Java Database Programming with JDBC
(Publisher: The Coriolis Group)
Author(s): Pratik Patel
ISBN: 1576100561
Publication Date: 10/01/96

Search this book:

Go!

Chapter 1
JDBC: Databases The Java Way!

The Internet has spurred the invention of several new technologies in client/server computing—the most recent of which is Java. Java is two-dimensional: It's a programming language and also a client/server system in which programs are automatically downloaded and run on the local machine (instead of the server machine). The wide embrace of Java has prompted its quick development. Java includes Java compilers, interpreters, tools, libraries, and integrated development environments (IDEs). Javasoft is leading the way in the development of libraries to extend the functionality and usability of Java as a serious platform for creating applications. One of these libraries, called Application Programming Interfaces (APIs), is the Java Database Connectivity API, or JDBC. Its primary purpose is to intimately tie connectivity to databases with the Java language.

We'll discuss the reasoning behind the JDBC in this chapter, as well as the design of the JDBC and its associated API. The Internet, or better yet, the technologies used in the operation of the Internet, are tied into the design of the JDBC. The other dominant design basis for the JDBC is the database standard known as SQL. Hence, the JDBC is a fusion of three discrete computer areas: Java, Internet technology, and SQL. With the growing implementation of these Internet technologies in "closed" networks, called *intranets*, the time was right for the development of Java-based enterprise APIs. In this book, intranet and Internet are both used to describe the software technology behind the network, such as the World Wide Web.

What Is The JDBC?

As I mentioned a moment ago, JDBC stands for Java Database Connectivity. What is this JDBC besides a nifty acronym? It refers to several things, depending on context:

- It's a specification for using data sources in Java applets and applications.
- It's an API for using low-level JDBC drivers.

- It's an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.
- It's based on the X/Open SQL Call Level Interface (CLI) that defines how client/server interactions are implemented for database systems.

Confused yet? It's really quite simple: The JDBC defines every aspect of making data-aware Java applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface. This interface is used by the developer so he doesn't need to worry about the database-specific syntax when connecting to and querying different databases. The JDBC is a package, much like other Java packages such as java.awt. It's not currently a part of the standard Java Developer's Kit (JDK) distribution, but it is slated to be included as a standard part of the general Java API as the java.sql package. Soon after its official incorporation into the JDK and Java API, it will also become a standard package in Java-enabled Web browsers, though there is no definite timeframe for this inclusion. The exciting aspect of the JDBC is that the drivers necessary for connection to their respective databases do not require any pre-installation on the clients: A JDBC driver can be downloaded along with an applet!

The JDBC project was started in January of 1996, and the specification was frozen in June of 1996. Javasoft sought the input of industry database vendors so that the JDBC would be as widely accepted as possible when it was ready for release. And, as you can see from this list of vendors who have already endorsed the JDBC, it's sure to be widely accepted by the software industry:

- Borland International, Inc.
- Bulletproof
- Cyber SQL Corporation
- DataRamp
- Dharma Systems, Inc.
- Gupta Corporation
- IBM's Database 2 (DB2)
- Imaginary (mSQL)
- Informix Software, Inc.
- Intersoft
- Intersolv
- Object Design, Inc.
- Open Horizon
- OpenLink Software
- Oracle Corporation
- Persistence Software
- Presence Information Design
- PRO-C, Inc.
- Recital Corporation
- RogueWave Software, Inc.

- SAS Institute, Inc. ™
- SCO
- Sybase, Inc.
- Symantec
- Thunderstone
- Visigenic Software, Inc.
- WebLogic, Inc.
- XDB Systems, Inc.

The JDBC is heavily based on the ANSI SQL-92 standard, which specifies that a JDBC driver should be SQL-92 entry-level compliant to be considered a 100 percent JDBC-compliant driver. This is not to say that a JDBC driver has to be written for an SQL-92 database; a JDBC driver can be written for a legacy database system and still function perfectly. As a matter of fact, the simple JDBC driver included with this book uses delimited text files to store table data. Even though the driver does not implement every single SQL-92 function, it is still a JDBC driver. This flexibility will be a major selling point for developers who are bound to legacy database systems but who still want to extend their client applications.

The JDBC Structure

As I mentioned at the beginning of this chapter, the JDBC is two-dimensional. The reasoning for the split is to separate the low-level programming from the high-level application interface. The low-level programming is the JDBC driver. The idea is that database vendors and third-party software vendors will supply pre-built drivers for connecting to different databases. JDBC drivers are quite flexible: They can be local data sources or remote database servers. The implementation of the actual connection to the data source/database is left entirely to the JDBC driver.

The structure of the JDBC includes these key concepts:

- The goal of the JDBC is a DBMS independent interface, a "generic SQL database access framework," and a uniform interface to different data sources.
- The programmer writes only *one* database interface; using JDBC, the program can access any data source without recoding.

Figure 1.1 shows the architecture of the JDBC. The **DriverManager** class is used to open a connection to a database via a JDBC driver, which must register with the **DriverManager** before the connection can be formed. When a connection is attempted, the **DriverManager** chooses from a given list of available drivers to suit the explict type of database connection. After a connection is formed, the calls to query and fetch results are made directly with the JDBC driver. The JDBC driver must implement the classes to process these functions for the specific database, but the rigid specification of the JDBC ensures that the drivers will perform as expected. Essentially, the developer who has JDBC drivers for a certain database does not need to worry about changing the code for the Java program if a different type of database is used (assuming that the JDBC driver for the other database is available). This is especially useful in the scenario of distributed databases.

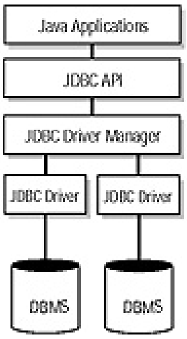


Figure 1.1 The architecture of the JDBC.

The JDBC uses a URL syntax for specifying a database. For example, a connection to a mSQL database, which was used to develop some of the Java applets in this book, is:

```
jdbc:msql://mydatabase.server.com:1112/testdb
```

This statement specifies the transport to use (jdbc), the database type (msql), the server name, the port (1112), and the database to connect to (testdb). We'll discuss specifying a database more thoroughly in Chapter 3.

The data types in SQL are mapped into native Java types whenever possible. When a native type is not present in Java, a class is available for retrieving data of that type. Consider, for example, the **Date** type in the JDBC. A developer can assign a date field in a database to a JDBC **Date** class, after which the developer can use the methods in the **Date** class to display or perform operations. The JDBC also includes support for binary large objects, or BLOB data types; you can retreive and store images, sound, documents, and other binary data in a database with the JDBC. In Chapter 6, we'll cover the SQL data types and their mapping into Java/JDBC, as well object-relational mapping.

ODBC's Part In The JDBC

The JDBC and ODBC share a common parent: Both are based on the same X/OPEN call level interface for SQL. Though there are JDBC drivers emerging for many databases, you can write database-aware Java programs using existing ODBC drivers. In fact, Javasoft and Intersolv have written a JDBC driver—the JDBC-ODBC Bridge—that allows developers to use exisiting ODBC drivers in Java programs. Figure 1.2 shows the place of the JDBC-ODBC Bridge in the overall architecture of the JDBC. However, the JDBC-ODBC Bridge requires pre-installation on the client, or wherever the Java program is actually running, because the Bridge must make native method calls to do the translation from ODBC to JDBC. This pre-installation issue is also true for JDBC drivers that use native methods. Only 100 percent Java JDBC drivers can be downloaded across a network with a Java applet, thus requiring no pre-installation of the driver.

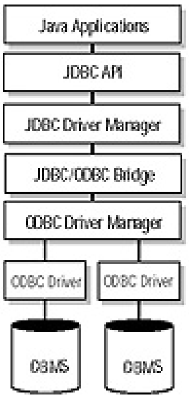


Figure 1.2 ODBC in the JDBC model.

ODBC drivers function in the same manner as “true” JDBC drivers; in fact, the JDBC-ODBC bridge is actually a sophisticated JDBC driver that does low-level translation to and from ODBC. When the JDBC driver for a certain database becomes available, you can easily switch from the ODBC driver to the new JDBC driver with few, if any, changes to the code of the Java program.

Summary

The JDBC is not only a specification for using data sources in Java applets and applications, but it also allows you to create and use low-level drivers to connect and “talk” with data sources. You have now explored the JDBC architecture and seen how the ODBC fits into the picture. The important concept to remember about the JDBC is that the modular design of the JDBC interface allows you to change between drivers—hence databases—without recoding your Java programs.

In the next chapter, we’ll take a step back to give you a quick primer on SQL, one of the pillars of the JDBC. If you are already familiar with SQL-92, feel free to skip the chapter. However, I think that you may find the chapter helpful in clarifying the SQL queries performed in the sample JDBC programs we develop in this book.

Chapter 2
SQL 101

SQL—the language of database. This chapter’s primary purpose is to serve as a primer on this data sublanguage. Although it would be impossible for me to cover the intricacies of SQL in just one chapter, I do intend to give you a solid introduction that we’ll build on in the remainder of this book. Because the JDBC requires that drivers support the ANSI SQL-92 standard to be “JDBC compliant,” I’ll be basing this chapter on that standard. SQL-92, which I’ll refer to as SQL, is based on the relational model of database management proposed in 1970 by Dr. E.F. Codd; over time, SQL evolved into the full-featured language it is today, and it continues to evolve with our ever-changing needs.

A JDBC driver doesn’t absolutely *have* to be SQL-92 compliant. The JDBC specification states the following: “In order to pass JDBC compliance tests and to be called ‘JDBC compliant, we require that a driver support at least ANSI SQL-92 Entry Level.” This requirement is clearly not possible with drivers for legacy database management systems (DBMS). The driver in these cases will not implement all of the functions of a “compliant” driver. In Chapter 10, *Writing JDBC Drivers*, we develop the basics of a JDBC driver that implements only some of the features of SQL, but is a JDBC driver nonetheless.

We’ll start our exploration of SQL by discussing the relational model, the basis for SQL. Then we’ll cover the essentials of building data tables using SQL. Finally, we’ll go into the manipulation and extraction of the data from a datasource.

The Relational Model And SQL

Although SQL is based on the relational model, it is not a rigid implementation of it. In this section, we’ll discuss the relational model as it pertains to SQL so we do not obfuscate our discussion of this standard, which is central to the JDBC specification. As part of its specification, the SQL-92 standard includes the definition of data types. We’ll cover these data types, and how to map to Java, in Chapter 6, *SQL Data Types in Java and the ORM.*

Understanding The Basics

The basic units in SQL are tables, columns, and rows. So where does the “relational” model fit into the SQL units? Strictly speaking, in terms of the relation model, the “relation” is mapped in the table: It provides a way to relate the data contained within the table in a simple manner. A column represents a data element present in a table, while a row represents an instance of a record, or entry, in a table. Each row contains one specific value for each of the columns; a value can be blank or undefined and still be considered valid. The table can be visualized, you guessed it, as a matrix, with the columns being the vertical fields and the rows being the horizontal fields. Figure 2.1 shows an example table that can be used to store information about a company’s employees.

Employee Number	Last Name	First Name	Function
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

Figure 2.1 An SQL table.

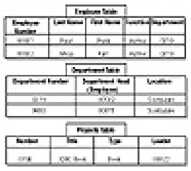
Before we push on, there are some syntax rules you need to be aware of:

- SQL is not whitespace sensitive. Carriage returns, tabs, and spaces don’t have any special meaning when executing queries. Keywords and tokens are delimited by commas, when applicable, and parentheses are used for grouping.
- When performing multiple queries at one time, you must use semicolons to separate distinct queries.
- Queries are *not* case sensitive.

