does not maintain a conversational state for a particular client, it may contain a transient state in the form of its instance variables, as shown in the code example. When a client invokes the method of a stateless bean, the values of the bean's instance variables represent such a transient state but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. Most of application servers take advantage of this feature and pool the stateless session beans to achieve better performance.

Because stateless session beans can support multiple clients and usually are pooled in the EJB container, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients. At times, the EJB container may write a stateful session bean to secondary storage (called passivation, discussed later). However, stateless session beans are never written to secondary storage. This further makes stateless beans offer better performance than stateful beans.

The major advantage of stateless session beans over stateful session beans is performance. A stateless session bean should be chosen if one of following is true:
- The bean's state has no data for a specific client.
- In a single-method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to retrieve stock quotes at any time.
- The bean fetches from a database a set of read-only data that is often used by clients. Such a bean, for example, can retrieve the table rows that represent the inventory that currently below certain level.

In general, the steps for developing EJBs include:
1. Write the remote interface.
2. Write the home interface.
3. Write the EJB implementation class.
4. Compile the EJB and all its supporting files.
5. Write the deployment descriptors.
6. Package and deploy.

Note

Many of the preceding steps can be performed automatically by a variety of IDEs. Don't write everything from scratch.

Although your favorite IDE may complete many steps for you, let's go through these steps manually in the development of our stateless session bean example. To keep things less confusing, a naming convention should be adopted for all the Java classes involved. A commonly accepted naming convention is listed in Table 20-1.

Table 20-1: EJB Name Convention

Item	Name	Example
Remote Interface	<name>	Customer
Home Interface	<name>Home	CustomerHome
Implementation Class	<name>Bean	CustomerBean
EJB Name	<name>EJB	CustomerEJB
EJB Jar Display Name	<name>Jar	CustomerJar

Remember the first program you have ever written in Java? Is it the "`Hello, world`"? The first EJB example developed in this chapter is a stateless session bean called `HelloEJB`. When it is invoked, a welcome message is delivered to the calling client.

Although the business logic is defined in the implementation class, the client can never directly access implementation-class instances. Instead, the client calls an EJB's remote interface to get its service. In other words, the remote interface defines the business methods that a remote client may invoke. The bean developer defines the types of the method arguments, the return type, and the exceptions the methods throw. The signatures of these methods must be identical to the signatures of the corresponding methods in the EJB implementation class.

Remote interface

Every EJB remote interface extends the `java.ejb.EJBObject` interface. Since EJBs are meant to work in a distributed system, the remote interface is a valid remote interface for RMI-IIOP, so each method must throw the `java.rmi.RemoteException`. The source code for the `HelloEJB` remote interface is shown in Listing 20-1. Three methods are defined by which a client can get all welcome messages, a specific welcome message, or the number of messages.

Listing 20-1: Remote interface of HelloEJB

```
package java_database.ch20.HelloSLBean;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface Hello extends EJBObject {
   public String[] getAllWelcomeMsgs() throws RemoteException;
   public String getWelcomeMsg(int i)  throws RemoteException;
   public int getNumberOfWelcomeMsgs() throws RemoteException;
}
```

Home interface

The home interface controls the life cycle of the EJB objects. For a session bean, the purpose of the home interface is to define the `create` methods that a remote client may invoke to create its reference to the EJB object. You may define multiple `create` methods with different signatures. The default method without any argument is used to instantiate EJB objects in the container.

Note that `create` methods are different from constructors. A constructor is an initializer for an object (which may exist for a very long time). A `create` method is used by clients to initialize an EJB instance in an EJB container. An EJB instance may be composed of one object or a variety of objects over its life cycle. As such, it has different initialization mechanisms.

Understanding the life cycle is critical in mastering EJBs. Unfortunately, that is beyond the scope of this book. The interested reader can find extensive discussions on EJB life cycles in numerous EJB books.

Note

Do not assume that `create` methods are the same as constructors.

As is the case for the remote interface, the signatures of the `create` methods defined in the home interface must correspond to those of its corresponding `ejbCreate` methods in the implementation class. The `throws` clause of the `create` method must include `java.rmi.RemoteException` and the `javax.ejb.CreateException`. The home interface of the `HelloEJB` is shown in Listing 20-2. Only one `create` method is defined in this example.

Listing 20-2: Home interface of HelloEJB

```
package java_database.ch20.HelloSLBean;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface HelloHome extends EJBHome {
   public Hello create() throws CreateException, RemoteException;
}
```

Implementation class

Most of the work that you have to do as a bean developer occurs in the bean class itself. There are a number of methods the bean class must provide. An important method, and perhaps the most confusing one, is `ejbCreate`.

Because an enterprise bean runs inside an EJB container, a client cannot directly instantiate the bean. Only the EJB container can instantiate an enterprise bean. During instantiation, the example program performs the following steps:

1. The client invokes a `create` method on the home object
2. The EJB container instantiates the EJB instance.
3. The EJB container invokes the appropriate `ejbCreate` method in the implementation class; typically, an `ejbCreate` method initializes the state of the EJB instance.

create and ejbCreate method guidelines

Typically, an `ejbCreate` method initializes the state of the EJB instance. The guidelines for writing such methods are:

- Each `create` method defined in the home interface must have a corresponding `ejbCreate` method in the bean-implementation class.
- The number of arguments and argument data types between the `ejbCreate` and the corresponding `create` methods must be the same.

- Since the `ejbCreate` is called by the container, there is nothing to return so its return type is `void`.
- The `create` method returns the remote interface.

The bean class extends the `java.ejb.SessionBean` interface, which declares the `ejbRemove, ejbActivate, ejbPassivate`, and `setSessionContext` methods. The `HelloBean` class does not use these methods, but it must implement them (as empty functions). Later sections on stateful session beans and entity beans explain the use of these methods.

EJBExceptions

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the remote object reference that the `create` method returns. From the client's perspective, the business methods appear to run locally, although they actually run remotely in the application server's EJB container. All the business methods declared in the remote interface need to be implemented. The signatures of these business methods are the same as those defined in the remote interface. However, since the bean object is running inside of the container, it does not need to throw the `java.rmi.RemoteException`.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw `javax.ejb.EJBException`. When a business method throws an `EJBException`, the container wraps it in a `RemoteException`, which is caught by the client. Since `EJBException` is a subclass of `RuntimeException`, you do not need to explicitly include it in the `throws` clause of the business method. The `HelloBean` class is shown in Listing 20-3. It should be noted that the method `getWelcomeMsg(int)` is coded defensively to prevent the index from going out of range.

Listing 20-3: HelloBean class

```
package java_database.ch20.HelloSLBean;

import javax.ejb.*;
import java.util.*;

public class HelloBean implements SessionBean
{
   // instance variables
   private SessionContext ctx;
   private String[] msgList = new String[3];

   // default constructor – different from ejbCreate()
   public HelloEJBBean() {
   }

   // Life cycle methods called by EJB container
   public void setSessionContext(SessionContext c) {
     System.out.println("setSessionContext called.");
     ctx = c;
   }
   public void ejbCreate() {
      System.out.println ("ejbCreate() called.");
```

```
        msgList[0] = "Hello!";
        msgList[1] = "Welcome to the EJB world.";
        msgList[2] = "Enjoy reading.";
    }
    public void ejbRemove() {
        System.out.println ("ejbRemove called.");
    }
    public void ejbPassivate() {
        System.out.println("ejbPassivate called.");
    }
    public void ejbActivate() {
        System.out.println("ejbActivate() called.");
    }

    // Business methods serving the client's need
    public String[] getAllWelcomeMsgs() {
        return msgList;
    }

    public String getWelcomeMsg(int i) {
        // first make sure index is not out of range
        i = (i >= msgList.length) ? (msgList.length - 1) : i;
        i = (i < 0) ? 0 : i;
        // return welcome message
        return msgList[i];
    }

    public int getNumberOfWelcomeMsgs() {
        return msgList.length;
    }
}
```


Where is the database-access code? After all, this is a book on Java database programming. Since EJB itself is complex enough, I have deliberately kept the first EJB example as simple as possible (the spirit of the `HelloWorld` example). As soon as you have this piece of code running, you can easily extend it to fulfill your needs, such as accessing databases using the JDBC programming skills you have learned from previous chapters. For example, the welcome messages may be stored in a database table. Then in the `ejbCreate` method, you do not need to initialize the `msgList` array; instead, an instance of `javax.sql.DataSource` should be initialized. And the business methods access the `DataSource` and retrieve the welcome message (or count the number of rows) from the message table. You can find code samples of initializing and accessing `DataSource` objects in next chapter.

After the bean classes are coded, they need to be packaged and deployed. Most application servers have built-in tools for EJB (and enterprise application) deployment. The key artifacts of the deployment process are the deployment descriptors, the XML files that contain the declarative information about the EJBs and the enterprise application. Most of the container vendors use some proprietary technology to enhance the performance of their product. As a consequence, the deployment descriptor usually contains two files: one is the strictly J2EE standard, and the other is vendor specific. As an example, the

deployment-descriptor files for WebLogic application server (version 6.1) are listed in Listing 20-4. If you use different application servers, the files may look different.

Listing 20-4: Deployment-descriptor files for HelloEJB

```
# First file: J2EE standard
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>HelloEJB</ejb-name>
      <home>HelloSLBean.HelloHome</home>
      <remote>HelloSLBean.Hello</remote>
      <ejb-class>HelloSLBean.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>

#Second file: WebLogic specific
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic
6.0.0 EJB//EN' 'http://www.bea.com/servers/wls60/ejb20/dtd/weblogic-ejb-
jar.dtd'>
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>HelloEJB</ejb-name>
    <jndi-name>Hello</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

You can see that the first XML file specifies the EJB name, remote interface name, home interface name, and bean class name. It specifies that `HelloEJB` is a stateless session bean, and the container manages its database transaction (transaction management is discussed later). The second file specifies the EJB's Java Naming and Directory Interface (JNDI) name, which the client uses to look up the EJB.

Now that the EJB has been developed, it is time to code the client. The steps for a client to invoke EJB services are as follows:

1. Instantiate an `InitialContext` instance.
2. Look up the home interface from JNDI.
3. Create a remote interface instance as the EJB reference.
4. Invoke EJB services via the reference.

These steps are illustrated in the JSP client listed in Listing 20-5. The output of the test is shown in Figure 20-1. Bounds checking coded into the method `getWelcomeMsg(int)` prevent the array index

from going out of range, so there is always a welcome message returned for any integers passed into the method.

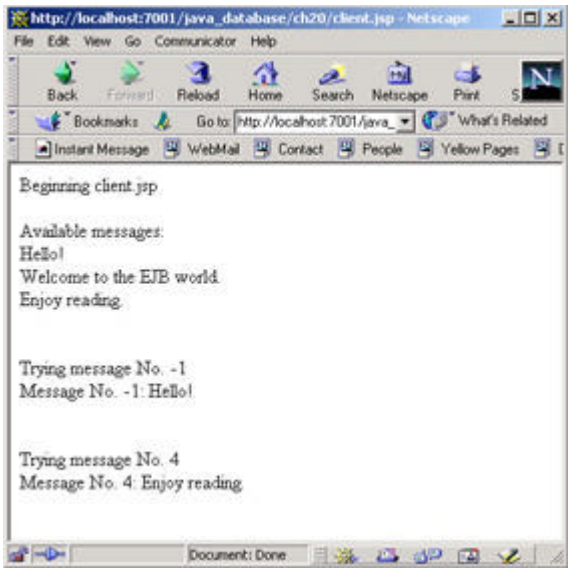


Figure 20-1: Test output after the JSP client shown in Listing 20-5

Listing 20-5: JSP client

```
<%@ page import="javax.naming.*, java.rmi.*,
java_database.ch20.HelloSLBean.*" %>

<%! private static Context ctx;
    // Instantiate InitialContext
    static {
       try {
          ctx = new InitialContext();
       } catch (Exception e) {
           System.out.println(e);
           System.out.println("Error trying to do one time initialization.");
       }
    }

    // A utility method that output a message to browser with a line
breaker <BR>
    public void toBrowser(String msg, JspWriter out) throws Exception {
        out.print(msg + "<BR>");
    }

    // Action starts
    toBrowser("Beginning client.jsp", out);
    // Look up home interface
    HelloEJBHome home = (HelloEJBHome)ctx.lookup("Hello");
```

```
    // Create remote interface
    HelloEJB bean = (HelloEJB)home.create();

    // Invoke EJB services through remote interface reference
    String[] messages= bean.getAllWelcomeMsgs();
       toBrowser("<BR>Available messages:", out);
    for(int i=0; i<bean.getNumberOfWelcomeMsgs(); i++) {
       toBrowser(messages[i], out);
    }

    //Test out-of-range index
    toBrowser("<BR>", out);
    toBrowser("Trying message No. -1", out);
    toBrowser("Message No. -1: " + bean.getWelcomeMsg(-1), out);

    toBrowser("<BR>", out);
    toBrowser("Trying message No. 4", out);
    toBrowser("Message No. 4: " + bean.getWelcomeMsg(4), out);
}
%>
```

Stateful Session Bean

A *stateful session bean* typically implements a conversational business process. A shopping cart of an online shopping application is a classical example of a stateful session bean. While a shopper searches the catalog and keeps dropping items into his or her shopping cart, the item list must be maintained. Obviously, different shoppers' shopping carts cannot be mixed. Only after a shopper finally checks out are the purchased items transferred into a persistent data store (such as a database).

A shopping cart application differs from a catalog-search application, for example, because each time the user searches the catalog, the search criteria are different. Such a service is usually implemented by a stateless session bean. This means that, unlike stateless session beans, a stateful session bean cannot serve multiple clients. An instance of a stateful session bean is associated with only one client. The instance retains the state on behalf of the client across multiple method invocations.

There is a one-to-one correspondence between user sessions (maintained as `HttpSession` objects) and the instances of a stateful session bean. The EJB container always delegates the method invocation from a given client to the same stateful session bean instance. The instance variables of the stateful session bean provide a convenient mechanism for the application developer to retain a client-specific state on the server. Note that such a state is not persistent on any data store. If the session is timed out, or if the server is crashed, the states are lost. If the states need to be persistent against server crash, entity beans must be used.

A client initiates the life cycle of a stateful session EJB in the same way as stateless session beans: by invoking the `create` method in its home interface. The EJB container instantiates the bean and then invokes the `setSessionContext` and `ejbCreate` methods in the session bean. The bean is now ready to have its business methods invoked.

Unlike stateless session beans, stateful session instances cannot be pooled because of the one-to-one correspondence between bean instances and session objects. At the end of the client session (for example, the online shopper checks out), the client invokes the `remove` method, and the EJB container calls the bean's `ejbRemove` method. The bean's instance is then ready for garbage collection.

Passivation and activation

A stateful session object lasts for the duration of the business process that typically spans multiple client-invoked business methods. The process may last for several minutes, hours, or even days. During its life cycle, the state of a stateful session instance may occupy a nontrivial amount of main memory on the server. In addition, the state may include expensive resources such as database connections. Because of these factors, it is important that the EJB container be able to reclaim the resources (when the available resources become too low) by saving the state into some form of secondary memory, such as a database or file systems. Later, when the state of the session object is once again needed for the invocation of a business method, the EJB container can restore the state from the saved image.

The process of saving the session objects' state to secondary storage is called *passivation*, whereas the process of restoring the state is called *activation*. The container typically passivates a session object when it needs to free resources in order to process requests from other clients or when it needs to transfer the session bean instance to a different process for load-balancing purposes. The container passivates the instance by invoking the `ejbPassivate` method and then serializing the instance and moving it to some secondary storage. When it activates the session objects, it restores the session bean's instance by deserializing the saved image of the passivated instance and then invoking the `ejbActivate` method.

For many session beans, including the example `YachtSessionEJB`, the passivation and activation processes do not require any programming effort from the bean developer. The bean developer has only to ensure that the objects held in the session bean instance variables are serializable at passivation. An object is serializable if it is an instance of a class that has implemented the `java.io.Serializable` interface.

Business processes and rules

In this chapter and the next two, we build a simple example application to demonstrate the use of stateful session beans and entity beans. Please note that these example EJBs are written for educational purposes only. They may not represent the best (or even appropriate) approaches for the hypothetical business process. The error and exception handling are not enough for these programs to be used in any a production release. Nevertheless, once you have fully understood the example code and have had it running, you can easily extend its functionality to meet your needs. In that sense, it serves as a good starting point for your own EJB application development.

The example application is used for a yacht club in its yacht cruise operation. From time to time, the club offers its members free yacht cruises. The business process includes the following tasks:

- Operate the yacht such as start, stop, accelerate and decelerate.
- Check the status of the yacht.
- Pick up a club members as a passengers (only members can come on board).
- Drop off passengers.

Since the business process involves multiple business-method invocations, it is implemented as a stateful session bean: `YachtSessionEJB`. The yacht and club members are business entities and can be modeled by entity beans. Although the `MemberEJB` and the `YachtEJB` are developed in the next two chapters, the `YachtSessionEJB` code is listed in Listings 20-6 and 20-7.

Listing 20-6: Remote and Home interfaces of YachtSessionEJB

```
/** YachtSessionEJB YachtSessionEJB
 *  @author: Andrew Yang
 *  @version: 1.0
 */
package java_database.YachtSessionSFBean;
```

```
import javax.ejb.*;
import java.rmi.*;
import java_database.common.*;
import java_database.MemberEBean.*;

public interface YachtSession extends EJBObject {
    public void start() throws RemoteException, YachtException;
    public void stop() throws RemoteException, YachtException;
    public int accelerate(int amount) throws RemoteException, YachtException;
    public int decelerate(int amount) throws RemoteException, YachtException;
    public void addPassenger(Member member) throws RemoteException,
YachtException;
    public boolean removePassenger(Member member) throws RemoteException;
    public YachtStatus getCurrentStatus() throws RemoteException;
}

/** YachtSessionEJB Home Interface
 *  @author: Andrew Yang
 *  @version: 1.0
 */
package java_database.YachtSessionSFBean;

import javax.ejb.*;
import java.rmi.*;
import java_database.YachtEBean.*;

public interface YachtSessionHome extends EJBHome {
    public YachtSession create(Yacht yacht) throws CreateException,
RemoteException;
}
```

Listing 20-7: YachtSessionEJB implementation class

```
/** YachtSessionEJB Implementation Class
 *  @author: Andrew Yang
 *  @version: 1.0
 */
package java_database.YachtSessionSFBean;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;
import java_database.common.*;
```

```
import java_database.YachtEBean.*;
import java_database.MemberEBean.*;

public class YachtSessionBean implements SessionBean, SessionSynchronization
{
   private int     currentVelocity;
   private boolean yachtRunning;
   private Vector  passengers;
   private Yacht   yacht;        // Remote reference to the Yacht

   private InitialContext ctx;
   private SessionContext context;

   public YachtSessionBean() {
      try {
         ctx = new InitialContext();
      } catch (Exception e) {
         System.out.println("Could not obtain InitialContext");
      }
   }

   public void ejbCreate(Yacht Yacht) throws RemoteException {
      yachtRunning = false;
      currentVelocity = 0;
      passengers = new Vector(10);
      this.yacht = yacht;
   }

   public void setSessionContext(SessionContext c) { this.context = c; }
   public void ejbRemove()    { }
   public void ejbActivate()  { }
   public void ejbPassivate() { }
   public void afterBegin()   { }
   public void beforeCompletion() { }
   public void afterCompletion(boolean committed) { }

   // business methods
   public void start() throws YachtException {
      if((passengers == null) || (passengers.size() == 0)) {
         throw new YachtException("Cannot start! No passengers in the
Yacht.");
      }
      yachtRunning = true;
   }
```

```
   public void stop() throws YachtException {
      if(currentVelocity > 2) {
         throw new YachtException("Too fast to stop. Decelerate first!");
      }
      yachtRunning = false;
   }

   public int accelerate(int amount) throws RemoteException{
      if(yachtRunning) {
         currentVelocity = (currentVelocity + amount <
yacht.getMaxVelocity()) ?
                           (currentVelocity + amount) :
yacht.getMaxVelocity();
      }
      return currentVelocity;
   }

   public int decelerate(int amount) {
      if(yachtRunning) {
         currentVelocity = (currentVelocity - amount > 0) ?
                           (currentVelocity – amount) : 0;
      }
      return currentVelocity;
   }

   public void addPassenger(Member member) throws RemoteException,
YachtException {
      if(passengers.size() == yacht.getCapacity()) {
         throw new YachtException("Yacht is full (" + Yacht.getCapacity() +
")");
      }
      passengers.addElement(member);
   }

   public boolean removePassenger(Member member) {
      return passengers.remove(member);
   }

  public YachtStatus getCurrentStatus() throws RemoteException {
     YachtStatus status = new YachtStatus();
     status.setCurrentVelocity(currentVelocity);
     status.setYachtRunning(yachtRunning);
     status.setMaxVelocity(yacht.getMaxVelocity());
```

```
      status.setMake(yacht.getMake());
      status.setModel(yacht.getModel());
      status.setCapacity(yacht.getCapacity());
      // Update the passenger list
      Passenger[] list = new Passenger[passengers.size()];
      int counter = 0;
      Enumeration enum = passengers.elements();
      while (enum.hasMoreElements()) {
         list[counter++] = (Member)enum.nextElement();
      }
      status.setPassengers(list);
      return status;
   }
}
```

You can see that a `YachtSessionEJB` instance is associated with a specific yacht, which is represented by an entity-bean instance. The business rules encapsulated by this entity bean include the following:

- If there are any passengers on board, the yacht can be started.
- If the speed has dropped to below a threshold (for example, 2 miles/hour), the yacht can be stopped.
- The yacht can be accelerated or decelerated between zero and maximum speed.
- The status of the yacht can be checked by calling the `getCurrentStatus` method.
- The session bean maintains a passenger list.
- Only club members can board the yacht.
- Whenever a new passenger is picked up, or a passenger leaves board, the passenger list is updated.

The YachtSessionEJB uses other helper or utility classes such as `YachtStatus`, `YachtException`, and so on. `YachtException` is the application exception that wraps the exceptions related to business rules. Do not be distracted by these utility classes at this time. They are discussed in subsequent chapters when we build the other parts of the example. Although the cruise process is implemented as a session bean, the business entities such as Yacht and Club Member would be better implemented as entity beans, as discussed next.

Entity Beans

An *entity bean* represents a business-entity object that exists in persistent storage mechanisms such as relational databases, object stores, or file systems. In practice, the persistent storage mechanism is usually a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. In a more complex situation, an entity bean may represent several related database tables, and each instance may correspond to a record in the table join. Some examples of business objects are customers, purchase orders, and products.

The syntax of the session bean and entity bean client-view API is almost identical. However, the two types of EJBs have different life cycles, different persistent management, and provide different programming styles to their clients.

Entity beans are normally used under the following conditions:

- The bean represents a business entity, not a procedure. For example, `MemberEJB` is be an entity bean, but `MemberRegistrationEJB` is likely a session bean.
- The state of the bean is required to be persistent. If the bean instance terminates or if the server is shut down, the bean's state still exists in persistent storage (for example, a database).

- The bean is shared (that is, accessed simultaneously) by multiple clients.

Primary keys

Similar to a row stored in a database table, each entity bean has a unique object identifier called a *primary key*. A customer entity bean, for example, might be identified by a customer number. Note that the primary key of an entity bean is an object. In most cases, it may be simply a `String` object, although it can be more complex. If the number (or integer) is used in the underlying database table, the Java wrapper classes (such as `java.lang.Integer` or `java.lang.Long`) need to be used for the primary key class. The primary key object enables the client to locate a particular entity-bean instance.

Persistent Storage

When an entity bean is created, the data that the EJB represents is placed into the persistent storage, typically through a database insert operation, and a copy of that data is stored in the memory as part of the EJB instance. Whenever the attributes of the in-memory EJB instance are modified, their underlying persistent counterparts are automatically updated by the EJB container.

Since an entity bean's value is stored in a persistent manner, multiple clients can access the same data at the same time. In others words, entity beans allow shared access just as a relational database allows multiple users to access its data simultaneously. EJB containers can implement two or more clients requesting accesses to the same data in a variety of ways. Because these clients might want to change the same data, it's important that entity beans work within transactions. Typically, the EJB container provides transaction management. In this case, bean developers or application assemblers specify the transaction attributes in the bean's deployment descriptor. A bean developer does not have to code the transaction boundaries in the bean — the container marks the boundaries based on the transaction attributes specified in the deployment descriptor. Transaction attributes are discussed later in this chapter.

Because the state of an entity bean is saved in a persistent storage, it exists beyond the lifetime of the application or the server process. For example, data stored in a database still exists even after you shut down the database server or the applications it serves.

Bean-managed persistence

There are two types of persistence for entity beans: bean managed and container managed. With bean-managed persistence (BMP), the EJB itself is responsible for writing all of the logic necessary for synchronizing the data between itself and the persistent store. The entity bean code that a bean developer writes contains all the calls that access the database. A BMP bean must manage the four following operations:

- Add an entry to the persistent store.
- Remove an entry from persistent store.
- Update the persistent store with the current attribute values of the entity bean instance.
- Update the attributes of bean instance with values stored in persistent store.

Effectively, a bean developer is responsible for coding all of the database queries. However, the container still controls the life cycle of the bean itself.

Cross-Reference BMP entity beans are discussed in more detail in Chapter 21.

Container-managed persistence

Container-managed persistence (CMP) means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if you redeploy the same entity bean on different J2EE-compliant application servers that use different databases, you will not need to modify or recompile the bean's code. In short, CMP entity beans are more portable. To generate the data-access calls, the container needs information that a bean developer provides in the entity bean's deployment descriptor.

Like a table in a relational database, an entity bean may be related to other entity beans. For example, in a college enrollment application, `StudentEJB` and `CourseEJB` are related because students enroll in classes. With container-managed persistence, the EJB container takes care of the relationships. For this reason, relationships in entity beans with container-managed persistence are often referred to as container-managed relationships (CMRs).

Cross-Reference You learn more on CMP entity beans in Chapter 22.

In addition to session beans and entity beans, the EJB 2.0 introduced a third type of EJB: message driven bean, as discussed next.

Message-Driven Beans

A message-driven bean is a new type of EJB. It acts as a listener for the Java Message Service (JMS) API and processes messages asynchronously. That means the client does not need to wait the complete of the tasks it delegated to the message driven bean. Instead, it can continue on other tasks as soon as it has dropped the message to the JMS. The Message-driven beans were introduced as recently as late 2001 in EJB Specification 2.0 to fill up the gap in interactions between the J2EE platform and the Java Message Service (JMS). The messages may be sent by any J2EE component (such as an application client, another enterprise bean, or a Web component) or by a JMS application or system that does not use J2EE technology at all.

A message-driven bean is similar to an event listener, except that it receives messages instead of events. The calling client does not need to wait for the completion of the services it requests. As soon as the message is dropped to the JMS message queue or the topic, the calling client moves on to other tasks. Message-driven beans currently process only JMS messages, but in the future they may be used to process other kinds of messages as well.

A visible difference between message-driven beans and session or entity beans is that clients do not access message-driven beans through interfaces. In fact, clients do not directly access message-driven beans at all. A message-driven bean can only be accessed by an EJB container once a JMS message is received. As a consequence, message-driven beans have no home or remote interfaces. Only the implementation class needs to be developed. As you can see from the example in Listing 20-8, there is actually only one specific method, `onMessage`, that the bean developer needs to code.

Listing 20-8: MessageEchoEJB source code

```
package java_database.MessageEchoMDEJB;

import javax.ejb.*;
import javax.jms.*;

/**
 * This message driven bean echos the message text it received on the
standard output.
 * It can be extended to implement any business rules upon receiving the
message.
 * @author: Andrew Yang
 * @version: 1.0
 */
public class MessageEchoBean implements MessageDrivenBean, MessageListener {
   private MessageDrivenContext context;
```

```
   /** Public, default constructor */
   public MessageEchoBean () {}

   /** Set the MessageDrivenContext */
   public void setMessageDrivenContext(MessageDrivenContext context) {
     this.context = context;
  }

   /** ejbCreate is required by EJB Specification */
   public void ejbCreate() { }

   /** ejbRemove is required by EJB Specification */
   public void ejbRemove() { }

   /**
    * Message handling, the business logic. The message text is printed on
the
    * output screen. <BR> It can be extended to implement any business rules
    * upon receiving the message.
    */
   public void onMessage(Message message) {
     TextMessage textmessage = (TextMessage)message;
     try {
         String s = textmessage.getText();
         System.out.println("A message received: " + s);
      } catch(JMSException e) {
         ex.printStackTrace();
      }
   }
}
```

In the following respects, a message-driven bean resembles a stateless session bean:

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message driven bean instance available. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages (for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object).

When a JMS message arrives, the container calls the message-driven bean's `onMessage` method to process the message. The `onMessage` method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The `onMessage` method may call helper methods, or it may invoke a session or entity bean to process the information in the message or to store it in a database.

A message may be delivered to a message-driven bean within a transaction context, so that all operations within the `onMessage` method are part of a single transaction. If message processing is rolled back, the message will be redelivered.

Session beans and entity beans are able to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, it may be better not to use blocking synchronous receives in a server-side component. To receive messages asynchronously, a message-driven bean has to be used.

You can see that the development of a message-driven bean is fairly straightforward. The `onMessage` method is the only method a bean developer has to write. Note that various application servers have different mechanisms to write text to their council screen. Before deploying the `MessageEchoEJB` to your favorite application server, you may need to replace `println` function in the following code with function calls appropriate to the server you use:

```
System.out.println("A message received: " + s);
```

During the deployment phase, the bean is associated to a JMS destination, either a message queue or a topic. The JMS destination is where the message-driven bean receives its message. It is specified in the deployment descriptor as follows:

```
<message-driven-destination>
    <jms-destination-type>javax.jms.Topic</jms-destination-type>
</message-driven-destination>
<message-driven-descriptor>
    <destination-jndi-name>SimpleTopic</destination-jndi-name>
</message-driven-descriptor>
```

Notice that the `MessageEchoEJB` is associated to the JMS topic, "`SimpleTopic`".

So far you have learned all three types of EJBs. Let us moved to EJB transaction management.

EJB Transactions

Transactions are a big part of most enterprise applications. A transaction consists of multiple data-updating steps as an indivisible unit of work. Execution of a transaction may end in two ways: *commit* or *rollback*. When a transaction commits, the data modifications made by its statements are saved. If one of the multiple steps within a transaction fails, the transaction rolls back, undoing the effects of all steps in the transaction.

The EJB architecture provides for two kinds of transaction demarcation: container-managed transaction and bean-managed transaction, as discussed in the following sections.

Container-Managed Transaction

For EJBs with *container-managed transactions*, the EJB container sets the boundaries of the transactions. Container-managed transactions can be used with any type of EJBs: session bean, entity beans, or message-driven beans. Container-managed transactions significantly simplify development because the EJB code does not explicitly mark the transaction boundaries. The code does not include statements that begin and end the transaction.

Typically, the container begins a transaction immediately before an EJB method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method in the current EJB standard. Container-managed transactions do not require all methods to be associated with transactions. When deploying an EJB, one specifies which of the bean's methods are associated with transactions by setting the transaction attributes.

A transaction attribute specified in the deployment descriptor controls the scope of a transaction. A transaction attribute may have one of the following values:

- `Required` means that the container ensures that the bean method is invoked with a transaction. If the calling client has a transaction, the container passes it to the bean method. If the calling client does not have a transaction, the container starts one and then invokes the bean method.
- `RequiresNew` means that the container always starts a new transaction when invoking the bean method. If the calling client has a transaction, the container suspends it and starts a new one. This is not a nested transaction because the outcome of the new transaction has no impact on the suspended one. If the calling client does not have a transaction, the container creates a new transaction and invokes the bean method.
- `Mandatory` states that the calling client must have a transaction, which is propagated to the bean method being invoked. If the calling method does not have a transaction, the container throws a `javax.transaction. TransactionRequiredException`.
- `NotSupported` means that the method cannot handle transactions. If the calling client has a transaction, the container suspends it before invoking the bean method. If the calling client does not have a transaction, the container immediately invokes the bean method.
- `Supports` states that the bean method accepts a transaction if available but does not require the container to create a new one. If the calling client has a transaction, the container propagates it to the bean method. If the calling client does not have a transaction, the container just invokes the bean method.
- `Never` means that the bean method is not expecting a transaction. If the calling client has a transaction, the container throws a `java.rmi.RemoteException`. If the calling client does not have a transaction, the container just invokes the bean method.

Table 20-2 summarizes the behavior of the container for each of these transaction attributes.

Table 20-2: Transaction Attributes

Transaction Attribute	Client Has Transaction	Client Has No Transaction
Required	Transaction Propagated	New Transaction Started
RequiresNew	Transaction Suspended	New Transaction Started
Mandatory	Transaction Propagated	Throws `TransactionRequiredExceptio n`
NotSupported	Transaction Suspended	No Action
Supports	Transaction Propagated	No Action
Never	Throws `RemoteException`	No Action

Because transaction attributes are stored in the deployment descriptor, they can be changed during several phases of J2EE application development: EJB creation, application assembly, and deployment. However, an enterprise bean developer is responsible for specifying the attributes when the bean is first created. The attributes should be modified only by other developers who are assembling components into larger applications. Do not expect the person who is deploying the J2EE application to specify the transaction attributes.

You can specify the transaction attributes for the entire enterprise bean or for individual methods. If you've specified one attribute for a method and another for the bean, the attribute for the method takes precedence. As an example, the transaction attribute of the `YachtSessionEJB` may be specified as follows:

```
<enterprise-beans>
   <session>
       <ejb-name>YachtSessionEJB</ejb-name>
       <transaction-type>Container</transaction-type>
   </session>
```

```
<enterprise-beans>

<assembly-descriptor>
   <container-transaction>
      <method>
        <ejb-name>YachtSessionEJB</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

The asterisk (*) between the `method-name` tags indicates all methods. This listing specifies the transaction attribute for all methods as `Required`. For finer granularity, you can assign each method a different transaction attribute.