A word of caution: While the keywords are not case sensitive, the string values that are stored as data in a table do preserve case, as you would expect. Keep this in mind when doing string comparisons in queries.

Putting It Into Perspective: Schema And Catalog

Though you can stick all of your data into a single table, it doesn't make sense logically to do this all the time. For example, in our EMPLOYEE table shown previously, we could add information about company departments; however, the purpose of the EMPLOYEE table is to store data on the employees. The solution is for us to create another table, called DEPARTMENT, which will contain information about the specific departments in the company. To associate an employee with a department, we can simply add a column to the EMPLOYEE table that contains the department name or number. Now that we have employees and departments neatly contained, we can add another table, called PROJECT, to keep track of the projects each employee is involved in. Figure 2.2 shows our tables.



Employee Table

Employee Number	Last Name	First Name	Location	Department
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Department Table

Department Number	Department Head (Employee)	Location
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Projects Table

Number	Title	Type	Leader
[illegible]	[illegible]	[illegible]	[illegible]

Figure 2.2 The EMPLOYEE, DEPARTMENT, and PROJECT tables track employees by department and project.

Now that you understand how to logically separate your data, it's time to take our model one step higher and introduce you to the schema/catalog relationship. The *schema* is a higher-level container that is defined as a collection of zero or more tables, where a table belongs to exactly one schema. In the same way, a *catalog* can contain zero or more schemas. This abstract is a necessary part of a robust relational database management system (RDBMS). The primary reason is access control: It facilitates who can read a table, who can change a table, and even who can create or destroy tables. Figure 2.3 demonstrates this point nicely. Here we have added another table, called CONFIDENTIAL. It contains the home address, home phone number, and salary of each employee. This information needs to belong in a separate schema so that anyone who is not in payroll cannot access the data, while allowing those in marketing to get the necessary data to do their job.

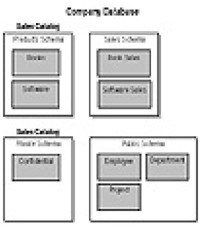


Figure 2.3 The table, schema, and catalog relationship allows you to limit access to confidential information.

Introducing Keys

As you can see in the previous example, we have purposely set up the three tables to link to one another. The EMPLOYEE table contains a column that has the department number that the employee belongs in. This department number also appears in the DEPARTMENT table, which describes each department in the company. The EMPLOYEE and CONFIDENTIAL tables are related, but we still need to add one corresponding entry (row) in one table for each entry in the other, the distinction coming from the employee's number.

The link—employee number and department number—we have set up can be thought of as a *key*. A key is used to identify information within a table. Each individual employee or department should have a unique key to aid in various functions performed on the tables. In keeping with the relational model, the key is supposed to be unique within the table: No other entry in the table may have the same primary key.

A single column is sometimes enough to uniquely identify a row, or entry. However, a combination of rows can be used to compose a *primary key*—for example, we might want to just use the combination of the title and city location of a department to comprise the primary key. In SQL, columns defined as primary keys must be defined. They cannot be "undefined" (also known as NULL).

Using Multiple Tables And Foreign Keys

As we have shown, it's best to split data into tables so that the data contained within a table is logically associated. Oftentimes, the data will belong logically in more than one table, as is the case of the employee number in the EMPLOYEE and CONFIDENTIAL tables. We can further define that if a row in one table exists, a corresponding row must exist in another table; that is, we can say that if there is an entry in the EMPLOYEE table, there must be a corresponding entry in the CONFIDENTIAL table. We can solidify this association with the use of *foreign keys*, where a specific column in the dependent table matches a column in a "parent" table. In essence, we are linking a "virtual" column in one table to a "real" column in another table. In our example database, we link the CONFIDENTIAL table's employee number column to the employee number column in the EMPLOYEE table. We are also specifying that the employee number is a key in the CONFIDENTIAL table (hence the term foreign key). A composite primary key can contain a foreign key if necessary.

We can create a logical structure to our data using the concept of a foreign key. However, in preparation, you'll have to put quite a bit of thought into creating your set of tables; an efficient and planned structure to the data by way of the tables and keys requires good knowledge of the data that is to be modeled. Unfortunately, a full discussion on the techniques of the subject is beyond the scope of this book. There are several different ways to efficiently model data; Figure 2.4 shows one visualization of the database we have created. The SQL queries we perform in the examples of this book are not very complex, so the information outlined in this section should suffice to convey a basic understanding of the example databases created throughout the following chapters.

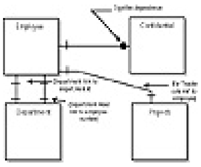


Figure 2.4 E-R diagram of relationships between tables.

Data Definition Language

Now that we have outlined the basic foundation of SQL, let's write some code to implement our database. The formal name for the language components used to create tables is Data Definition Language, or DDL. The DDL is also used to drop tables and perform a variety of other functions, such as adding and deleting rows (entries) from a table, and adding and deleting columns from a table. I'll show you some of these along the way.

Declaring Domains

One of the handy shortcuts that the DDL offers is a way to create predefined data objects. Though we haven't really talked about the data types available in SQL, you can probably guess the common ones like integer, character, decimal (floating point), date, etc. Domains allow you to declare a data type of specific length and then give the declared type a name. This can come in handy if you have numerous data columns that are of the same data type and characteristics. Here's the SQL statement you use to declare a domain:

```
CREATE DOMAIN EMP_NUMBER AS CHAR(5)
```

Tip: Smart domain declaration habits.
When you are actually creating or altering tables, this domain can be used instead of specifying CHAR(20) each time. There are a number of reasons why this is good practice. Notice that we chose to make EMP_NUMBER a domain. This is a column that appears in several tables.

If we mistakenly use the wrong type or length in one of the table definitions where we have employee numbers, it could cause havoc when running SQL queries. You'll have to keep reading to find out the other reason.

Performing Checks

Predefining a data object is also useful for making sure that a certain entry in a column matches the data we expect to find there. For example, our **empno** field should contain a number. If it doesn't, performing a *check* of that data will alert us to the error. These checks can exist in the actual table definition, but it's efficient to localize a check in a domain. Hence, we can add a check to our employee number domain:

```
CREATE DOMAIN EMP_NUMBER AS CHAR(5) CHECK (VALUE IS NOT NULL);
```

Now our domain automatically checks for any null entries in columns defined as EMP_NUMBER. This statement avoids problems that crop up from non-existent entries, as well as allowing us to catch any rogue SQL queries that add an incorrect (those that do not set the employee number) entry to the table.

Creating Tables

Creating a table in SQL is really pretty easy. The one thing you need to keep in mind is that you should define the referenced table, in this case EMPLOYEE, before defining the referencing table, CONFIDENTIAL. The following code creates the EMPLOYEE table shown in Figure 2.2:

```
CREATE TABLE EMPLOYEE
(
empno                       CHAR(5) PRIMARY KEY,
lastname                    VARCHAR(20) NOT NULL,
firstname                   VARCHAR(20) NOT NULL,
function                    VARCHAR(20) NOT NULL,
department                  VARCHAR(20)
);
```

We also could have easily incorporated the domain that we defined earlier into the creation of the table, as shown here:

```
CREATE DOMAIN EMP_NUMBER AS CHAR(5) CHECK (VALUE IS NOT NULL);

CREATE TABLE EMPLOYEE
(
empno                       EMP_NUMBER PRIMARY KEY,
lastname                    VARCHAR(20) NOT NULL,
firstname                   VARCHAR(20) NOT NULL,
function                    VARCHAR(20) NOT NULL,
department                  VARCHAR(20)
);
```

I can hear you now, “What’s this VARCHAR data type?” SQL has two defined string types: CHAR and VARCHAR. The RDBMS allocates exactly the amount of space you specify when you use a CHAR data type; when you set an entry that is defined as a CHAR(N) to a string smaller than the size of N, the remaining number of characters is set to be blank. On the other hand, VARCHAR simply stores the exact string entered; the size you have specified is strictly a limit on how big the entered value can be.

We also see the NOT NULL directive again, which institutes the check on the specific column entry. We discussed primary and foreign keys earlier, now let’s see how we actually implement them. Note that you should define the referenced table before defining the referencing table.

Now it’s time to create the CONFIDENTIAL table. This table uses the empno attribute of the EMPLOYEE table as its primary key, via the REFERENCES keyword.

```
CREATE DOMAIN EMP_NUMBER AS CHAR(5) CHECK (VALUE IS NOT NULL);

CREATE TABLE CONFIDENTIAL
(
empno                EMP_NUMBER PRIMARY KEY,
homeaddress          VARCHAR(50),
homephone            VARCHAR(12),
salary               DECIMAL,
FOREIGN KEY ( empno ) REFERENCES EMPLOYEE ( empno )
)
```

We have tied the empno field in the CONFIDENTIAL table to the empno field in the EMPLOYEE table. The fact that we used the same name, empno, is a matter of choice rather than a matter of syntax. We could have named the empno field whatever we wanted in the CONFIDENTIAL table, but we would need to change the first field referred to in the FOREIGN KEY declaration accordingly.

Manipulating Tables

Database management often requires you to make minor modifications to tables. However, careful planning can help you keep these alterations to a minimum. Let's begin by dropping, or removing, a table from a database:

```
DROP TABLE EMPLOYEE;
```

This is all we have to do to remove the EMPLOYEE table from our database. However, if the table is referenced by another table, as is the case with the CONFIDENTIAL table, a RDBMS may not allow this operation to occur. In this situation, you would have to drop any referencing tables first, and then rebuild them without the referencing.

Altering a table definition is as straightforward as dropping a table. To remove a column from a table, issue a command like this:

```
ALTER TABLE EMPLOYEE
DROP firstname;
```

Of course, if this column is part of the table's key, you won't be able to remove it. Also, if the column is referenced by another table, or there is another column in any table that is dependent on this column, the operation is not allowed.

To add a column to a table, run a query like this:

```
ALTER TABLE CONFIDENTIAL
ADD dateofbirth DATE NOT NULL;
```

You can also make multiple "alterations" at one time with the ALTER clause.

Data Maintenance Language

The subset of commands for adding, removing, and changing the data contained in tables is the Data Maintenance Language (DML). As pointed out earlier, the data is manifest in the form of rows. So, basically, DML performs row-based operations. Let's see how this works by inserting an entry (row) in the EMPLOYEE table:

```
INSERT INTO EMPLOYEE
VALUES (
'00201',
'Pratik',
'Patel',
'Author',
''
);
```

Here we have inserted the appropriate information *in the correct order* into the EMPLOYEE table. To be safe, you can specify which field each of the listed tokens goes into:

```
INSERT INTO EMPLOYEE (empno, lastname, firstname, function, department)
VALUES (
'00201', 'Pratik', 'Patel',  'Author', ''
);
```

If you don't want to add all the fields in the row, you can specify only the fields you wish to add:

```
INSERT INTO EMPLOYEE (empno, lastname, firstname, function)
VALUES (
'00201', 'Pratik', 'Patel',  'Author'
);
```

As you can see, I chose not to add anything in the department field. Note that if a field's check constraint is not met, or a table check is not met, an error will be produced. For example, if we did not add something under the firstname field, an error would have been returned because we defined the table's firstname column check as NOT NULL. We did not set up a check for the department field, so the previous command would not produce an error.

To delete a table's contents without removing the table completely, you can run a command like this:

```
DELETE FROM EMPLOYEE;
```

This statement will wipe the table clean, leaving no data in any of the columns, and, essentially, deleting all of the rows in the table. Deleting a single entry requires that you specify some criteria for deletion:

```
DELETE FROM EMPLOYEE
WHERE empno='00201';
```

You can delete multiple rows with this type of operation, as well. If the WHERE clause matches more than one row, all of the rows will be deleted. You can also delete multiple entries by using the SELECT command in the WHERE clause; we will get to the SELECT command in the next section.