Bean-Managed Transaction

In a *bean-managed transaction*, the code explicitly marks the boundaries of the transaction. Note that only session or message-driven beans can use bean-managed transactions. An entity bean cannot have bean-managed transactions; it must use container-managed transactions instead. Although beans with container-managed transactions require less coding, they have one limitation: when a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation makes coding your bean difficult, you should consider using bean-managed transactions. For example, if multiple databases are accessed and a two-phase commit is required, a bean-managed transaction should be used.

Note

Entity beans must use container-managed transactions. Session beans and message-driven beans can use either container-managed transactions or bean-managed transactions.

When coding a bean-managed transaction for session or message-driven beans, the bean developer must decide whether to use Java Database Connectivity (JDBC) or Java Transaction Architecture (JTA) transactions. The JDBC transaction has been discussed intensively in previous chapters and is not repeated here. However, using JDBC transactions is not recommended in EJB development. JDBC transactions are usually only used when wrapping legacy code inside a session bean.

In many enterprise applications, the client needs to combine the invocation of multiple methods into a single transaction. The methods can be on the same EJB, or they can be on multiple EJBs. To demarcate transactions across multiple method invocations, it is recommended that you use the JTA APIs. Actually, only one interface in the JTA APIs, `javax.transaction.UserTransaction`, needs to be used to demarcate a JTA transaction. This interface has a few useful methods, such as `begin`, `commit`, and `rollback`. A bean method may look like this:

```
public void withdrawCash(double amount) {
   UserTransaction ut = context.getUserTransaction();
   try {
      // Start transaction
      ut.begin();
      // perform tasks
      updateChecking(amount);
      machineBalance -= amount;
      insertMachine(machineBalance);
```

```
      // Commit when all tasks succeed
      ut.commit();
   } catch (Exception ex) {
      try {
         //Some tasks failed. Rollback.
         ut.rollback();
      } catch (SystemException syex) {
         throw new EJBException("Rollback failed: " + syex.getMessage())
      }
      throw new EJBException("Transaction failed: " + ex.getMessage());
   }
}
```

The preceding code snippet demonstrates the usage of `UserTransaction` methods. The `begin` and `commit` invocations delimit the updates to the database. If the updates fail, the code invokes the `rollback` method and throws an `EJBException`.

To summarize the discussions in this section, Table 20-3 lists the types of transactions allowed for the different types of EJBs.

Table 20-3: Allowed Transaction Types for EJBs

EJB Type	Container-Managed Transaction	Bean-Managed Transaction JTA Transaction	JBDC Transaction
Entity	Allowed	Not Allowed	Not Allowed
Session	Allowed	Allowed and Recommended	Allowed but not Recommended
Message Driven	Allowed	Allowed and Recommended	Allowed but not Necommended

Summary

This chapter provides a brief introduction to the vastly broad area of EJB. You learn how to develop EJBs by writing a simple session bean: `HelloEJB`. You also study a stateful session bean, `YachtSessionEJB`. The following specific topics are discussed:

- Session beans
- Entity beans
- Message-driven beans
- EJB transactions

In the next chapter, you will learn about the entity beans using bean-managed persistence.

Chapter 21: Bean-Managed Persistence

In This Chapter

Bean-managed persistence is discussed in detail in this chapter. The entity object persistence model is explained. and the development of bean managed persistence EJBs are illustrated with examples. After reading this chapter, you should be able to develop your own EJBs using bean managed persistence.

Entity-Object Persistence

A *business entity* is a business object representing some information an enterprise maintains. A business entity has state, represented by the values of its data fields, and this state is kept persistently in a persistent store, typically in a database. As discussed in Chapter 20, an entity-bean instance represents a business-entity object stored in such persistent storage.

The methods of the entity bean class allow the client to access the business-entity object's state through the bean instance. In other words, the state of an entity object is maintained and persisted outside the bean instance, in persistent storage. This mechanism is illustrated in Figure 21-1.

Figure 21-1: Entity object's state maintained in persistent store

Separating the state of the entity-bean objects from the bean instances has the following advantages:

- **Facilitates data persistence.** The separation permits the life cycle of the entity object's state to go beyond the life cycle of the entity-bean instances and even beyond the life cycle of the JVMs in which the instances are created.
- **Facilitates the transaction.** Persistent stores (for example, relational databases), instead of the EJB container, are responsible for the implementation of transaction behavior to keep the atomicity, consistency, isolation, and durability (the so-called ACID properties). This takes advantage of the built-in functionality of a relational database management system and greatly simplifies the implementation of EJB container.
- **Promotes distributed component model.** The separation makes it possible to implement the entity object's state so that it is accessable concurrently from multiple JVMs running on different network nodes. This is essential to the implementation of EJB server clusters that provide load balancing and fail-over for large-scale applications.
- **Improves accessibility of non-Java applications.** The separation makes it possible to externalize an entity object's state in a representation suitable for non-Java applications. For instance, if a relational database maintains the state of the entity objects, the state is available to any application that can access the database via SQL statements.

The entity-bean architecture is flexible regarding the choice of the type of the persistent store. It can be a relational database; a hierarchical database; an object-oriented database; an XML database; an LDAP server; a file system; an application; and so on. In practice, however, most applications use relational databases as persistent stores. When the state of an entity object is maintained in a persistent data store, an entity-bean instance must use an API specific to the persistent store to access the state of the associated entity object.

As you have learn in Chapter 20, an entity-bean instance can access the state of its associated entity object using two access styles: BMP and CMP. The BMP implements the persistence in the EJB class or in one or more helper classes, whereas the CMP delegates the handling of its persistence to the EJB container. BMP is discussed in detail in following sections, and CMP is the topic of the next chapter.

Bean-Managed Persistence

With bean-managed persistence, the bean developer writes database-access calls using a persistent storage-specific API. In most cases, the persistent stores are relational databases, and the JDBC provides a unified API for data access. Effectively, the bean developer is responsible for coding all of the database queries. The data-access calls can be coded directly into the EJB implementation class, as you see in the examples given later in this chapter, or can be encapsulated in a data-access object (DAO).

If database queries are coded directly in the EJB class, it may be difficult to adapt the entity bean to work with a different type of database or a database that has a different schema. Encapsulating data-access calls in a DAO makes it easier to adapt the EJB's data access to different schemas or different databases, since only the DAO, instead of the whole EJB, needs to be modified or rewritten. Detailed discussion on DAO is beyond the scope of this book. Interested readers can find the use of DAOs in many J2EE design-pattern books.

The main advantage of BMP is its simple deployment process. When an entity bean uses BMP, no deployment tasks are necessary to adapt the bean to the database type or database schema. Everything is *hard coded* in the implementation. As you can see, this is also the main disadvantage of BMP, because an entity bean using BMP is generally tied up with the database type and schema. The tight coupling between the EJB and database makes BMP entity bean less flexible and less reusable across different operational environments.

However, an entity bean using BMP can achieve some degree of independence of the EJB code from the database type and schema by using the DAOs discussed earlier in this section. Since the entity bean uses the DAO to access the entity object's state stored in the database, the EJB implementation class is not tightly coupled with the database. The DAO provides the appropriate interface for customizing the data-access logic to a different database type or schema. For example, a `CustomerEJB` may use a DAO to access the Customer Relationship database on Oracle and use another DAO to access the Sales database on Sybase. If access to a new database schema (or database type) is needed, you need only to code a new DAO and do not need to change the EJB code at all.

BMP reduces the complexity of the deployment process, but the price you pay is the loss of flexibility. You should choose BMP or CMP based on your own requirements.

Primary Key

Understanding primary keys is an essential part of understanding entity EJBs (both BMP and CMP beans). One of the fundamental concepts of entity EJBs is that they must be accessible concurrently by multiple clients. This does not mean that multiple clients need to access the same bean instance; rather, they need to have access to the state of the same business-entity object stored in the database. To achieve this, the EJB architects incorporate an entity bean primary key concept similar to the primary key of a relational database. The primary key is a *unique identifier* for the entity object. The primary key enables the client to locate a particular business entity it needs to access.

Since every entity bean must have a unique primary key, you can compare two entity EJB instances without actually using the instances themselves. Rather, you can just compare the contents of their primary keys. If two EJB instances with the same home interface have the same primary key, they are considered identical, since they represent the same underlying entity object. For example, a customer entity bean may be identified by a customer number. Two `CustomerEJB` instances with the same customer number are considered identical because they represent the same customer. The synchronization between these two instances and the state of the `Customer` object stored in the database is the EJB container's responsibility.

Note that the primary key of an entity bean is a Java class object. In most cases, your primary key class is a `String`, an `Integer`, or some other class that belongs to the J2SE or J2EE standard libraries. For some entity beans, you need to define your own primary key class. For instance, if the bean has a composite primary key (that is, one composed of multiple fields), you must create a primary key class.

For BMP entity beans, a primary key class must meet these requirements:

- The class is serializable.

- All fields are declared as public.
- The class has a public default constructor.
- The accessors are public.
- The class implements the `hashCode()` and `equals(Object other)` methods.

Listing 21-1 shows a primary class for the `PartEJB` that represents the business object stored in the `Part` table in the database. The `Part` table has a composite key: the `productId` and `vendorId` fields. The primary key class then has two attributes: `productId` and `vendorId`.

Listing 21-1: A primary key class example

```
public class PartKey implements java.io.Serializable {
   public String productId;
   public String vendorId;

   public ItemKey() { };

   public ItemKey(String productId, String vendorId) {
     this.productId = productId;
     this.vendorId  = vendorId;
   }

   public String getProductId() {  return productId;  }
   public String getVendorId()  {  return vendorId;   }

   public boolean equals(Object other) {
      if(other instanceof PartKey) {
         return (productId.equals(((ItemKey)other).productId)
                && vendorId.equals(((ItemKey)other).vendorId));
      }
      return false;
   }

   public int hashCode() {
      return productId.concat(vendorId).hashCode();
   }
}
```

With BMP, the `ejbCreate` method (to be discussed in next section) assigns the input parameters to instance variables and returns the primary key object. A client can fetch the primary key of an entity bean by invoking the `getPrimaryKey` method of the `EJBObject` class as follows:

```
PartKey id = (PartKey)Part.getPrimaryKey();
```

The entity bean retrieves its own primary key by calling the `getPrimaryKey` method of the `EntityContext` class as shown here:

```
PartKey id = (PartKey)context.getPrimaryKey();
```

At the deployment phase, you specify the primary key class in the entity bean's deployment descriptor, as shown next:

```
<entity>
   <ejb-class>PartEBean.PartBean</ejb-class>
   <home>PartEBean.PartHome</home>
   <remote>PartEBean.Part</remote>
   <persistence-type>Bean</persistence-type>
   <prim-key-class> PartEBean.PartKey</prim-key-class>
   … …
</entity>
```

The entity bean's deployment descriptor specifies that the `PartEJB` uses bean-managed persistence and that its primary key class is `PartKey`. In the next chapter, you see that the deployment descriptor for a CMP entity bean contains much more contents than that for a BMP bean.

In the rest of this chapter, a BMP entity bean, `MemberEJB`, is built as an example. A `MemberEJB` instance represents a row in the Member Table. The Member Table has four columns, as shown in Table 21-1. Since the Member Table has a simple primary key, `member_id`, the `java.lang.String` class is used as the `MemberEJB`'s primary key class.

Table 21-1: Sample Data Stored in the Member Table

member_id	last_name	first_name	membership_year
m001	Dole	Jane	20
m002	Dole	John	4
m003	Corleone	Fredo	12
m004	Smith	Mike	7

Create and Delete Entity Objects via Entity Beans

With BMP, the bean developer writes all the database-access calls. These database access-operations include the creation and deletion of the business objects, the synchronization between the attribute values of entity beans and the state of the corresponding business object, and the search for specific business objects.

The `EJBHome` interface and `EntityBean` interface specify the life-cycle methods for these operations, and the bean developer implements these methods by using database-access APIs such as JDBC. As an example, the home interface of the `MemberEJB` is shown in Listing 21-2. A Member EJB represents a row in the Member Table, which has these four columns, as seen in Table 21-1:

- `member_id`
- `last_name`
- `first_name`
- `membership_year`.

Listing 21-2: Home interface of MemberEJB

```
/** MemberEJB Home Interface
 * @author: Andrew Yang
 * @version: 1.0
 */
package java_database.MemberEBean;
```

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface MemberHome extends EJBHome {
  public Member create(String id, String lastName, String firstName, int
membershipYear)
                throws CreateException, RemoteException;
  public Member findByPrimaryKey(String id)
                throws FinderException, RemoteException;
  public Collection findByMembershipYear(int minYear)
                throws FinderException, RemoteException;
}
```

The home interface may define one or more `create` methods. The overloaded `create` methods must have different signatures. The home interface `create` methods all return the EJB's remote interface, which in the preceding example is `Member`, and they all throw `CreationException` and `RemoteException`. The bean developer may define a `create` method to throw additional application-specific exceptions to address their specific requirement in exception handling.

Note that the `MemberHome` interface extends the `EJBHome` interface, which is listed in Listing 21-3. The `EJBHome` interface defines two `remove` methods. The first method removes an EJB object identified by a handle. A *handle* is an object that provides a reference to an EJB object and can be stored in persistent storage. The second method removes an EJB object identified by its primary key.

Listing 21-3: EJBHome interface

```
Import java.rmi.RemoteExceptions;

Public interface EJBHome extends java.rmi.Remote {
   Void remove(Handle handle)     throws RemoteException,
RemoveException;
   Void remove(Object primaryKey) throws RemoteException,
RemoveException;
   EJBMetaDate getEJBMetaData()   throws RemoteException;
   HomeHandle  getHomeHandle()    throws RemoteException;
}
```

Like the session bean implementation discussed in the previous chapter, the BMP entity-bean implementation class implements the `create` and `remove` methods defined in the home interface. However, the corresponding methods for the `create` methods are named `ejbCreate` in the implementation class, and they return a primary key object instead of the remote interface. The corresponding methods for the `remove` methods are named `ejbRemove` in the implementation class, and they do not take any argument. The differences in name and signature between the interface methods and implementation class methods are due to the fact that the life cycle of an entity bean is managed by the EJB container. It is critical to understand the EJB life cycles in order to master the use of EJB.

When a client invokes a `create` method, the EJB container invokes the corresponding `ejbCreate` method. Typically, an `ejbCreate` method in an entity bean performs the following tasks:

- Inserts the entity state into the database
- Initializes the instance variables
- Returns the primary key

Thus, when an entity bean is *created*, the data that the EJB represents is placed in the database, and a copy of that data is stored in the EJB container's memory as part of the EJB instance. If a record with the same primary key already exists in the database, a `CreationException` will be thrown, and the EJB object will not be instantiated.

For each `ejbCreate` method, you must write an `ejbPostCreate` method in the entity-bean implementation class. (The `ejbPostCreate` method is defined in the `EntityBean` interface that discussed later.) The EJB container invokes `ejbPostCreate` immediately after it calls `ejbCreate`. Unlike the `ejbCreate` method, the `ejbPostCreate` method can invoke the `getPrimaryKey` method. In most of situations, however, your `ejbPostCreate` methods are empty.

A client deletes an entity bean by invoking the `remove` method. This invocation causes the EJB container to call the `ejbRemove` method, which deletes the business-entity object from the database. It should be noted that the business-entity object is deleted from the database. The entity-bean instance is not necessarily garbage collected. It is just disassociated with a specific entity object and may be returned to the EJB pool maintained by the EJB container and ready to represent another business-entity object (for example, another row in the same table). If the `ejbRemove` method encounters a system problem, it should throw the `EJBException`. If it encounters an application error, it should throw a `RemoveException`.

Note

Calling `ejbCreate` creates a business entity (for example, a row in a database table) in the persistent storage. Calling `ejbRemove` deletes a business-entity object from the persistent storage.

You can find the implementation of the `ejbCreate` and `ejbRemove` methods for `MemberEJB` later in this chapter, under "An Example BMP Entity Bean — MemberEJB." You can see that the SQL commands are coded to insert into and delete from the database.

An entity object can also be created or deleted directly by native database operations. For example, if a SQL script deletes a row from a table, the entity object represented by the row is deleted, and the corresponding entity-bean instances become disassociated with the entity object. If a client attempts to invoke a business method on an entity bean instance after its underlying business object has been deleted from the database, the client receives `NoSuchObjectException`.

Find Entity Object

Calling the `ejbCreate` method creates a business entity in the database. In many situations, the business entity already exists in the database, and you just need to instantiate an EJB instance to represent it. The `finder` methods are designed just for this purpose.

All entity beans have `finder` methods. Similar to the select command in SQL statements, `finder` methods are used by clients to locate business objects stored in the database and to associate them with entity-bean instances. Each `finder` method can have different logic for locating the entity object. The logic may find one entity object or a group of entity objects. If a `finder` method returns a single reference to the remote reference, it will return the first valid occurrence of the bean that is located. If a `finder` method returns the `Collection` interface, it will return zero or more references to entity beans. The client can then iterate over the collection to access each of the available beans.

All home interfaces must have a `findByPrimaryKey(PrimaryKeyClass key)` method. Since lookup operations are common for all entity beans, a standard mechanism for looking up one entity bean by its unique identifier (that is, the primary key) is defined as the `findByPrimaryKey()` method. All entity beans have this method available and return exactly one reference to a bean in the form of the

bean's remote interface. In addition to the `findByPrimaryKey()method`, `other findByxxx` methods can be defined in the home interface to implement application-specific business logic. As seen in Listing 21-2, the `MemberEJB`'s home interface defines two `finder` methods. The `findByPrimaryKey(String id)` method returns the remote interface. But the `findByMembershipYear(int minYear)` returns a `Collection` because there may be zero or more members that have established their membership for a certain number of years.

Like the `create` methods, each `finder` method must have a matching method in the implementation classes. The method name in the implementation class is the same as that in the home interface, except a prefix `ejb` is added, for example, `ejbFindByPrimaryKey`. The method must have the same signature. However, the return type may be different. If only one entity-bean reference is returned, the `ejbFindByxxx` method returns the primary key, instead of the remote interface.

The implementation of the `MemberEJB`'s `finder` methods is shown in Listing 21-6. With BMP, the SQL code is written in these implementations to locate the entity objects in the database. The implementation of `ejbFindByPrimaryKey` method may look strange to you because it uses a primary key (`String` in this case) for both the method argument and return value. However, remember that a client does not call `ejbFindByPrimaryKey` directly. The EJB container calls the `ejbFindByPrimaryKey` method. The client invokes the `findByPrimaryKey` method, which is defined in the home interface and returns the remote interface.

The following list summarizes the rules for the `finder` methods you implement in an entity bean class with BMP:

- All `finder` methods defined in the home interface must be implemented.
- At a minimum, the `ejbFindByPrimaryKey` method must be implemented.
- A `finder` method name must match the name of the corresponding method in the home interface and must start with the prefix `ejb`.
- The method must be public and cannot be final or static
- The return type must be the primary key or a collection of primary keys.

The throws clause may include the `javax.ejb.FinderException` and exceptions that are specific to your application. If a `finder` method returns a single primary key but the requested entity does not exist, the method should throw the `javax.ejb.ObjectNotFoundException` (a subclass of `FinderException`). If a `finder` method returns a collection of primary keys, but it does not find any objects, it should return an empty collection.

Synchronization of Bean Instance Variable and State of Persistent Object

Recall from earlier in this chapter that the state of an entity object is kept in the database. The attribute values of the EJB instance are merely the image of the entity object's state. Since multiple clients can access the same entity objects via multiple EJB instances, the EJB container must keep the attribute values of the EJB instances and the state of the corresponding entity object synchronized. The synchronization mechanisms are different between the BMP and CMP entity beans. With BMP, the EJB container maintains the synchronization by calling the `ejbLoad` and `ejbStore` methods you have coded in the EJB implementation class.

The `ejbLoad` and `ejbStore` methods are defined in the `EntityBean` interface, which is shown in Listing 21-4. The `EntityBean` interface defines a group of life-cycle methods for the EJB container to use. All entity-bean implementation classes extend the `EntityBean` interface. You may have noticed that the implementation class shown in Listing 21-6 (in Section: An Example BMP Entity EJB) extends the `EntityBean` interface. Therefore, all the methods defined in the `EntityBean` interface must be implemented in the EJB class. With BMP, you need to write a certain amount of code to implement the `ejbLoad` and `ejbStore` methods. The other methods are typically empty or have only a few lines of code.

Listing 21-4: EntityBean home interface

```
public interface EntityBean extends EnterpriseBean{
   public void setEntityContext(EntityContext ctx)
               throws EJBException, RemoteException;
   public void unsetEntityContext()
               throws EJBException, RemoteException;
   public void ejbRemove()
               throws RemoveException, EJBException, RemoteException;
   public void ejbActivate()
               throws EJBException, RemoteException;
   public void ejbPassivate()
               throws EJBException, RemoteException;
   public void ejbLoad()
               throws EJBException, RemoteException;
   public void ejbStore()
               throws EJBException, RemoteException;
}
```

If the EJB container needs to synchronize the instance variables of an entity bean with the corresponding values stored in a database, it invokes the `ejbLoad` and `ejbStore` methods. The `ejbLoad` method refreshes the instance variables from the database, and the `ejbStore` method writes the variables to the database. The client may not call `ejbLoad` and `ejbStore` directly. In fact, the synchronization between the EJB instance variables and the entity-object state is completely transparent to the client. From the client's point of view, the EJB instance is the same as the entity object.

If a business method is associated with a transaction, the container invokes `ejbLoad` before the business method executes. Immediately after the business method executes, the container calls `ejbStore`. Because the container invokes `ejbLoad` and `ejbStore`, you do not have to refresh and store the instance variables in your business methods. Since the EJB classes rely on the container to synchronize the instance variables with the database, their business methods should be associated with transactions.

Normally, the database resides on a different network node from the EJB container in which EJBs are deployed. Because the implementation of a business method typically accesses the entity object's state, each invocation of a business method may result in a network trip to the database. If a transaction includes multiple business invocations, the resulting multiple accesses to the database over the network may increase the transaction overhead. Many bean developers want to reduce such overhead. To accomplish this, the EJB architecture allows the entity bean to cache the entity object's state, or part of the state, within a transaction. Rather than making repeated calls to the database to access the entity object's state, the EJB instance loads the object's state at the beginning of a transaction and caches it in its instance variables. The database is not updated until the transaction is ready to commit.

To facilitate such caching, the EJB container invokes the `ejbLoad` method on the instance prior to the first business method invocation in a transaction. The instance can utilize the `ejbLoad` method to load the entity object's state, or part of its state, into the instance's variables. Then, subsequent calls to business methods on the instance read and update the cached state instead of making calls to the database. When the transaction ends, the EJB container invokes the `ejbStore` method on the instance. If the previously invoked business methods update the state cached in the instance variables, calling the `ejbStore` will resynchronize the entity object's state stored in the database with the cached state.

The container invokes the `ejbLoad` and `ejbStore` methods, plus the business methods between the `ejbLoad` and `ejbStore` methods, in the same transaction context. When, from these methods, the EJB instance accesses the entity object's state in the database, the database properly associates all the multiple database accesses with the transaction.

Because the EJB container needs a transaction context to drive the `ejbLoad` and `ejbStore` methods on an EJB instance, caching of the entity object's state in the instance variables works reliably only if the entity-bean methods execute in a transaction context.

The `ejbLoad` and `ejbStore` methods must be used with great caution for entity beans that do not execute with a defined transaction context. These are entity beans with methods that use transaction attributes `NotSupported`, `Never`, and `Supports` (the transaction attribute in Chapter 20). If the business methods can execute without a defined transaction context, the instance should cache only the state of the immutable entity objects. Fore these entity beans, an instance can use the `ejbLoad` method to cache the entity object's state but should never call the `ejbStore` method.

Cross-Reference See the section "Container-Managed Transaction" in Chapter 20 for discussions of the transaction attributes.

If the `ejbLoad` and `ejbStore` methods cannot locate an entity in the underlying database, they should throw the `javax.ejb.NoSuchEntityException`. This exception is a subclass of `java.ejb.EJBException`. Because `EJBException` is a subclass of `RuntimeException`, you do not have to include it in the `throws` clause. When `NoSuchEntityException` is thrown, the EJB container wraps it in a `RemoteException` before returning it to the client.

Like stateful session beans, the entity bean instances may be passivated and activated, that is, saved into the secondary storage and moved back to the memory. The EJB container calls the `ejbPassivate` and `ejbActivate` (both defined in `EntityBean` interface) for EJB passivation and activation. To maintain synchronization, the `ejbStore` is called before calling `ejbPassivate` so that the latest version of the EJB instance variable is saved to the database. The `ejbLoad` is called immediately after calling the `ejbActivate;` thus, the variables of the activated EJB instance are refreshed with the current state of the entity object.

Business Methods

The business methods contain the business logic you want to encapsulate within the entity bean. Usually, the business methods do not access the database, allowing you to separate the business logic from the database-access code. All business methods that can be invoked by clients are defined in the remote interface. Note that some utility methods that are not meant for the client to use are typically not defined in the remote interface. Instead, they are normally declared as `private` (or `protected`) methods and are implemented in the bean class.

Listing 21-5 shows the remote interface of the example `MemberEJB`. In this simple example, only `getters` and `setters` of the instance variables are defined. Because the client can get the primary key by calling the `context.getPrimaryKey()` method, the `getter` for the primary key is not necessary. Since the bean should not change its unique identifier during its life cycle, the `setter` for the primary key is not defined.

Listing 21-5: Remote interface of MemberEJB

```
/** MemberEJB Remote Interface
 * Note: It does not provide accessor for memberId since it is the
primary key and will
 * be accessible by the context.getPrimaryKey() method on the client.
 * @author: Andrew Yang
```

```
 * @version: 1.0
 */
package java_database.MemberEBean;

import java.rmi.*;
import javax.ejb.*;

public interface Member extends EJBObject {
   // accessors
   public String getLastName()    throws RemoteException;
   public String getFirstName()   throws RemoteException;
   public int getMembershipYear() throws RemoteException;

   // mutators
   public void setLastName(String s)    throws RemoteException;
   public void setFirstName(String s)   throws RemoteException;
   public void setMembershipYear(int n) throws RemoteException;
}
```

Note that the `MemberEJB` is a simplified example intended to illustrate the fundamentals of EJB. For a real-life entity bean, more business methods are defined in the remote interface. For example, the yacht club may allow the members who leave the club and then rejoin to reinstate their seniority after they have been members for more than three years recently. To implement this business logic, a business method may be added as follows:

```
void reinstateMembershipYear(int formerYear) thorws YachtException
{
   if (membershipYear < 3) {
      throw new YachtException("Have not stayed 3 years this round!");
   }
   membershipYear =+ formerYear;
}
```

But then you have to add more logic to make sure the seniority is only reinstated once. The code must prevent a member from reinstating his seniority multiple times.

The following requirements for the signature of an entity bean business method are the same as those for session beans:

- The method name must not conflict with a method name defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The methods must be public because clients call them.
- The method modifier cannot be final or static.
- The arguments and return types must be legal types for the Java RMI API. Typically, that means the arguments and return types must either be Java datatypes or implement the `Serilizable` interface.
- The `throws` clause must include `java.rmi.RemoteException`.

The `throws` clause may include the exceptions that you define for your application. The `reinstateMembershipYear` method, for example, throws `YachtException`. To indicate a system-level problem, a business method should throw `EJBException`.

All the business methods defined in the remote interface must be implemented in the bean class. The methods in the bean-implementations class must have the same function name, function signature, and return type as the corresponding methods defined in the remote interface. You will see the implementation of `MemberEJB` in the next section.

An Example BMP Entity Bean — MemberEJB

An example application is being built through Chapters 20-22. The example application is used by a yacht club for its cruise operation. From time to time, the club offers its member free yacht cruises. The cruising yacht allows only the club members on board. The business entity, *club member*, is implemented as a BMP entity bean. The underlying database table, member, is illustrated in Table 21-1. The home interface and remote interface of `MemberEJB` are shown in Listing 21-3 and Listing 21-5, respectively. The implementation class is shown in Listing 21-6.

Listing 21-6: MemberEJB implementation class

```
/** MemberEJB Implementation class.
 * System.out.printline() is used to display the error message to the
standard output.
 * You should replace System.out.println() with approperate log functions of
the EJB
 * server you use, so that the error message will be written to the log file.
 * @author: Andrew Yang
 * @version: 1.0
 */
package java_database.MemberEBean;

import java.rmi.*;
import java.util.*;
import java.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class MemberBean implements EntityBean {
   // instance attributes
   private String memberId;
   private String lastName;
   private String firstName;
   private int membershipYear;

   private InitialContext ctx;
   private EntityContext context;

   /** default constructor */
   public MemberBean() {
      try {
         ctx = new InitialContext();
```

```
      } catch (NamingException ne) {
         System.out.println("Could not obtain InitialContext");
      }
   }

   public String ejbCreate(String id, String nLast, String nFirst, int mYear)
               throws CreateException {
      Connection con = null;
      PreparedStatement ps = null;
      try {
         con = getConnection();
         ps = con.prepareStatement("INSERT into member values(?, ?, ?, ?)");
         ps.setString(1, id);
         ps.setString(2, nLast);
         ps.setString(3, nFirst);
         ps.setInt(4, mYear);
         if (ps.executeUpdate() != 1) {
            System.out.println("Insert data failed!");
            throw new CreateException("JDBC could not create a row.");
         }
         // assign instance attrubute values
         memberId = id;
         lastName = nLast;
         firstName = nFirst;
         membershipYear = mYear;
      } catch (SQLException sqe) {
         // Check if the exception was due to an existing entry in the
database.
         try {
            ejbFindByPrimaryKey(id);
         } catch (ObjectNotFoundException e) {
            System.out.println("Ambiguous SQLException: " + sqe);
            throw new CreateException("Ambiguous SQLException");
         }
         System.out.println("A member with this ID already exists.");
         throw new DuplicateKeyException("A member with this ID already
exists.");
      } finally {
         cleanup(con, ps);
      }
      return memberId;
   }

   public void ejbPostCreate(String name, int age) {
```

```
      // do nothing
   }

   /**
    * The ejbRemove gets 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.
    */
   public void ejbRemove() {
      Connection con = null;
      PreparedStatement ps = null;
      try {
         con = getConnection();
         memberId = (String) context.getPrimaryKey();
         ps = con.prepareStatement("DELETE FROM member WHERE member_id=?");
         ps.setString(1, memberId);
         if (ps.executeUpdate() < 1) {
            String error = "Member (" + memberId + ") not found";
            System.out.println(error);
            throw new NoSuchEntityException(error);
         }
      } catch (SQLException sqe) {
         System.out.println("SQLException:  " + sqe);
         throw new EJBException (sqe);
      } finally {
         cleanup(con, ps);
      }
   }

   public String ejbFindByPrimaryKey(String pk) throws
ObjectNotFoundException {
      Connection con = null;
      PreparedStatement ps = null;
      try {
         con = getConnection();
         ps = con.prepareStatement("SELECT last_name, first_name,
membership_year " +
                                   "FROM member WHERE member_id=?");
         ps.setString(1, pk);
         ps.executeQuery();
         ResultSet rs = ps.getResultSet();
         if (rs.next()) {
            memberId = pk;
            lastName = rs.getString(1);
```

```
            firstName = rs.getString(2);
            membershipYear = rs.getInt(3);
         } else {
            String error = "ejbFindByPrimaryKey: Member (" + pk + ") not
found";
            System.out.println(error);
            throw new ObjectNotFoundException (error);
         }
      } catch (SQLException sqe) {
         System.out.println("SQLException:  " + sqe);
         throw new EJBException (sqe);
      } finally {
         cleanup(con, ps);
      }
      return pk;
   }

   public Collection ejbFindByMembershipYear(int minYear)
          throws ObjectNotFoundException {
      Connection con = null;
      PreparedStatement ps = null;
      Vector v = new Vector();       // returing object
      try {
         con = getConnection();
         ps  = con.prepareStatement("SELECT memberId FROM member " +
                                    "WHERE membershipYear>?");
         ps.setInt(1, minYear);
         ps.executeQuery();
         ResultSet rs = ps.getResultSet();
         String pk;
         while (rs.next()) {
            pk = rs.getString(1);
            v.addElement(pk);
         }
      } catch (SQLException sqe) {
         System.out.println("SQLException:  " + sqe);
         throw new EJBException (sqe);
      } finally {
         cleanup(con, ps);
      }
      return v;
   }
```

```
   // methods defined in EntityBean interface
   public void ejbActivate() {  }
   public void ejbPassivate() {  }
   public void setEntityContext(EntityContext c) { this.context = c; }
   public void unsetEntityContext() {  }

   public void ejbLoad() {
      Connection con = null;
      PreparedStatement ps = null;
      memberId = (String)context.getPrimaryKey();
      try {
         con = getConnection();
         ps = con.prepareStatement("Select last_name, first_name,
membership_year " +
                                   "FROM member WHERE member_id=?");
         ps.setString(1, memberId);
         ps.executeQuery();
         ResultSet rs = ps.getResultSet();
         if (rs.next()) {
            lastName = rs.getString(1);
            firstName = rs.getString(2);
            membershipYear = rs.getInt(3);
         } else {
            System.out.println ("Member EJB could not load data");
            throw new NoSuchEntityException("Could not locate Member with
ID:" +
                                            memberId);
         }
      } catch (SQLException sqe) {
         System.out.println ("SQLException: " + sqe);
         throw new EJBException(sqe);
      } finally {
         cleanup(con, ps);
      }
   }

   public void ejbStore() {
      Connection con = null;
      PreparedStatement ps = null;

      try {
         con = getConnection();
         ps = con.prepareStatement("UPDATE member SET last_name=?,
firstName=?, " +
```

```
                                    "membership_year=? WHERE member_id=?");
          ps.setString(1, lastName);
          ps.setString(2, firstName);
          ps.setInt(3, membershipYear);
          ps.setString(4, memberId);

          if (ps.executeUpdate() < 1) {
             System.out.println("Could not locate member with ID: " +
memberId);
              throw new NoSuchEntityException("Could not locate member with ID:
" +
                                              memberId);
          }
       } catch (SQLException sqe) {
          System.out.println ("SQLException: " + sqe);
          throw new EJBException(sqe);
       } finally {
          cleanup(con, ps);
       }
    }

    // business methods defined in the remote interface
    public String getLastName()    { return lastName;       }
    public String getFirstName()   { return firstName;      }
    public int getMembershipYear() { return membershipYear; }

    public void setLastName(String s)    { lastName = s;       }
    public void setFirstName(String s)   { firstName = s;      }
    public void setMembershipYear(int n) { membershipYear = n; }

    // utility methods
    private Connection getConnection() throws java.sql.SQLException {
       try {
          javax.sql.DataSource ds = (javax.sql.DataSource)
          ctx.lookup("java:comp/env/jdbc/YachtClubDB");
          return ds.getConnection();
       } catch(NamingException ne) {
          System.out.println("UNABLE to get a connection!");
          throw new EJBException(ne);
       }
    }

    private void cleanup(Connection con, PreparedStatement ps) {
       try {
```

```
            if (ps != null) {
               ps.close();
            }
         } catch (Exception e) {
            System.out.println("Error closing PreparedStatement: "+e);
            throw new EJBException (e);
         }
         try {
            if (con != null) {
               con.close();
            }
         } catch (Exception e) {
            System.out.println("Error closing Connection: " + e);
            throw new EJBException (e);
         }
      }
}
```

The implementation class uses all the methods defined in the home and remote interfaces, as well as the methods defined in the `EntityBean` interface. Table 21-2 summarizes the database-access calls in the `MemberBean` class.

Table 21-2: Database-access Operations in MemberBean

Method	SQL Statement	Functionality
ejbCreate	INSERT	Create entity object in database and initialize instance variables
ejbRemove	DELETE	Delete entity object from database and disacssociate bean instance
ejbFindByPrimaryKey	SELECT	Locate entity object with given primary key and associate bean instance with the object
ejbFindByMembershipYear	SELECT	Locate all entity objects that have been members longer than a given number of years
ejbLoad	SELECT	Refresh instance variables with current object state stored in database
ejbStore	UPDATE	Update state stored in database with current instance variable values

The business methods of the `MemberBean` class (such as `getLastname()`, `setLastName(String)`, and so on) are absent from this table because they do not access the database. Instead, these business methods update the instance variables, which are written to the database when the EJB container calls `ejbStore`. After the EJB container calls the `ejbLoad` method, the instance variables are refreshed with the current entity-object state, and the `getter` function retrieves the refreshed variables. Another developer might have chosen to access the database in the business methods of the `MemberBean` class. This choice is one of those design decisions that depend on the specific needs of your application. For the simple entity bean like the example `MemberEJB`, database access from business methods is not warranted. But in some cases, the business logic may be complex, and there may be a need to access the database from the business method.

Before accessing a database, you must connect to it. With BMP, you must write the program to get a database connection. In the `MemberBean` class, this is achieved by calling the utility method `getConnection()`. Typically, you get a database connection by following these steps:

1. Specify the database name. In the example, the database name is "`java:comp/env/jdbc/YachtClubDB`".
2. Obtain the `DataSource` associated with the logical name by searching the Java Naming and Directory Interface (JNDI).
3. Get the `Connection` from the `DataSource`.

The database's JNDI name is specified in the deployment descriptor during the deployment phase as shown here:

```
<entity>
   … …
   <resource-ref>
      <res-ref-name>jdbc/YachtClubDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
   </resource-ref>
</entity>
```

When coding an EJB, you must decide how long it will retain the connection. Generally, you have two choices: either hold the connection for the lifetime of the bean, or hold it only during each database call. Your choice determines the method (or methods) in which your bean connects to a database.

With long-term connections, an EJB holds a database connection for its entire lifetime. Because the bean connects and disconnects just once, its code is slightly easier to write. But there's a trade-off: other components cannot acquire the connection. This is normally not recommended.

Briefly held connections allow many components to share the same connection. Because the EJB container manages a pool of database connections, enterprise beans can quickly obtain and release the connections. In the `MemberBean` class, all methods hold the connection briefly. Before the method returns, another utility method, `cleanup()`, is called, which returns the connection to the EJB container's connection pool by calling `con.close()`. In addition, the `cleanup` method closes resources such as `preparedStatement`.

In almost all the application servers on the market today, the EJB container maintains a pool of database connections. This pool is transparent to the enterprise beans. When an EJB requests a connection, the container fetches one from the pool and assigns it to the bean. Because the connection has already been made, the bean quickly gets a connection. The bean may release the connection after each database call (as seen in `MemberBean`), since it can rapidly get another connection. Because such a bean holds the connection for a short time, the same connection can be shared sequentially by many beans. To further boost the performance of your application, consider using EJB design patterns and best practices such as value objects (also called data transfer objects sometimes) as described next.

Using Value Objects for Better Performance

You may have noticed that the `MemberEJB` has `getters` and `setters` for all the EJB instance variables except the primary key (see Listing 21-5). The example is oversimplified for demonstrating the basic concept of the entity bean. In practice, the member table usually has more columns, such as `street_address`, `city`, `state`, `zip_code`, `country`, `phone`, `fax`, `email`, `last_due_payment_date`, `spouse_name`, and so on. In that case, you may want to consider different strategies to reduce the network-performance overhead.

Typically, the clients reside in different network nodes from the EJB container where entity beans and their instances are deployed. It is important to realize that every invocation of an entity bean method is

typically a remote call that requires using a network resource. (The local EJB specified in EJB 2.0 is an exception). Even the entity bean caches the object state and therefore does not need to access the database every time; the network trip from the client box to EJB container box cannot be avoided. A client of an EJB needs to be aware of this potential performance hit when using an entity bean. More important, entity-bean developers should consider this when designing and developing an entity bean. You must look at how the bean's clients might use the methods in the bean's remote interface and then design the bean to be used efficiently and effectively.

For certain applications, most of the clients may typically want access to only a few attributes of the bean (or columns in a row of a database table), and they may usually only retrieve or update only one or two attributes per interaction. That means that in the yacht club example, a given client may want to see the member's `ID` and `membership_year;` another client may want to see only a member's `phone` and `e-mail;` yet another client may only want to see a member's `street_address, city, state` and `zip_code`.

In this situation, you can design the entity bean to promote individual access to each persistent attribute. You design the bean's remote interface just as I do in the `MemberEJB`'s remote interface: there is a separate `getter` and `setter` for each attribute. To address the real-life scenario, you add `getters` and `setters` for other attributes such as `streetAddress, city, state, zipCode, country, phone, fax, e-mail, lastDuePaymentDate, spouseName`, and so on.

In most real-life situations, it is difficult to predict how the clients will use the EJB. If an entity object has many attributes, it is unrealistic to assume that the client only needs very few of them for every invocation. If clients normally need most of the attributes when they invoke a bean's business method, the multiple network trips may cause significant performance overhead. In addition to multiple network trips, this approach has the following drawbacks:

- **Transaction overhead.** The EJB architecture recommends letting the EJB container handle transaction demarcation. Notice that each method invocation involves a separate transaction and that the EJB container must perform some house keeping work such as calling `ejbLoad` and `ejbStore`. If several attributes need to be updated, the separate methods cause the EJB container a lot of unnecessary work. Although it is possible for the client to get around this by using client-side transaction demarcation, the client developer must write transaction code that violates the sprit of EJB architecture.
- **Difficulty in validating business logic.** It is difficult to validate business logic when such validation requires more than one attribute value. For example, suppose you need to add a new business logic to the `MemberEJB` that invites the member and the member's spouse to a party if the member has paid his due in the past month and lives in a specific city. Inside the `inviteMemberCouple` method, you have to have code to check `lastDuePayment`, `city`, and `spouseName` (to make sure it is not empty). This requires three invocations of the `getter` method. For a large-scale system, since the entity objects are shared by many clients, the three separate `setters` may have the risk of certain temporary data inconsistency.

A recommended practice to overcome such drawbacks is to use a value object. Rather than accessing each entity attribute individually, you can set up your entity bean's remote interface to access all attributes in one call. Essentially, the client makes one request to either retrieve or update all the attributes of the bean instance. All persistent attributes are accessed via one remote call and within one transaction. This reduces the network and transaction overhead to a minimum. This approach is especially suitable for situations where clients generally require access to most (or even all) of the attributes at the same time.

To access all persistent attributes in one method invocation, your entity bean uses a value object. A value object is nothing more than a JavaBean, that is, a Java class with public `getters` and `setters` that implement the `Serializable` interface. You basically remove the individual `getters` and `setters` from the remote interface (and from the implementation class) and place the equivalent methods in the value object class. Then you add methods to the EJB remote interface: one to retrieve the entire value object and the other to set new values. Of course, you include the implementation for these new methods in the EJB's implementation class.

For the `MemberEJB` example, assuming the Member Table contains more columns, you may create a value object called `MemberInfoVO` as shown in Listing 21-7. The remote interface is then modified as shown in Listing 21-8.

Listing 21-7: Value object MemberInfoVO

```
/** Value object for MemberEJB. Class name is MemberInfoVO
 * @author: Andrew Yang
 * @version: 1.0
 */
package java_database.MemberEBean;

import java.util.*;

public class MemberInfoVO {
   // attributes
   private String memberId;
   private String lastName;
   private String firstName;
   private String streetAddress;
   private String city;
   private String state;
   private String zipCode;
   private String country;
   private String phone;
   private String fax;
   private String email;
   private String spouseName;
   private int membershipYear;
   private Date lastDuePaymentDate;

   // accessors
   public String getMemberId()         { return memberId;          }
   public String getLastName()         { return lastName;          }
   public String getFirstName()        { return firstName;         }
   public String getStreetAddress()    { return streetAddress;     }
   public String getCity()             { return city;              }
   public String getState()            { return state;             }
   public String getZipCode()          { return zipCode;           }
   public String getCountry()          { return country;           }
   public String getPhone()            { return phone;             }
   public String getFax()              { return fax;               }
   public String getEmail()            { return email;             }
   public String getSpouseName()       { return spouseName;        }
```

```
   public int  getMembershipYear()     { return membershipYear;     }
   public Date getLastDuePaymentDate() { return lastDuePaymentDate; }

   // mutators
   public void setMemberId(String s)      { memberId = s;           }
   public void setLastName(String s)      { lastName = s;           }
   public void setFirstName(String s)     { firstName = s;          }
   public void setStreetAddress(String s) { streetAddress = s;      }
   public void setCity(String s)          { city = s;               }
   public void setState(String s)         { state = s;              }
   public void setZipCode(String s)       { zipCode = s;            }
   public void setCountry(String s)       { country = s;            }
   public void setPhone(String s)         { phone = s;              }
   public void setFax(String s)           { fax = s;                }
   public void setEmail(String s)         { email = s;              }
   public void setSpouseName(String s)    { spouseName = s;         }
   public void setMembershipYear(int n)   { membershipYear = n;     }
   public void lastDuePaymentDate(Date d) { lastDuePaymentDate = d; }
}
```

Listing 21-8: Remote interface of MemberEJB using value object

```
/** MemberEJB Remote Interface. It use a value object, MemberInfoVO, to
reduce network
 * and transaction overhead.
* @author: Andrew Yang
 * @version: 1.0
 */
package java_database.MemberEBean;

import java.rmi.*;
import javax.ejb.*;

public interface Member extends EJBObject {
   public MemberInfoVO getMemberInfo()    throws RemoteException;
   public void updateMemberInfo(MemberInfoVO mInfo)    throws
RemoteException;
}
```

In practice, you may need to address a variety of data-access requirements. The required granularity may well fall between two extremes. Then you may want to design multiple value objects for your entity bean. This approach gives you more fine-grain control for accessing an entity object's state and still lets you retain the performance benefits that value objects provide. It works best for entity beans with many individual attributes, but where the bean's clients typically need access to only a small number of them.

In this approach, you group those individual `getters` and `setters` into subsets that clients logically want to access together. You can have duplicated `getters` and `setters` in different subsets. Then

you set up separate value objects for each subset, that is, you define each value object to contain and handle the attributes required by a particular client's use of the entity bean. In the `MemberEJB` example, you may set up a value object to handle all the contact information (for example, address, phone number, and so on) and another value object to handle all the membership related information (for example, membership year, last due payment date, and so on).

It is important to keep performance considerations in mind when developing entity beans, especially for BMP entity beans, since you are responsible for writing all database-access code. Using value objects to improve performance is a proven best practice. Use it whenever you feel it is appropriate.

Summary

This chapter provides an overview of the following topics:

- The implementation and use of the `ejbCreate`, `ejbRemove`, `ejbLoad`, and `ejbStore` methods as well as finder methods
- The implementation of EJB's business methods
- The use of value objects for improved performance

Chapter 22: Container-Managed Persistence

In This Chapter

Extending the discussion in previous chapter, in this chapter you will learn the container managed persistence in details. You also see how the EJBs developed since Chapter 20 can be put together to build a simple application.

CMP Entity Bean — a Rebirth after EJB2.0

The term *container-managed persistence* means that the EJB container handles all database access the entity bean requires. The bean's code contains no database-access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if you redeploy the same entity bean on different J2EE-compliant application servers that use different types of databases, you will not need to modify or recompile the bean's code. In short, your entity beans are more portable and easier to develop.

It sounds like a very nice concept. However, in the early stage of EJB adoption (up to EJB specification version 1.1), CMP entity beans have been labeled as slow or even as a performance nightmare. The persistent state was stored as bean-instance variables, and bean developers often had to use third-party tools to map bean attributes to database fields (the so called O/R mapping). The integration between such tools and application servers had given developers enough headaches. It was also difficult to handle relationships between related objects such as `Orders` and `Line-Items`.

Fortunately, all these problems have been addressed in the EJB 2.0 specification released in September 2001. With EJB 2.0, the EJB container uses the information that bean developers provide in the entity bean's abstract schema to generate all the data-access calls. As part of an entity bean's deployment descriptor, the abstract schema defines the bean's persistent fields as well as relationships. The term *abstract* distinguishes this schema from the physical schema of the underlying data store.

Bean developers specify the name of an abstract schema in the deployment descriptor. This name is referenced by queries written in the Enterprise JavaBeans Query Language (EJB QL). For a CMP entity bean, you must define an EJB QL query for every `finder` method (except `findByPrimaryKey`). The EJB QL query determines the database query the EJB container executes when the `finder` method is invoked. You see the examples of EJB QL later in this chapter.

There are two types of container-managed fields in a CMP bean: *persistent* and *relational*. The persistent fields of an entity bean are stored in the underlying data store. Collectively, these fields constitute the state of the bean. At runtime, the EJB container automatically synchronizes this state with the database. During deployment, the container typically maps the entity bean to a database table and maps the persistent fields to the table's columns.

A relationship field is like a foreign key in a database table — it identifies a related bean. Like a persistent field, a relationship field is virtual and is defined in the enterprise bean class with access methods. But unlike a persistent field, a relationship field does not represent the bean's state.

According to EJB 2.0 specification, the implementation classes for CMP beans must be abstract. That means no instance of these implementation classes can be directly instantiated. The EJB container generates a concrete class based on the code you have written and all the information you have provided in the deployment descriptor. These concrete classes contain all the database-access calls that deal with the persistent state as well as the relationship between business entities. Instances of such concrete classes are instantiated during runtime and invoked by the client. The generation and instantiation of such concrete classes are totally transparent to bean developers and client programmers.

To further boost the performance, EJB 2.0 also introduced the local (both home and remote) interfaces that provide support for lightweight access by local clients. The local interfaces are standard Java interfaces that do not inherit from RMI. When a client (it may be another EJB) accesses the EJB on the

same network note, going through the local interfaces avoids the network-service overhead and significantly improves performance.

With all the new features introduced in EJB 2.0, the CMP bean has been proliferating rapidly in the past year. For the first time, many enterprises are seriously considering using CMP entity beans in their mission-critical applications. In 2002 JavaOne conferences, a significant number of technical sessions are devoted to the development, deployment, and proper uses of CMP entity beans. It is fair to say that EJB 2.0 has given CMP EJB a new life. You should probably consider using CMP beans instead of BMP beans for all applications in which the use of entity bean is appropriate. Now let us move to the development of CMP EJB.

Developing CMP EJBs

CMP and BMP entity beans are very similar. They both represent persistent business objects and have the same client-side behavior. The major difference is the database- access code. In BMP, all database access calls are implemented by the bean developer. In CMP, the implementation is generated by a persistent manager.

A *persistent manager* is the software that takes care of persistence in place of the bean developer. It is normally part of the EJB container. In many EJB books and documents (including this book), *persistent manager* and *EJB container* are used interchangeably.

Home and Remote Interfaces

Since CMP entity beans have the same client-side views as BMP entity beans, there is no difference between coding the remote interfaces and home interfaces. The home interface defines all the life-cycle methods, whereas the remote interface defines all the business methods accessible by clients. You have to define at least one `create` method and at least one `finder` (that is, the `findByPrimaryKey`) method in the home interface. You may define as many optional `create` and `finder` methods as you need.

The remote interface defines all the business methods that a client can invoke. You may also want to define local interfaces for the local clients' access. The local home interface looks similar to the remote interface, except that it extends `EJBLocalHome` instead of `EJBHome;` and the local interface is the same as the remote interface, except that it extends `EJBLocalObject` instead of `EJBObject`. In response to this similarity, I focus the discussion primarily on remote interfaces.

As an example, a CMP entity bean is developed in our yacht club application to represent a yacht entity. There is a `yacht` table in the underlying database; the table has five columns, as seen in Table 22-1.

Table 22-1: Sample Data Stored in Yacht Table

yacht_name	builder	Engine_type	capacity	Max_velocity
Whaler	Grand Banks	Twin Diesel	12	35
Liberty	Bristol	Single Diesel	10	25
Yifei	Eastbay	Twin Diesel	8	27
Lighting	Eastbay	Twin Diesel	8	30

The `yacht_name` column is the primary key. All of the five columns are persistent and therefore have corresponding persistent fields in the entity bean `YachtEJB`. The home interface and remote interface of the `YachtEJB` are shown in Listing 22-1 and Listing 22-2, respectively.

Listing 22-1: Home interface of YachtEJB

```
/** YachtEJB Home Interface. CMP is used.
```

```
 * @author: Andrew Yang
 * @version: 1.0
 */
package java_database.YachtEBean;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface YachtHome extends EJBHome {
  public Yacht create(String yachtName, String builder, String
engineType,
                      int capacity, int maxVelocity)
               throws CreateException, RemoteException;
  public Yacht findByPrimaryKey(String yachtName)
               throws FinderException, RemoteException;
  public Collection findAllYachts()
               throws FinderException, RemoteException;
  public Collection findYachtsCapacityMoreThan(int minCapacity)
               throws FinderException, RemoteException;
}
```

Listing 22-2: Remote interface of YachtEJB

```
/** YachtEJB Remote Interface. CMP is used.
 * @author: Andrew Yang
 * @version: 1.0
 */
package java_database.YachtEBean;

import java.rmi.*;
import javax.ejb.*;
import common.*;
import YachtSessionSFBean.*;

public interface Yacht extends EJBObject {
   public YachtSession createYachtSession() throws RemoteException;
   public String       getBuilder()         throws RemoteException;
   public String       getEngineType()      throws RemoteException;
   public int          getCapacity()        throws RemoteException;
   public int          getMaxVelocity()     throws RemoteException;
}
```

Although multiple `create` methods can be defined, only one `create` method is defined in the home interface for simplicity. The `create` method takes all of the five persistent fields as argument. The remote interface defines the `getters` for four out of the five persistent fields, except the primary key field. The client can get the primary key field, `yachtName`, by calling the `getPrimaryKey` method of the `EJBHome` class or `EntityContext` class.

Three `finder` methods are defined in the home interface with different data-retrieval criteria. The `findByPrimaryKey` is required and returns only one reference to `YachtEJB` that the primary key identifies. It may return null if no match is found. The other two methods return a collection of references to `YachtEJB` objects.

There are only `getters`; no `setters` are defined in the remote interface; therefore the `YachtEJB` is apparently defined as read only (after created) from the clients' point of view. If the clients also need to modify the persistent state, `setters` for the persistent fields must be defined in the remote interface. Although `getters` and `setters` are defined in the remote interface, you do not need to code their implementations, as you see in the next section.

Implementation Class with Minimum Code

In EJB 1.1, a persistent field was identified in the deployment descriptor and also identified as a public instance variable of your bean implementation class. In EJB 2.0, this approach has been radically changed. Persistent fields are still identified in the deployment descriptor, but they are not identified as public-instance variables. Instead, they are identified through specialized `getters` and `setters` that you must write.

For example, you have to write `getYachtName`, `setBuilder`, and so on in the `YachtEJB` implementation class. What is intriguing is that these methods are declared as `abstract` and are implemented automatically by the EJB container during the deployment phase. That makes the implementation class also abstract; thus, no instance can be instantiated directly for the implementation class. The EJB container uses the information you provide in the deployment descriptor to automatically generate a concrete class with all the database-access implementations. The objects of these container-generated, concrete classes are used at runtime for clients' invocation. The implementation class of the example `YachtEJB` is shown in Listing 22-3.

Listing 22-3: Implementation class of YachtEJB

```
/** YachtEJB Implementation Class. CMP is used.
 * @author: Andrew Yang
 * @version: 1.0
 */
package java_database.YachtEBean;

import java.rmi.*;
import java.util.*;
import java.sql.*;
import javax.ejb.*;
import javax.naming.*;
import common.*;
import YachtSessionSFBean.*;

public abstract class YachtBean implements EntityBean {
   private EntityContext  context;
```

```
   private InitialContext ctx;

   public YachtBean() {
      try {
         ctx = new InitialContext();
      } catch (Exception e) {
         System.out.println("Problem getting InitialContext!");
      }
   }

   public void setEntityContext(EntityContext ctx) {  this.context = ctx;  }
   public void unsetEntityContext() {   }
   public void ejbActivate()  {  }
   public void ejbPassivate() {  }
   public void ejbLoad()      {  }
   public void ejbStore()     {  }
   public void ejbRemove() throws RemoveException {  // nothing to code }

   // Container managed fields
   public abstract String getYachtName();
   public abstract String getBuilder();
   public abstract String getEngineType();
   public abstract int getCapacity();
   public abstract int getMaxVelocity();
   public abstract void setYachtName(String s);
   public abstract void setBuilder(String s);
   public abstract void setEngineType(String s);
   public abstract void setCapacity(int n);
   public abstract void setMaxVelocity(int n);

   public String ejbCreate(String yachtName, String builder, String
engineType,
                           int capacity, int maxVelocity)
                throws CreateException {
      // You have to call accessor methods here.
      setYachtName(yachtName);
      setBuilder(builder);
      setEngineType(engineType);
      setCapacity(capacity);
      setMaxVelocity(maxVelocity);
      // If int values passed in are zero, pull the value from the constant
file.
      if(capacity <= 0)    { setCapacity(YachtConstants.CAPACITY); }
      if(maxVelocity <= 0) { setMaxVelocity(YachtConstants.MAX_VELOCITY); }
```

```
      // Always return null
      return null;
  }

   public void ejbPostCreate(String yachtName, String builder, String
engineType,
                             int capacity, int maxVelocity) {
      // nothing to code
   }

   // Business Methods
   public YachtSession createYachtSession() {
      YachtSession session = null;
      try {
         YachtSessionHome home = (YachtSessionHome)ctx.lookup(

"java:comp/env/ejb/YachtSessionEJB");
         // In order to create a YachtSession instance, we must pass
         // in a reference to this Yacht EJB's remote stub.
         session = (YachtSession)home.create((Yacht)context.getEJBObject());
      } catch (Exception e) {
         System.out.println("Failed to create YachtSession: " + e);
      }
      return session;
   }
}
```

You may be impressed by how little you have to code. Compared with the BMP implementation given in Listing 21-6, the CMP implementation class has much less code. This leads to one of the major advantages of CMP entity bean — a fast development cycle.

From Listing 22-3, you see that the implementation classes have defined five abstract `getters` and five abstract `setters` for the five persistent fields. The concrete implementation is automatically generated by the EJB container based on the information provided in the deployment descriptor. As an example, the `YachtEJB`'s deployment descriptor for WebLogic Application Server 6.0 is shown in Listing 22-4. If you use another application server, your deployment descriptor files may look slightly different. When your application contains multiple EJBs, and some other components such as servlets, the deployment descriptor can be very long and complex. Therefore, you should never write your deployment descriptors with a text editor. Instead, always use the deployment tool provided by your application server. These XML files should always be generated by your application server, just as all the concrete implementations of the abstract EJB methods are automatically generated by EJB container.

Note

Do not use a text editor to write a deployment descriptor. Use the deployment tool provided by your application server.

Listing 22-4: Deployment descriptor for YachtEJB

```
# First DD File – J2EE Standard
```

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>YachtEJB</ejb-name>
      <home>YachtEBean.YachtHome</home>
      <remote>YachtEBean.Yacht</remote>
      <ejb-class>YachtEBean.YachtBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>YachtBean</abstract-schema-name>
      <primkey-field>yachtName</primkey-field>
      <cmp-field><field-name>builder</field-name></cmp-field>
      <cmp-field><field-name>engineType</field-name></cmp-field>
      <cmp-field><field-name>capacity</field-name></cmp-field>
      <cmp-field><field-name>maxVelocity</field-name></cmp-field>
      <ejb-ref>
         <description>The YachtEJB does a lookup for YachtSession
beans.</description>
         <ejb-ref-name>ejb/YachtSessionEJB</ejb-ref-name>
         <ejb-ref-type>Session</ejb-ref-type>
         <home>YachtSessionSFBean.YachtSessionHome</home>
         <remote>YachtSessionSFBean.YachtSession</remote>
      </ejb-ref>
      <query>
        <query-method>
          <method-name>findAllYachts</method-name>
          <method-params></method-params>
       </query-method>
       <ejb-ql><![CDATA[WHERE yachtName IS NOT NULL]]></ejb-ql>
      </query>
      <query>
        <query-method>
          <method-name>findYachtsCapacityMoreThan</method-name>
           <method-params>
             <method-param>int</method-param>
           </method-params>
        </query-method>
        <ejb-ql><![CDATA[FROM YachtBean cb WHERE cb.capacity > ?1]]></ejb-
ql>
```

```
      </query>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <security-role>
      <description>The group of users allowed to access
YachtEJBs.</description>
      <role-name>ValidYachtClubUsers</role-name>
    </security-role>
    <container-transaction>
      <method>
        <ejb-name>YachtEJB</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>


# Second DD File – Weblogic Specific
<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 6.0.0
EJB//EN'
'http://www.bea.com/servers/wls60/ejb20/dtd/weblogic-ejb-jar.dtd'>
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>YachtEJB</ejb-name>
    <entity-descriptor>
      <entity-cache>
        <max-beans-in-cache>150</max-beans-in-cache>
      </entity-cache>
      <persistence>
       <persistence-type>
         <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
         <type-version>6.0</type-version>
         <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml</type-storage>
       </persistence-type>
       <persistence-use>
         <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
         <type-version>6.0</type-version>
       </persistence-use>
      </persistence>
    </entity-descriptor>
    <reference-descriptor>
```

```
      <ejb-reference-description>
        <ejb-ref-name>ejb/YachtSessionEJB</ejb-ref-name>
        <jndi-name>YachtSessionEJB</jndi-name>
      </ejb-reference-description>
    </reference-descriptor>
    <jndi-name>YachtEJB</jndi-name>
  </weblogic-enterprise-bean>
  <security-role-assignment>
    <role-name>ValidYachtClubUsers</role-name>
    <principal-name>system</principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>

# Third DD File – Persistent Field Mapping, Weblogic Specific
<!DOCTYPE weblogic-rdbms-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic
6.0.0 EJB RDBMS
Persistence//EN' 'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms20-
persistence-
600.dtd'>
<weblogic-rdbms-jar>
  <weblogic-rdbms-bean>
    <ejb-name>YachtEJB</ejb-name>
    <data-source-name>yachtClub-datasource</data-source-name>
    <table-name>yacht</table-name>
    <field-map>
      <cmp-field>yachtName</cmp-field>
      <dbms-column>yacht_name</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>builder</cmp-field>
      <dbms-column>builder</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>engineType</cmp-field>
      <dbms-column>engine_type</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>capacity</cmp-field>
      <dbms-column>capacity</dbms-column>
    </field-map>
    <field-map>
      <cmp-field>maxVelocity</cmp-field>
      <dbms-column>max_velocity</dbms-column>
    </field-map>
```

```
  </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

Recall that the deployment-descriptor files are supposedly read by the EJB container, not by people. I list them here just for the demonstration of some key concepts. Don't try to write or read these files using a text editor. Use the deployment tools instead.

As you see from the Listing 22-4, you specify the abstract schema name (`YachtBean`), the primary key field (`yachtName`), and all other persistent fields (`builder`, `engineType`, `capacity` and `maxVelocity`) in the deployment descriptor, as follows:

```
<persistence-type>Container</persistence-type>


      <abstract-schema-name>YachtBean</abstract-schema-name>
      <primkey-field>yachtName</primkey-field>
      <cmp-field><field-name>builder</field-name></cmp-field>
      <cmp-field><field-name>engineType</field-name></cmp-field>
      <cmp-field><field-name>capacity</field-name></cmp-field>
      <cmp-field><field-name>maxVelocity</field-name></cmp-field>
```

You further declare the persistent type as CMP, as shown here:

```
<persistence-type>Container</persistence-type>


<persistence-type>
  <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
  … …
</persistence-type>
```

You then define the mapping between `YachtEJB` and the underlying persistent store (`yachtClub-datasource`) by specifying the mapping between each persistent field and its corresponding table-column name (such as `yachtName` mapped to `yacht_name`, `maxVelocity` mapped to `max_velocity`, and so on), as follows:

```
<ejb-name>YachtEJB</ejb-name>
<data-source-name>yachtClub-datasource</data-source-name>
<table-name>yacht</table-name>
<field-map>
  <cmp-field>yachtName</cmp-field>
  <dbms-column>yacht_name</dbms-column>
</field-map>
<field-map>
  <cmp-field>builder</cmp-field>
  <dbms-column>builder</dbms-column>
</field-map>
 … …
```

Such information tells the EJB container to implement the access calls for these persistent fields. Based on the deployment information, the EJB container determines the approperate JDBC implementations (that is, the SQL calls) for the persistent fields and keeps a CMP bean's persistent field synchronized with the state of the database record it represents. After the concrete classes are generated during the

deployment phase, the life cycle of these CMP bean instances are same as that of BMP instances discussed in the previous Chapter 21.

For each `create` method defined in the home interface, you need to write a corresponding `ejbCreate` method. As a bean developer, your job is to assign the persistent fields with their initial values by calling the `setters`. You may find something weird by looking at the implementation shown in Listing 21-3. Although the return type is supposed to be the primary key, the `ejbCreate` method actually returns `null` at the end of the code. This is required by EJB specification. The rationale is that this method will only be called by the EJB container and that the container always knows exactly what the primary key is for each EJB.

Note

For a BMP bean, you must write code for the `ejbRemove` method. For a CMP bean, since the database-access logic is implemented by the EJB container, you typically do not need to write any code.

In the code shown in Listing 22-3, you do not see even the empty implementation of any `finder` method defined in the home interface. How does the EJB container generate the implementation for `finder` methods? For the method `findByPrimaryKey`, the container knows how to implement it by looking at the primary key class type in the deployment descriptor and the corresponding database column specified. For implanting all other methods, the EJB container follows your orders, given in the form of EJB QL in the deployment descriptor. The EJB QL is discussed in detail later.

You still need to implement all the business methods defined in the remote interface, except for the `getters` and `setters`. In the `YachtBean`, you only need to code the business method `createYachtSession`. You first look up the `YachtSessionEJB`'s home interface from JNDI, then create a remote interface handle. Since this will always be an EJB-to-EJB call, you may want to use `YachtSessionEJB`'s local interface for better performance.

To summarize this discussion, Table 22-2 lists major differences between coding a BMP bean implementation class and coding a CMP implementation class. All the database-access calls by bean developers for BMP beans are automatically generated by the EJB container. Since the bean-implementation classes you write contain no implementations, it is important to declare them as abstract. The corresponding concrete classes are automatically generated by EJB container at the deployment phase.

Table 22-2: Coding Differences between CMP and BMP

Item	CMP	BMP
Class Definition	Abstract	Not abctract
Database access calls	Generated by tools	Coded by developers
Persistent state	Represented by virtual persistent fields	Coded as instance variables
Accessor to persistent and relationship fields	Required	None
Customized `finder` methods	Handled by EJB container (but the developer must define the EJB QL queries)	Coded by developers
Select Methods (??)	Handled by EJB container	None
Return type of `ejbCreate` method	null	Primary key

With CM entity bean, no database code is needed. The database access functionality is specified by EJB developers or application assemblers in description descriptor in EJB Query language that is discussed next.

EJB Query Language

The EJB Query Language (EJB QL) is used to define query methods (for example, `finder` and `select` methods) for CMP entity beans. EJB QL, which is based on SQL-92, can be compiled automatically by the EJB container to a target language, such as SQL, of a database or other types of persistent stores. This makes CMP entity beans more portable and much easier to deploy.

An EJB QL query has these three clauses:
- `SELECT`
- `FROM`
- `WHERE`

The `SELECT` and `FROM` clauses are required, but the `WHERE` clause is optional. Here is the high-level BNF syntax of an EJB QL query:

```
EJB QL :: = select_clause from_clause [where_clause]
```

The `SELECT` clause defines the types of the objects or values that the query returns. A return type is a remote interface, a local interface, or a persistent field.

The `FROM` clause defines the scope of the query by declaring one or more identification variables, which may be referenced in the `SELECT` and `WHERE` clauses. An identification variable represents one of the following elements:
- The abstract schema name of an entity bean
- A member of a collection that is the multiple side of a one-to-many relationship

The `WHERE` clause is a conditional expression that restricts the objects or values retrieved by the query. Although this is optional, most queries have a `WHERE` clause.

You now may have found that the syntax of EJB QL is quite similar to the syntax of SQL. They do have a lot of similarities. However, EJB QL is not like SQL in the following aspects:
- SQL deals with tables and rows, but EJB QL deals with objects and instances.
- SQL has many built-in functions that EJB QL does not have.
- The result of an EJB QL is a remote interface or a collection of remote interfaces.

For each method (except the `findByPrimaryKey` method) in your CMP entity bean, there must be a `<query>` tag that describes this `finder` method. In the deployment descriptor, the EJB QL must be wrapped in an expression that looks like this:

```
<!CDATA[expression]]>
```

`expression` is a valid EJB QL statement. The CDATA statement is not necessary but is recommended because it escapes the reserved characters of XML. Since the object to be selected is obvious for these `finder` methods, you do not need to put the `SELECT` clause into the expression. In many cases, if the data is selected from a single entity bean object and there is no relationship that needs to be specified, the `FROM` clause can be omitted too.

Look at the deployment descriptor shown in Listing 22-4. The EJB QL for the `findAllYachts` method is as follows:

```
<ejb-ql><![CDATA[WHERE yachtName IS NOT NULL]]></ejb-ql>
```

This is translated to the following SQL statement by EJB container at the deployment phase if, for example, an Oracle database is used:

```
SELECT * FROM yacht
```

The EJB QL for the `findYachtsCapacityMoreThan` method is as follows:

```
<ejb-ql><![CDATA[FROM YachtBean cb WHERE cb.capacity > ?1]]></ejb-ql>
```

It may be translated into a SQL statement like this:

```
SELECT * FROM yacht WHERE yacht.capacity > parameter_1
```

The `parameter_1` is passed into the statement at runtime.

In addition to `finder` methods, you can use EJB QL to do any number of querying activities. EJB QL allows you to do simple queries; compound queries; queries that invoke the persistent fields of more than one EJB; queries that use `finder` methods on other EJBs; and queries that use persistent fields accessible through a relationship to other EJBs. In other words, EJB QL is a very powerful tool. However, it has also the following restrictions:

- Comments are not allowed.
- Date and time values are in milliseconds and use `Java long` data type. A date or time literal should be an integer literal. To generate a millisecond value, you may use the `java.util.Calendar` class.
- Currently, CMP does not support inheritance. For this reason, two entity beans of different types cannot be compared.

Note

This section covers only the simplified syntax of EJB QL. The full syntax is beyond the scope of this book. Interested readers can find a detailed description on EJB QL in many EJB books.

By now you have learnt how to develop and deploy CMP EJBs. Let us move on to run the example application.

Running the Example Application

Remember the yacht club application discussed first in Chapter 20? It is uses by a yacht club to manage its yacht-cruise operation. As a treat, the club offers its member free yacht cruises. The business process includes the following:

- Operating the yacht — such as starting, stoping, speeding up and slowing down
- Checking the status of the yacht — such as current velocity, maximum velocity, current passenger on board, and so on
- Picking up club members if there is enough room
- Dropping off passengers

Over the last three chapters, you have built these three EJBs:

- Stateful session bean `YachtSessionEJB`
- BMP entity bean `MemberEJB`
- CMP entity bean `YachtEJB`.

You can use them to build the simple yacht club application.

As an example, Listing 22-5 shows a JSP client that allows you to manage the yachts that the club owns.

Listing 22-5: YachtManager.jsp

```
<%@ page import="javax.naming.*, java.rmi.*, javax.ejb.*, YachtEBean.*,
common.*"
   session="true" %>
<%
{
   YachtHome home = (YachtHome)ctx.lookup("YachtEJB");

   if (request.getParameter("DestroyYacht") != null) {
      String pk = null;
```

```
      try {
         pk = request.getParameter("DestroyYacht");
         home.remove(pk);
         session.removeAttribute(pk);   // destroy the associated session
      } catch (NumberFormatException e) {
         log("Failed to destroy a Yacht.", out);
      }
   }
   else if (request.getParameter("CreateNewYacht") != null) {
      String yachtName = request.getParameter("YachtName");
      if(yachtName == null) {
         yachtName = "DefaulName";
      }
      String builder = request.getParameter("Builder");
      if (builder == null) {
         builder = "Unknown";
      }
      String engineType = request.getParameter("EngineType");
      if (engineType == null) {
         engineType = "Unknown";
      }
      int capacity = 0;
      int maxVelocity = 0;
      try {
         capacity = Integer.parseInt(request.getParameter("Capacity"));
      } catch (Exception e) {
         capacity = 10;
      }
      try {
         maxVelocity = Integer.parseInt(request.getParameter("MaxVelocity"));
      } catch (Exception e) {
         maxVelocity = 25;
      }
      // finally create the Yacht
      try {
         Yacht yacht = (Yacht) home.create(yachtName, builder, engineType,
capacity,
                                           maxVelocity);
      } catch (CreateException e) {
         log("CreateException caught while trying to create a new yacht." +
e, out);
      }
   }
```

```
    Collection coll = null;
    if (request.getParameter("MinCapacity") != null) {
       int minCapacity = 0;
       try {
           minCapacity = Integer.parseInt(request.getParameter("MinCapacity"));
       } catch (Exception e) {
          minCapacity = 10;
       }
       coll = home.findYachtsCapacityMoreThan(minCapacity);
    } else {
       coll = home.findAllYachts();
    }
 %>
    <html><head><title>Manage Yacht</title></head><body>
    <b>Yachts</b><br>
 <%
    Iterator iter = coll.iterator();
 %>
    <table width="400" border="thin" cellpadding="0" cellspacing="0">
 <%
    while (iter.hasNext()) {
       Yacht yacht = (Yacht)iter.next();
 %>
    <tr><td width="25%"><%= (String)yacht.getPrimaryKey() %></td>
        <td width="25%"><%= yacht.getCapacity() %></td>
        <td width="25%"><a href=YachtSessionManager.jsp?YachtPK=<%=
                        (String)yacht.getPrimaryKey() %>&Action=View>View
 Session</a></td>
        <td width="25%"><a href=YachtManager.jsp?DestroyYacht=<%=
                        (String)yacht.getPrimaryKey() %>>Destroy</a></td></tr>
 <%
    }
 %>
    </table>
 <%
 }
 %>

 <FORM action=YachtManager.jsp>
 <b>Create a New Yacht:</b><BR>
   <table>
     <tr><td>Yacht Name:</td>
         <td><INPUT TYPE=TEXT NAME=YachtName></td>
```

```
        <td>Builder:</td>
        <td><INPUT TYPE=TEXT NAME=Builder></td></tr>
    <tr><td>Engine Type:</td>
        <td><INPUT TYPE=TEXT NAME=EngineType></td>
        <td>Capacity:</td>
        <td><INPUT TYPE=TEXT NAME=Capacity></td></tr>
    <tr><td>Maximum Velocity:</td>
        <td><INPUT TYPE=TEXT NAME=MaxVelocity></td>
        <td></td>
        <td><><INPUT TYPE=SUBMIT NAME="CreateNewYacht"
VALUE=Create></td></tr>
  </table>
<b>Search Yachts Big Enough:</b><br>
  <table>
    <tr><td>Find yachts with capacity more than: </td>
        <td><INPUT TYPE=TEXT NAME=MinCapacity></td>
        <td><INPUT TYPE=SUBMIT Value="Find"></td></tr>
  </table>
</FORM>

<%!
    private static Context ctx;
    static {
        try {
            ctx = new InitialContext();
        } catch (Exception e) {
            System.out.println("Error trying to do one time
initialization.\n" + e);
        }
    }

    public void log(String logMsg, JspWriter out) throws Exception {
        out.print(logMsg + "<BR>");
    }
%>
</body>
</html>
```

This JSP client allow you to add a new yacht to the yacht club's possession, to remove a yacht, and to search for yachts that have a capacity over a give number. The name and capacity of the yacht that meet the searching criteria are listed at the top of the browser screen. By clicking the hyperlink marked "Destroy," the corresponding yacht is removed from the database. When adding a new yacht, you need to provide the five persistent attributes of the yacht, namely, yacht name, its builder, the engine type, capacity, and maximum velocity. An output screen for running this JSP client is illustrated in Figure 22-1.

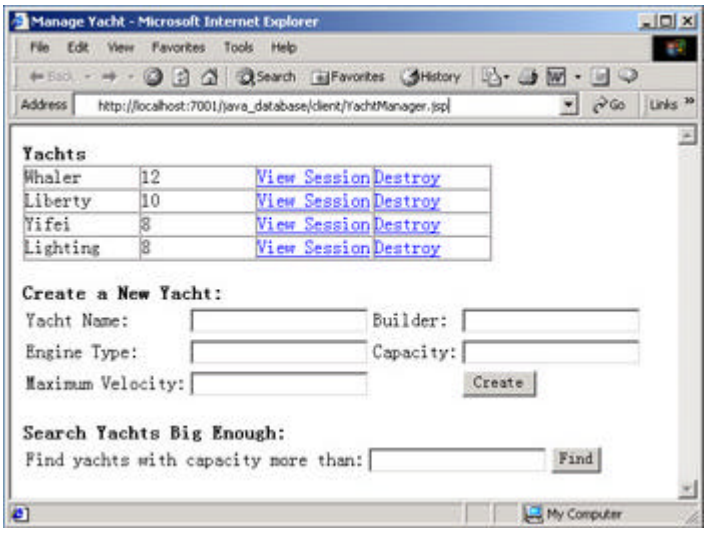


Figure 22-1: Output of ManageYacht client

By clicking the hyperlink marked "View Session," you should be able to view the session associated with this yacht (that is, whether it is in an active cruise operation, the current status such as velocity and passenger list, and so on). From the code in Listing 22-5, you can see that clicking the View Session hyperlink sends a request to another JSP client: `YachtSessionjManager. jsp`, as shown here:

```
<a href=YachtSessionManager.jsp?YachtPK=<%= (String)yacht.getPrimaryKey() %>
        &Action=View>View Session</a>
```

The code for `YachtSessionjManager.jsp` is not provided here, and I do it on purpose. By now you should be able to write your own client to use these EJBs to meet your own needs. You have learned all the skills you need to access these EJBs from your own client. Do this as an exercise!

Your yacht-session-management client should allow a user to check whether a cruise session is active. If no active session is associated with the yacht the user has selected, the user should be prompt to create a session. Once a cruise session is created (or retrieved), the user should be able to operate the cruising yacht. That means the user is able to start, stop, accelerate and decelerate the yacht, check the yacht status, drop off passengers, pick up members, and so on.

Because only members can come on board, you do need the help of `MemberEJB` to implement the preceding functionality. You can write a JSP client, a Swing client, or a stand-along client. If you decide to use the JSP client, you may be able to take advantage of the functionality provided by the `HttpSession` interface. For example, once a `YachtSessionEJB` instance session is created, you can save it to `HttpSession` as follows:

```
session.setAttribute(yachtPrimaryKey, theYachtSession);
```

When a user wants to view the `YachtSessionEJB` instance for a specific yacht, you just need to retrieve it as follows:

```
myYachtSession = (YachtSession)session.getAttribute(yachtPrimaryKey);
```

When an active session is destroyed (that is, the cruise is completed), you remove the `YachtSessionEJB` instance as follows:

```
session.removeAttribute(yachtPrimaryKey);
```

The `session.removeAttribute(yachtPrimaryKey)` method is called in the `YachtManager.jsp` shown in Listing 22-5. When a yacht is removed, its associated `YachtSession` is also removed.

To give you more hints, you may write a JSP client that provides a user interface similar to what is shown in Figure 22-2. The yacht name and current status are shown on the top of the screen. A user can start and stop the yacht. He or she can also speed up or slow down the yacht by a certain velocity.

Remember that business logic is built into the `YachtSessionEJB` that you cannot accelerate a stopped yacht. You must start the yacht and then speed it up. You cannot stop a yacht that is running too fast. You must slow it down to certain speed before stopping it. If you want to add business logic, you can revisit the `YachtSessionEJB` code listed in Chapter 20 and make any modification you need.

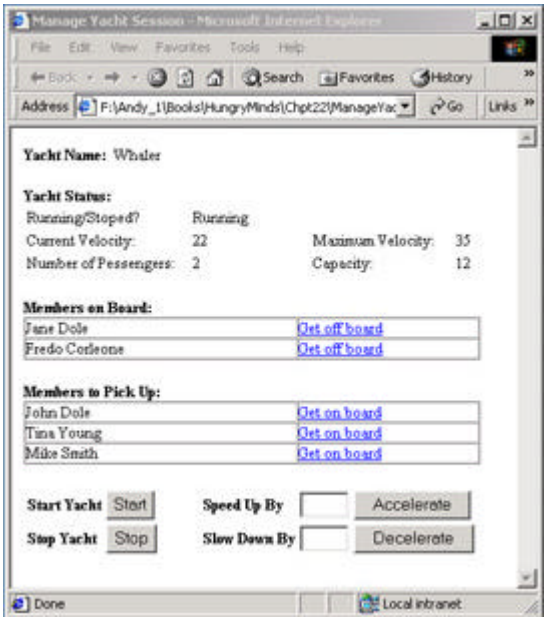


Figure 22-2: Possible output screen of your yacht-session client

This example application is simplified to demonstrate the fundamentals of CMP EJB. However, an important feature of CMP entity bean brought by EJB 2.0 is missing: the container managed relationship. You will learn it in the next section.

Container-Managed Relationship

The entity beans you have seen so far in this book are detached objects that do not relate with each other. This type of entity bean has only limited use because in real life objects are often linked and depend upon each other. This kind of behavior has always existed in databases through primary keys and foreign keys. With EJB 1.1, CMP entity beans had no easy way of representing the natural interaction between entity objects. This has partially contributed to the slow adoption of CMP entity beans in the early stage. To address this problem, the EJB 2.0 specification introduces a way to support simple and complex relationships by introducing the container-managed relationship through the *relationship fields*.

Relationship Field

A relationship field is like a foreign key in a database table — it identifies a related bean. Like a persistent field, a relationship field is virtual and is defined in the enterprise-bean class with access methods. But unlike a persistent field, a relationship field does not represent the bean's state. For example, each yacht has an engine. Assume an `EngineEJB` is developed; it has a one-to-one relationship with the YachtEJB you have written. To model `YachtEJB`'s relationship to `EngineEJB`, it has a relationship field: `engine`. In the deployment descriptor, you specify this relationship as follows:

```
  <ejb-relation>
    <ejb-relation-name>Yacht-Engine</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Yacht-Has-Engine</ejb-relationship-role-
name>
        <multiplicity>one</multiplicity>
```

```
        <role-source>
          <ejb-name>YachtEJB</ejb-name>
        </role-source>
        <cmr-field>
          <cmr-field-name>engine</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>Engine-In-Yacht</ejb-relationship-role-
name>
        <multiplicity>one</multiplicity>
        <role-source>
          <ejb-name>EngineEJB</ejb-name>
        </role-source>
      </ejb-relationship-role>
  </ejb-relation>
```

This part of the deployment descriptor tells the EJB container that each `YachtEJB` has one engine and each `EngineEJB` may belong to a yacht. Thus, this is a one-to-one relationship. Since you can only find the `EngineEJB` instance through a `YachtEJB` instance, not vice verse, this relationship is unidirectional. Although similar to the primary-key and foreign-key relationships in a database, the EJB relationships do not work the same way as relationships in database. An EJB relationship binds two EJBs together through object graphs to have in-memory object graphs mapped to an underlying database schema. For example, the `YachtEJB` owns an `EngineEJB;` thus, when a `YachtEJB` instance is instantiated, an associated `EngineEJB` instance must also be instantiated and placed in the EJB container's memory for a client to access. In other words, the EJB relationship is enforced and adhered to by the EJBs and the EJB container.

Cardinality and Direction of Relationship

The XML elements used in the deployment descriptor to describe the container managed can become very complex, as they must deal with both the *cardinality* and direction (unidirectional vs. bidirectional) of the relationships.

Cardinality indicates the number of EJBs. The four types of multiplicities are as follows:

- *One-to-one*: Each entity-bean instance is related to a single instance of another entity bean. For example, if each yacht has only one engine, `YachtEJB` and `EngineEJB` will have a one-to-one relationship.
- *One-to-many*: An entity-bean instance may be related to multiple instances of the other entity bean. In real life, yachts have twin engines, and some have even more engines. To reflect this fact, the `YachtEJB` has a one-to-many relationship with `EngineEJB`.
- *Many-to-one*: Multiple instances of an entity bean may be related to a single instance of the other entity bean. This multiplicity is the opposite of a one-to-many relationship. In the example mentioned in the previous item, from the perspective of `EngineEJB` the relationship to `YachtEJB` is many-to-one.
- *Many-to-many*: The entity-bean instances may be related to multiple instances of each other. For example, in college, each course has many students, and every student may take several courses. Therefore, in an enrollment application, `CourseEJB` and `StudentEJB` have a many-to-many relationship.

The direction of a relationship may be either bidirectional or unidirectional. In a *bidirectional* relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its related object. If an entity bean has a relative field, we often say that it

"knows" about its related object. For example, if `CourseEJB` knows which `StudentEJB` instances it has and, at the same time, `StudentEJB` knows which `CourseEJB` it is associated with, they have a bidirectional relationship.

In a *unidirectional* relationship, only one entity bean has a relationship field that refers to the other. Look at the snipet of the deployment descriptor given on the previous page; `YachtEJB` has a relationship field that identifies `EngineEJB`, but `EngineEJB` does not have a relationship field for `YachtEJB`. In other words, `YachtEJB` knows about `EngineEJB`, but `EngineEJB` doesn't know which `YachtEJB` instances refer to it.

EJB QL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another. For example, a query can navigate from `YachtEJB` to `EngineEJB` but cannot navigate in the opposite direction. For `CourseEJB` and `StudentEJB`, a query can navigate in both directions, since these two beans have a bidirectional relationship.

Access to Relationship Field

During development, you implement the relationship fields in a similar way to persistent fields. They are defined in the deployment descriptor, and they have their `getters` and `setters` defined in the bean-implementation class. They can even be exposed in the remote interface. By following a strict syntax for authoring relationship fields in the bean-implementation class and in the deployment descriptor, the EJB container is able to implement the relationship automatically behind the scene.

The rules for writing relationship-field `accessor` methods in a bean-implementation class are listed here:

- Both `getters` and `setters` for every relationship field must exist in the implementation class.
- These `getters` and `setters` must be declared as abstract and must contain no implementation code.
- These `accessor` methods must begin with `get` or `set`; and the text following `get/set` must match the name of the relationship field as it is declared in the deployment descriptor.
- These `getters` and `setters` that do not access `Collections` may be optionally placed in the remote interface.

If you want the clients to use the relationship-field `accessor` method, put the `getters` or `setters` in the remote interface. But the last rule says that you may only do this if the method does not access a `Collection` of objects. Only the entity bean's other business methods can use its own `Collection` relationship.

Why does such a restriction exist? It is imposed for better performance. For a one-to-many relationship, a `getter` may return a `Collection` of the related EJB objects. For example, the `OrderEJB` and `LineItemEJB` are linked by `lineItem` field of the `OrderEJB`. The `getLineItems()` method may return tens or hundreds of `LineItemEJB` instances, but you may want to work only on one of these `LineItemEJB` instances. Imagine the network traffic it produces! To avoid the potential performance nightmare, the last rule given in the preceding list is imposed. If you really need to get the whole list of the elements to the client, you must define your own (nonabstract) utility `accessor` method like this:

```
Public ArrayList getAlLineItems() {
   ArrayList list = new ArrayList();
   // call the abstract relationship field accessor and walk through the
Collection
   Iterator iter = getLineItems().iterator();
   While (iter.hasNext()) {
      List.add(iter.next())
   }
}
```

Inside your own utility `accessor` method, you can call the bean's abstract `getter` that returns a `Collection`. This tells the EJB container that you really need to get the whole list and that it is not the container's responsibility to ensure good performance.

The `ejbPostCreate` method in Listing 22-3 is empty. However, if there are any relationship fields, you must put these fields' initialization code in this method. Although all the persistent fields must be set in the `ejbCreate` method, it is important to not set any relationship fields in the `ejbCreate` methods. When `ejbCreate` is called, the bean has not yet been inserted into the underlying database. When calling a `setter` method, the other EJB in the relationship also tries to update its references in the related fields. This is not possible, since the EJB that is having `ejbCreate` method invoked has not yet been created. You should initialize the relationship fields in the `ejbPostCreate` method.

Thus, if the `YachtEJB` is related to `EngineEJB`, the `ejbPostCreate` method may look like this:

```
   public void ejbPostCreate(String yachtName, String builder, String
engineType,
                             int capacity, int maxVelocity, Engine engine) {
      // initialize relationship field
      setEngine(null);
   }
```

In summary, implement relationships differently for BMP entity beans and CMP entity beans. With BMP, the code you write implements the relationships. But with CMP, the EJB container takes care of the relationships for you. Most information of the relationships is given in the deployment descriptor. A bean developer needs to write very little code for the simple abstract `getters` and `setters` and some initialization in the `ejbPostCreate` method. All these features make the CMP entity bean more appealing because they are easier to develop and more flexible.

Summary

In this chapter, you learn how CMP entity beans handle the data persistence and object relationship. Specifically, you learned:

- The differences between CMP and BMP
- How to achieve persistence through persistent fields
- How to handle entity relationship through relationship fields
- How to specify database access in EJB query language

This chapter concludes the discussion on EJBs. Over the past three chapters, three EJB have been developed. You are encouraged to enhance their functionality and write your own client programs to use these EJBs. In next chapter, you will learn another mechanism for data persistence: the Java data object.

Chapter 23: Java Data Objects and Transparent Persistence

In This Chapter

The focus of this chapter is on the transparent persistence and the standard way to achieve it: the Java data object. After reading this chapter, you should have one more tool in you toolkit to design and develop enterprise applications.

JDO for Transparent Persistence

So far, you have learned many ways to persist your application data such as Java serialization, JDBC, entity EJBs, and so on. All these persistence mechanisms require that application programmers know the details of the underlying database structure; most of them even require programmers to be responsible for handling the details of persistence. To relieve application programmers from having to know the details of the database structure, the recently released Java data object (JDO) specification provides a high level of abstraction: *transparent persistence*.

Transparent persistence means that the persistence of data objects is automatic and that all logic for processing persistent objects is expressed in pure Java language. The application programmers do not need to know any database query languages such as SQL. The mapping of Java objects and the persisted state of objects stored in the database is achieved behind the scene by the JDO provider implementation and is totally transparent to application developers. From the application developer's point of view, persistent objects are treated exactly the same as transient objects — instances that only reside in JVM memory and do not persist outside of an application.

The two major goals of the JDO specification are:

- Providing a standard interface between application objects and data stores (for instance, relational databases, file systems, and so on)
- Simplifying secure and scalable applications by providing developers with a Java-centric mechanism for working with persistent data

Although lower-level abstractions for interacting with databases are still useful, the goal of JDO is to reduce the need for explicit code for SQL and transaction handling in common business applications.

In addition to shielding the Java developers from the details of the underlying methods for providing persistence, JDO acts as a standard layer between the application program and any back-end data stores, whether it be a relational database, an XML database, a legacy application, a file system, or flash RAM. Applications using the JDO interface can automatically plug in any data store that provides a JDO implementation. This generally provides portability and increases the longevity of code.

JDO has come a long way to get here. It originated from Java Specification Request (JSR-012), proposed in 1999. After three years of lengthy Java community process, it was finally approved as an official specification in March 2002. In the meantime, many other requested specifications have become standards, and the JDO work force has been dealing with the fact that JDO is able to be integrated into the frameworks provided by these related specifications (mostly notably J2EE). Indeed, servlets and session EJBs can directly manipulate JDO persistent instances instead of dealing with the underlying data stores. Entity EJBs with bean-managed persistence can delegate business logic and persistence management to JDO classes instead of forcing the developers writing all SQL commands in the implementation classes. Integration of JDO with J2EE is discussed later in this chapter. First let us see what makes JDO different from other data persistence mechanisms.

What Makes JDO an Unique Persistence Mechanism

In most cases, instances of Java classes reside in the memory of the running application. They are destroyed when the program terminates. However, it is often desirable for the objects to persist even

after applications terminate or sessions end so that their state may be saved for the next execution or so that they may be shared between different applications.

You know several mechanisms to serve this purpose. The simplest way is through Java serialization. The `java.io.Serializable` interface gives the programmer a way to explicitly persist objects to an output stream and later retrieve them by calling, for example, `writeObject(ObjectOutputStream out)` or `readObject(ObjectInputStream in)`. As a developer, you only need to declare that the class you are writing implements the `Serializable` interface; the JVM handles the lower-level details for you transparently. Since the persisted data is coded as Java classes, serialization persistence supports the object-oriented design and programming paradigm.

Although Java serialization provides a simple and transparent mechanism for persisting objects to an output stream (mostly to a file system or local disk), it suffers from many limitations. It does not provide query capability and cannot handle transaction. It does not support partial read and update. The whole object is read or written in a single operation. Because of these limitations, it is usually not used to persist business objects in enterprise applications.

JDBC provides a mechanism to store and retrieve data objects to and from a database. It allows an application access to many types of relational databases through a standard API. The transaction API ensures concurrency control and therefore allows multiple applications to share persisted data. In the previous chapters of this book, you have learned how to use JDBC APIs and have seen what great tools they are. The downside is that, as a Java programmer, you must know the database structure and manually map your class attributes to database fields and write all the SQL commands in your Java code. In other words, persistence is not transparent. In addition, because of the SQL variants among different types of databases (for example, Oracle and Sybase), your code is not 100-percent portable.

Although Java is a highly object-oriented language, the JDBC uses the relational data model of SQL. It is based on tables, rows, and columns. The relationships are specified as primary and foreign keys. As a consequence, a developer has to struggle between the OO object model and the relational data model. Although you may get used to it after while, the use of different models in the same application is generally not considered the best approach, and a better approach using a unified model (most favorably an object-oriented model) is always preferred.

In Chapters 20- 22, you learn that entity EJBs provide another mechanism for data persistence. If CMP beans are used, you enjoy guaranteed portability because all the database-access calls are declared in the deployment descriptor. With the so-called "write once, deploy everywhere" approach, you only need to modify the deployment descriptor when the EJBs are deployed to a different database type or to a different database schema. The Java code does not need to be modified or recompiled. The EJB container provides many system-level services such as transaction, security, transparent remote invocation, and so on. The synchronization between the instance variables and the persisted object state is also handled by the EJB container in an automatic and optimized manner.

As Java classes (and interfaces), EJBs also support the object-oriented paradigm. However, the current EJB specification does not support inheritance, and you cannot have a complex object model. Besides, if BMP entity beans are used, you will have to write all SQL commands in your implementation class.

The JDO specification provides a new persistent mechanism. It has a set of very simple APIs to support transparent persistence. The Java code is totally decoupled from the underlying data store, which leads to the "write once, persist everywhere" approach. JDO is fully object oriented and hence is able to support complex domain object models. The optimization of database read and update is performed at the JDO implementation layer and is transparent to application developers. Unlike EJBs, it can be used outside a container and used for batch processes. If combined with J2EE components (for instance, servlets, JSPs, and EJBs), it enjoys the system-level services that containers provide.

As an example, the `Yacht` class listed in Listing 23-1 is all you need to code to persist `Yacht` objects. It is basically a JavaBean class with some persistable attributes and access methods to these attributes. Compared with the `YachtEJB` you see in Chapter 22, the code is much simpler, and there is not even a slight hint of an underlying database as the persistent store. Similar to EJB's deployment descriptor, you declare that the class `Yacht` is persistence-capable in an XML MetaData file. But you see later that an XML MetaData file is normally much simpler and shorter than an EJB deployment descriptor.

Listing 23-1: A persistent class — Yacht