If you really want to get fancy, you can use one statement to delete the same row from more than one table:

```
DELETE FROM EMPLOYEE, CONFIDENTIAL
WHERE empno='00201';
```

The final command I want to cover in this section is UPDATE. This command allows you to change one or more existing fields in a row. Here is a simple example of how to change the firstname field in the EMPLOYEE table:

```
UPDATE EMPLOYEE
SET firstname = 'PR'
WHERE empno='00201';
```

We can set more than one field, if we wish, by adding more expressions, separated by commas, like this:

```
UPDATE EMPLOYEE
SET firstname='PR', function='Writer'
WHERE empno='00201';
```

As you'll see in the next section, the WHERE clause can take the form of a SELECT query so that you can change multiple rows according to certain criteria.

Data Query Language

You have seen how to create your tables and add data to them, now let's see how to retrieve data from them. The SQL commands that you use to retrieve data from a table are part of the Data Query Language (DQL). DQL's primary command is SELECT, but there are a host of predicates you can use to enhance SELECT's flexibility and specificity. Oftentimes, the key to understanding the process of querying is to think in terms of mathematical sets. SQL, like all fourth-generation languages, is designed to pose the question, "What do I want?" as opposed to other computer languages, like Java and C++, which pose the question, "How do I do it?"

Let's look at a set representation of our example database as shown in Figure 2.3. When making queries, you'll want to ask these questions:

- Where is the data located in terms of the table?
- What are the references?
- How can I use them to specify what I want?

Mastering SQL querying is not an easy task, but with the proper mind set, it is intuitive and efficient, thanks to the relational model upon which SQL is based.

The syntax of the SELECT statement is shown here:

```
SELECT column_names
FROM table_names
WHERE predicates
```

Let's take a look at the various functions of the SELECT command. To retrieve a complete table, run this query:

```
SELECT * FROM EMPLOYEE;
```

To get a list of employees in the Editorial department, run this query:

```
SELECT * FROM EMPLOYEE
WHERE department = 'Editorial';
```

To sort the list based on the employees' last names, use the ORDER BY directive:

```
SELECT * FROM EMPLOYEE
WHERE department= 'Editorial'
ORDER BY lastname;
```

To get this ordered list but only see the employee number, enter the following statements:

```
SELECT empno FROM EMPLOYEE
```

```
WHERE department = 'Editorial'
ORDER BY lastname;
```

To get a list of users with the name Pratik Patel, you would enter:

```
SELECT * FROM EMPLOYEE
WHERE (firstname='Pratik') AND (lastname='Patel');
```

What if we want to show two tables at once? No problem, as shown here:

```
SELECT EMPLOYEE.*, CONFIDENTIAL.*
FROM EMPLOYEE, CONFIDENTIAL;
```

Here's a more challenging query: Show the salary for employees in the Editorial department. According to our tables, the salary information is in the CONFIDENTIAL table, and the department in which an employee belongs is in the EMPLOYEE table. How do we associate a comparison in one table to another? Since we used the reference of the employee number in the CONFIDENTIAL table from the EMPLOYEE table, we can specify the employees that match a specified department, and then use the resulting employee number to retrieve the salary information from the CONFIDENTIAL table:

```
SELECT c.salary
FROM EMPLOYEE as e, CONFIDENTIAL as c
WHERE e.department = 'Editorial'
           AND c.empno = e.empno;
```

We have declared something like a variable using the **as** keyword. We can now reference the specific fields in the table using a ".", just like an object. Let's begin by determining which people in the entire company are making more than $25,000:

```
SELECT salary
FROM CONFIDENTIAL
WHERE salary > 25000;
```

Now let's see who in the Editorial department is making more than $25,000:

```
SELECT c.salary
FROM EMPLOYEE as e, CONFIDENTIAL as c
WHERE e.department = 'Editorial'
           AND c.empno = e.empno
           AND c.salary > 25000;
```

You can perform a number of other functions in SQL, including averages. Here's how to get the average salary of the people in the Editorial department:

```
SELECT AVG (c.salary)
FROM EMPLOYEE as e, CONFIDENTIAL as c
WHERE e.department = 'Editorial'
           AND c.empno = e.empno;
```

Of course, the possibilities with SQL exceed the relatively few examples shown in this chapter. Because this book's goal is to introduce the JDBC specifically, I didn't use complex queries in the examples. And now our discussion on SQL is complete. If you are interested in learning more about SQL, I recommend that you check out our book's Website, where I have posted a list of recommended books on the topic of SQL and distributed databases.

Coming Up Next

The next chapter begins our journey into JDBC. I'll show you how to use JDBC drivers for connecting to data sources. Then we'll cover installing drivers, as well as the proper way to use drivers that are dynamically fetched with an applet. Finally, we'll discuss the security restrictions of using directly downloaded drivers as opposed to locally installed drivers.

Chapter 3
Using JDBC Drivers

As a developer who's using the JDBC, one of the first things you need to understand is how to use JDBC drivers and the JDBC API to connect to a data source. This chapter outlines the steps necessary for you to begin that process. We'll be covering the details of getting JDBC drivers to work, as well as the driver registration process we touched on in Chapter 1. We'll also take some time to explore JavaSoft's JDBC-ODBC Bridge, which allows your Java programs to use ODBC drivers to call ODBC data sources.

Before our discussion gets underway though, I need to point out a few things about JDBC drivers. First, there are no drivers packaged with the JDBC API; you must get them yourself from software vendors. Check out this book's Web site for links to demo versions of drivers for your favorite database server, as well as free JDBC drivers available on the Internet. Second, if you want to use ODBC, don't forget that you'll need ODBC drivers, as well. If you don't have a database server, but you want to use JDBC, don't despair: You can use the ODBC drivers packaged with Microsoft Access. Using the JDBC-ODBC Bridge, you can write Java applications that can interact with an Access database.

Unfortunately, applets enforce a security restriction that does not allow access to the local disk, so ODBC drivers might *not* work in the applet context (inside a Web browser). A future release of the Java Development Kit (JDK) may change or relax this security restriction. A workaround for Java-enabled Web browsers is being prepared, and by the time you read this, it may very well be possible to use the JDBC-ODBC bridge. Using ODBC drivers in Java programs also requires pre-installation of the ODBC drivers and JDBC-ODBC Bridge on the client machine. In contrast, JDBC drivers that are 100 percent Java class files can be downloaded dynamically over the network, along with the calling applet's class file. I'll provide a more thorough discussion of this point in Chapter 9.

Quick Start Guide

So you're a regular Java hacker, and you've already figured out how to install the JDBC API package. Now you want to jump right into it. This section will outline the four basic steps for running your first query and getting the results. The steps are explained in greater detail in Chapter 4. Figure 3.1 is a diagram relating the four classes that you'll call on in your JDBC Java program, and it is the skeleton around which you can build database-aware Java programs. The diagram does not list all of the methods available in the respective classes. See Chapter 12, the JDBC API reference, for the complete class and method list.

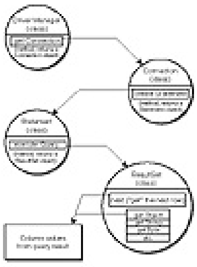


Figure 3.1 The JDBC classes to call.

The following (Listing 3.1) is a very simple JDBC application that follows these four steps. It runs a query and gets one row from the returned result. If you don't understand everything going on here, don't worry—it's all explained in detail in Chapter 4.

Listing 3.1 Example JDBC application.

```
import  java.net.URL;
import  java.sql.*;

class Select {
    public static void main(String argv[]) {
     try {
          new imaginary.sql.iMsqlDriver();
          String url = "jdbc:msql://elanor.oit.unc.edu:1112/bcancer";
         Connection con = DriverManager.getConnection(url, "prpatel",
"");
          Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM Users");
            System.out.println("Got results:");
          while(rs.next()) {

                                   String UID=  rs.getString(1);
                                   String Password=  rs.getString(2);
                                   String Last=  rs.getString(3);
                                   String First= rs.getString(4);
                                   String OfficeID= rs.getString(5);

                                   System.out.print(UID +" "+ Password+"
                                   "+Last+" "+First+" "+OfficeID );
                                   System.out.print("\n");


         }
          stmt.close();
          con.close();
      }
       catch( Exception e ) {
           e.printStackTrace();
      }
   }
}
```

Installing java.sql.*

The java.sql.* package contains the JDBC base API classes, which are supposed to be

Installing java.sql.*

The java.sql.* package contains the JDBC base API classes, which are supposed to be in the normal java.* hierachy that is distributed as part of the Java API (which includes the java.awt, java.io, and java.lang packages). Currently, the JDBC API is not distributed with the JDK, but it is slated to be included in the next release. I have a sneaking suspicion that the java.sql.* package will also be included in the future APIs of popular Java-enabled Web browsers.

However, you don't have to wait for this updated software to be released. You can grab the JDBC API classes from the accompanying CD-ROM or from the JavaSoft Web site at http://splash.java.com/jdbc. As I was writing this chapter, the classes were stored in a file named "jdbc.100.tar.Z." By the time you read this chapter, however, the file name may be slightly different. Once you have your software, simply follow these easy instructions to install the API classes in the proper place on your computer's hard disk. The method shown here allows you to compile and run Java applications *and* applets (using the Appletviewer) that use the JDBC:

1. Download the JDBC API package from the JavaSoft Web site or make a copy of the file from the CD-ROM.
2. On your hard drive, locate the directory that stores the Java API packages. (On my PC, the directory is C:\JAVA\SRC, and on my Sun box, the directory is \usr\local\java\src.) You do not need to install the JDBC API package in the same directory as the rest of the Java API, but I strongly recommend that you do because, as I mentioned earlier, the JDBC API will soon be a standard part of the Java API distribution and will be packaged in the Java API hierarchy.
3. Unpack the JDBC API classes using one of the following methods (for Unix-based machines or PCs), substituting the location where you downloaded the JDBC class file and the location where you want to install the JDBC classes.

Unix Procedure:

- To upack the file, enter *prompt> uncompress \home\prpatel\jdbc.100.tar.Z*.
- To create a jdbc directory with the classes and their source in separate directories, enter *prompt> tar xvf \home\prpatel\jdbc.100.tar.Z*.
- To install the JDBC classes, enter *prompt> cd \usr\local\java\src*, then enter *prompt> mv \home\prpatel\jdbc\classes\java*, and finally enter *prompt> mv \home\prpatel\jdbc\src\java*.

Windows 95 Procedure:

- Using a Windows 95 ZIP utility such as WinZip, uncompress and untar the file. Be sure the file name ends with *.tar* when you uncompress the file so that utilities will recognize the file. Untar the file to a tempory folder. Then do the following:
- Copy the java folder from the JDBC\CLASSES directory (from the temp directory where you untarred the downloaded file) to the C:\JAVA\SRC directory.
- Copy the java folder from the JDBC\SRC directory to C:\JAVA\SRC.

4. Set the CLASSPATH to point to c:/usr/local/java/src (for Unix-based machines) or C:\JAVA\SRC (for PCs). Again, remember to substitute your location if this is not where you installed the downloaded file.

Tip: Save the API documentation.
The only item left from the JDBC package you downloaded is the API documentation, which is in the jdbc\html directory that was created when you untarred the downloaded file. You may want to save that somewhere for reference. You can view the file using a Web browser.