```
package java_database.jdo;

public class Yacht {
   private String yachtName;
   private String builder;
   private String engineType;
   private int    capacity;
   private int    maxVelocity;

   /** constructor */
   public Yacht(String yachtName, String builder, String engineType,
                int capacity, int maxVelocity) {
      this.yachtName = yachtName;
      this.builder = builder;
      this.engineType = engineType;
      this.capacity = capacity;
      this.maxVelocity = maxVelocity;
   }

   /** default constructor */
   public Yacht() { }

   // getters and setters
   public String getYachtName()   { return yachtName;   }
   public String getBuilder()     { return builder;     }
   public String getEngineType()  { return engineType;  }
   public int    getCapacity()    { return capacity;    }
   public int    getMaxVelocity() { return maxVelocity; }

   public void setYachtName(String v)  { yachtName = v;   }
   public void setBuilder(String v)    {  builder = v;    }
   public void setEngineType(String v) { engineType = v;  }
   public void setCapacity(int v)      { capacity = v;    }
   public void setmaxVelocity(int v)   { maxVelocity = v; }
}
```

All these features — transparent persistence; extended query capability; adherence to the object-oriented paradigm and support for the complex data model; simplicity; and so on — make the JDO a unique persistent mechanism and should be in every enterprise application developer's toolkit. Table 23-1 summarizes the features of the persistence mechanisms discussed in this section. You learn the details of JDO in the next section.

Table 23-1: Comparison of Major Persistence Mechanisms

	Serialization	JDBC	EJB	JDO
Transparent Persistence	Transparent	Not transparent	Session and BMP entity bean is not transparent. CMP EJB is partially transparent.	Transparent
Domain Object Model	Fully object oriented.	Inherently not object oriented.	Simple domain object model. No inheritance.	Fully object oriented. Support complex domain object model.
Query	Not supported	Supported by writing SQL code.	Supported by declarative query by CMP entity beans, and SQL code by session and BMP entity beans.	Extended support via JDO QL
Transaction	Not surported	Supported	Supported	Supported
Database Portability	N/A	Weak support for relational DBs. May have to recode due to SQL variant. Do not suport OO and XML DBs.	Session and BMP entity bean has same level of support as JDBC. CMP entity bean has better support.	Good support – "write once, persist everywhere"

Major JDO APIs

Compared with other Java technologies, JDO has a set of very simple APIs. The package `javax.jdo` contains 12 interfaces, five classes and nine exceptions. These interfaces and classes are all you need when you develop Java classes whose instances are to be stored in persistence stores. These APIs specify the contracts between your persistent-capable classes and the runtime environment that is part of the JDO implementation. The JDO implementation is provided by JDO vendors, and there are currently quite a few implementations available on the market. A set of contracts between application developers and JDO vendors is defined in the JDO architecture and specified through these APIs.

The following sections discuss these interfaces:

- `PersistenceCapable`
- `PersistenceManagerFactory`
- `PersistenceManager`
- `Query`
- `Transaction`

PersistenceCapable Interface

The `javax.jdo.PersistenceCapable` interface makes a Java class capable of being persisted by a persistence manager through a JDO implementation. Every class whose instances can be managed by a JDO `PersistenceManager` must implement the `PersistenceCapable` interface.

This interface defines methods that allow the implementation to manage the instances. It also defines methods that allow a JDO-aware application to examine the runtime state of instances. For example, an application can discover whether the instance is persistent, transactional, dirty, new, or deleted and can get its associated `PersistenceManager` if it has one.

Unlike with the `java.io.Serializable` that makes a class serializable, you do not explicitly declare your class as "`implements PersistenceCapable`". Look at the persistent class (`Yacht`) shown in Listing 23-1; simply declare the class as follows, without mentioning the `PersistenceCapable` interface:

```
public class Yacht {
    … …
}
```

This is the beauty of transparent persistence. In most JDO implementations, you specify that the class meant to be persistent in an XML MetaData file read by the JDO enhancer. The JDO enhancer modifies the class's bytecode to ensure that it implements `PersistenceCapable` prior to loading the class into the runtime environment. The JDO enhancer also adds code to implement the methods defined by `PersistenceCapable`.

As an example, the XML MetaData file for the persistent class `Yacht` is shown in Listing 23-2. The document-type definition file, `jdo.dtd`, is provided by the vendor of the JDO implementation that you use.

Listing 23-2: XML MetaData file for the persistent class Yacht

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
   <package name="java_database.jdo">
      <class name="Yacht" identity-type="datastore">
      </class>
   </package>
</jdo>
```

You tell the JDO enhancer that the class `java_database.jdo.Yacht` is to be persisted in a data store. If you have a complex persistable object model with inheritance and other types of object relationships, the XML MetaData file is, of course, more complex than what you see in Listing 23-2. But it is almost never so lengthy and so complicated as a CMP entity bean's deployment descriptor.

Since the JDO enhancer does all the work behind the scenes, mapping your persistent classes with the persistent store, you do not have to know the details of the `PersistenceCapable` interface.

PersistenceManagerFactory Interface

The `javax.jdo.PersistenceManagerFactory` interface obtains `PersistenceManager` instances. All `PersistenceManager` instances obtained from the same `PersistenceManagerFactory` have the same default properties.

`PersistenceManagerFactory` instances may be configured and serialized for later use. They may be stored via the Java Naming and Directory interface (JNDI) and looked up and used later. Any properties configured are saved and restored. Once the first `PersistenceManager` is obtained from the `PersistenceManagerFactory`, the factory can no longer be configured.

The application acquires an instance of JDO `PersistentManager` by calling the `getPersistentManager` method of an instance of JDO `PersistenceManagerFactory`. The code may looks like this:

…

```
InitialContext ctx = new InitialContext();
```

```
PersistentenceManagerFactory pmf = (PersistenceManagerFactory) ctx.lookup(

"java:comp/env/jdo/JNDI_NAME_FOR_YOUR_PMF");
PersistenceManager pm = pmf.getPersistenceManager();
…
```

Most JDO-implementation vendors provide some proprietary APIs to instantiate instances of `PersistenceManagerFactory` by other means such as properties.

PersistenceManager Interface

The `javax.jdo.PersistenceManager` interface is the primary one for JDO-aware application components. It is the factory for `Query` and `Transaction` instances and contains methods to manage the life cycle of `PersistenceCapable` instances. The most commonly used methods include the following:

- Make instances persistent
  - `void makePersistent(Object pc)`
  - `void makePersistentAll(Object[] pcs)`
  - `void makePersistentAll(Collection pcs)`
- Delete persistent instances
  - `void deletePersistent(Object pc)`
  - `void deletePersistentAll(Object[] pcs)`
  - `void deletePersistentAll(Collection pcs)`
- Make instance transient — disassociate the instance from the persistence manager. The data stored in the data store is not deleted.
  - `void makeTransient(Object pc)`
  - `void makeTransientAll(Object[] pcs)`
  - `void makeTransientAll(Collection pcs)`
- Handle persisted object IDs
  - `Object GetObjectId(Object pc)`
  - `Object getObjectById(Object oid, boolean validate)`
- Give access to current transaction interface
  - `Transaction currentTransaction()`
- Serve as the factory for the Query Interface
  - `Query newQuery()`
  - `Query newQuery(java.lang.Class cls)`
  - `Query newQuery(A variety of parameters can be passed in)`

The usage of some of these APIs is seen from the example test client of the `Yacht` class listed later in this chapter under "A Test Client Example."

A JDO `PersistenceManager` instance supports one transaction at a time and uses one connection to the underlying data source at a time. The JDO `PersistenceManager` instance might use multiple transactions serially and might use multiple connections serially.

Normally, cache management is automatic and transparent. When instances are queried, navigated to, or modified, instantiation of instances and their fields and garbage collection of unreferenced instances occur without any explicit control. When the transaction commits in which persistent instances are created, deleted, or modified, eviction is automatically handled by the transaction-completion mechanisms.

Query Interface

The `javax.jdo.Query` interface allows applications to obtain persistent instances from the data store. The `PersistenceManager` is the factory for `Query` instances. There may be many `Query` instances associated with a `PersistenceManager`. Multiple queries might be executed simultaneously by

different threads, but the implementation might choose to execute them serially. In either case, the implementation must be thread-safe.

There are three required elements in a query: the class of the results, the candidate collection of instances, and the filter. There are optional elements: parameter declarations, variable declarations, import statements, and an ordering specification.

The query namespace is modeled after these methods in Java:

- `setClass` corresponds to the class definition.
- `declareParameters` corresponds to formal parameters of a method.
- `declareVariables` corresponds to local variables of a method.
- `setFilter` and `setOrdering` correspond to the method body.

Note

You can find more details of these methods in the JDO document (visit http://access1.sun.com/jdo).

The `Query` interface provides the following methods that execute the query based on the parameters given:

- `Object execute()`
- `Object execute(Object param)`
- `Object execute(Object[] params)`

They return a `Collection` that the user can iterate to get results. For future extension, the signature of the `execute` methods specifies that they return an `Object` that must be cast to `Collection` by the user. Any parameters passed to the `execute` methods are used only for this execution and are not remembered for future execution.

All queries must conform to the object query language (OQL) grammar. Unlike SQL, the JDO OQL operates on Java classes and objects and has a strong object-oriented flavor. A JDO OQL has at least three elements: the class of results, the JDO instances' candidate collection (usually `extent`), and the query filter. The query filter is where you specify the query criteria. Query filters use syntax very similar to Java syntax such as: `"name.startWith("Liberty")"` or `"getCapacity() > 12"`. The following code snippet illustrates the use of some `Query` APIs.

…

```
Class target = Yacht.class;
Extent extent = pm.getExtent(target, false);
String filter = "getCapacity() > 12";
Query query = pm.newQuery(excent, filter);
Query.setClass(target);
Query.compile();
Collection result = (Collection) query.execute();
```

…

This piece of code searches the persistent store and returns a collection of `Yacht` objects that has a capacity of more than 12 passengers.

Transaction Interface

Operations on persistent JDO instances at the user's choice might be performed in the context of a transaction. That is, the view of data in the data store is transactionally consistent, according to the standard definition of Atomicity, Consistency, Isolation, and Durability (ACID) transactions.

The `javax.jdo.Transaction` interface is used to mark the beginning and end of a application-defined unit of work. The `PersistenceManager` allows the application to get the instance that manages these transactional boundaries via the `currentTransaction` method.

Transaction-completion methods have the same semantics as `javax.transaction.UserTransaction` and are valid only in the nonmanaged, nondistributed transaction environment. Do not be surprised if you are told that the most useful methods in the `javax.jdo.Transaction` interface are as follows:

- `void begin()`
- `void commit()`
- `void rollback()`

You may use these APIs as follows:

```
… …
PersistentManager pm = pmf.getPersistentManager();  // get a PM instance
Transaction txn = pm.currentTransaction();          // get current
transaction context
pm.deletePersistent(pm.getObjectById(oid), false);  // delete a persisted
object
txn.commit();                                       // commit the
transaction
pm.close();
… …
```

For operation in the distributed environment, `Transaction` is declared to implement `javax.transaction.Synchronization`. This allows for flushing the cache to the data store during externally managed transaction completion.

A Test Client Example

The most commonly used APIs are discussed in the preceding sections. To illustrate the use of these APIs, a test client for the persistence-capable class `Yacht` is developed as shown in Listing 23-2. You can see from this example that the coding for JDO applications is very simple and straightforward.

Listing 23-2: A test client for the persistent class Yacht

```
package java_database.jdo;

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

import com.prismt.j2ee.connector.jdbc.ManagedConnectionFactoryImpl;

public class TestClient {
   private final static int SIZE         = 4;
   private PersistenceManagerFactory pmf = null;
   private PersistenceManager pm         = null;
   private Transaction transaction       = null;

   private Yacht[] yachts;               // Array of yachts for persistence
test
   private Vector id = new Vector(SIZE); // Vector of object identifiers
```

```
    /** constructor */
    public TestClient() {
        System.out.println("First initializing JDO PersistenceManagerFactory");
       try {
          Properties props = new Properties();
          props.setProperty("javax.jdo.PersistenceManagerFactoryClass",

 "com.prismt.j2ee.jdo.PersistenceManagerFactoryImpl");
          // Following part uses vendor specif APIs. Modify it as needed!
          pmf = JDOHelper.getPersistenceManagerFactory(props);
          pmf.setConnectionFactory( createConnectionFactory() );
       } catch(Exception ex) {
          ex.printStackTrace();
          System.exit(1);
       }
    }

    public static Object createConnectionFactory() {
        ManagedConnectionFactoryImpl mcfi = new ManagedConnectionFactoryImpl();
       Object connectionFactory = null;
       try {
          mcfi.setUserName("system");
          mcfi.setPassword("manager");
          mcfi.setConnectionURL("jdbc:oracle:thin:@localhost:1521:thedb");
          mcfi.setDBDriver("oracle.jdbc.driver.OracleDriver");
          connectionFactory = mcfi.createConnectionFactory();
       } catch(Exception e) {
          e.printStackTrace();
          System.exit(1);
       }
       return connectionFactory;
    }

    /**  Create a group of Yacht objects for persistence test */
    public void yachtPersistor() {
       // Create an array of three yachts to persist
       yachts = new Yacht[SIZE];
       yachts[0] = new Yacht("Whaler", "Grand Banks", "Twin Diesel", 12, 35);
       yachts[1] = new Yacht("Liberty", "Bristol", "Single Diesel", 10, 25);
       yachts[2] = new Yacht("Yifei", "Eastbay", "Twin Diesel", 8, 27 );
       yachts[3] = new Yacht("Lightning", "Eastbay", "Twin Diesel", 8, 30 );
       // get a PM and set transaction
       pm = pmf.getPersistenceManager();
```

```
      transaction = pm.currentTransaction();
      // make all of the objects in the graph persistent
      pm.makePersistentAll(yachts);
      transaction.commit();
      // retrieve object ids for the persisted objects
      for(int i = 0; i < yachts.length; i++) {
         id.add(pm.getObjectId(yachts[i]));
         System.out.println("Object id is: " + id.elementAt(i));
      }
      // close current PM to ensure that objects are read from the datastore
      // rather than the PM's memory cache.
      pm.close();
   }

   /** Display the persisted objects' state on standard output */
   public void display(int endIndex) {
      Yacht aYacht;
      int max = endIndex <= SIZE ? endIndex : SIZE;
      System.out.println("\n-----------------------------------------------
---------
");
      System.out.println(" Display: Persisted Yachts");
      System.out.println("-------------------------------------------------
-------");
      // get a new PM
      pm = pmf.getPersistenceManager();
      // retrieve objects from datastore and display their state
      for(int i = 0; i < max; i++) {
         aYacht = (Yacht)pm.getObjectById(id.elementAt(i), false);
         System.out.println("------------ " + i + " ------------");
         System.out.println("YachtName   : " + aYacht.getYachtName());
         System.out.println("Builder     : " + aYacht.getBuilder());
         System.out.println("Engine Type : " + aYacht.getEngineType());
         System.out.println("Capacity    : " + aYacht.getCapacity());
         System.out.println("Max Velocity: " + aYacht.getMaxVelocity());
         System.out.println("-------------------------------");
      }
      pm.close();
  }

   /** Change a Yacht's name and make the change persistent */
   public void change() {
      Yacht aYacht;
      // get a PM and set transaction
```

```
      pm = pmf.getPersistenceManager();
      transaction = pm.currentTransaction();
      // change DataString field of the second persisted object
      aYacht = (Yacht)pm.getObjectById(id.elementAt(1), false);
      aYacht.setYachtName("New_Name");
      // commit the change and close the PM
      transaction.commit();
      pm.close();
   }

   /** Delete a Yacht object from persistent datastore */
   public void delete() {
      // get a PM and set transaction
      pm = pmf.getPersistenceManager();
      transaction = pm.currentTransaction();
      // delete the 2nd persisted object from datastore and its ID from
Vector id.
      pm.deletePersistent(pm.getObjectById(id.remove(1), false));
      // commit the change and close the persistence manager
      transaction.commit();
      pm.close();
   }



   /**
    * The main method of the Test Client program.
    */
   public static void main(String[] args) {
      System.out.println("Start Test");
      // Instantiate a TestClient
      TestClient aTestClient = new TestClient();

      // Setup and persist a group of yachts, then display tieir persised
state.
      System.out.println("\nThree yachts are persisted");
      aTestClient.yachtPersistor();
      aTestClient.display(SIZE);

      // Change a yacht'sname, and then display the yachts' state again.
      System.out.println("\nSecond yacht's name is changed");
      aTestClient.change();
      aTestClient.display(SIZE);

      // Delete a person and display the yachts'state again.
```

```
        System.out.println("\nA yacht is deleted");
        aTestClient.delete();
        aTestClient.display(SIZE - 1);

        System.out.println("Test Completed");
    }
}
```

There are many JDO vendors on the market. The JDO implementation used in the example is the OpenFusion JDO from Prism Technologies, but I am not endorsing any specific vendor. Choose the vendor based on your specific requirements. For simplicity, the initialization of the `PersistenceManagerFactory` instance part uses the vendor-specific APIs included in the package: `com.prism.j2ee.connector.jdbc.ManagedConnectFactoryImpl`. This is the only place that uses a JDO vendor-specific API; everything else uses the standard JDO APIs, with the vendor-specific implementation behind the scenes. If you use another JDO implementation, you can simply modify that portion of the code. If your application has access to a JNDI service, you can get a `PersistentManagerFactory` instance via JNDI lookup. Then you do not need to modify code at all when you switch to another vendor's JDO implementation.

The test client program first instantiates four `Yacht` objects and persists their state into a persistent store. The persisted objects are then retrieved one by one, and their states are displayed on the standard output screen. The program then changes the second yacht's name from "Liberty" to "New_Name." You see the change when the program redisplays the persisted objects' states on the screen. Finally, the test program deletes the second `Yacht` object from the persistent store. When the display is refreshed, the second `Yacht` object is gone.

The business logic in the above example is very simple. In the real life, the business rules and logic are much more complicated and complex domain objects must be handled. You will learn the support of complex domain objects in the next section.

Support for the Complex Domain Object Model

As a Java developer, you must be familiar with object-oriented design and programming. In almost all enterprise applications, the domain object models are fairly complex, with all kinds of relationships between classes: association, aggregation, composition, inheritances, and so on. The most notable relationships are extension or inheritance. These allow the abstraction of common attributes and behaviors of a set of classes into a base class. All the subclasses inherit the fields and methods from the super class.

If the objects are to be persisted, however, there is a gap between such an object model and the relationship model used by relationship databases. Typically, object-to-relationship mapping (O/R mapping) must be made, and developers must deal with low-level constructs of the database model, such as rows and columns, and constantly translate them back and forth. In the previous chapters of this book, you have learned all the tricks using JDBC APIs to couple your Java objects with underlying database tables. It is not only tedious, but the code is not 100-percent portable due to the variant of SQL flavors different databases use.

Container-managed entity EJBs partially solve this problem by postponing the O/R mapping to the deployment phase. Since you do not need to write any data-store access code, the CMP entity beans are decoupled from specific database type or even the specific database schema. As *write once, deploy everywhere* components, they are portable and reusable. However, when you deploy them to a specific database, you still have to know the relational data model in order to map the persistent fields to the corresponding table columns and to specify the relational fields in the deployment descriptor.

Moreover, the current CMP entity-bean specification does not support inheritance, one of the most important features of object-oriented design. This makes it impossible to build a complex domain object

model with CMP entity beans. Although quite a few design workarounds (or design patterns) have been proposed in the recent years, none of them eliminates this fundamental limitation.

As a Java developer, you certainly favor a mechanism that abstracts away any persistent details and has a clean, simple, object-oriented API to perform data persistence. With the recent official release of the JDO specification, you finally have an object-oriented, data-persistent mechanism to use.

Since the JDO persistent classes are simply Java classes, you build your domain model as usual and without worrying about the details of data persistence. This can be illustrated by using a classic example. Assume you have three classes: `Employee`, `ParttimeEmployee` and `FulltimeEmployee`. Their relationship is shown in Figure 23-1.

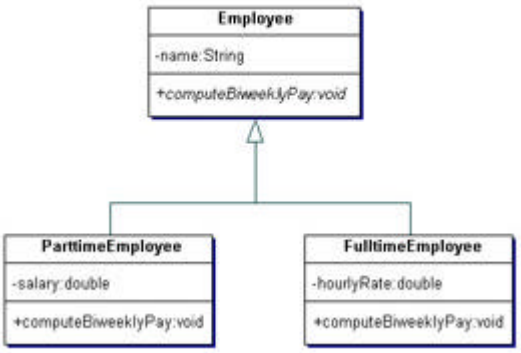


Figure 23-1: Class diagrams of the employee object model

The employee name is common for both part-time and full-time employees and thus is defined in the base class. The part-time employee is paid by hour and has an attribute `hourlyRate`. The full-time employ is paid by annual salary and hence has an attribute `salary`. An abstract method, `computeBiweeklyPay()`, is defined in the base class, and the implementation is provided in the subclasses. The implementations of this method are certainly different in two subclasses. For example, the part-time employee may be paid simple by the hours worked multiplied by the hourly rate. For the full-time employees, you may have to deduct all payroll deductions, and you may also have to handle holidays and vacations differently from the working days.

The coding of these persistent classes is straightforward. After you have written these classes, you need to tell the JDO enhancer that they should be persisted. Your XML MetaData file may look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
   <package name="java_database.jdo">
      <class name="Employee">
      </class>
      <class name="ParttimeEmployee" persistence-capable-
superclass="Employee">
      </class>
      <class name="FulltimeEmployee" persistence-capable-
superclass="Employee">
      </class>
   </package>
</jdo>
```

There may be slight variances in the XML MetaData file depending on the JDO implementation you use. Basically, you specify the inheritance using the `persistence-capable-superclass` attribute within the `class` tag. The JDO implementation handles all the persistence details for you transparently. It may even create the database schema, with referential integrity enforced to implement the inheritance relationship.

The client program accesses the JDO persistent class instances in the same way as it accesses any Java class instance. As an example, your client code may have the following code snippet:

… …

```
PersistenceManager pm = pmf.getPersistenceManager();
Employee emp = (Employee) pm.getObjectById(empId);
emp.computeBiweeklyPay();
```

… …

When the client calls `getObjectById` method, the JDO implementation automatically instantiates an instance of the correct class, either `ParttimeEmployee` or `FulltimeEmployee`. When the instance's `computeBiweeklyPay` method is called, the correct version is invoked. The polymorphism of the object-oriented paradigm makes the client-code simple and elegant. You cannot achieve such a level of simplicity and elegance by using any other data-persistence mechanisms.

In addition to inheritance, your classes can have any type of object relationships such as aggregation and composition, association, utilization, and so on. All you need to do is specify these relationships in the XML MetaData file and feed the file to the JDO enhancer.

After leaning the nuts and bolts of JDO, you must be eager to learn how to develop applications using JDO. This is discussed in the next section.

JDO Application Development Process

When designing a JDO-aware application, developers do not have to worry about the database type and database schema; but there are some best practices you probably want to follow. You should divide the Java classes into two categories: the *persistent classes* and *business classes*. The persistent classes are those that contain the data managed in the database, and the business classes are those that contain the business logic and query the persisted data. In the example code shown in Listing 23-1 and Listing 23-2, the class `Yacht` is persistent, and the class `TestClient` is business.

The persistent classes are very simple Java classes with data members, basic accessor methods, and (if necessary) some data-manipulation methods. There is no restriction from the JDO specification that you cannot put the business logic into these persistent classes. As a matter of fact, putting some business logic into these classes is a more object-oriented approach. However, since the persistent classes have to be further processed by the JDO implementation, as you will see later, it is recommended as a best practice to separate business logic from date persistence. The separation makes the data-persistent class more like simple JavaBean classes and can be generated automatically by many integrated development environments (IDEs) such as Together Control Center, Forte for Java, Visual Age, and so on.

The business classes contain all the business logic. They access the persisted data through the instances of persistent classes. Since JDO provides transparent persistence, there are no database calls (no SQL) in the business class code. You do not need to make these business classes persistent capable, which reduces the runtime overhead of the JDO implementation and hence improves the performance of the application.

The data represented by persistent objects are ultimately stored in the underlying database, and therefore the objects must be mapped to database entities. This mapping is done by the JDO enhancer based on the information provided in the XML MetaData file. In many cases, the JDO enhancer also creates a database schema to support the persistence.

In the XML MetaData file, all persistent-capable classes, as well as their relationships, are defined. Although the XML MetaData files can be created manually, many JDO implementations provide tools to help you with their creation. You should use the tools whenever they are available.

After the persistent classes are compiled by a Java compiler into bytecode, the Java class files are fed into the JDO enhancer, along with the XML MetaData file. The enhancer modifies the bytecode by doing the following:

- Making the class implement the `javax.jdo.PersistenceCapable` interface
- Adding implementations of `javax.jdo.PersistenceCapable` required methods based on the object relationship and underlying database structure
- Creating a database schema, if needed, for data persistence
- Adding any other necessary functionalities for optimized and transparent persistence

These JDO-enhanced classes are loaded into the JVM at the runtime. The XML MetaData file is also read by the JVM at the runtime to guide data persistence. The JDO-aware applications' development and execution process is shown in Figure 23-2.

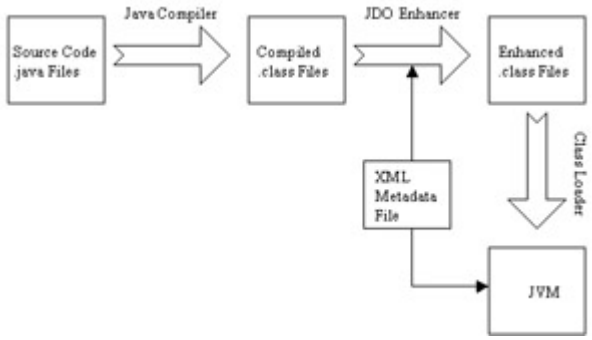


Figure 23-2: The JDO application-development and execution process

You can see from Figure 23-2 that only JDO enhanced classes are loaded into the JVM. Since the enhancer does all the persistence-support work, developers can focus on the implementation of business logic. On the other hand, you have to pay attention to the so-called performance penalty. Since the enhanced classes are generally larger than the plain class files, business logic is normally separated from the persistent classes and instead placed into the business classes. The business classes are not enhanced and thus have smaller footprints. At runtime, it is obvious that unnecessary network traffic on the database (typically residing on another network note) deteriorates performance. To overcome this potential weak point, most JDO implementations have a complex cache system.

Another important design target for all JDO enhancers is to provide compatibility among different database vendors and different JDO implementations. As the technology matures, more and more high-quality JDO implementations will become available.

In the previous chapters you learned about EJBs, the fundamental components of J2EE framework. Can you fit the JDO into the J2EE framework? Absolutely yes. The next section explains how it can be achieved.

Integration of JDO with the J2EE Framework

Unlike J2EE components, JDOs do not need to run inside a container. They can be used in any standalone applications. This makes them suitable for many batch processes.

On the other hand, JDOs do not enjoy the system-level support provided by containers such as transaction, security, transparent remote invocation, and so on. Application developers may have to code these system-lever services if they are needed, which prolongs the development cycle and may make the application more vulnerable to all kinds of defects.

The best way to develop an enterprise application may be by combining the better of the two worlds: using the container services provided by the J2EE framework and the transparent persistence provided

by the JDO specification. There are numerous ways to integrate the JDO into the J2EE framework. For example, you can build a two-tier system servlet and JSP pages on the client tier and JDO and other business processes on the second tier. You can also build a three-tier system with a servlet and/or JSP at the client tier, the session EJB or BMP entity beans in the middle tier, and the JDO at the back end (that is, the resource tier). You can certainly combine JDO with J2EE in any other innovative ways.

When you integrate the JDOs into the J2EE framework, the J2EE components typically take advantage of JDO's simple, object-oriented, and transparent persistence service. The servlet, JSP, or EJBs access the persisted object state via JDOs instead of by accessing the database directly. From these J2EE components' points of view, accessing JDOs and accessing databases share a great deal of logical similarity. This process is illustrated in Figure 23-3.

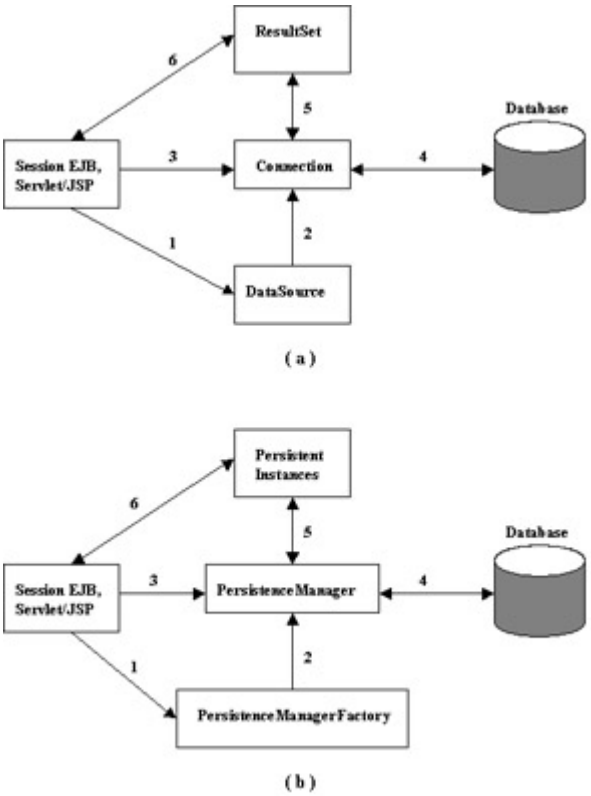


Figure 23-3: Data persistence with (a) JDBC and (b) JDO

Figure 23-3 (a) illustrates the scenario that session EJBs, servlet, or JSPs access the persisted data via JDBC. The J2EE components first get a transactional `Connection` to the database from the `DataSource` that represents the underlying database. The `DataSource` serves as a `Connection` factory. The J2EE components then access the database through the `Connection`, which returns a `ResultSet` object that wraps up the retrieved rows of data. The client then walks through the `ResultSet` to access each record.

In Figure 23-3 (b), JDO is used for data persistence. Instead of getting a database `Connection`, the J2EE clients first get a transactional `PersistenceManager` from the `PersistenceManagerFactory`. They then access the persisted data through the `PersistenceManager`. The `PersistenceManager` returns either a single persisted object or a collection of persisted objects. The J2EE components retrieve or update the persisted object state using the `accessor` methods (that is, `getters` and `setters`) of the persistent classes.

The code examples in the two scenarios illustrated in Figure 23-3 also share a great deal of similarity. For the JDBC approach, the code may look like what is shown here:

… …