I must stress that you should make sure that you have the CLASSPATH set properly. The package will be called in the following way in your Java program:

```
import  java.sql.*
```

You need to point the CLASSPATH at the *parent* of the java directory you copied in Step 2, which is why we set the CLASSPATH in Step 3. The package is contained in the java/sql/ folder, which is exactly as it should be according to the calling code snippet above.

Registering And Calling JDBC Drivers

Now that we've installed the JDBC classes, let's cover how you load a JDBC driver. Note that the java.sql.* must be imported into your Java program if you want to use a JDBC driver. These JDBC base classes contain the necessary elements for properly instantiating JDBC drivers, and they serve as the "middleman" between you and the low-level code in the JDBC driver. The JDBC API provides you with an easy-to-use interface for interacting with data sources, independent of the driver you are using. The following sections cover three different ways to tell the JDBC's **DriverManager** to load a JDBC driver.

The sql.drivers Property

When you want to identify a list of drivers that can be loaded with the **DriverManager**, you can set the sql.drivers system property. Because this is a system property, it can be set at the command line using the -D option:

```
java  -Dsql.drivers=imaginary.sql.iMsqlDriver classname
```

If there is more than one driver to include, just separate them using colons. If you do include more than one driver in this list, the **DriverManager** will look at each driver once the connection is created and decide which one matches the JDBC URL supplied in the **Connection** class' instantiation. (I'll provide more detail on the JDBC URL and the Connection class later on.) The first driver specified in the URL that is a successful candidate for establishing the connection will be used.

There's Always A Class For A Name

You can explicitly load a driver using the standard **Class.forName** method. This technique is a more direct way of instantiating the driver class that you want to use in the Java program. To load the mSQL JDBC driver, insert this line into your code:

```
Class.forName("imaginary.sql.iMsqlDriver");
```

This method first tries to load the imaginary/sql/iMsqlDriver from the local CLASSPATH. It then tries to load the driver using the same class loader as the Java program—the applet class loader, which is stored on the network.

Just Do It

Another approach is what I call the "quick and dirty" way of loading a JDBC driver. In this case, you simply instantiate the driver's class. Of course, I don't advise you to take this route because the driver may not properly register with the JDBC **DriverManager**. The code for this technique, however, is quite simple and worth mentioning:

```
new  imaginary.sql.iMsqlDriver;
```

Again, if this is in the applet context, this code will first try to find this driver in the local CLASSPATH, then it will try to load it from the network.

JDBC URL And The Connection

The format for specifying a data source is an extended Universal Resource Locator (URL). The JDBC URL structure is broadly defined as follows

```
jdbc:<subprotocol>:<subname>
```

where *jdbc* is the standard base, *subprotocol* is the particular data source type, and *subname* is an additional specification that can be used by the subprotocol. The subname is based solely on the subprotocol. The subprotocol (which can be "odbc," "oracle," etc.) is used by the JDBC drivers to identify themselves and then to connect to that specific subprotocol. The subprotocol is also used by the **DriverManager** to match the proper driver to a specific subprotocol. The subname can contain additional information used by the satisfying subprotocol (i.e. driver), such as the location of the data source, as well as a port number or catalog. Again, this is dependent on the subprotocol's JDBC driver. JavaSoft suggests that a network name follow the URL syntax:

```
jdbc:<subprotocol>://hostname:port/subsubname
```

The mSQL JDBC driver used in this book follows this syntax. Here's the URL you will see in some of the example code:

```
jdbc:msql://mycomputer.com:1112/databasename
```

The **DriverManager.getConnection** method in the JDBC API uses this URL when attempting to start a connection. Remember that a valid driver must be registered with the JDBC **DriverManager** before attempting to create this connection (as I discussed earlier in the *Registering and Calling JDBC Drivers* section). The **DriverManager.getConnection** method can be passed in a **Property** object where the keys "user," "password," and even "server" are set accordingly. The direct way of using the **getConnection** method involves passing these attributes in the constructor. The following is an example of how to create a **Connection** object from the **DriverManager.getConnection** method. This method returns a **Connection** object which is to be assigned to an instantiated **Connection** class:

```
String url="jdbc:msql://mydatabaseserver.com:1112/databasename";
Name = "pratik";
password = "";
Connection con;
con = DriverManager.getConnection(url, Name, password);
// remember to register the driver before doing this!
```

Chapter 4 shows a complete example of how to use the **DriverManager** and **Connection** classes, as well as how to execute queries against the database server and get the results.

Using ODBC Drivers

In an effort to close the gap between existing ODBC drivers for data sources and the emerging pure Java JDBC drivers, JavaSoft and Intersolv released the JDBC-ODBC Bridge. Note that there is a Java interface (hidden as a JDBC driver called JdbcOdbcDriver and found in the jdbc/odbc/ directory below) that does the necessary JDBC to ODBC translation with the native method library that is part of the JDBC-ODBC bridge package. Although Chapter 5 covers the inner workings of the Bridge, I would like to show you how to install it here. Once the Bridge is set up, the JDBC handles access to the ODBC data sources just like access to normal JDBC drivers; in essence, you can use the same Java code with either JDBC drivers or ODBC drivers that use the Bridge—all you have to do is change the JDBC URL to reflect a different driver.

Installing The JDBC-ODBC Bridge

There are three steps to installing the JDBC-ODBC Bridge. You'll need to get the package first. Look on the CD-ROM, or grab the latest version from JavaSoft's Web site at http://splash.javasoft.com/jdbc.

1. Uncompress the package.

2. Move the jdbc directory (located in the jdbc-odbc/classes directory) into a directory listed in your CLASSPATH, or move it to your regular Java API tree.
3. Move JdbcOdbc.dll into your java/bin directory to make sure that the system and Java executables can find the file. You can also:

For Unix:

- Add the path location of the JdbcOdbc.dll to your LD_LIBRARY_PATH, or move the DLL into a directory covered by this environment variable.

For Windows 95:

- Move the DLL into the \WINDOWS\SYSTEM directory.

Setting Up ODBC Drivers

The data sources for the ODBC driver and the drivers themselves must be configured before you can run Java programs that access them. Consult your platform documentation and ODBC server's documentation for specific information.

One of the great features of the Bridge is that it allows you to use existing data sources to start developing database-aware Java applications. And with Access, you don't even need a database server! In Chapter 11, I present the full source code for writing an application server that can use the JDBC-ODBC Bridge, the Access ODBC drivers that come with Access 95, and an Access database to develop Java applets that can interact with a database without having a database server.

To set up an Access database for ODBC, follow these steps (I'm assuming that you are using Windows 95):

1. Make sure you have the Access 95 ODBC drivers installed. These ODBC drivers can be installed from the Access install program.
2. Select Start Menu|Settings|Control Panels.
3. Click on 32 bit ODBC.
4. Click on the Add button and choose the Access Driver.
5. Type in a Data Source Name and Description (anything you like).
6. In the Database area, click on Select.
7. Select the Access database file; a sample database is located in MSoffice\ACCESS\Samples (if you installed it during the Access installation). However, you can specify any Access database you want.
8. You may want to click on the Advanced button and set the Username and Password. Click on OK and then on Close to complete the configuration.

That is all you need to do to set up the ODBC data source. Now you can write Java applications to interact with the data source on the machine in which you performed the configuration; the ODBC driver is not directly accessible over the network. You can access the data source by using the name you supplied in Step 5. For example, the URL would be something like

```
jdbc:odbc:DataSourceName
```

and the statement

```
Class.forName("jdbc.odbc.JdbcOdbcDriver")
```

would load the JDBC-ODBC bridge.

Summary

The next chapter works through a complete example of using a JDBC driver. I use the mSQL driver to query an mSQL database server over the network. The JDBC driver can easily be changed to use an ODBC driver or another JDBC driver to connect to a different data source.

Chapter 4
The Interactive
SQL Applet

Now that you have seen how to use JDBC drivers, it's time we ante up. In this chapter, we jump into the JDBC with an example applet that we'll build on and derive from through the rest of the book. Our Interactive Query applet will accomplish a number of tasks. It will:

- Connect to a database server, using a JDBC driver
- Wait for a user to enter an SQL query for the database server to process
- Display the results of the query in another text area

Of course, before we can get to the programming details of the applet, we need to take a step back, review the basics, and put together a plan. I know, plans take time to develop, and you want to get into the good stuff right away. But trust me, we'll be saving ourselves a lot of trouble later on by figuring out just the right way to get to where we want to go.

Your First JDBC Applet

Our first step in creating a quality applet is understanding exactly what we need to do. This section covers some applet basics, at a high level. We'll begin by discussing the functionality of the Interactive Query applet, and then we'll explore how to fit the data-aware components contained in the JDBC into the Java applet model. As I said before, every great program starts with a well-thought-out plan, so we'll work through the steps to create one. If you are familiar with Java, take the time to at least review the following section before moving on to *Getting A Handle On The JDBC Essentials*. However, if you are unsure about what an applet really is, and why it's different from a generic application, you will want to read this section all the way through.

The Blueprint

The applet structure has a well-defined flow, and is an event-driven development. Let's begin by defining what we want the SQL query applet to do at a high level. First, we want to connect to a database, which requires some user input: the database we want to connect to, a user name, and, possibly, a password. Next, we want to let the user enter an SQL query, which will then be executed on the connected data source. Finally, we need to retrieve and display the results of the query. We'll make this applet as simple as possible (for now), so that you understand the details of using the JDBC API and have a firm grasp of the foundations of making database-aware Java applets.

Our next task is to fill in some of the technical details of our plan. The absolute first thing we need to do, besides setting up the constructors for the various objects we use, is design and layout the user interface. We aren't quite to that phase yet (remember, we're still in the planning phase), so we'll defer the design details for a later section of this chapter, *The Look of the Applet*.

We need to get some preliminary input from the user; we need to have some event handlers to signal the applet that the user has entered some information that needs to be processed, like the SQL query. Finally, we need to clean up when the applet is terminated, like closing the connection to the data source.

Figure 4.1 shows the flow diagram for the applet. As you can see, we do most of our real work in the **Select** method. The dispatcher is the event handler method, **handleEvent()**. We use several global objects so that we don't have to pass around globally used objects (and the data contained within). This approach also adds to the overall efficiency; the code shows how to deal with some of the events directly in the event handler.

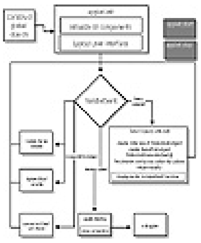


Figure 4.1 Flow diagram of the Interactive Query applet.

The Applet "Four-Step"

As indicated in Figure 4.2, Java applets have a distinct life cycle of four basic steps: initialization, execution, termination, and clean up. It's often unnecessary to implement all four, but we can use them to our advantage to make our database-aware applet more robust. Why does an applet have this flow? Applets run inside a Java Virtual Machine (JVM), or Java interpreter, like the one embedded within a Java-enabled Web browser. The interpreter handles the allocation of memory and resources for the applet, thus the applet must live within the context of the JVM. This is a pre-defined specification of the

Java environment, designed to control the applet's behavior. Note that Java *applications* do not follow this life-cycle, as they are not bound to run in the context of Java applets. Here's a synopsis of what the four overridable methods, or steps, do in the context of Java applets:

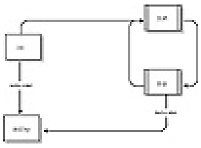


Figure 4.2 An applet's life cycle.