```
InitialContext ctx = new InitialContext();
```

```
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/YachtClubDB");
Connection conn = ds.getConnection();
PreparedStatement pStmt = cx.preparedStatement("SELECT yacht_name,
engine_type FROM
                                                Yacht WHERE yacht_name=?")
PStmt.setString(1, yachtName);
ResultSet rs = pStmt.execute();
String engineType = rs.getString(2);
```

Note that the code snippet is for illustration only. When you develop real-life applications, you must code more defensively against exceptions. For example, you should check whether the result is `null` before you try to get the value of `engine_type`.

For the JDO approach, the code may look like this:

```
… …
InitialContext ctx = new InitialContext();
PersistenceManagerFactory pmf = (PersistenceManagerFactory) ctx.lookup(

"java:comp/env/jdbc/YachtClubPMF");
PersistenceManager pm = pmf.getPersistenceManager();
YachtKey yKey = new YachtKey(yachtName);
Yacht aYacht = (Yacht) pm.getObjectById(yKey);
String engineType = aYacht.getEngineType();
… …
```

Note that the code snippet is again simplified to demonstrate the core concept. The preceding code can be part of the session EJBs, servlets, or JSPs, depending upon your application design. It is seen that the session EJBs and servlets/JSPs use either JDBC or JDO in a similar manner.

Entity beans with bean-manager persistence can also take advantage of the simple and transparent persistence JDO provides. Instead of writing SQL code, developers access persisted objects' states via JDO-persistent class instances. Recall the discussion in Chapter 21 regarding that data-access codes are mostly in the following methods: `ejbCreate`, `ejbRemove`, `ejbFindByxxx`, `ejbLoad` and `ejbStore`. If JDO is used, you will write these methods using JDO APIs. For example, `ejbCreate` creates a persistent instance by first instantiating a persistent object and then calling the `PersistenceManager`'s `makePpersistent` method. `ejbRemove` deletes a persistent instance by calling the `PersistenceManager`'s `deletePersistent` method. `ejbLoad` retrieves a persistent instance by calling a `getObjectByxxx` method.

Entity beans with container-managed persistence have a unique method for persistence management. The concrete bean class is automatically generated by the EJB container at the deployment phase based on the information provided in the deployment descriptor. Database access is transparent to the bean developer, although certain knowledge regarding database structure is required to map the persistent and relational fields to corresponding database columns for deployment. There is typically no need or viable mechanism to integrate CMP entity beans with JDO. You may have to choose either one in your application based on your specific needs and constraints.

Summary

JDO bridges the gap between the Java object model and relational data model by providing a very natural yet very simple object-oriented mechanism to store and retrieve java objects. In this chapter, you have learned the following:

- How JDO achieves transparent persistence
- How to develop JDO-aware applications using standard JDO APIs
- How to integrate JDO into J2EE framework

This chapter concludes the discussion on data persistence. In the last four chapters, you have learned various ways to persist an object's state. Every approach has its strengths and weaknesses. Therefore, you should consider all options and choose the one that most suits your specific requirements and constraints.

Part VI: Database Administration

Chapter List

Part Overview

Part VI covers database-administration issues such as user management, security, and tuning. Although these issues may lack the glamour of creating, updating, and querying a database, they are nevertheless an important part of a practical application.

Good user management is one of the keys to database integrity. Understanding how and when to assign privileges to a user can make all the difference in avoiding accidental wipeouts of valuable databases and securing sensitive information from the prowling eyes.

Database tuning is the key to achieving great performance from a database-driven application. Badly written statements and poorly designed queries can bring an application to its knees. The correct use of indexes can make a huge difference to response times, and misusing them can make updates incredibly slow. The use of joins and views can also help speed the execution of a query. Finally, denormalization of the database is another key technique widely used to improve database performance.

Chapter 24: User Management and Database Security

In This Chapter

This chapter discusses how to create and manage users and groups. It goes on to define a database schema and shows how to create and manage these schemas. It further explains the concepts of user privileges and the assignment of permissions to the schema objects. The grouping of privileges into roles extend benefits such as ease of administration.

Groups, Users, and Roles

A user is a person who has been assigned certain privileges or permissions to perform certain tasks on the database. A logical collection of these users is a group. Most database management systems provide the capability of defining users and groups of users with different access privileges and operational roles. Typically, there is always a database administrator with full access privileges, as well as a number of other users who can access individual databases within the database management system.

Many systems support the concept of *groups*, which allow the administrator to order certain users logically. Database management systems allow you to manage these groups and the users within them via the Structured Query Language (SQL). Database users are completely separate from the operating system users, at least in concept. In practice, it might be convenient to maintain a correspondence, but this is not required.

Cross-Reference See Part II for discussions of SQL.

Groups have certain permissions assigned to them. Users that belong to the group inherit the permissions of that group. A database *role* defines what operations a user or users in a group can perform on the database, such as "Create Databases," "Backup Databases," and so on. A role is not the same as a group. Role definitions are specific to a particular DBMS, so look at the documentation provided with your specific database for these roles.

Working with Groups

A database management system uses the concepts of groups and users to assign certain privileges to perform tasks. We create groups, then users, and finally we assign roles or privileges to the users.

Creating a group

The first task is to create a group within which you can put users. You must be a database super user or administrator to use this command, which creates a group with no users:

```
CREATE GROUP group_name
```

Alternatively, you can create a group and assign users to it in one command; first, create users. If you try to run the following command without creating users, you will get an error message.

```
CREATE GROUP group_name WITH USER user1, user2
```

The general syntax for `CREATE GROUP` is as follows:

```
CREATE GROUP name
    [ WITH
     [ SYSID gid ]
     [ USER  username [, ...] ] ]
```

Dropping a group

Only database super users and administrators can use the `DROP GROUP` command. This command deletes a group, not the users:

```
DROP GROUP group_name
```

The users are simply left as they are, without any group assignments. You can always add these users to another group you create.

The JDBC code to create and delete a group is shown in Listing 24-1.

Listing 24-1: Working with groups

```
package jdbc_bible.part2;

import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;

public class GroupMaker {

  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String groupName = "";
  static String url = "jdbc:odbc:dummyDB";


  static String SQL_GroupCreate =
      "CREATE GROUP ";

  static String SQL_GroupDelete =
      "DROP GROUP ";

  public GroupMaker (){
    registerDriver();
  }

  public void setGroupName(String groupName){
    this.groupName = groupName;
  }

  public void registerDriver() {
    try
    {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      System.err.print(e.getMessage());
```

```
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
  }

  public void createGroup(){
    Connection con;
    Statement stmt;
    try
    {
      con = DriverManager.getConnection(url);
      stmt = con.createStatement();
      stmt.execute(SQL_GroupCreate + this.groupName);
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
    finally
    {
      try
      {
         if (con != null) {
           con.close();
         }
         if (stmt !=null) {
              stmt.close();
         }
      }
      catch (Exception ex) { // ignore }
    }
  }

  public void deleteGroup(){
    Connection con;
    Statement stmt;
    try
    {
      con = DriverManager.getConnection(url);
      stmt = con.createStatement();
      stmt.execute(SQL_GroupDelete + this.groupName);
    }
    catch(SQLException e) {
```

```
      System.err.println(e.getMessage());
    }
       finally {
            try
      {
              if (con != null) {
                  con.close();
              }
              if (stmt !=null) {
                  stmt.close();
             }
          }
      catch (Exception ex) { // ignore }
      }
  }

  public static void main(String[] args) {
    GroupMaker groupMaker = new GroupMaker ();
    groupMaker.setGroupName("Managers"); // which group to work with
     // Create a group
    groupMaker.createGroup();
       // Drop the group
     groupMaker.deleteGroup();
  }
}
```

Altering a group

Only database super users and administrators can use the `ALTER GROUP` command, which is useful when you want to change the group assignments for users. You use the following `ALTER GROUP ADD` command to add users to the group and `ALTER GROUP DROP` to delete users from the group:

```
ALTER GROUP group_name ADD USER username [, ... ]
ALTER GROUP group_name DROP USER username [, ... ]
```

Working with Users

This section describes the processes of creating, dropping, and altering users in a database.

Creating users

To create users and assign basic privileges to them, use the `CREATE USER` command. When you create a user, you can assign a password, certain basic permissions, and an expiration date, all in one command. You can also assign the group they belong to in the same command. However, if the group does not exist, it will not be created by using this command. You will have to use the `CREATE GROUP` command described earlier in this chapter.

The general syntax for the `CREATE USER` command is as follows:

```
CREATE USER username
```

```
[ WITH
 [ SYSID uid ]
 [ PASSWORD 'password'' ] ]
[ CREATEDB   | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
[ IN GROUP     groupname [, ...] ]
[ VALID UNTIL  'abstime' ]
```

The following code example creates a user with no password. This is not a very safe thing to do from a security perspective, so we do not recommend using this command. Nevertheless, the command is valid and creates a user with the given username, as shown here:

```
CREATE USER user_name
```

Next, create a user with a password, whose account is valid until the end of 2001. This user also has permission to create other users but not to create other databases. Here's an example:

```
CREATE USER user_name WITH PASSWORD 'jw8s0F4' NOCREATEDB
CREATEUSER VALID UNTIL 'Jan 1 2002'
```

Recall that we can assign users to groups only if the user exists. Similarly, the `CREATE USER` command allows us to assign users to groups only if the groups exist. For example, let us say we want to create a new group called "Managers" and assign two users to the group, "John Doe" and "Jack Smith." Neither of these users currently exists in the system. Here's one way to assign them to a group:

```
CREATE USER 'jdoe' WITH PASSWORD 'temppassword'
CREATE USER 'jsmith' WITH PASSWORD 'temppassword2'
CREATE GROUP 'managers' WITH USER jdoe, jsmith
```

Adding a user to a group does not create the user. Similarly, removing a user from a group does not drop the user itself. To create a new group and assign two new users to that group, you have to issue three separate commands to the database as shown below.

```
CREATE GROUP 'managers'
CREATE USER 'jdoe' WITH PASSWORD 'temppassword' IN GROUP managers
CREATE USER 'jsmith' WITH PASSWORD 'temppassword2' IN GROUP managers
```

Finally, you can use the ALTER commands to do the same as above.

```
CREATE GROUP 'managers'
CREATE USER 'jdoe' WITH PASSWORD 'temppassword'
CREATE USER 'jsmith' WITH PASSWORD 'temppassword2'
ALTER GROUP 'managers' ADD USER jdoe, jsmith
```

As you can see, there are several ways to create users. It is very hard to tell which method is better than another other. This depends on how well defined are your requirements for the groups and users. Is this something that will change frequently or is it something that you can define once? Depending on your organizational structure and how well defined is your Org Chart, you can decide which combinations of commands minimize your work.

Dropping a user

Only database super users and administrators can use the `DROP USER` command, which removes the specified user from the database. It does not remove tables, views, or other objects the user owns. If the user owns any database objects, you get an error message. Thus, to delete a user, you need to delete all objects the user owns or to change the ownership of the objects the user owns. Here's the general syntax for this command:

```
DROP USER user_name
```

For example, you will get the following error messages if the user does not exist or owns some object.

```
ERROR: DROP USER: user "user_name" does not exist
ERROR: DROP USER: user "user_name" owns database "name", cannot
be removed
```

Listing 24-2 displays the JDBC code to create and delete users.

Listing 24-2: Working with Users

```
package jdbc_bible.part2;

import java.sql.*;
import sun.jdbc.odbc.JdbcOdbcDriver;

public class UserManager {
  static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  static String url = "jdbc:odbc:dummyDB";

  static String SQL_UserDelete =
        "DROP USER ";

  public UserManager(){
    registerDriver();
  }

  public void registerDriver(){
    try
    {
      Class.forName(jdbcDriver);
      DriverManager.registerDriver(new JdbcOdbcDriver());
    }
    catch(ClassNotFoundException e){
      System.err.print(e.getMessage());
    }
    catch(SQLException e){
      System.err.println(e.getMessage());
    }
  }

  public void createUser(String username, String password) {
        Connection con;
       Statement stmt;
    try
    {
```

```
      con = DriverManager.getConnection(url);
      stmt = con.createStatement();
      stmt.execute("CREATE USER " + username + " WITH PASSWORD " +
password);
    }
    catch(SQLException e) {
      System.err.println(e.getMessage());
    }
       finally {
            try
       {
              if (con != null) {
                  con.close();
              }
              if (stmt !=null) {
                  stmt.close();
             }
          }
      catch (Exception ex) { // ignore }
      }
  }

  public void deleteUser(String username){
       Connection con;
      Statement stmt;
    try
    {
      con = DriverManager.getConnection(url);
      stmt = con.createStatement();
      stmt.execute(SQL_GroupDelete + username);
    }
    catch(SQLException e) {
      System.err.println(e.getMessage());
    }
       finally {
            try
       {
              if (con != null) {
                  con.close();
              }
              if (stmt !=null) {
                  stmt.close();
             }
```

```
          }
       catch (Exception ex) { // ignore }
       }
  }

  public static void main(String[] args) {
    UserManager userMgr = new UserManager ();
     // Create a user
    userMgr.createUser("john", "j23xqt3#");
       // Drop the user
     userMgr.deleteUser("john");
  }
}
```

Altering a user

The `ALTER USER` commands are useful for changing the permissions for a user, the password assigned to the user, or the expiration date for the user. If the user does not exist, you will get a similar error message as you will get if you try to delete a user that does not exist. The following command shows examples of what you can do with the `ALTER USER` command.

```
ALTER USER user_name WITH PASSWORD 'hu8jmn3'
ALTER USER user_name VALID UNTIL 'Jan 31 2030'
ALTER USER user_name CREATEUSER CREATEDB
```

Understanding Database Schemas

A database *schema* defines the objects to be stored within the database. These objects generally are tables that contain columns, indexes on these tables, views of tables, stored procedures and triggers, and security instructions. Other objects such as synonyms, links, and sequences should also be part of the schema, depending on whether they are implemented by the database vendor and used within the database. A database schema also includes storage instructions for the objects that are often defaults that go unnoticed.

A schema is a high-level abstraction of a container object that the SQL standard defines to contain other database objects. In many database management systems, a schema is the same as the database owner. In others, where a database can have multiple schemas, a schema denotes a collection of objects a single user owns.

Schemas are a named entity and generally follow the dot-naming conventions (as for Java packages and classes). Typically, schemas are named like this:

```
OWNER_NAME.OBJECT_NAME
```

You can create a schema and assign an owner to it. By default, if the user John Doe logs into the database and creates a bunch of objects, all the objects created belong to John's schema. A super user can create objects and assign them to be owned by other users.

The following is the general syntax for the `CREATE SCHEMA` command.

```
CREATE SCHEMA AUTHORIZATION authorization_name
create_oject_statement
    [ create_object_statement ... ]
[ permission_statement ... ]
```

The following command creates a new schema for the user who is logged in. Thus, if user John Doe is logged in to the database and issues this command, a new schema will be created, and John Doe will be assigned as the owner of the schema.

```
CREATE SCHEME myschema
```

Now let us assume that our other friend Jack Smith is logged in to the database and wants to create a schema for John Doe. Jack can issue the following command, which creates a new schema for John Doe. It is traditional to use the first initial and last name as the user name. However, this is organization specific.

```
CREATE SCHEMA johns_schema AUTHORIZATION jdoe
```

Alternatively, you can use the more complex form of the command by nesting several `CREATE OBJECT` statements when you are creating a schema. You can also nest the appropriate `GRANT` and `REVOKE` commands, described toward the end of this chapter.

```
CREATE SCHEMA johns_schema AUTHORIZATION jdoe
CREATE TABLE products (
  ProductID int (4) PRIMARY_KEY,
  ProductName varchar (40) NOT NULL,
  ProductPrice float(5) NOT NULL
)
GRANT ALL ON PRODUCTS TO jdoe   // The GRANT command is explained later
in this chapter
```

We have just learned how to create a schema. Next, we will talk about how to delete and how to make changes to the schema.

Managing a Schema

In most database management systems, you do not have to create a schema explicitly. If that is the case, a schema simply becomes the collection of objects a particular user owns. As you change those objects, you are, in effect, changing the schema for the user. When you explicitly create a schema as shown in the preceding section, you are creating various objects as part of the schema. Again, changing those objects changes the schema.

To manage objects within your schema, use one of the following `ALTER OBJECT` or `DROP OBJECT` commands, such as:

- `ALTER VIEW|TABLE`
- `DROP VIEW|TABLE`

The ALTER VIEW command is used to alter a previously created view (created by executing CREATE VIEW without affecting dependent stored procedures or triggers and without changing permissions. The basic syntax of the ALTER View command is shown below:

```
ALTER VIEW [ < database_name > . ] [ < owner > . ] view_name
[ ( column [,...i] ) ]

[ WITH < view_attribute > [ , ...i] ]

AS

    select_statement
```

The ALTER TABLE command can be used to change the schema of the table that you create. The following examples show various formats of the ALTER TABLE command:

1. Add a column to a table

```
ALTER TABLE [ ONLY ] table [ * ] ADD [ COLUMN ] column type
```

2. Rename a column within a table
3. `ALTER TABLE table [ * ] RENAME [ COLUMN ] column TO newcolumn`
4. Change the owner of a table
5. `ALTER TABLE table OWNER TO new owner`
6. Rename a table
7. `ALTER TABLE table RENAME TO newtable`

You can use the DROP SCHEMA or DROP VIEW command to drop a schema or view that you have created. This is shown in the next example:

```
DROP SCHEMA schema_name
```

The `DROP` will command succeed only if the schema does not contain any objects. If the schema contains database objects, the `DROP SCHEMA` command will return an error message.

User Privileges

Database management systems have a scheme of privileges that can be assigned to users. Privileges are actions that can be performed on schema objects. This assignment of privileges allows granular control of the database, allowing certain users to do certain tasks but not others tasks.

When you create a new database, you need to assign an owner for the database. By default, the owner of the database is the user who executes the `CREATE` command. By default, only the owner and the super user (administrator) can do anything on the database or on the object within the database.

To allow other users to work with the database, assign them the privileges to do so. Alternatively, you can assign privileges at the group level, so that certain groups of users can or cannot do certain things on specific database objects. This security schema is very flexible. If you have thousands of users, you do not want to have to assign individual permissions to each of them. Thus, organizing them in groups and assigning permissions to groups can help reduce the administrative effort.

Most database management systems support at least these three basic privileges:

- Select (read)
- Insert (append)
- Update/delete (write)

Certain database management systems might support extended privilege types such as Alter, Create, Process, Usage, and Shutdown. Refer to the documentation with the DBMS you are using for a complete list of the privileges it supports.

Some commonly implemented privilege types include the following:

- ALL (ALL PRIVILEGES)
- FILE
- RELOAD
- ALTER
- INDEX
- SELECT
- CREATE
- INSERT
- SHUTDOWN
- DELETE
- PROCESS
- UPDATE
- DROP
- REFERENCES
- USAGE

These privileges can be assigned to various objects in the database at the following levels:

- *Global* privileges apply to all databases on a given server.

- *Database* privileges apply to all tables in a given database.
- *Table* privileges apply to all columns in a given table.
- *Column* privileges apply to single columns in a given table.

Generally, you need to worry about only global or database privileges, and in most cases it is sufficient to assign privileges at one of these levels. Your major aim should be to assign the privileges at the highest level possible so that you save administration time. However, you do have the power to restrict access or to grant access to some critical columns or tables.

Next, we will talk about the management of user roles.

User Roles

Most database management systems support *user roles*, which are simply a grouping of user privileges. User roles are a neat administrative feature that saves time for the database administrator. The concept of roles is similar to that of groups. Just as groups can contain other groups, roles can contain other roles. Thus, a typical scenario where you would use all these concepts of users, groups, privileges, and roles can be explained as follows.

Putting It All Together

Imagine that Company A has 100 employees, five of which make up the management team, with access to all the information in the database. Another five make up the finance department, with access to all financial information. The remaining 90 are normal employees, with no specific access to the database except for their individual employee records.

For our example, assume a very simple database model, with the three following tables:

- COMPANY_DATA
- FINANCE_DATA
- EMPLOYEE_DATA

The management team has access to all the data (that is all the tables), the finance team has access to FINANCE_DATA and EMPLOYEE_DATA, and the employees can only view the EMPLOYEE_DATA. How would you organize this data using groups and roles? In this scenario, one solution is to follow these steps:

1. Create the 100 users in the database with the following syntax:

```
CREATE USER user1 with password 'temppassword'

CREATE USER user2 with password 'tempassword'
```

3. Create a `MANAGEMENT` group, and assign the five employees who are part of the management team of the company to that group:

```
CREATE GROUP MANAGEMENT USERS user1, user2, user3, user4, user5
```

4. Create a `FINANCE` group, and assign the five employees who are part of the finance team to that group:

```
CREATE GROUP FINANCE USERS user6, user7, user8, user9, user10
```

5. Create an `EMPLOYEES` group, and assign all the users to this group:

```
CREATE GROUP EMPLOYEES USERS user11, user12, user3, .. user100
```

6. Create an `EMPLOYEE_ACCESS` role, and assign the `SELECT` privilege for the EMPLOYEE_DATA table to the role:

```
CREATE ROLE EMPLOYEE_ACCESS
```

7. Assign the role to the `EMPLOYEES` group:

```
GRANT EMPLOYEE_ACCESS

     ON EMPLOYEE_DATA

    TO EMPLOYEES
```

10. Similarily, create a `FINANCE_ACCESS` role, and assign `SELECT` and `UPDATE` privileges for the EMPLOYEE_DATA and FINANCE_DATA Table to the role:

```
CREATE ROLE FINANCE_ACCESS
```

11. Assign the role to the `FINANCE` group:

```
12. GRANT FINANCE_ACCESS
13.     ON FINANCE_DATA, EMPLOYEE_DATA
       TO FINANCE
```

14. Finally, create a `MANAGEMENT_ACCESS` role, and assign `ALL` privileges for all tables to the role:

```
CREATE ROLE MANAGEMENT_ACCESS
```

15. Assign the role to the `MANAGEMENT` group:

```
16. GRANT MANAGEMENT_ACCESS
17.     ON EMPLOYEE_DATA, FINANCE_DATA, COMPANY_DATA
18.     TO MANAGEMENT
19.     WITH ADMIN OPTION
```

The creation of roles is a very database dependent task. MS SQL server has a stored procedure call to do this, whereas Oracle has a `CREATE ROLE` command. Thus, we are not explaining this syntax.

The GRANT Command

The `GRANT` command is used to give privileges to users so that they can perform certain tasks on the database. Recall that there are many types of privileges and that they can be assigned at various degrees of granularity (global, database, table, or column). It is important to note that the exact syntax of this command might differ as per your database. Still, here is an example:

```
GRANT PRIVILEGE ON table_name TO user_name
```

The `GRANT` command is more powerful. For example, you can `GRANT` a privilege to a user and allow the user to be able to grant that privilege to other users. Do this using the `WITH GRANT OPTION` clause. Now the grantee can grant the privileges specified in the `GRANT` command to other valid users. The following command gives user John Doe `SELECT` privileges on the Products Table and allows him to `GRANT` this privilege to others:

```
GRANT SELECT ON PRODUCTS WITH GRANT OPTION TO jdoe
```

There is also a `HIERARCHY` option. However, this is not yet widely supported. It grants privileges on all subtables and related tables. The complete syntax for the `GRANT` command is as follows:

```
GRANT priv_type [, priv_type]
    ON {tbl_name | * | *.* | db_name.*}
    TO user_name [, user_name]
    [WITH [GRANT OPTION | HIERARCHY OPTION]]
```

There is an equivalent `GRANT ROLE` command that enables you to grant roles instead of privileges. The syntax is identical to that of the GRANT PRIVILIEGES command. The only difference is that instead of `GRANT OPTION`, it is called `ADMIN OPTION`.

```
GRANT ROLE name [, role_name ]
    ON {tbl_name | * | *.* | db_name.*}
    TO user_name [, user_name]
    [WITH ADMIN OPTION]
```

The REVOKE Command

The `REVOKE` command is used to take away privileges from users so that they cannot perform certain tasks on the database. Just like the `GRANT` command, this command can be applied at various levels. It

is important to note that the exact syntax of this command might differ as per your database. For example, the following command revokes the SELECT privileges from John Doe on the Products Table:

```
REVOKE SELECT ON PRODUCTS FROM jdoe
```

The general syntax of this command is as follows:

```
REVOKE priv_type [, priv_type ]
    ON {tbl_name | * | *.* | db_name.*}
    FROM user_name [, user_name ...]
```

Summary

In this chapter, you have learned to create and manage users. Users are essential to any database system in order to regulate access to the database and to allow for granular control of the data. Furthermore, you have learned about the following topics:

- Organizing users within groups
- Creating and managing schemas
- Working with user privileges and user roles
- Granting and revoking these privileges or roles from users and groups

Chapter 25 shows you how to tune the database for enhanced performance.

Chapter 25: Tuning for Performance

In This Chapter

In this chapter, we talk about various performance enhancements that are possible to make on the database. The database is generally the bottleneck in most data-intensive applications. Tuning the database is the job of an experienced database administrator (DBA). As developers, there are many simple operations we can perform to get the maximum performance from the database. These include the appropriate use of indexes, joins, and views.

Database Tuning

It is hard to anticipate how the database will be used when we are initially designing it. Thus, tuning a database after it has been designed and deployed is important. There is a subtle distinction between *designing* and *tuning* a database. Generally, database design involves the database schema and a set of indexes and clustering decisions. Any subsequent changes to the schema or indexes can be considered *database tuning*. These changes include the addition of indexes and the adding or removing of columns and tables. This distinction is not critical as long as we understand that database tuning will usually be required even if the database design is excellent.

Database tuning involves the following types of activities:

- *Statement tuning* involves tuning the SQL queries and stored procedures that are run on the system.
- *Tuning of JOINS and indexes changes the ways the SQL queries are executed internally.*
- *Denormalization or normalization* changes the database schema to improve performance.
- *Horizontal partitioning* breaks up the tables by month, year, and so on to reduce the size of each table.
- *Vertical partitioning* decomposes relations to improve queries that use few attributes.
- *Views* create conceptual schemas for users.

Each of these is explained in more detail in the following sections.

Statement Tuning

Statement tuning is a complex operation on a database. In a DBMS, the query is passed through a query optimizer, which is responsible for identifying an efficient execution path for evaluating the query. These optimizers produce alternative plans, choosing the ones with the lowest cost in terms of execution time. Figure 25-1 depicts this process

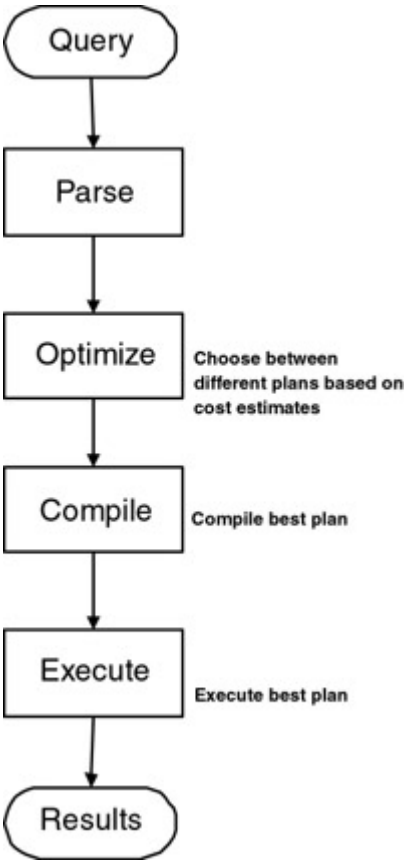


Figure 25-1: Typical execution path for a SQL query

To understand the details of these query plans, it is sufficient to realize that many database management systems do some level of query optimization for you. However, they cannot change the query or add indexes and so on. They can merely rearrange the order of execution of the query to use the existing indexes and table structures more efficiently. The manual tasks of creating indexes, rearranging the schema, and other operations are the focus of this chapter.

The first step in statement tuning is to define a *database workload*, which consists of the following components:

- A list of SQL statements and their frequenices, as a fraction of the total number of statements
- A list of updates and their frequenices
- Performance goals for each query and update

Once we have a representative database workload, we can identify which statements are not performing as well as others. These statements need tuning. Most database management systems provide tools to profile the SQL statements and see where the pain points lie. For example, you can use tools such as TKPROF, the SQL trace facility, and Oracle Trace to find the problem statements and stored procedures that are consuming the maximum resources. These resources generally indicate central processing unit (CPU) or Disk input/output (I/O) problems. The execution paths the query optimizer creates generally define the cost of a query based on the number of Disk I/O, since that is the most expensive resource. You can always add hardware to solve the issue, but that can prove expensive and is not the ideal solution to the problem.

Most database management systems allow you to see a graphical representation of the "Execution Path" of your statement and highlight the percentage of time taken for each operation the statement performs. This can be helpful in pinpointing the exact operations with a statement that need to be tuned or that are the performance bottlenecks. Remember, the aim of statement tuning is to get your SQL statements or stored procedures to be more efficient. So, when the query is passed through the query optimizer, the query optimizer should return a more efficient path than was the case previously.

We have the two following options to improve the performance of the system as a whole:

- **Modify your statement so that it uses fewer resources**— However, in many cases, it might not be possible to solve the problem by only tuning the statements. The performance might be due to the database design; thus, this might not always help.
- **Use slow statements less frequently** — This is obviously the ideal solution, but frequently it is not possible. We cannot base the performance of our system by restricting the users to certain operations. This might be a feasible solution or the only solution at times; however, as technical professionals, we should not dictate the business needs of an application.

We need to concentrate on the first option. There are three basic ways to achieve statement tuning, all of which involve restructuring the following items:

- Statements, working with JOINS (discussed next) and `WHERE` clauses
- Restructuring Indexes, and choosing between clustered and non-clustered indexes
- Data and rows that are being accessed. This is covered when we talk about normalization and denormalization of the schema, later in this chapter under "Changing the Database Schema."

We will first talk about JOINS and the various ways of improving the performance of a SQL query by working with JOINS.

Tuning JOINS

A JOIN is a SQL construct that allows a developer to obtain data from more then one table using a single query. JOINS are the primary reasons for database performance-related issues. Optimizing JOIN queries is extremely important for system performance, since relational databases make heavy use of JOINS. The order in which we join tables can have a significant impact on performance. The main objective of statement tuning is to avoid doing unnecessary work to access rows, which do not affect the result. That is, it does not change the rows that would be returned by the query. JOINS should always return the minimum number of rows so that they are manipulating the least amount of data.

These are the three rules of thumb involved when considering how to improve the performance of a query:

- Use indexes instead of doing full-table scans.
- Use the appropriate indexes to retrieve the minimum number of possible rows.
- Choose the join order so as to join fewer rows to tables later in the join order.

In general, you should follow these rules when using JOINS: :

- All the columns being joined on should have their own indexes.
- For maximum performance, always try to join on columns of similar data types.
- For maximum performance, always try to join on numeric columns instead of on chars and varchars.
- If your JOIN contains four or more tables, consider denormalizing your database or using VIEWS.
- Do not use SELECT * when performing JOINS.
- JOINS should be performed on columns with unique values; otherwise, most database management systems do a full table scan even if an index exists.
- JOINS are generally faster than subqueries or nested queries.
- Be careful of the type of JOIN you use, and determine whether it is the best JOIN for your goals. If the outer table is larger than the inner table, OUTER JOINS might be the right choice, or vice versa. The idea is to anticipate the JOIN that has to access the minimum number of records to produce the desired result.

Cross-Reference

In Chapter 9, we talk about the various types of JOINS. Here we introduce another category of JOINS based on the algorithms they use for implementation.

There are cases in which the choice of JOINS can result in a significantly different number of rows, which has a direct relation to the performance of the query. For example, instead of creating a LEFT JOIN, a developer might use a CROSS JOIN. There can be a situation in which fewer than 1000 rows should have resulted from the LEFT JOIN, but because a CROSS JOIN has been used, over 100,000 rows are returned instead. Then the developer uses a SELECT DISTINCT to filter the extra rows the CROSS JOIN creates. This is clearly an example in which we need to watch for the type of JOIN used. As a general caution, avoid using CROSS JOINS unless that is the only way to accomplish what you

need. The same is true for LEFT OUTER JOINS. This type of JOIN returns all rows from the outer table; assuming that the outer table is large, we would get rows we would need to filter out using a DISTINCT or some other method.

The choice of which JOIN algorithm is used is not always in the hands of the DBA or programmer. It is generally decided by the DBMS. However, if you know how the underlying plumbing works, you can make some intelligent decisions on the use of JOINS. We introduce the various algorithms later in this chapter; however, a detailed explanation is out of scope for this book. *Database Management Systems*, by Raghu Ramakrishnan, Mc-Graw Hill, August 1997, ISBN 0070507759, covers the internals of database-performance concepts, using relational algebra techniques.

The following JOIN algorithms are discussed next:

- Nested loop JOINS
- Block nested loop JOINS
- Index nested loop JOINS
- Sort merge JOINS
- Hash JOINS

Nested loop JOINS

This join algorithm forces the scan of the entire inner table for each row of the outer table. Basically, this algorithm and its variants are computing a cross product. You can see that this can be a huge performance hindrance.

Block nested loop JOINS

This is a refinement of the simple nested loop join algorithm, which allows the caching of the smaller relation in memory, which can improve the performance of the query. If the smaller relation cannot fit entirely in memory, the algorithm creates blocks of the table that can fit in memory and performs the computations. Thus, even though we are still computing a cross product, we are using some caching and buffering for performance.

Index nested loop JOINS

If there is an index on one of the relations in the join, the indexed relation can be treated as the inner table. An in-memory hash is created on the inner table. This avoids the computation of a cross product.

Sort merge JOINS

This algorithm sorts both the relations on the join attribute and computes the matching rows. Thus, we can consider only the chunk of rows from the outer table that match the criteria and do not have to compute a cross product. This algorithm can be further refined to combine the merging phase of the sort algorithm with the merging phase of the join.

Hash JOINS

Hash JOINS are similar to the sort merge join; however, they use hashing instead of sorting to avoid the cross-product computation. We hash both relations on the sort attribute using the same hash function. This way, we can read in the entire partition of the smaller relation and scan only the corresponding rows from the larger relation.

In general, if we can control the JOIN algorithm, we should force the use of hash JOINS and sort merge JOINSJOINS. There might be cases in which index JOINS are a good choice. However, this should be left to a good DBA and the underlying database support.

Next, let us talk about how the choice of indexes, and the type of indexes can help improve the performance of queries.

Tuning Indexes

The query optimizer most database management systems use first looks for indexes when creating the query-execution paths. Indexes can be used in several ways and can lead to execution plans that are significantly faster than plans that do not use indexes. The query optimizer would generate different plans based on the indexes available and then use the plan that is the fastest in terms of execution time. The underlying problems related to indexing and poor performance are as follows:

- No indexes, or too few indexes on a table, causing a full table scan for query execution.
- Existing indexes are not selective enough for a particular query, so the index is not used by the optimizer.
- Too many indexes are assigned to a table, so data modifications are slow. There is a cost associated with creating and maintaining an index. We should try to keep this to a minimum.Drop indexes that the query optimizer does not use and ensure that no duplicate indexes are present.
- The index key is too large, so using the index generates high I/O. This generally happens when we concatenate various columns in an index (composite indexes).

The optimizer consists of one of these four types of access paths:

- **Single-index access paths:** If there are multiple indexes that match, due to a `WHERE` clause, each index offers an alternative execution path. The optimizer chooses the access path that requires the lowest disk I/O (or cost).
- **Multiple-index access paths:** A query optimizer might choose an execution path where the query is broken down to use multiple indexes for the selects and then use an intersect operation to get the final data set.
- **Sorted-index access path:** In cases in which a GROUP by or SORT option is part of the query, the query optimizer might use a sorted index to retrieve the data and then filter by the select and other operations on the query.
- **Index-only access path:** If indexes exist on all the attributes the statement uses, an index-only access path might be chosen.

Note

It is important to choose your indexes based on the types of JOINS and `WHERE` clauses in the query.

Since we have decided that the use of indexes is important in query optimization, how do we decide which indexes to use and whether these should be clustered or nonclustered?

Clustered indexes

In *clustered indexes*, the physical order of the rows in the table is the same as the logical order of the index key values. Since most database management systems allow only one clustered index on each table, it is important to choose the clustered index wisely. Choose the queries that run most frequently, and decide the candidate for the clustered index. All tables should have a clustered index. By default, the DBMS creates a clustered index on the primary key. However, in many cases, we want to change this based on the following guidelines. Candidates for clustered indexes are the following kinds of columns:

- Ones used by the `ORDER BY` and `GROUP BY` clauses. The data is presorted for you, and aggregates such as MAX, MIN, and predicates such as <, >, and BETWEEN are faster.
- Ones that contain a limited number of distinct values, such as columns that hold country and state information. However, distintive columns such as binary and sex should not be indexed at all.
- Ones found in the `SELECT` clause
- Ones that are not frequently changed
- Ones used in JOINS

Nonclustered indexes

Candidates for *nonclustered indexes* are all cases in which a query does not meet the requirements for a clustered index (just discussed). Since most database management sytems allow multiple nonclustered indexes, you can have a nonclustered index for all columns that are using SQL statements. The only considerations are space and the cost of the creation of indexes. Do not create indexes unnecessarily, as the cost associated with the creation and maintainance of an index can become prohibitively high.

Consider the following example:

```
SELECT E.employee_name, D.department_name FROM Employee E, Department D
WHERE E.salary BETWEEN 10000 AND 20000
AND E.hobby='Stamps' AND E.department_no = D.department_no
```

The question is which indexes to use on the Employee and Department Table for maximum query performance. Based on the guidelines for each type of query, use a clustered index on Employee.salary to increase the performance of range query. Use a non-clustered index on Employee.hobby, Department.department_no, and Employee.department_no. However, for this query, indexes on Employee.employee_name and Department.department_name are not required. We might decide to create these to increase the performance of queries such as Find Employee by Employee Name, but not for this specific query.

Composite indexes

Composite indexes include more than one column as the key to the index. The query optimizer will not use the index if the first column is not the only column that the WHERE clause uses. Composite indexes are also useful for accessing columns that are frequently accessed together, such as city and state.

Consider the following example, in which our primary query is a select on city and state and order by city:

```
SELECT E.employee_name, E.employee_city, E.employee_state
FROM Employee E
WHERE E.employee_state = "Dummy State"
ORDER BY City
```

A composite index on city and state can improve the performance of the `ORDER BY` clause; however, it cannot help the `WHERE` clause. A composite index on state and city can help the `WHERE` clause but not the `ORDER BY` clause. Thus, in this case, we might just use two separate nonclustered indexes. However, if we did not have the `ORDER BY` clause in this query, a composite index on state and city might be useful.

The major advantages of composite indexes are that since more columns are used for a single index, they can reduce the total number of indexes on a table. This reduces the overhead to create a larger number of indexes. Composite indexes can serve more queries than simple indexes can.

Working with the database schema and performing normalization or denormalization is another technique used to improve the performance of a database.

Changing the Database Schema

Another important aspect of database tuning is the identification of problem tables. Even by optimizing JOINS and indexes, if the queries do not perform as well as we need them too, it is possible that the schema is too normalized. We need to understand a trade off. Normalization is good and in theory the right way to go. However, normalization comes at the cost of performance.

Normalization

It is good practice to normalize the database to at least second- or third-normal form during initial database design and to denormalize parts of the database during database tuning, using a representative database workload. This process is sometimes termed "schema evolution." The normal forms are covered in detail in Chapter 2 of this book.

Denormalization

Once we identify which tables are leading to performance problems, they become candidates for *denormalization*. Denormalization breaks a relation from a higher normal form into a lower normal form.

This might be breaking the tables into two or more tables, with some redundant data. We settle for the problems of maintaining redundant data if the process improves the overall performance of the system. Relational algebra concepts define how denormalization can be done using mathematics to preserve referential integrity and minimize redundancy. We spare our readers such mathematics and concentrate on some conceptual techniques that can be easily followed in the real world.

Multiple Data Tables

There is another technique often followed — we keep a set of normalized tables for updates and a set of denormalized tables for the queries. We then have to manage at a database level the synchronization of these two sets of tables. We allow the system to write to the normalized tables and then use triggers to update the denormalized tables. The denormalized tables are read-only from the application, used to serve the SQL queries.

The process of creating and maintaining the multiple data tables is described in the next section.

Creating Redundant Data

Repeating data in two tables can lead to producing redundant data. This, in turn, might lead to avoiding the use of a JOIN to run a query. This process should only be used if it increases the overall performance of the system. We can keep the normalized table for updates and use a trigger to update the denormalized tables the queries use.

Consider Figure 25-2, in which we might choose to have some redundant data in order to avoid a JOIN. This example uses City_Id in both the Customer and Customer_Order Tables and JOINS them with the City Table.

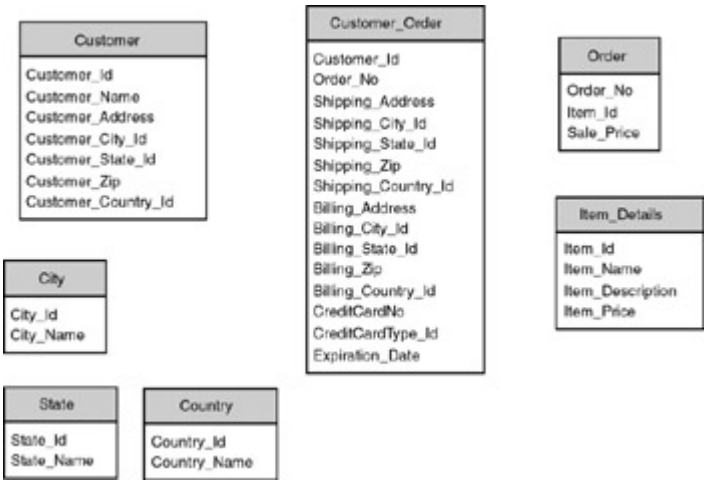


Figure 25-2: Example of schema with redundant data

Suppose a query to retrieve all orders for a customer is leading to performance problems. In the tables shown in Figure 25-2, we have to JOIN all the tables to return the results of the desired query. A simple query to retrieve all orders needs a join of seven tables. This can clearly lead to performance issues. Thus, we need to denormalize the data.

We decide to keep the city, state, and country names in the tables as redundant data. Furthermore, we can also choose to keep some of the customer details such as Customer_Name in the Customer_Order Table and so on. However, for this example, we just show you how to remove the City, State, and Country Tables. This is shown in Figure 25-3.

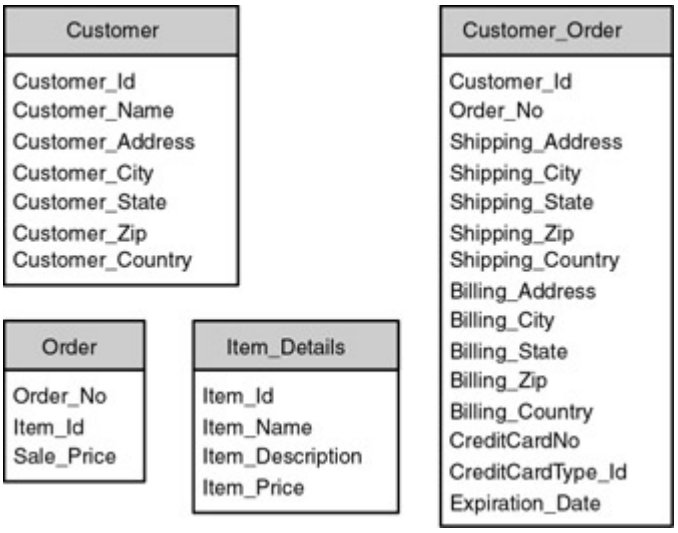


Figure 25-3: Example of normalized data schema

As we see in Figure 25-3, this schema gets rid of the City, Country, and State Tables. This does lead to redundant data. The names of the city, state, and country need to be stored individually for each customer and twice for each order (shipping and billing address). However, to retrieve all orders for a customer, we now need to JOIN only four tables. This is definitely faster than joining seven tables. With the appropriate indexes on the four tables, the performance increase we achieve by reducing from seven tables to four tables is well worth the efforts to maintain the redundant data.

We will now talk about derived columns and tables and how they affect database performance.

Using Derived Columns and Tables

Another technique used to improve performance is to create certain tables that maintain summary information such as totals, collections of columns, and so on. These are used when the performance of the queries is affected by the use of aggregations or operations. Operations such as summing data, concatenating string, and so on as part of a SQL query can be time consuming. We can create tables that are updated using triggers to maintain this aggregated data.

Consider Figure 25-4, which shows a table that maintains the price of a product by month. We create a temporary average price table that is updated using a trigger. This table maintains the average price for a product over a year. Another table maintains the average price by quarter. This is an example of a *derived table* and is useful if there are many requests to know the average price of a product, generally in reporting activities. Depending on the number and types of queries (average price by quarter and average price by year), we can choose to have only the TMP_Avg_Price_Quarter Table and use that to derive the TMP_Avg_Price_Year data. These are decisions we make after evaluating a good workload.

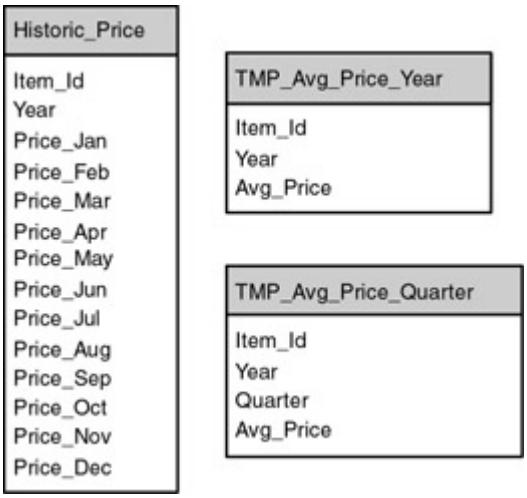


Figure 25-4: Data schema showing derived tables

Another commonly used technique to improve performance of a database is called *data partitioning*.

Data Partitioning

Data partitioning is the breaking up of large amounts of data, into small, more manageable chunks. It comes in two flavors -

Horizontal Partitioning

Horizontal partitioning of data is used with large amounts of data to make the number of records per table more manageable. Many systems that use horizontal partitioning need to build flexible SQL statements to be able to manage these random tables. This practice is often called *row partitioning*.

In the tables shown in Figure 25-4, we see that the historic price table contains a row for each product for each year. We can choose to partition this table horizontally to maintain the data, so that a separate table maintains data for each year. Thus, for five years, we can have five tables. Another valid partition option might be a table for each product. However, this does not sound feasible, as for 50 products we need to have fifty tables. However, in the first case, it is feasible to have five tables, one for each year.

Vertical Partitioning

Vertical partitioning, or *subsetting*, allows you to create smaller tables in terms of the number of attributes for a table. This makes smaller tables from which data can be retrieved faster. Row splitting divides the original table vertically into tables with fewer columns. The decomposed tables generally have one-to-one relationships with each other. The various types of relationships are covered in Chapter 1 of this book.

Like horizontal partitioning, vertical partitioning allows queries to scan less data, hence increasing query performance. For example, a table containing 10 columns, of which only the first four are frequently accessed, may benefit from splitting the last six columns into a separate table. This way, we would the table with four columns, instead of 10, increasing the performance of a query. However, we need to be careful; when we need all 10 columns, we have to do a JOIN, and this might be expensive.

The last technique that we will talk about to improve the performance of a database is the use of Views.

Using Views

A *view* is a representation of some subset of data from a table. This is created at the database layer. To increase the performance, we might define subsets of tables as views and run our queries of these

views. This is mainly useful for reporting components where the user wants to see different views (no pun intended) of the data, based on some criteria.

All the techniques described in this chapter help us improve the performance of the database. However, the database needs to be monitored closely on an on-going basis. The change is usage patterns of the database, can arise the need for further tuning.

On-Going Monitoring

Now that you know how to tune your database, are you done? Most definitely not. It is important to monitor your database and tune it based on the most current usage patterns. A good DBA maintains logs of all queries on the system and is able to choose a good workload that represents usage patterns of the database. For example, when you deploy a new system, the users are just starting to learn it, entering profile information and so on. Tuning the system to optimize these operations might not reflect the workload once the users are experts and as the data in the system continues to grow. Thus on-going monitoring of the system is required to identify new performance bottlenecks.

Summary

In this chapter, you learned various ways to improve the performance of your database. These include

- Statement tuning using JOINS and indexes
- Normalization and De-normalization of the schema
- Data partitioning
- Use of views

The database tends to be the bottleneck for most applications. Thus, it is important to design and tune your database properly to achieve the desired performance results. The tools DBMS provides can be used efficiently to pinpoint the trouble points of a database and effectively tune them.

Appendix A: A Brief Guide to SQL Syntax

Overview

This guide is intended to serve as a handy reference to essential SQL syntax. In practice, much of what the average Java database programmer needs to accomplish can be handled within the framework of the core statements covered in this appendix. This appendix is not intended to be a comprehensive summary of all the nonstandard variations that the various RDBMS suppliers implement.

Although standardization has been an important step in the universal adoption of SQL as the common language of relational database systems, the ANSI SQL standard has turned out in practice to be little more than a common starting point for a wide range of proprietary dialects. However, there is sufficient commonality for the basic commands to work with most database management systems and to handle most, if not all, SQL programming requirements.

The differences among SQL implementations arise from differences in their underlying design goals. MySQL, for example, has not supported transactions until very recently. Stored procedures and triggers are scheduled for implementation in future releases. The reason for this is that MySQL was originally intended to fill the need for a fast database suitable for such applications as membership management in Internet-based applications. Similarly, Oracle's goal of achieving pre-eminence at the high end of the RDBMS market place is accompanied by a very rich implementation of the SQL language.

Since there are significant underlying differences among database systems, you are strongly advised to refer to the documentation your DBMS supplier provides for the details of your own version. Bear in mind, however, that writing SQL that relies heavily on the unique features of a given implementation makes porting applications far more difficult.

Because it is intended to help you find the syntax of a SQL command you want to use, this appendix is organized alphabetically. The following SQL commands are reviewed in this appendix:

▪ ALTER PROCEDURE	▪ CALL
▪ ALTER TABLE	▪ CASE
▪ ALTER TRIGGER	▪ CAST
▪ ALTER VIEW	▪ COMMIT
▪ CREATE DATABASE	▪ GRANT
▪ CREATE FUNCTION	▪ INSERT
▪ CREATE INDEX	▪ REVOKE
▪ CREATE PROCEDURE	▪ ROLLBACK
▪ CREATE TABLE	▪ SAVEPOINT
▪ CREATE TRIGGER	▪ SELECT
▪ CREATE VIEW	▪ SET TRANSACTION
▪ DELETE	▪ START TRANSACTION
▪ DROP	▪ UPDATE

ALTER

This section discusses the following ALTER commands:

- ALTER PROCEDURE
- ALTER TABLE
- ALTER TRIGGER
- ALTER VIEW

ALTER PROCEDURE

ALTER PROCEDURE is supported by Oracle and MS SQL Server but is not widely supported by other database management systems. This is the basic syntax of the command:

```
ALTER PROCEDURE procedure_name
    [ { @parameter data_type }
    ] [ ,...n ]
AS
    sql_statement [ ...n ]
```

ALTER TABLE

ALTER TABLE provides a means of making changes to a table without losing the contents. It is supported by most RDBMS systems. When adding a column, the datatype and attributes are exactly as defined in the CREATE TABLE command shown below:

```
ALTER TABLE table_name
[ADD ColumnName DATATYPE[(LENGTH)][ATTRIBUTES]] |
[ALTER ColumnName SET | DROP ATTRIBUTE] |
[DROP ColumnName] |
[ADD TABLE_CONSTRAINT] |
[DROP TABLE_CONSTRAINT];
```

An example of this command might look like this:

```
ALTER TABLE CONTACTS ADD PHONE VARCHAR(20)
CHECK (PHONE LIKE '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]');
```

There are a number of variations in the behavior of this command from system to system. It is important to read your documentation.

ALTER TRIGGER

`ALTER TRIGGER` is supported by Oracle and MS SQL Server but is not widely supported by other database management systems. This is the basic syntax of the command:

```
ALTER TRIGGER trigger_name
ON ( table | view )
  {( FOR | AFTER | INSTEAD OF ) { [ DELETE ] [ , ] [ INSERT ] [ , ]
[ UPDATE ] }
AS
   sql_statement [ ...n ]
```

ALTER VIEW

`ALTER VIEW` is supported rather differently by Oracle and MS SQL Server but is not widely supported by other database management systems. Use of the `ALTER VIEW` command is shown below:

```
ALTER VIEW [ < db_name > . ] [ < owner > . ] view_name [ ( column
[ ,...n ] ) ]
    [ WITH < view_attribute > [ ,...n ] ]
AS
    select_statement
    [ WITH CHECK OPTION ]
```

CALL

The `CALL` command is used to invoke a stored procedure in Oracle and PostgreSQL. It is not supported by MySQL or MS SQL Server. In SQL Server, stored procedures are simply invoked by name.

CASE

The `CASE` statement can be used in `SELECT` and `UPDATE` statements in many versions of SQL. Here's an example of its usage:

```
SELECT First_Name, Last_Name,
    CASE WHEN Last_Name = 'Corleone' THEN 'Mafioso' ELSE '' END
    Comment
FROM CUSTOMERS;
```

This statement generates a result set like this:

First_Name	Last_Name	Comment
Michael	Corleone	Mafioso
Fredo	Corleone	Mafioso

First_Name	Last_Name	Comment
Francis	Corleone	Mafioso
Vito	Corleone	Mafioso
Tom	Hagen	
Kay	Adams	
Mario	Puzo	

Oracle does not support the `CASE` statement but provides similar functionality through the `DECODE` statement.

CAST

The `CAST` command casts one data type to another. This is the basic syntax of the command:

```
SELECT CAST(ZIP AS INT)
FROM CUSTOMERS;
```

`CAST` is not widely supported by database management systems.

COMMIT

The `COMMIT` command closes an open transaction. It is supported by most DBMS systems, with the notable exception of MySQL. This is the basic syntax of the command:

```
COMMIT [TRANSACTION | WORK]
```

Some systems allow you to name a transaction and to commit the transaction by name. This is a feature you should check in your documentation, since it is not supported by Oracle and does not commit the named transaction in SQL Server.

Note

Transactions can be opened explicitly with the `BEGIN TRANSACTION` command and are opened implicitly by `INSERT`, `UPDATE` or `DELETE` statements.

CREATE

This section discusses the following `CREATE` commands:

- `CREATE DATABASE`
- `CREATE FUNCTION`
- `CREATE INDEX`
- `CREATE PROCEDURE`
- `CREATE TABLE`
- `CREATE TRIGGER`
- `CREATE VIEW`

CREATE DATABASE

Although it is not a SQL-99 command, `CREATE DATABASE` is supported by most database management systems. This is the basic syntax:

```
CREATE DATABASE dbName;
```

This command has implementation-specific variants used to define physical-storage attributes such as file paths, so it is a good idea to look up the details in your documentation if you need to go beyond your system defaults.

CREATE FUNCTION

The CREATE FUNCTION command is defined in SQL-99 as a means of creating user-defined functions. The SQL-99 definition allows the user to define a function in any of a variety of programming languages. PostgreSQL supports this capability, but MySQL supports only C/C++. SQL Server supports functions defined in SQL, using syntax like this:

```
CREATE  FUNCTION [ owner_name. ] function_name
    ( [ { @parameter_name [AS] data_type [ = default ] } [ ,...n ] ] )
RETURNS data_type
 [ WITH < function_option> [ [,] ...n] ]
AS
 BEGIN
    function_body
    RETURN scalar_expression
 END
```

In MySQL, user-definable functions are written in C or C++ as dynamically loaded libraries. They are then defined to MySQL using this syntax:

```
CREATE FUNCTION <function_name> RETURNS [string|real|integer]
       SONAME <shared_library_name.so>
```

The CREATE FUNCTION command is both useful and widely supported. However, it is supported in such a variety of different forms that you have to look up your own system's version before you can make any use of it.

CREATE INDEX

Although the basic CREATE INDEX command is supported by all major relational database management systems, it, too, has many implementation-specific variants. The basic form of the command, which is directly supported by most systems, is as follows:

```
CREATE INDEX Index_Name ON Table_Name(Column_Name);
```

Here's an example:

```
CREATE INDEX ZIP_INDEX ON CUSTOMERS(ZIP);
```

Most systems also support composite indexes, which are based on more than one column, as shown here:

```
CREATE INDEX STATE_AND_ZIP_INDEX ON CUSTOMERS(STATE, ZIP);
```

CREATE PROCEDURE

The CREATE PROCEDURE command is used to create stored procedures. Stored procedures are precompiled SQL statements, which offer better performance than statements that have to be compiled at runtime.

The basic syntax defines the procedure name, followed by an argument list with data types, the key word AS, and the SQL statement to be compiled, as shown here:

```
CREATE PROCEDURE INSERT_CONTACT_INFO
 @ID INT, @FName VARCHAR(20), @MI CHAR(1),
 @LName VARCHAR(30),@Street VARCHAR(50), @City VARCHAR(30), @ST CHAR(2),
 @ZIP VARCHAR(10), @Phone VARCHAR(20), @Email VARCHAR(50)
```

```
AS
INSERT INTO CONTACT_INFO
 (ID, FName, MI, LName, Street, City, ST, ZIP, Phone, Email)
VALUES
 (@ID, @FName, @MI, @LName, @Street, @City, @ST, @ZIP, @Phone, @Email);
```

Stored procedures are not supported by either PostgreSQL or MySQL, the second of which relies on user-defined functions to support precompiled statements.

CREATE TABLE

The `CREATE TABLE` command is used to create tables and to define the columns they contain. It is supported, with variations, by all database management systems. This is the basic form of the command:

```
CREATE TABLE table_name
(ColumnName DATATYPE[(LENGTH)]  // define datatype and length if applicable
[DEFAULT]                       // provide an optional default value
[CHECK]                         // perform a validity check
[REFERENCES]                    // validate against a look up table
[NULL | NOT NULL]               // column can | can not contain NULL
[UNIQUE]                        // column values must be unique
[PRIMARY KEY]                   // column is primary key
[,...n]);                       // additional columns
```

An example of this command might look like this:

```
CREATE TABLE CONTACTS
(ID INT IDENTITY (1001, 1) PRIMARY KEY,
 FNAME VARCHAR(20),
 MI CHAR(1) NULL,
 LNAME VARCHAR(30) NOT NULL,
 STREET VARCHAR(30),
 CITY VARCHAR(20),
 STATE CHAR(2) REFERENCES STATES(STATE),
 ZIP CHAR(5) CHECK (ZIP LIKE '[0-9][0-9][0-9][0-9][0-9]'));
```

Minor constraint variants used by some popular DBMS systems include the following constraints for MySQL and SQL Server, respectively:

```
AUTO_INCREMENT    //a single autoincrementing integer may be assigned to
a table


IDENTITY (seed, increment)
```

The column definitions used in the `CREATE TABLE` command require a data type for each column. Table A-1 lists common SQL data types, together with the corresponding Java data type.

Table A-1: SQL Data Types

SQL Type	Java Type	Description
CHAR	String	Fixed length character string. For a CHAR

Table A-1: SQL Data Types

SQL Type	Java Type	Description
		type of length n, the DBMS will invariably assign n characters of storage, padding unused space.
VARCHAR	String	Variable length character string. For a VARCHAR of length n, the DBMS will assign upto n charcters of storage, as required.
LONGVARCHAR	String	Variable length character string. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
NUMERIC	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
DECIMAL	java.math.BigDecimal	Arbitrary-precision signed decimal numbers. Can be retrieved using either BigDecimal or String.
BIT	boolean	Yes / No or True / False value
TINYINT	byte	8 bit integer values
SMALLINT	short	16 bit integer values
INTEGER	int	32 bit integer values
BIGINT	long	64 bit integer values
REAL	float	Floating point number, mapped to float
FLOAT	double	Floating point number, mapped to double
DOUBLE	double	Floating point number, mapped to double
BINARY	byte[]	Retrieve as byte array
VARBINARY	byte[]	Retrieve as byte array
LONGVARBINARY	byte[]	Retrieve as byte array. JDBC allows retrieval of a LONGVARCHAR as a Java input stream.
DATE	java.sql.Date	Thin wrapper around java.util.Date
TIME	java.sql.Time	Thin wrapper around java.util.Date
TIMESTAMP	java.sql.Timestamp	Composite of a java.util.Date and a separate nanosecond value

The `CREATE TABLE` command is one of the richest in proprietary features in all DBMS systems, particularly as regards defining physical storage. You are well advised to look these up in your documentation. Oracle's variations on `CREATE TABLE` are so extensive that it may be the most complex command in any programming language.

CREATE TRIGGER

Triggers are a special form of stored procedures, executed when a data item is modified. Triggers are associated with data items in the following ways:

- Specific tables
- Specific data modification statements such as these:
  - `INSERT`

- UPDATE
- DELETE

- Specific firing times such as these:
  - BEFORE processing a statement
  - AFTER processing a statement
  - INSTEAD OF processing a statement

For example, if you want to track changes of address by your clients, you can create a trigger as follows to save the old data when a change is made to a customer address:

```
CREATE TRIGGER SavePreviousAddress ON CUSTOMERS FOR
INSERT,UPDATE
AS
 INSERT INTO PREVIOUS_ADDRESS
 SELECT * FROM Deleted
```

Like stored procedures, triggers can use variables, defined using the `@VariableName` convention.

Note

The example is written for SQL Server, which uses the keyword `FOR` instead of `BEFORE`. This is yet another indication of the need to read the documentation relevant to the specific DBMS you are using.

CREATE VIEW

`CREATE VIEW` is used to create temporary tables or views. Views are like tables in that they can be queried and updated, but they are created using a query, so they can also be considered a way of saving named queries. This is the basic form of the command:

```
CREATE VIEW ViewName AS
SELECT [* | ColumnList]
FROM TableName
WHERE ...
```

A simple example might look like this:

```
CREATE VIEW ViewCorleones AS
SELECT *
FROM CUSTOMERS
WHERE Last_Name = 'Corleone'
```

There are various restrictions on using views to update the underlying table or tables. Needless to say, the capabilities and implementation of views differ from system to system.

DELETE

The `DELETE` command is used to delete rows from a table. It is supported by all versions of SQL. This is the basic form of the command:

```
DELETE FROM TableName [WHERE ...]
```

Although optional, the `WHERE` clause is used in almost every instance of the `DELETE` command. Without it, the entire content of the table is deleted.

DROP

All objects created with a `CREATE` statement can be destroyed with a `DROP` statement. These are the main variants of the `DROP` command:

```
DROP DATABASE DatabaseName
DROP FUNCTION FunctionName
DROP INDEX IndexName
DROP PROCEDURE ProcedureName
DROP TABLE TableName
DROP TRIGGER TriggerName
DROP VIEW ViewName
```

Caution

You should never DROP a system-vendor-supplied table. These tables are essential to the proper functioning of the system.

GRANT

The GRANT command is used to grant user privileges and roles and is supported by most major DBMS systems. This is the basic syntax of the command:

```
GRANT {
ALL
| CREATE {DATABASE | DEFAULT | FUNCTION | PROCEDURE | TABLE | VIEW }
| SELECT
| INSERT [(ColumnName,...)]
| DELETE
| UPDATE [(ColumnName,...)]
| REFERENCES
ON TableName
| DOMAIN DomainName
| COLLATION CollationName
| CHARACTER SET CharacterSetName
| TRANSLATION
TO { user | PUBLIC }
[WITH GRANT OPTION]
```

As is true of most SQL commands, each RDBMS offers its own variations, depending on the system of privileges implemented. Oracle, for example, supports a wide range of privileges, but PosgreSQL supports far fewer ones.

INSERT

The INSERT command is used to insert data into a table. It is supported with variations by all RDBMS systems. This is the basic syntax of the command:

```
INSERT INTO tableName (colName1, colName2, ...) VALUES (value1, value2,
 ...);
```

You can omit the column-name list if your VALUES list exactly matches the column data types and sizes defined for the table.

The INSERT statement can also incorporate a SELECT statement, allowing you to transfer data from one table to another, as shown here:

```
INSERT INTO TableName1
SELECT (colName1, colName2, ...)
```

```
FROM TableName2
WHERE ...;
```

Note To insert a NULL, simply use the word NULL as the inserted value.

REVOKE

The REVOKE command is used to revoke privileges granted with the GRANT command. The syntax of the REVOKE command is similar to that of the GRANT command, as you can see here:

```
REVOKE {
ALL
| CREATE {DATABASE | DEFAULT | FUNCTION | PROCEDURE | TABLE | VIEW }
| SELECT
| INSERT [(ColumnName,...)]
| DELETE
| UPDATE [(ColumnName,...)]
| REFERENCES
ON TableName
| DOMAIN DomainName | COLLATION CollationName| CHARACTER SET
CharacterSetName | TRANSLATION
FROM { user | PUBLIC } {CASCADE | RESTRICT }
```

The CASCADE option revokes the specified privilege and any privileges dependent upon it, whereas the RESTRICT option revokes only the specified privilege.

ROLLBACK

The ROLLBACK command undoes a transaction back to the previous savepoint or to its beginning if no savepoint is specified. It also closes open cursors and releases locks. This is the syntax of the command:

```
ROLLBACK [WORK] [TO SAVEPOINT SavepointName];
```

SAVEPOINT

The SAVEPOINT command identifies a named savepoint for use in rolling back transactions. This is the basic syntax of the command:

```
SAVEPOINT SavepointName;
```

SAVEPOINT is not directly supported by most versions of SQL, so you should refer to your documentation.

SELECT

One of the most important SQL commands, and one of the most powerful, SELECT is supported by all versions of SQL. The syntax of the basic SELECT statement is straightforward, but it can be extended to create very complex queries, as discussed elsewhere in this book. This is the basic form of the command:

```
SELECT [DISTINCT] column1[,column2]
[ INTO new_table ]
FROM table1[,table2]
```

```
[ WHERE "conditions"]
[ GROUP BY "column-list"]
[ HAVING "conditions]
[ ORDER BY "column-list" [ASC | DESC] ]
```

The SELECT clause specifies the columns the query is to return. It supports a number of variations. Here's an example:

```
SELECT
 [ ALL | DISTINCT ][ TOP n [ PERCENT ]
    { * | { column_name | expression } [ [ AS ] column_alias ]
    }    [ ,...n ]
```

Arguments

Here are explanations of the arguments:

- ALL — Specifies that duplicate rows can be returned. ALL is the default, so it is rarely specified.
- DISTINCT — Specifies that only unique rows be returned
- TOP *n* [PERCENT] — Specifies that only the first *n* rows or *n* percent of rows matching the query are returned. When using PERCENT, *n* must be an integer between 0 and 100. If the query includes an ORDER BY clause, the filter is applied to the ordered result set.

Select list

The select list specifies the columns to be returned in the result set. It is supplied as a series of expressions separated by commas.

- * — Specifies that all columns should be returned. The columns are returned in the order in which they exist in the tables or views specified in the query.
- column_name — specifies the name of a column to return. Some implementations allow you to remove ambiguities by qualifying a column name by prefixing its table name, followed by a period.
- expression — specifies an expression evaluated to obtain the result to be returned.
- column_alias — specifies the name to be returned as the column name in the result set. Aliases are particularly useful for naming calculated columns.

INTO clause

The INTO clause creates a new table and inserts the result set the query returns into the new table. Each column in the new table has the same name, data type, and value as the corresponding expression in the select list. If a computed column is included in the select list, the corresponding column in the new table will contain the values computed at the time SELECT...INTO is executed.

FROM clause

The FROM clause specifies the tables, views or JOINS from which to retrieve rows. This is the basic syntax of the FROM clause:

```
FROM
 table_name [ [ AS ] table_alias ] |
 view_name  [ [ AS ] table_alias ] |
 joined_table
```

A joined table is a result set that is the product of two or more tables. The syntax of a JOIN looks like the following example:

```
SELECT *
FROM tab1e_1
```

```
 [ INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } ] JOIN table_2
ON
 table_1.ID = table_2.ID
```

For a detailed discussion of JOINS, with plenty of examples of the different kinds of JOINS, please refer to Chapter 9.

WHERE clause

The WHERE clause allows you to restrict the rows returned to those matching a specified search condition. This is the basic syntax of the WHERE clause:

```
WHERE search_condition
```

Comparison operators

Search conditions are defined by expressions using comparison operators. SQL supports the following standard comparison operators, as well as a special operator used to test for a NULL value in a column:

- Equality (=)
- Inequality (<>)
- Greater Than (>) and Greater Than or Equal To (>=)
- Less Than (<) and Less Than or Equal To (<=)
- IS NULL
- IS NOT NULL

Some examples of simple comparisons are shown here:

```
SELECT * FROM Inventory WHERE Name = 'Corn Flakes';
SELECT * FROM Inventory WHERE Price >= 2.67;
SELECT * FROM Contact_Info WHERE Cell_Phone IS NOT NULL;
```

In addition to letting you use the comparison operators to work with Strings, SQL provides these dedicated String operators for use with CHAR and VARCHAR variables:

- LIKE
- NOT LIKE

The LIKE operator supports wildcards to provide a very powerful tool for String comparison. The wildcards are as follows:

- Underscore (_), the single-character wild card
- Percent (%), the multicharacter wild card

For example, to find all records in the Contact_Info Table with a last name starting with "C," write a query using LIKE. Here's an example:

```
SELECT * FROM Contact_Info WHERE Last_Name LIKE 'C%';
```

Strings can also be concatenated. For example, to return a concatenated last name, comma (,), and a first name, use this query:

```
SELECT Last_Name + ', ' + First_Name AS NAME FROM Contact_Info;
```

Logical operators

SQL provides these logical operators to combine two or more conditions in the WHERE clause of a SQL statement:

- AND
- OR
- NOT

The AND operator is used to combine two or more comparisons, all of which must evaluate to TRUE for the comparison to be valid, as shown here:

```
SELECT * FROM Contact_Info WHERE Last_Name = 'Corleone' AND City = 'New
York';
```

The OR operator is used to combine two or more comparisons, any one of which can evaluate to TRUE for the comparison to be valid. Here's an example:

```
SELECT * FROM Contact_Info WHERE City = 'New York' OR State = 'NJ';
```

Like arithmetic operators, logical operators can be combined using parentheses (()), as shown here:

```
SELECT * FROM Contact_Info
WHERE Last_Name = 'Corleone' AND ( City = 'New York' OR State = 'NJ' );
```

The NOT operator is used to reverse the result of a comparison as in this example:

```
SELECT * FROM Contact_Info
WHERE Last_Name = 'Corleone' AND NOT ( City = 'New York' OR State = 'NJ' );
```

Arithmetic operators

SQL supports the common arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/). In addition, SQL supports the modulo operator (%), which returns the remainder of the division of one integer by another.

IN and BETWEEN

The IN operator provides a simple way to compare fields against a list. For example, to find contacts in New York State or New Jersey, you can use this query:

```
SELECT *
FROM Contact_Info
WHERE State IN ('NY', 'NJ');
```

The BETWEEN operator selects values between specified inclusive limits, as shown here:

```
SELECT *
FROM Inventory
WHERE Cost BETWEEN 1.03 AND 1.95;
```

Set operators

Set operators allow you to combine ResultSets returned by different queries into a single ResultSet. These are the main set operators:

- UNION returns the combined results of two queries.
- INTERSECT returns only the rows both queries find.
- EXCEPT returns the rows from the first query that are not present in the second.

UNION ALL returns the results of two queries, and UNION does the same thing, but it removes duplicate results. Here's an example:

```
SELECT *
FROM Contact_Info
WHERE Last_Name = 'Corleone' AND (City = 'New York' OR
    State = 'NJ')
UNION
SELECT *
FROM contact_info
WHERE first_name = 'Kay';
```

UNION, used alone, returns the results of the two queries without any repetitions. UNION ALL, on the other hand, returns the results of the two queries, including all repetitions.

Subqueries and the ANY, SOME, ALL, and EXISTS predicates

A *query* is a SQL command that uses the SELECT keyword to return an array of data fields from one or more tables. A *subquery* is simply used as part of another SQL statement.

In many cases, a subquery used in a comparison returns more than one value, so special predicates are required to operate on the results of the subquery before making the comparison.

ANY, SOME and ALL predicates

The ANY or SOME predicates, which are synonymous, can be used to retrieve records in the main query that satisfy the comparison with any records retrieved in the subquery. The following example returns all inventory items with a cost greater than the lowest-cost cookies in the Inventory Table:

```
SELECT * FROM INVENTORY
WHERE cost >= ANY
        (SELECT cost FROM inventory
         WHERE Description = 'Cookies');
```

The ALL predicate can be used to retrieve only those records in the main query that satisfy the comparison with all records retrieved in the subquery. If you change ANY to ALL in the preceding example, the query will return only those inventory items that cost more than all cookies.

EXISTS predicate

The EXISTS and NOT EXISTS predicates are used in true/false comparisons to determine whether the subquery returns any records. Conventionally, you use an asterisk (*) with the EXISTS predicate because EXISTS only returns true or false, so there is nothing to be gained by being more specific. Here's an example:

```
SELECT *
FROM Customers
WHERE EXISTS
        (SELECT *
      FROM Orders
      WHERE Orders.Customer_Number = Customers.Customer_Number)
```

The EXISTS clause stops the search as soon as it finds a single match and is therefore much faster and more efficient than a query that continues to check for additional rows that match.

Nesting subqueries

Just as you can use a subquery within a query, you can use a subquery within another subquery. Subqueries can be nested as deeply as your implementation of SQL allows. The syntax for nesting subqueries looks like this:

```
SELECT *
FROM Tables
WHERE
 ( SUBQUERY
  (SUBQUERY
   (SUBQUERY)));
```

ORDER BY Clause

The ORDER BY clause specifies the sort for the result set. Multiple sort columns can be specified. The sequence of the sort columns in the ORDER BY clause defines the sequence of the sorts. Here's an example:

```
[ ORDER BY { order_by_expression [ ASC | DESC ] }     [ ,...n] ]
```

You can use the ASC and DESC keywords to define sort order:

- ASC – defines sort order to be ascending
- DESC – defines sort order to be descending

The ORDER BY clause can include items not appearing in the select list.

GROUP BY clause

The GROUP BY clause specifies how to group rows for aggregate calculations such as COUNT and AVERAGE. Commonly supported aggregation functions include the following:

- AVG — Average of the values in the numeric expression
- COUNT — Number of selected rows
- MAX — Highest value in the expression
- MIN — Lowest value in the expression
- STDEV — Statistical standard deviation for all values in the expression
- SUM — Total of the values in the numeric expression

When GROUP BY is specified, columns other than aggregate calculations defined in the select list must be included in the GROUP BY list.

HAVING clause

The HAVING clause specifies a search condition for a group or an aggregate. This example illustrates the syntax for the HAVING clause:

```
SELECT State, COUNT(State)
FROM Customers
GROUP BY state
HAVING COUNT(State) > 3
```

SET TRANSACTION

The SET TRANSACTION command is used to set the properties of a transaction, such as the isolation level. This is the syntax of the command:

```
SET TRANSACTION
{ READ_ONLY | READ_WRITE }
ISOLATION LEVEL
{ READ_COMMITTED
| READ_UNCOMMITTED
| REPEATABLE_READ
| SERIALIZABLE}
```

START TRANSACTION

The START TRANSACTION command, often implemented as BEGIN TRANSACTION, is used to set the properties of a transaction, such as the isolation level, and to identify the point at which it starts. The syntax is very similar to SET TRANSACTION, as you can see here:

```
SET TRANSACTION
{ READ_ONLY | READ_WRITE }
ISOLATION LEVEL
{ READ_COMMITTED
| READ_UNCOMMITTED
| REPEATABLE_READ
| SERIALIZABLE}
```

UPDATE

The UPDATE command is used to modify records. Records to be modified are usually selected using a WHERE clause. This is the syntax of the statement:

```
UPDATE { TableName | ViewName }
SET ColumnName = { DEFAULT | expression }
WHERE conditions.
```

Appendix B: Installing Apache and Tomcat

Overview

Apache has been the most popular Web server on the Internet since April 1996. The January 2002 Netcraft Web Server Survey found that nearly 60 percent of the Web sites on the Internet are using Apache, nearly double the number of sites using the second-rated Web server and exceeding all other Web servers combined.

Tomcat is under ongoing development as an open-source project released under the Apache Software License. Like all open-source development projects, Tomcat is intended to be a collaboration of the best-of-breed developers from around the world. Tomcat has been selected as the JSP reference implementation by Sun.

Installing an HTTP Server — Apache

The Apache HTTPD server is powerful, flexible, and HTTP/1.1compliant. It is extremely easy to install on virtually any platform and offers a number of features such as support for virtual hosts. It is available with full source code and comes with an unrestrictive license. Better yet, you can download and use it free of charge. An open-source project, Apache development is driven by a genuine urge to produce a better product.

The original Apache server evolved from the NCSA Web daemon developed at the National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign. Development of that had stalled, and many Web masters had developed their own extensions and bug fixes that were in need of a common distribution. A small group of these Web masters formed the foundation of the original Apache Group. The name originally derived from the fact that Apache was "A PAtCHy server," as it was based on some existing code and a series of "patch files."

Despite the name, Apache has been shown to be substantially faster, more stable, and more feature-full than many other Web servers. According to the Netcraft Web Server Survey (http://www.netcraft.com/survey/) of web server software usage, of over 37,000,000 sites surveyed, more than 21,000,000 run Apache. It has been tested thoroughly by both developers and users on sites that get millions of hits per day, with no performance difficulties.

Apache Features

Apache provides easily used support for a wide range of useful features through its configuration files. Among these features are the following:

- DBM databases for authentication
- Customized responses to errors and problems
- Virtual hosts

DBM databases for authentication

This feature allows you to set up password-protected pages with enormous numbers of authorized users, without bogging down the server.

Customized responses to errors and problems

This feature allows you to set up custom files or even CGI scripts to be returned by the server in response to errors and problems. This is a great help when you need to perform diagnostics during Web-site development

Virtual hosting

Virtual hosting is an extremely useful and much requested feature that allows the server to distinguish between requests made to different IP addresses or names mapped to the same machine. This means you can use the same IP address for multiple domain names and have Apache resolve them to different paths on a single host machine.

Downloading and Installing Apache

You can download the latest version of Apache directly from the Apache Web site at `http://www.apache.org/dist/httpd/`. Apache is also available from a large number of mirror sites, listed at `http://www.apache.org/dyn/closer.cgi`. These sites list the current release as well more recent beta releases and have links to older versions and binary distributions for a variety of platforms.

Windows installation

According to its developers, Apache 1.3 is designed to run on Windows NT 4.0, Service Pack 6, and Windows 2000. In my own experience, it also runs fine on Windows 98; however, for reasons of reliability, I would not use Windows 98 as a production platform. In all cases, TCP/IP networking must be installed.

There are two ways to install Apache under Windows. If you want the source code, you can download the binary build of Apache for Windows named apache `1_3_#-win32-src.msi`. Otherwise, you can simply download apache`_1_3_#-win32-no_src.msi`. Each of these files contains the complete Apache runtime. You must have the Microsoft Installer version 1.10 installed on your PC before you can install the Apache runtime distributions.

Run the Apache `.msi` file you have downloaded according to the information provided at the beginning of this section. Doing so prompts you for various data about your installation, including the following:

- Whether you want to install Apache as a service or to run it in a console window when you choose the Start Apache shortcut
- Your server name, domain name, and administrative e-mail account
- The Apache installation directory. The default is `C:\Program Files\Apache Group\Apache`, but I prefer `C:\Apache` because many programs, including many Windows programs, have trouble handling path names containing spaces.

Caution

Avoid installing anything under `C:\Program Files\` because many programs, including many Windows programs, have trouble handling path names containing spaces.

Apache uses configuration files to maintain configuration information. These configuration files are kept in the `conf` directory. During the installation, Apache installs them in your chosen installation directory path. However, if any of the files in this directory already exist, as might be the case if you are installing an upgrade, they will not be overwritten. Instead, the new copy of the corresponding file is left with the extension `.default.conf`. Similarly, if you already have a file called `htdocs\index.html`, Apache will not overwrite it with the default version of `htdocs\index.html`. (`htdocs\index.html` is the file which is sent to the browser when your server is first accessed, and no other file is specifically requested.) This should mean it is safe to install Apache over an existing installation.

After installing Apache, you can edit the configuration files in the `conf` directory as required. The file you are most likely to want to edit is `httpd.conf`, which defines such features of the server as default file paths and virtual-host setups. However, to get started quickly, the files should work as installed.

Running Apache under Windows

There are two ways you can run Apache:

- From a console window. (If you close the console window, Apache will terminate.)
- As a "service," so that Apache starts automatically when your machine boots and keeps running when you log off.

Using the console window

To run Apache from a console window, select the "Start Apache as console app" option from the Start menu. This opens a console window and starts Apache running inside it. The window remains active until you stop Apache. To stop Apache, press the "Shutdown Apache console app" icon option from the Start menu.

In Apache 1.3.13 and above, you can also stop Apache by pressing Ctrl+C or Ctrl+Break. On Windows NT/2000 with version 1.3.13, Apache stops if you select "Close" from the system menu or click the close button in the top-right corner of the console window. The Close menu item and close (X) button also work on Windows 95/98 as of Apache version 1.3.15.

Starting as a service

To start Apache as a service, it has to be installed as a service. The easiest way to do this is to install the default Apache service when launching the Apache installation package. Once this is done, you can start the "Apache" service by opening the Services window, selecting Apache, and clicking Start. Apache is now running, hidden in the background.

Apache comes with extensive documentation, so you should have no problems installing and starting Apache. I was pleasantly surprised at how smoothly it went the first time I tried it.

Linux installation

Apache was first developed to run on Unix-type operating systems and is available in a wide variety of flavors from `Apache.org`. Figure B-1 displays the Apache download page, giving you an idea of the varieties of binary available. To download a binary, point your browser at Apache.org's download site at `http://www.apache.org/dist/binaries` and download the binary for your operating system.

Figure B-1: Flavors of Apache available for download

Now you need to uncompress the archive using gunzip and tar. You should end up with an apache_1.3.x directory (x is the particular subversion of Apache 1.3 you have downloaded). Move into the following newly created directory:

```
cd apache_1.3.x
```

As of Apache 1.3.11, binary distributions contain an install script. If your binary does not contain an install script, refer to the `README.bindist` and/or `INSTALL.bindist` documents for further information. Run the install script as follows:

```
./install.bindist.sh
```

This command should install the various components of the Apache distribution into the appropriate locations. Usually, the default is to install everything under `/usr/local/apache`.

Running Apache under Linux

Become root, and type the following:

```
/usr/local/apache/bin/apachectl start
```

Point your browser at your server, `http://localhost`, and you should see the Apache default home page.

Installing a JSP Container — Jakarta Tomcat

The Jakarta Tomcat Servlet engine and JSP container has been chosen for these reasons:

- Tomcat is the servlet container used in the official reference Implementation for the Java Servlet and JavaServer Pages technologies.
- Tomcat can be downloaded free from the Jakarta Web site, so, with Tomcat, there is no "barrier to entry" in getting your JSP applications up and running.

Tomcat is developed in an open-source development project released under the Apache Software License. Like all open-source development projects, Tomcat is intended to be a collaboration of the best-of-breed developers from around the world.

Since Tomcat development is an ongoing project, there are several versions of Tomcat available for download at any given time. The Jakarta Tomcat Web site tells you which is the current production version. At the time of writing this, Tomcat 3.2.3 is the current production quality release.

Tomcat 4.0 is the next generation of Tomcat. The 4.0 servlet container (Catalina) has been developed from the ground up for flexibility and performance. Version 4.0 implements the final released versions of the Servlet 2.3 and JSP 1.2 specifications. As required by the specifications, Tomcat 4.0 also supports Web applications built for the Servlet 2.2 and JSP 1.1 specifications with no changes.

Downloading and Installing Tomcat

All code in the book has been tested using Tomcat 3.3.1. You may, however, wish to download the most recent version, which is available at `http://jakarta.apache.org/tomcat/index.html`.

Go to the Downloads page, and find the section headed "Release Builds." To quote the copy on the download page: "Release Builds are those that are ready for Prime Time. This build is "as good as it gets!" Select the highest numbered version of Tomcat. (There may be a legacy version with a lower revision number available for those who need it.) This takes you to a file-index page that looks like Figure B-2.

Figure B-2: Tomcat download directory

If you are running Windows, you need the zip file, in this case: `jakarta-tomcat-3.2.3.zip`. In my experience, the Apache/Tomcat combination works flawlessly under Windows, offering more capabilities and an easier installation than other available products. Linux users will want the `tar.gz` version of the same file.

To unpack the zip file, you need a copy of Winzip or an equivalent decompression utility. Winzip is available for download from `http://www.winzip.com/`. First, install Winzip by clicking the file icon and following the directions. During installation, Winzip associates itself with all Zip format files, so you will see the Winzip icon next to your Tomcat zip file in the download directory.

You also need a copy of the `java jdk`. You can download this from `http://java.sun.com/j2se/`. The `jdk` is packaged as a self-extracting executable, so all you need is to execute it; it prompts you

through the installation. I normally create a `C:\Java` directory and install the `jdk` in a subdirectory with a path like `C:\Java\jdk1.3`.

Installation

To unzip and install Tomcat, simply click on the Tomcat zip file, and Winzip opens a dialog box and prompts you to select the install directory and then guides you through the installation. The actual installation takes place in a subdirectory called `jakarta-tomcat-3.2.3` or whatever the version number of your copy may be. I normally just rename this directory "Tomcat" to avoid the problems frequently encountered with long file names under Windows.

I prefer to install Tomcat in its own directory, either directly under C: or under Apache, since Tomcat and Apache are designed to work well together. Do not install Tomcat under `C:\Program Files\` because Tomcat's script files will not work properly. Frequently, you will find that a program appears to work fine, but after a while you'll find a feature you haven't used before that objects to spaces in directory names and fails to work properly.

Caution

Avoid installing anything under `C:\Program Files\` because many programs, including many Windows programs, have trouble handling path names containing spaces.

Starting and Stopping Tomcat

Tomcat is a Java program, so it is possible to execute it from the command line. However, this involves setting several environment variables. It is easier to use the scripts provided with the Tomcat distribution to start and stop Tomcat.

For the average user, the most important scripts are listed in Table B-1:

Table B-1: Tomcat Scripts

Script	Purpose
tomcat	Sets the environment variables, including `CLASSPATH`, `TOMCAT_HOME` and `JAVA_HOME`, and starts Tomcat with the proper command-line parameters
startup	Starts tomcat in the background
shutdown	Stops tomcat

The most important of these scripts is `(tomcat.sh/tomcat.bat)`. The other scripts serve as a simplified, single-task-oriented entry point to the tomcat script (set different command-line parameters and so on).

Setting Environment Variables

Before you can use these scripts to start Tomcat, you need to set the following environment variables.

`TOMCAT_HOME` points to the root directory of your Tomcat hierarchy, and you should type the following:

- On Win32, type **"set TOMCAT_HOME= c:\apache\tomcat "**
- On UNIX , for bash/sh and tcsh, respectively, type the following:
  - **"TOMCAT_HOME= apache\tomcat ; export TOMCAT_HOME"**
  - **"setenv TOMCAT_HOME apache\tomcat"**

`JAVA_HOME` points to the root directory of your JDK hierarchy, and you should type the following:

- On Win32, type **"set JAVA_HOME= c:\java\jdk1.3"**
- On UNIX, for bash/sh and tcsh, respectively, type the following:
  - **"JAVA_HOME= java\jdk1.3; export TOMCAT_HOME"**
  - **"setenv TOMCAT_HOME java\jdk1.3"**

PATH points to the javac executable, and you should type the following:

- On Win32, type

path=c:\java\jdk1.3;"%PATH%" - don't forget the quotes around %PATH% - Windows will get confused by its own space separated path names without the quotes.

CLASSPATH points to the java classes required by Tomcat and you should type the following:

- On Win32, type

```
SET CLASSPATH=.;C:\apache\tomcat\lib\servlet.jar;
SET CLASSPATH="%CLASSPATH%;C:\apache\tomcat\lib\jasper.jar;
```

The easiest way to do all this is to create your own script file containing these lines:

```
PATH=C:\JAVA\JDK1.3\BIN;"%PATH%"
SET TOMCAT_HOME=C:\APACHE\TOMCAT
SET JAVA_HOME=C:\JAVA\JDK1.3
SET CLASSPATH=.;C:\apache\tomcat\lib\servlet.jar;
SET CLASSPATH="%CLASSPATH%;C:\apache\tomcat\lib\jasper.jar;
STARTUP
```

The final command, "STARTUP", runs Tomcat's own startup.bat script. Save this script as tcstart.bat. Now start Tomcat by typing "tcstart".

Note

The scripts are only a convenient way to start and stop Tomcat. You can modify them to customize the CLASSPATH, environment variables such as PATH and LD_LIBRARY_PATH, and so on, so long as a correct command line is generated for Tomcat.

Test

To check out your installation, open your Tomcat/bin directory, and run the startup script. Tomcat should start running. To check it out, open a browser window, and point it to:

```
http://localhost:8080/
```

You should see the Tomcat default home page. This page has links to a variety of servlet and JSP examples, as well as to the API documentation for servlets and Java Server Pages. Now that you are at this page, you can check out the JSP and servlet examples by clicking the links. The snoop example is particularly worth looking at, since it shows the wealth of information the server receives from the browser.

Notice the port number 8080 after "localhost". As a servlet/JSP container, Tomcat defaults to port 8080 so that a primary server can service the standard HTTP port at 80.

You probably also notice that your browser now points to this:

```
http://localhost:8080/index.html
```

If you look at the directory listing, you will notice that index.html is one of the files in the tomcat\webapps\ROOT directory. This is the default Web directory for Tomcat Web pages.

Apache and Tomcat are frequently installed together, so that Apache can be used to serve static web pages, while Tomcat handles dynamic web pages. The next section shows how to configure Tomcat to work with Apache.

Configuring Tomcat to Work with Apache

The reason for using Apache and Tomcat together is that Apache is the premier Web server, but it doesn't do servlets and JSP pages. Tomcat, on the other hand, is the reference servlet/JSP container, but it doesn't do Web pages as well as Apache. Together, they qualify as the dream team.

The Web server's job is to wait for client HTTP requests and, when these requests arrive, to do whatever is needed to serve the requests by providing the necessary content. Adding a servlet container changes this behavior, adding the need to perform these tasks:

- Load the servlet container adapter library and initialize it prior to serving requests.
- Check and see if a certain request belongs to a servlet, and, if so, let Tomcat take the request and handle it.

Tomcat needs to know what requests it is going to serve, usually based on some pattern in the request URL, and where to direct these requests.

First, make sure that you can run Tomcat 3.x as a standalone, and be able to run the servlets and JSPs from the examples. Now you can edit the various configuration files.

The following examples reflect my configuration, which uses an Apache directory under C, with the Apache server and Tomcat in subdirectories:

- Apache in `c:/apache/apache/`
- Tomcat in `c:/apache/tomcat/`

You should modify the paths to reflect your own configuration.

When Tomcat starts up, it automatically generates a configuration file for Apache in the path:

```
TOMCAT_HOME/conf/jserv/tomcat-apache.conf
```

You need to edit the apache configuration files so Apache can find the file. To do so, edit `httpd.conf`. Then add the following code to the end of the file:

```
Include c:/apache/tomcat/conf/tomcat.conf
```

This adds your Tomcat configuration to Apache.

Next, you need to copy the jserv module to the Apache `libexec` directory (for Win32, the `modules` directory) and restart Apache. The jserv module acts as a Web server adapter to sit in Apache and redirect requests to Tomcat. It should now be able to connect to Tomcat.

Note

You should refer to the documentation provided with the versions of Apache and Tomcat you download, since there may be some changes as newer revisions are released. You will also find troubleshooting information in the documentation.

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y–Z

List of Figures

Chapter 1: Relational Databases

Chapter 2: Designing a Database

Chapter 3: SQL Basics

Chapter 4: Introduction to JDBC

Chapter 5: Creating a Table withJDBC and SQL

Chapter 6: Inserting, Updating,and Deleting Data

Chapter 7: Retrieving Data withSQL Queries

Chapter 8: Organizing Search Results and Using Indexes

Chapter 9: Joins and Compound Queries

Chapter 10: Building a Client/Server Application

Chapter 11: Building a Membership Web Site

Chapter 12: Using JDBC DataSources with Servlets and Java Server Pages

Chapter 13: Using PreparedStatements and CallableStatements

Chapter 14: Using Blobs and Clobs to Manage Images and Documents

Chapter 15: Using JSPs, XSL, and Scrollable ResultSets to Display Data

Chapter 17: The XML Document Object Model and JDBC

Chapter 18: Using Rowsets to Display Data

Chapter 20: Enterprise JavaBeans

Chapter 21: Bean-Managed Persistence

Chapter 22: Container-Managed Persistence

Chapter 23: Java Data Objects and Transparent Persistence

Chapter 25: Tuning for Performance

Appendix B: Installing Apache and Tomcat

List of Tables

Chapter 1: Relational Databases

Chapter 2: Designing a Database

Chapter 3: SQL Basics

Chapter 4: Introduction to JDBC

Chapter 5: Creating a Table withJDBC and SQL

Chapter 7: Retrieving Data withSQL Queries

List of Listings

Chapter 2: Designing a Database

Chapter 3: SQL Basics

Chapter 4: Introduction to JDBC

Chapter 5: Creating a Table withJDBC and SQL

Chapter 6: Inserting, Updating,and Deleting Data

Chapter 13: Using PreparedStatements and CallableStatements

Chapter 14: Using Blobs and Clobs to Manage Images and Documents

Chapter 15: Using JSPs, XSL, and Scrollable ResultSets to Display Data

Chapter 16: Using the JavaMail API with JDBC

Chapter 17: The XML Document Object Model and JDBC

Chapter 18: Using Rowsets to Display Data

Chapter 19: Accessing XML Documents Using SQL

Chapter 20: Enterprise JavaBeans

List of Sidebars

Chapter 3: SQL Basics

Chapter 9: Joins and Compound Queries