- **init** This is the method called when the applet is first started. It is only called once, and it is the place where the initialization of objects (via construction or assignment) should be done. It is also a good place to set up the user interface.
- **start** Once the applet has been initialized, this method is called to begin the execution of the applet. If you are using threads, this is the ideal place to begin threads that you create to use in the applet. This method is called when the Web browser (or appletviewer) becomes *active*; that is, when the user brings up the window or focuses attention to the window.
- **stop** This method is called when the applet window (which can be within a Web browser) becomes *inactive*. For instance, iconifying the Web browser calls this method. This can be used to suspend the execution of the applet when the user's attention is somewhere else.
- **destroy** Before the applet is wiped from memory and its resources returned to the operating system, this method is called. This is a great place to flush buffers and close connections, and generally to clean house.

As I said earlier, you don't need to have all four steps in your applet. For instance, our simple applet doesn't need the **start** and **stop** methods. Because we aren't running an animation or any other CPU-consuming process continuously, we aren't stealing many precious system cycles. Besides, if you are connected to a database across the Internet and execute a query that takes time to process and download the results from, you may want to check your email instead of staring at the computer while the applet is working. These methods are meant to be overriden, since a minimal "default" for each method exists; the default depends on the individual intended function of the four methods.

Events To Watch For

The flow chart in Figure 4.1 shows some of the events we want to process. In this applet, we are only looking for keystrokes and mouse clicks. We override the **handleEvent** method to allow us to program for these events. We use the target property of the **Event** object, which is passed into the event handler, to look for a specific object. Then we can look for more specific events. Listing 4.1 contains a snippet of code that shows how we deal with the user entering a name in the **TextArea** NameField.

Listing 4.1 Trapping for the Enter key event in a specific object.

```
if (evt.target == NameField)
      {char c=(char)evt.key;
      if (c  == '\n')
          { Name=NameField.getText();
            return true;
         }
         else { return false; }
   }
```

The object **evt** is the local instantiation of the Event parameter that is part of the **handleEvent** method, as we'll see later in the complete source code listing. We use the **target** property to see which object the event occurred in, then we look at the **key** property to see if the Enter key was pressed. The Java escape sequence for Enter is **\n**. The rest of the code shown in the listing is fairly straightforward: We compare the pressed key to the "enter" escape sequence, and if we come up with a match, we set the **Name** string variable to the text in the NameField using the **TextArea getText** method. Because we have processed the event, and we want to let the rest of the event handler know that we've dealt with it, we return true. If this wasn't the key we were looking for in this specific object (NameField), we would return false so that the other event handling code could attempt to process this event.

Finishing Up

One of Java's great strengths lies in its ability to automatically allocate and de-allocate memory for objects created in the program, so the programmer doesn't have to. We primarily use the **destroy** method to close the database connection that we open in the applet. The JDBC driver that we used to connect to the data source is alerted to the fact that the program is exiting, so it can gracefully close the connection and flush input and output buffers.

Getting A Handle On The JDBC Essentials: The Complete Applet Source Code

Okay, enough talk, let's get busy! The complete source code is shown in Listings 4.2 though 4.9. The HTML file that we use to call our applet is shown in Listing 4.10. I bet you're not too keen on entering all that code. But wait! There's no need to type it all in, just pull out the CD-ROM and load the source into your favorite editor or IDE. Don't forget, though, that you need to have the JDBC driver installed, and you may need your CLASSPATH set so that the applet can find the driver. If you're planning on loading the driver as a class along with the applet, make sure you put the driver in the same place as the applet. See Chapter 3 if you have trouble getting the applet to run and you keep getting the “Can’t Find a Driver” or “Class not found” error.

Tip: Source code on the CD-ROM.
There’s no need to type in the source code because the Interactive Query applet can be found on the CD-ROM, as is true for all source code in this book.

The Look Of The Applet

As I promised earlier, we’re going to cover the details of user interface design and layout. Listing 4.2 covers the initialization of the user interface, as well as the normal “preliminaries” associated with Java programs. To help you along, I’ve included some comments to elaborate on the fine points that will help you to understand what’s going on and what we are doing.

Listing 4.2 Setting up the objects.

```
import  java.net.URL;
import  java.awt.*;
import  java.applet.Applet;
// These are standard issue with applets, we need the net.URL
// class because the database identifier is a glorified URL.

import  java.sql.*;
// These are the packages needed to load the JDBC kernel, known as the
// DriverManager.
import  imaginary.sql.*;
// These are the actual driver classes! We are using the msql JDBC
// drivers to access our msql database.

public class IQ extends java.applet.Applet {
// This is the constructor for the base applet. Remember that the applet
// name must match the file name the applet is stored in--this applet
// should be saved in a file called "IQ.java".

 Button ConnectBtn = new Button("Connect to Database");
 TextField QueryField = new TextField(40);
 TextArea OutputField = new TextArea(10,75);
 TextField NameField = new TextField(40);
 TextField DBurl = new TextField(40);
 Connection con;
// Here we create the objects we plan to use in the applet.
// The Connection object is part of the JDBC API, and is the primary way
// of tying the JDBC's function to the applet.

  String url = "";
  String Name = "";
```

GridBagLayout: It’s Easier Than It Seems!

In Listing 4.2, we set up the objects we’ll be using in the user interface. We loaded the necessary classes and the specific driver we will use in the applet. In Listing 4.3, we go through the init phase of the applet, where we set up the user interface. We use

GridBagLayout, a Java layout manager, to position the components in the applet window. **GridBagLayout** is flexible and offers us a quick way of producing an attractive interface.

Listing 4.3 Setting up the user interface.

```
public void init() {
      QueryField.setEditable(true);
      OutputField.setEditable(false);
     NameField.setEditable(true);
     DBurl.setEditable(true);
// We want to set the individual TextArea and TextField to be editable
so
// the user can edit the OutputField, where we plan on showing the
// results of the query.

 GridBagLayout gridbag = new GridBagLayout();
 GridBagConstraints Con = new GridBagConstraints();
// create a new instance of GridBagLayout and the complementary
// GridBagConstraints.

 setLayout(gridbag);
// Set the layout of the applet to the gridbag that we created above.
 setFont(new Font("Helvetica", Font.PLAIN, 12));
 setBackground(Color.gray);
// Set the font and color of the applet.

 Con.weightx=1.0;
 Con.weighty=0.0;
 Con.anchor = GridBagConstraints.CENTER;
 Con.fill = GridBagConstraints.NONE;
 Con.gridwidth = GridBagConstraints.REMAINDER;
```

This code requires some explanation. The **weightx** and **weighty** properties determine how the space in the respective direction is distributed. If either weight property is set to 0, the default, any extra space is distributed around the outside of the components in the corresponding direction. The components are also centered automatically in that direction. If either weight property is set to 1, any extra space is distributed within the spaces between components in the corresponding direction. Hence, in setting the **weightx=1**, we have told the **GridBagLayout** layout manager to position the components on each row so that extra space is added equally between the components on that row. However, the rows of components are vertically “clumped” together because the **weighty** property is set to 0.0. Later on, we’ll change **weighty** to 1 so that the large **TextArea** (the OutputField) takes up extra space equal to all the components added before it. Take a look at Figure 4.3, shown at the end of the chapter, to see what I mean.

We also set the anchor property to tell the **GridBagLayout** to position the components on the center, relative to each other. The **fill** property is set to NONE so that the components are not stretched to fill empty space. You will find this technique to be useful when you want a large graphics area (Canvas) to take up any empty space that is available around it, respective to the other components. The **gridwidth** is set to

REMAINDER to signal that any component assigned the **GridBagContstraint Con** takes up the rest of the space on a row. Similarly, we can set **gridheight** to REMAINDER so that a component assigned this constraint takes up the remaining vertical space. The last detail associated with **GridBagLayout** involves assigning the properties to the component. This is done via the **setConstraints** method in **GridBagLayout**.

Listing 4.4 shows how we do this. Notice that we assign properties for the **TextArea**, but not for the **Labels**. Because we're positioning the **Labels** on the "right" side of the screen (the default), there is no need to assign constraints. There are more properties you can set with **GridBagLayout**, but it's beyond the scope of this book.

Listing 4.4 Assigning properties to components.

```
   add(new Label("Name"));
     gridbag.setConstraints(NameField, Con);
   add(NameField);
// Note that we did not setConstraints for the Label. The GridbagLayout
// manager assumes they carry the default constraints. The NameField is
// assigned to be the last component on its row via the constraints Con,
// then added to the user interface.

   add(new Label("Database URL"));
     gridbag.setConstraints(DBurl, Con);
   add(DBurl);

     gridbag.setConstraints(ConnectBtn, Con);
   add(ConnectBtn);
// Here, we only want the ConnectBtn button on a row, by itself, so we
// set the constraints, and add it.

   add(new Label("SQL Query"));
     gridbag.setConstraints(QueryField, Con);
   add(QueryField);

   Label result_label = new Label("Result");
   result_label.setFont(new Font("Helvetica", Font.PLAIN, 16));
      result_label.setForeground(Color.blue);
    gridbag.setConstraints(result_label, Con);
   add(result_label);
// Here we add a label on its own line. We also set the colors for it.

  Con.weighty=1.0;
    gridbag.setConstraints(OutputField, Con);
      OutputField.setForeground(Color.white);
      OutputField.setBackground(Color.black);
   add(OutputField);
// This is what we were talking about before. We want the large
OutputField to
// take up as much of the remaining space as possible, so we set the
// weighty=1 at this point. This sets the field apart from the
previously
// added components, and gives it more room to exist in.

  show();
```

```
    } //init
```

Everything has been added to the user interface, so let's show it! We also don't need to do anything else as far as preparation, so that ends the init method of our applet. Now we can move on to handling events.

Handling Events

We want to watch for four events when our applet is running: the user pressing the Enter key in the DBurl, NameField, and QueryField TextAreas, and the user clicking on the Connect button. Earlier in the chapter, we saw how to watch for events, but now we get to see what we do once the event is trapped, as shown in Listing 4.5. The event handling code is contained in the generic **handleEvent** method.

Listing 4.5 Handling events.

```
public boolean handleEvent(Event evt) {
// The standard format for this method includes the Event class where
// all the properties are set.

  if (evt.target == NameField)
     {char c=(char)evt.key;
// Look for the Enter key pressed in the NameField.
      if (c  == '\n')
          { Name=NameField.getText();
// Set the global Name variable to the contents in the NameField.
           return true;
        }
       else { return false; }
    }

if (evt.target == DBurl)
      {char c=(char)evt.key;
// Look for the enter key pressed in the DBurl TextArea.
     if (c  == '\n')
          { url=DBurl.getText();
// Set the global url variable to the contents of the DBurl TextArea.
           return true;
        }
       else { return false; }
    }

if (evt.target  ==  QueryField)
      {char c=(char)evt.key;
//  Look for the Enter key pressed in the QueryField.
      if (c  == '\n')
         {
                OutputField.setText(Select(QueryField.getText()));
// Get the contents of the QueryField, and pass them to the Select
// method that is defined in Listing 4.7. The Select method executes the
// entered query, and returns the results. These results are shown in
the
// OutputField using the setText method.
            return true;
        }
       else { return false; }
    }
```

Opening The Connection

Our next step is to connect to the database that will process the user's query, as shown in Listing 4.6.

Listing 4.6 Opening a database connection.

```
if (evt.target == ConnectBtn)
   {
// If the user clicks the "Connect" button, connect to the database
// specified in the DBurl TextArea and the user name specified in the
// NameField TextArea.

        url=DBurl.getText();
       Name=NameField.getText();
    try {
         new imaginary.sql.iMsqlDriver();
// This creates a new instance of the Driver we want to use. There are a
// number of ways to specify which driver you want to use, and there is
// even a way to let the JDBC DriverManager choose which driver it
thinks
// it needs to connect to the data source.

        con = DriverManager.getConnection(url, Name, "");
// Actually make the connection. Use the entered URL and the entered
// user name when making the connection. We haven't specified a password,
// so just send nothing ("").
      ConnectBtn.setLabel("Reconnect to Database");
// Finally, change what the ConnectBtn  to show "Reconnect to Database".
    }
    catch( Exception e ) {
        e.printStackTrace();
        OutputField.setText(e.getMessage());
    }
// The creation of the connection throws an exception if there was a
// problem connecting using the specified parameters. We have to enclose
// the getConnection method in a try-catch block to catch any
// exceptions that may be thrown. If there is a problem and an exception
// thrown, print it out to the console, and to the OutputField.

   return true;
  }
return false;
} // handleEvent() end
```

No Guts, No Glory: Executing Queries And Processing Results

Now that we have opened the connection to the data source (Listing 4.6), it's time to set up the mechanism for executing queries and getting the results, as shown in Listings 4.7 and 4.8. The parameter that we need in this method is a String containing the SQL query the user entered into the QueryField. We will return the results of the query as a string because we only want to pipe all of the results into the OutputField **TextArea**. We cast all of the returned results into a String—however, if the database contains binary data, we could get some weird output, or even cause the program to break. When I tested the applet, the data source that I queried contained numerical and strings only. In Chapter 7, I'll show you how to deal with different data types in the ANSI SQL-2 specification, upon which the data types for the JDBC are based.

Listing 4.7 Executing a statement.

```
public String Select(String QueryLine) {
// This is the method we called above in Listing 4.5.
// We return a String, and use a String parameter for the entered query.

  String Output="";
  int columns;
  int pos;
    try {
// Several of the following methods can throw exceptions if there was a
// problem with the query, or if the connection breaks, or if
// we improperly try to retrieve results.

      Statement stmt = con.createStatement();
// First, we instantiate a Statement class that is required to execute
// the query. The Connection class returns a Statement object in its
// createStatement method, which links the opened connection to
// the passed-back Statement object. This is how the stmt instance
// is linked to the actual connection to the data source.

      ResultSet rs = stmt.executeQuery(QueryLine);
// The ResultSet in turn is linked to the connection to the data source
// via the Statement class. The Statement class contains the
executeQuery
// method, which returns a ResultSet class. This is analagous to a
// pointer that can be used to retrieve the results from the JDBC
// connection.

      columns=(rs.getMetaData()).getColumnCount();
// Here we use the getMetaData method in the result set to return a
// Metadata object. The MetaData object contains a getColumnCount
// method which we use to determine how many columns of data
// are present in the result. We set this equal to an integer
// variable.
```

Listing 4.8 Getting the Result and MetaData Information.

```
      while(rs.next()) {
// Now, we use the next method of the ResultSet instance rs to fetch
// each row, one by one. There are more optimized ways of doing
// this--namely using the inputStream feature of the JDBC driver.
// I show you an example of this in Chapter 9.

        for( pos=1; pos<=columns; pos++) {
// Now let's get each column in the row ( each cell ), one by one.

          Output+=rs.getObject(pos)+" ";
// Here we've used the general method for getting a result. The
// getObject method will attempt to caste the result in the form
// of its assignee, in this case the String variable Output.
// We simply get each "cell" and add a space to it, then append it onto
// the Output variable.

        }
// End for loop (end looping through the columns for a specific row ).

        Output+="\n";
// For each row that we fetch, we need to add a carriage return so that
// the next fetched row starts on the next line.
      }
// End while loop ( end fetching rows when no more rows are left ).

      stmt.close();
// Clean up, close the stmt, in effect, close the input-output query
// connection streams, but stay connected to the data source.
    }
    catch( Exception e ) {
      e.printStackTrace();
      Output=e.getMessage();
    }
// We have to catch any exceptions that were thrown while we were
// querying or retrieving the data. Print the exception
// to the console and return it so it can be shown to the user
// in the applet.

return Output;
// Before exiting, return the result that we got.
  }
```

Wrapping It Up

The last part of the applet, shown in Listing 4.9, involves terminating the connection to the data source. This is done in the **destroy** method of the applet. We have to catch an exception, if one occurs, while the **close** method is called on the connection.

Listing 4.9 Terminating the connection.

```
public void destroy() {

  try {con.close();}
   catch( Exception e ) {
      e.printStackTrace();
     System.out.println(e.getMessage());
    }
 } // end destroy
} // end applet
```

The HTML File That Calls The Applet

We need to call this applet from an HTML file, which is shown in Listing 4.10. We don't pass in any properties, but we could easily include a default data source URL and user name that the applet would read in before initializing the user interface, and then set the appropriate **TextField** to show these defaults. Note that we set the width and height carefully in the **<APPLET>** tag. This is to make sure that our applet's user interface has enough room to be properly laid out.

Listing 4.10 HTML code to call the interactive query applet.

```
<HTML>
<HEAD>
<TITLE>JDBC Client Applet - Interactive SQL Command Util</TITLE>
</HEAD>
<BODY>
<H1>Interactive JDBC SQL Query Applet</H1>
<hr>

<applet code=IQ.class width=450 height=350>
</applet>

<hr>
</BODY>
</HTML>
```

The Final Product

Figure 4.3 shows a screen shot of the completed applet, and Figure 4.4 shows the applet running. Not too shabby for our first try. We've covered a lot of ground in creating this applet, so let's take some time to recap the important details. We learned how to:

Figure 4.3 The completed Interactive Query applet.

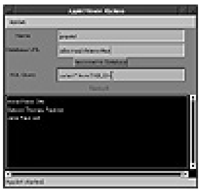


Figure 4.4 The Interactive Query applet running.

- Open a connection to a data source
- Connect a Statement object to the data source via the connection
- Execute a query
- Get MetaData information about the result of the query
- Use the MetaData information to properly get the results row-by-row, column-by-column
- Close the connection

To use the applet, you can load the HTML file in a Java-enabled Web browser, or you can start the applet from the command line:

```
bash$ appletviewer IQ.html &
```

Don't forget, if you have problems finding the class file or the driver, set the CLASSPATH. See Chapter 3 for more help on this topic.

Coming Up Next

In the next chapter, we'll explore the bridge between ODBC and JDBC. You'll see how easy it is to use existing ODBC drivers with JDBC, and learn some of the fine points of the relation, similarity, and difference between the two database connectivity standards. You won't want to miss this one; the author, Karl Moss, is also the author of the Sun/Intersolv ODBC-JDBC bridge included in the JDBC package.

Chapter 5
Accessing ODBC Services Using JDBC

One of JavaSoft's first tasks in developing the JDBC API was to get it into the hands of developers. Defining the API specification was a major step, but JDBC drivers must be implemented in order to actually access data. Because ODBC has already established itself as an industry standard, what better way to make JDBC usable by a large community of developers than to provide a JDBC driver that uses ODBC. JavaSoft turned to Intersolv to provide resources to develop a bridge between the two, and the resulting JDBC driver—the Bridge—is now included with the Java Developer's kit.

The Bridge works great, but there are some things you need to understand before you can implement it properly. In this chapter, we'll cover the requirements necessary to use the Bridge, the limitations of the Bridge, and the most elegant way to make a connection to a JDBC URL. I'll also provide you with a list of each JDBC method and the corresponding ODBC call (broken down by the type of call).

Bridge Requirements

One thing to note about the JDBC-ODBC Bridge is that it contains a very thin layer of native code. This library's sole purpose is to accept an ODBC call from Java, execute that call, and return any results back to the driver. There is no other magic happening within this library; all processing, including memory management, is contained within the Java side of the Bridge. Unfortunately, this means that there is a library containing C code that must be ported to each of the operating systems that the Bridge will execute on. This is obviously not an ideal situation, and invalidates one of Java's major advantages—portability. So, instead of being able to download Java class files and execute on the fly, you must first install and configure additional software in order to use the Bridge. Here's a short checklist of required components:

- The Java Developer's Kit
- The JDBC Interface classes (java.sql.*)
- The JDBC-ODBC Bridge classes (jdbc.odbc.* or sun.jdbc.odbc.* for JDBC version 1.1 and higher)

- An ODBC Driver Manager (such as the one provided by Microsoft for Win95/NT); do not confuse this with the JDBC **DriverManager** class
- Any ODBC drivers to be used from the Bridge (from vendors such as Intersolv, Microsoft, and Visigenic)

Before actually attempting to use the Bridge, save yourself lots of headaches—be sure to test the ODBC drivers that you will be using! I have pursued countless reported problems that ended up being nothing more than an ODBC configuration issue. Make sure you setup your data sources properly, and then test them to make sure you can connect and perform work. You can accomplish this by either using an existing tool or writing your own sample ODBC application. Most vendors include sample source code to create an ODBC application, and Microsoft provides a tool named Gator (a.k.a ODBCTE32.EXE) which can fully exercise ODBC data sources on Win95/NT.

The Bridge Is Great, But...

All looks good for the Bridge; it gives you access to any ODBC data source, and it's *free*! But wait, there are a few limitations that I need to make you aware of before you start.

First, as I mentioned before, a lot of software must be installed and configured on each system that will be using the Bridge. In today's environment, this feat cannot be accomplished automatically. Unfortunately, this task can be a major limitation, not only from the standpoint of getting the software installed and configured properly, but ODBC drivers may not be readily available (or may be quite costly) for the operating system that you are using.

Second, understand the limitations of the ODBC driver that you will be using. If the ODBC driver can't do it, neither can the Bridge. The Bridge is not going to add any value to the ODBC driver that you are using other than allowing you to use it via JDBC. One of the most frequently asked questions I get is: "If I use the Bridge, can I access my data over the Internet?" If the ODBC driver that you are using can, then the Bridge can; if it can't, then neither can the Bridge.

Third, keep in mind the quality of the ODBC driver. In order for the Bridge to properly use an ODBC driver, it must be ODBC version 2.0 or higher. Also, if there are bugs in the ODBC driver, they will surely be present when you use it from JDBC.

Finally, there are Java security considerations. From the JDBC API specification, all JDBC drivers must follow the standard security model, most importantly:

- JDBC should not allow untrusted applets access to local database data
- An untrusted applet will normally only be allowed to open a database connection back to the server from which it was downloaded

For trusted applets and any type of application, the Bridge can be used in any fashion to connect to any data source. For untrusted applets, the prognosis is bleak. Untrusted applets can only access databases on the server from which they were downloaded. Normally, the Java Security Manager will prohibit a TCP connection from being made to an unauthorized hostname; that is, if the TCP connection is being made from within the Java Virtual Machine (JVM). In the case of the Bridge, this connection would be made from within the ODBC driver, outside the control of the JVM. If the Bridge could determine the hostname that it will be connected to, a call to the Java Security Manager could easily check to ensure that a connection is allowed. Unfortunately, it is not always possible to determine the hostname for a given ODBC data source name. For this reason, the Bridge always assumes the worst. An untrusted applet is not allowed to access any ODBC data source. What this means is that if you can't convince the Internet browser in use that an applet is trusted, you can't use the Bridge from that applet.

The ODBC URL

To make a connection to a JDBC driver, you must supply a URL. The general structure of the JDBC URL is

```
jdbc:<subprotocol>:<subname>
```

where *subprotocol* is the kind of database connectivity being requested, and *subname* provides additional information for the subprotocol. For the Bridge, the specific URL structure is:

```
jdbc:odbc:<ODBC datasource name>[;attribute-name=attribute-value]...
```

The Bridge can only provide services for URLs that have a subprotocol of **odbc**. If a different subprotocol is given, the Bridge will simply tell the JDBC **DriverManager** that it has no idea what the URL means, and that it can't support it. The subname specifies the ODBC data source name to use, followed by any additional connection string attributes. Here's a code snippet that you can use to connect to an ODBC data source named *Accounting*, with a user name of *dept12* and a password of *Julie*:

```
// Create a new instance of the JDBC-ODBC Bridge.

new jdbc.odbc.JdbcOdbcDriver();

// The JDBC-ODBC Bridge will have registered itself with the JDBC
// DriverManager. We can now let the DriverManager choose the right
// driver to connect to the given URL.

Connection con = DriverManager.getConnection("jdbc:odbc:Accounting",
   "dept12", "Julie");
```

An alternative way of connecting to this same data source would be to pass the user name and password as connection string attributes:

```
Connection con = DriverManager.getConnection("jdbc:odbc:Accounting;UID=
   dept12;PWD=Julie");
```

A third, more robust way of connecting would be to use a **java.util.Properties** object. **DriverManager.getConnection** is overloaded to support three versions of the interface:

```
public static synchronized Connection getConnection(String url, String
   user, String password) throws SQLException;
```

```
public static synchronized Connection getConnection(String url);
public static synchronized Connection getConnection(String url,
      java.util.Properties info);
```

The third method listed here is by far the most elegant way of connecting to any JDBC driver. An intelligent Java application/applet will use **Driver.getPropertyInfo** (which will not be covered here) to get a list of all of the required and optional properties for the driver. The Java program can then prompt the user for this information, and then create a **java.util.Properties** object that contains an element for each of the driver properties to be used for the JDBC connection. The following code shows how to setup the **java.util.Properties** object:

```
// Create the Properties object.

java.util.Properties prop = new java.util.Properties();

// Populate the Properties object with each property to be passed to the
// JDBC driver.

prop.put("UID", "dept12");
prop.put("PWD", "Julie");

Connection con = DriverManager.getConnection("jdbc:odbc:Accounting",
   prop);
```

JDBC To ODBC Calls: A Roadmap

For all of you ODBC junkies, Tables 5.1 through 5.8 show each JDBC method and the corresponding ODBC call (only JDBC methods that actually make an ODBC call are included). I can hear you now: “But isn’t this a closely guarded national secret? What if someone takes this information to write another Bridge?” First of all, the information provided here can be easily gathered by turning on the JDBC logging facility (**DriverManager.setLogStream**). The Bridge is nice enough to log every ODBC call as it is made, providing a log stream has been set via the **DriverManager** (all good JDBC drivers should provide adequate logging to aid in debugging). And second, the Bridge is provided for free. No one could possibly take this information to create a better Bridge at a lower price. It simply can’t be done. I provide this information in an effort to help you better understand how the Bridge operates, and, if you are well versed in ODBC, to give you the direct correlation between the Bridge and ODBC. This should enable you to write advanced JDBC applications right off the starting line.

Table 5.1 Driver ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
connect	SQLDriverConnect	The Bridge creates a connection string using the java.util. Properties

		attribute given
Each property returned is converted into a DriverPropertyInfo object		

Table 5.2 Connection ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
prepareStatement	SQLPrepare	Prepares the statement for use with IN parameters
prepareCall	SQLPrepare	Prepares the statement for use with IN and OUT parameters (JDBC has not defined the use of IN/OUT parameters together)
nativeSQL	SQLNativeSql	Converts the given SQL into native format, expanding escape sequences
setAutoCommit	SQLSetConnectOption	fOption = SQL_AUTOCOMMIT
getAutoCommit	SQLGetConnectOption	fOption = SQL_AUTOCOMMIT
commit	SQLTransact	fType = SQL_COMMIT
rollback	SQLTransact	fType = SQL_ROLLBACK
close	SQLFreeConnect	Frees the connection handle associated with the connection
setReadOnly	SQLSetConnectOption	fOption = SQL_ACCESS_MODE; this is only a hint to the ODBC driver; the underlying driver may not actually change its behavior

isReadOnly	SQLGetConnectOption	fOption = SQL_ACCESS_MODE
setCatalog	SQLSetConnectOption	fOption = SQL_CURRENT_ QUALIFIER
getCatalog	SQLGetInfo	fInfoType = SQL_DATABASE_NAME
setTransactionIsolation	SQLSetConnectOption	fOption = SQL_TXN_ISOLATION
getTransactionIsolation	SQLGetConnectOption	fOption = SQL_TXN_ISOLATION
setAutoClose		ODBC does not provide a method to modify this behavior

fInfoType = SQL_CURSOR_COMMIT_ BEHAVIOR and fInfoType = SQL_CURSOR_ ROLLBACK_BEHAVIOR; the Bridge makes both calls, and if either are true, then getAutoClose returns true

Table 5.3 DatabaseMetaData ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
allProceduresAreCallable	SQLGetInfo	fInfoType = SQL_ACCESSAB PROCEDURES
allTablesAreSelectable	SQLGetInfo	fInfoType = SQL_ACCESSAB TABLES
getUserName	SQLGetInfo	fInfoType = SQL_USER_NA
isReadOnly	SQLGetInfo	fInfoType = SQL_DATA_ SOURCE_READ_ONLY
nullsAreSortedHigh	SQLGetInfo	fInfoType = SQL_NULL_COLLATION; res must be SQL_NC_HIGH
nullsAreSortedLow	SQLGetInfo	fInfoType = SQL_NULL_COLLATION; res must be SQL_NC_LOW
nullsAreSortedAtStart	SQLGetInfo	fInfoType =

		SQL_NULL_COLLATION; res must be SQL_NC_START
nullsAreSortedAtEnd	SQLGetInfo	fInfoType = SQL_NULL_COLLATION; res must be SQL_NC_END
getDatabaseProductName	SQLGetInfo	fInfoType = SQL_DBMS_NA
getDatabaseProductVersion	SQLGetInfo	fInfoType = SQL_DBMS_V
usesLocalFiles	SQLGetInfo	fInfoType = SQL_FILE_USA the result must be SQL_FILE_QUALIFIER
usesLocalFilePerTable	SQLGetInfo	fInfoType = SQL_FILE_USA the result must be SQL_FILE_TABLE
supportsMixedCaseIdentifiers	SQLGetInfo	fInfoType = SQL_IDENTIFIER_CASE; th result must be SQL_IC_UPP SQL_IC_LOWER or SQL_IC_MIXED
storesUpperCaseIdentifiers	SQLGetInfo	fInfoType = SQL_IDENTIFIER_CASE, th result must be SQL_IC_UPP
storesLowerCaseIdentifiers	SQLGetInfo	fInfoType = SQL_IDENTIFIER_CASE; th result must be SQL_IC_LOW
storesMixedCaseIdentifiers	SQLGetInfo	fInfoType = SQL_IDENTIFIER_CASE; th result must be SQL_IC_MIX
supportsMixedCaseQuoted Identifiers	SQLGetInfo	fInfoType = SQL_QUOTED_IDENTIFIER_C the result must be SQL_IC_UPPER, SQL_IC_LOW or SQL_IC_MIXED
storesUpperCaseQuoted Identifiers	SQLGetInfo	fInfoType = SQL_QUOTED_IDENTIFIER_C the result must be SQL_IC_UPPER
storesLowerCaseQuoted Identifiers	SQLGetInfo	fInfoType = SQL_QUOTED_IDENTIFIER_C the result must be SQL_IC_LOWER

storesMixedCaseQuoted Identifiers	SQLGetInfo	fInfoType = SQL_QUOTED_IDENTIFIER_C the result must be SQL_IC_MIXED
getIdentifierQuoteString	SQLGetInfo	fInfoType = SQL_IDENTIFIER_QUOTE_CI
getSQLKeywords	SQLGetInfo	fInfoType = SQL_KEYWORI
getNumericFunctions	SQLGetInfo	fInfoType = SQL_NUMERIC_FUNCTIONS; result is a bitmask enumera the scalar numeric functior this bitmask is used to crea comma-separated list of functions
getStringFunctions	SQLGetInfo	fInfoType = SQL_STRING_FUNCTIONS; result is a bitmask enumera the scalar string functions; bitmask is used to create comma-separated list of functions
getSystemFunctions	SQLGetInfo	fInfoType = SQL_SYSTEM FUNCTIONS; the result is bitmask enumerating the sc system functions; this bitma used to create a comma-separated list of function
getTimeDateFunctions	SQLGetInfo	fInfoType = SQL_TIMEDAT FUNCTIONS; the result is bitmask enumerating the sc date and time functions; T bitmask is used to create comma-separated list of functions
getSearchStringEscape	SQLGetInfo	fInfoType = SQL_SEARCH_PATTERN_ ESCAPE
getExtraNameCharacters	SQLGetInfo	fInfoType = SQL_SPECIAL CHARACTERS
supportsAlterTableWithAdd Column	SQLGetInfo	fInfoType = SQL_ALTER_TAE result must have the

		SQL_AT_ADD_COLUMN bit
supportsAlterTableWithDropColumn	SQLGetInfo	fInfoType =SQL_ALTER_TAB the result must have the SQL_AT_DROP_COLUMN bit set
supportsColumnAliasing	SQLGetInfo	fInfoType = SQL_COLUMN_A
nullPlusNonNullIsNull	SQLGetInfo	fInfoType = SQL_CONCAT_NULL_BEHAVI the result must be SQL_CB_
supportsConvert	SQLGetInfo	fInfoType = SQL_CONVERT FUNCTIONS; the result mus SQL_FN_CVT_CONVERT
supportsTableCorrelationNames	SQLGetInfo	fInfoType = SQL_CORRELATI NAME; the result must b SQL_CN_DIFFERENT or SQL_CN_AN
supportsDifferentTableCorrelationNames	SQLGetInfo	fInfoType = SQL_CORRELATI NAMES; the result must b SQL_CN_DIFFERENT
supportsExpressionsInOrderBy	SQLGetInfo	fInfoType = SQL_EXPRESSIO IN_ORDER_BY
supportsOrderByUnrelated	SQLGetInfo	fInfoType = SQL_ORDER_B COLUMNS_IN_SELECT
supportsGroupBy	SQLGetInfo	fInfoType = SQL_GROUP_BY, result must not be SQL_GB_NOT_SUPPORTED
supportsGroupByUnrelated	SQLGetInfo	fInfoType = SQL_GROUP_BY, result must be SQL_GB_NO RELATION
supportsGroupByBeyondSelect	SQLGetInfo	fInfoType = SQL_GROUP_BY, result must be SQL_GB_GROUP_BY_CONTAINS_SELECT
supportsLikeEscapeClause	SQLGetInfo	fInfoType = SQL_LIKE_ESCA CLAUSE
supportsMultipleResultSets	SQLGetInfo	fInfoType = SQL_MULT_RESU SETS

supportsMultipleTransactions	SQLGetInfo	fInfoType = SQL_MULTIPL ACTIVE_TXN
supportsNonNullableColumns	SQLGetInfo	fInfoType = SQL_NON_NULLABLE_COLUMNS; the re must be SQL_NNC_NON_NULL
supportsMinimumSQL Grammar	SQLGetInfo	fInfoType = SQL_ODBC_SQ CONFORMANCE; result must SQL_OSC_MINIMUM, SQL_OSC_CORE, or SQL_OSC_EXTENDED
supportsCoreSQLGrammar	SQLGetInfo	fInfoType = SQL_ODBC_SQL_CONFORMANCE; the re must be SQL_OSC_CORE c SQL_OSC_EXTENDED
supportsExtendedSQL Grammar	SQLGetInfo	fInfoType = SQL_ODBC_SQL_CONFORMANCE; the re must be SQL_OSC_EXTENDED
supportsIntegrityEnhancement Facility	SQLGetInfo	fInfoType = SQL_ODBC_SQ OPT_IEF
supportsOuterJoins	SQLGetInfo	fInfoType = SQL_OUTER_JO the result must not be "N
supportsFullOuterJoins	SQLGetInfo	fInfoType = SQL_OUTER_JO the result must be "F"
supportsLimitedOuterJoins	SQLGetInfo	fInfoType = SQL_OUTER_JO the result must be "P"
getSchemaTerm	SQLGetInfo	fInfoType = SQL_OWNER_T
getProcedureTerm	SQLGetInfo	fInfoType = SQL_PROCEDURE_TERM
getCatalogTerm	SQLGetInfo	fInfoType = SQL_QUALIFIER_TERM
isCatalogAtStart	SQLGetInfo	fInfoType = SQL_QUALIFIE LOCATION; the result must SQL_QL_START
getCatalogSeparator	SQLGetInfo	fInfoType = SQL_QUALIFIER_NAME_SEPARATOR
supportsSchemasInData	SQLGetInfo	fInfoType =

Manipulation		SQL_OWNER_USAGE; the re must have the SQL_OU_DM STATEMENTS bit set
supportsSchemasInProcedure Calls	SQLGetInfo	fInfoType = SQL_OWNER_USAGE; the re must have the SQL_OU_ PROCEDURE_INVOCATION bi
supportsSchemasInTable Definitions	SQLGetInfo	fInfoType = SQL_OWNER_USAGE; the re must have the SQL_OU_TAB DEFINITION bit set
supportsSchemasInIndex Definitions	SQLGetInfo	fInfoType = SQL_OWNER_USAGE; the re must have the SQL_OU_IND DEFINITION bit set
supportsSchemasInPrivilege Definitions	SQLGetInfo	fInfoType = SQL_OWNER_USAGE; the re must have the SQL_OU_ PRIVILEGE_DEFINITION bit
supportsCatalogsInData Manipulation	SQLGetInfo	fInfoType = SQL_QUALIFIER_USAGE; t result must have the SQL_QU_DML_STATEMENTS set
supportsCatalogsInProcedure Calls	SQLGetInfo	fInfoType = SQL_QUALIFIER_USAGE; t result must have the SQL_C PROCEDURE_INVOCATION bi
supportsCatalogsInTable Definitions	SQLGetInfo	fInfoType = SQL_QUALIFIE USAGE; the result must have SQL_QU_TABLE_DEFINITION set
supportsCatalogsInIndex Definitions	SQLGetInfo	fInfoType = SQL_QUALIFIER_USAGE; t result must have the SQL_QU_INDEX_DEFINITION set
supportsCatalogsInPrivilege Definitions	SQLGetInfo	fInfoType = SQL_QUALIFIER_USAGE; t result must have the SQL_C PRIVILEGE_DEFINITION bit

supportsPositionedDelete	SQLGetInfo	fInfoType = SQL_POSITION STATEMENTS; the result m have the SQL_PS_POSITIONED_DELET set
supportsPositionedUpdate	SQLGetInfo	fInfoType = SQL_POSITION STATEMENTS; the result m have the SQL_PS_POSITIONED_UPDA bit set
supportsSelectForUpdate	SQLGetInfo	fInfoType = SQL_POSITION STATEMENTS; the result m have the SQL_PS_SELECT_FOR_UPDA bit set
supportsStoredProcedures	SQLGetInfo	fInfoType = SQL_PROCEDU
supportsSubqueriesIn Comparisons	SQLGetInfo	fInfoType = SQL_SUBQUERI the result must have the SQL_SQ_ COMPARISON bit set
supportsSubqueriesInExists	SQLGetInfo	fInfoType = SQL_SUBQUERI the result must have the SQL_SQ_EXISTS bit set
supportsSubqueriesInIns	SQLGetInfo	fInfoType = SQL_SUBQUERI the result must have the SQL_SQ_IN bit set
supportsSubqueriesIn Quantifieds	SQLGetInfo	fInfoType = SQL_SUBQUERI the result must have the SQL_SQ_ QUANTIFIED bit set
supportsCorrelatedSubqueries	SQLGetInfo	fInfoType = SQL_SUBQUERI the result must have the SQL_SQ_ CORRELATED_SUBQUERIES set
supportsUnion	SQLGetInfo	fInfoType = SQL_UNION; t result must have the SQL_U_UNION bit set
supportsUnionAll	SQLGetInfo	fInfoType = SQL_UNION; t result must have the

		SQL_U_UNION_ALL bit se
supportsOpenCursors Across Commit	SQLGetInfo	fInfoType = SQL_CURSOR_COMMIT_ BEHAVIOR; the result must SQL_CB_PRESERVE
supportsOpenCursors Across Rollback	SQLGetInfo	fInfoType = SQL_CURSOR ROLLBACK_BEHAVIOR; the r must be SQL_CB_PRESERV
supportsOpenStatements Across Commit	SQLGetInfo	fInfoType = SQL_CURSOR COMMIT_BEHAVIOR; the res must be SQL_CB_PRESERVE SQL_CB_CLOSE
supportsOpenStatements Across Rollback	SQLGetInfo	fInfoType = SQL_CURSOR ROLLBACK_BEHAVIOR; the r must be SQL_CB_PRESERVE SQL_CB_CLOSE
getMaxBinaryLiteralLength	SQLGetInfo	fInfoType = SQL_MAX_BINA LITERAL_LEN
getMaxCharLiteralLength	SQLGetInfo	fInfoType = SQL_MAX_CHA LITERAL_LEN
getMaxColumnNameLength	SQLGetInfo	fInfoType = SQL_MAX_COLU NAME_LEN
getMaxColumnsInGroupBy	SQLGetInfo	fInfoType = SQL_MAX_COLUMNS_IN_GROUP_BY
getMaxColumnsInIndex	SQLGetInfo	fInfoType = SQL_MAX_COLUMNS_IN_INDEX
getMaxColumnsInOrderBy	SQLGetInfo	fInfoType = SQL_MAX_COLUMNS_IN_ORDER_BY
getMaxColumnsInSelect	SQLGetInfo	fInfoType = SQL_MAX_COLUMNS_IN_SELECT
getMaxColumnsInTable	SQLGetInfo	fInfoType = SQL_MAX_COLUMNS_IN_TABLE
getMaxConnections	SQLGetInfo	fInfoType = SQL_ACTIVE_ CONNECTIONS

getMaxCursorNameLength	SQLGetInfo	fInfoType = SQL_MAX_CURS NAME_LEN
getMaxIndexLength	SQLGetInfo	fInfoType = SQL_MAX_INDEX_SIZE
getMaxSchemaNameLength	SQLGetInfo	fInfoType = SQL_MAX_OWN NAME_LEN
getMaxProcedureNameLength	SQLGetInfo	fInfoType = SQL_MAX_ PROCEDURE_NAME_LEN
getMaxCatalogNameLength	SQLGetInfo	fInfoType = SQL_MAX_ QUALIFIER_NAME_LEN
getMaxRowSize	SQLGetInfo	fInfoType = SQL_MAX_ROW_SIZE
doesMaxRowSizeIncludeBlobs	SQLGetInfo	fInfoType = SQL_MAX_ROW_SIZE_ INCLUDES_LONG
getMaxStatementLength	SQLGetInfo	fInfoType = SQL_MAX_ STATEMENT_LEN
getMaxStatements	SQLGetInfo	fInfoType = SQL_ACTIVE_ STATEMENTS
getMaxTableNameLength	SQLGetInfo	fInfoType = SQL_MAX_TABI NAME_LEN
getMaxTablesInSelect	SQLGetInfo	fInfoType = SQL_MAX_TABL IN_SELECT
getMaxUserNameLength	SQLGetInfo	fInfoType = SQL_MAX_USE NAME_LEN
getDefaultTransactionIsolation	SQLGetInfo	fInfoType = SQL_DEFAULT_T ISOLATION
supportsTransactions	SQLGetInfo	fInfoType = SQL_TXN_CAPAI the result must not be SQL_TC_NONE
supportsTransactionIsolation Level	SQLGetInfo	fInfoType = SQL_TXN_ISOLATION_ OPTION
supportsDataDefinitionAnd DataManipulationTransactions	SQLGetInfo	fInfoType = SQL_TXN_CAPAI the result must have the SQL_TC_ALL bit set
supportsDataManipulation TransactionsOnly	SQLGetInfo	fInfoType = SQL_TXN_CAPAI the result must have the SQL_TC_DML bit set

dataDefinitionCauses Transaction Commit	SQLGetInfo	fInfoType = SQL_TXN_CAPA the result must have the SQL_TC_DDL_COMMIT bit s
dataDefinition IgnoredIn Transactions	SQLGetInfo	fInfoType = SQL_TXN_CAPA the result must have the SQL_TC_DDL_IGNORE bit s
getProcedures	SQL Procedures	Returns a list of procedur names
getProcedureColumns	SQLProcedure Columns	Returns a list of input and ou parameters used for procedu
getTables	SQLTables	Returns a list of tables
getSchemas	SQLTables	Catalog = “”, Schema = “% Table = “”, TableType = NU only the TABLE_SCHEM colur returned
getCatalogs	SQLTables	Catalog = “%”, Schema = Table = “”, TableType = NU only the TABLE_CAT colum returned
getTableTypes	SQLTables	Catalog = “”, Schema = “”, T = “”, TableType = “%”
getColumns	SQLColumns	Returns a list of column na in specified tables
getColumnPrivileges	SQLColumn Privileges	Returns a list of columns a associated privileges for t specified table
getTablePrivileges	SQLTable Privileges	Returns a list of tables and privileges associated with e table
getBestRowIdentifier	SQLSpecial Columns	fColType = SQL_BEST_ROW
getVersionColumns	SQLSpecial Columns	fColType = SQL_ROWVEF
getPrimaryKeys	SQLPrimary Keys	Returns a list of column na that comprise the primary for a table
getImportedKeys	SQLForeign Keys	PKTableCatalog = NULL, PKTableSchema = NULL, PKTableName = NULL

getExportedKeys	SQLForeign Keys	FKTableCatalog = NULL,
		FKTableSchema = NULL,
		FKTableName = NULL

getCrossReference	SQLForeign Keys	Returns a list of foreign key the specified table
getTypeInfo	SQLGetType Info	fSqlType = SQL_ALL_TYPE

Returns a list of statistics about the specified table and the indexes associ with the table

Table 5.4 Statement ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
close	SQLFreeStmt	fOption = SQL_CLOSE
getMaxFieldSize	SQLGetStmtOption	fOption = SQL_MAX_LENGTH
setMaxFieldSize	SQLSetStmtOption	fOption = SQL_MAX_LENGTH
getMaxRows	SQLGetStmtOption	fOption = SQL_MAX_ROWS
setMaxRows	SQLSetStmtOption	fOption = SQL_MAX_ROWS
setEscapeProcessing	SQLSetStmtOption	fOption = SQL_NOSCAN
getQueryTimeout	SQLGetStmtOption	fOption = SQL_QUERY_TIMEOUT
setQueryTimeout	SQLSetStmtOption	fOption = SQL_QUERY_TIMEOUT
cancel	SQLCancel	Cancels the processing on a statement
setCursorName	SQLSetCursorName	Associates a cursor name with a statement
execute	SQLExecDirect	The Bridge checks for

		a SQL statement containing a ‘FOR UPDATE’ clause; if present, the cursor concurrency level for the statement is changed to SQL_CONCUR_LOCK
getUpdateCount	SQLRowCount	Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement
Determines whether there are more results available on a statement and, if so, initializes processing for those results		

Table 5.5 PreparedStatement ODBC calls.

JDBC Interface Method	**ODBC Call**	**Comments**
setNull	SQLBindParameter	fParamType = SQL_PARAM_INPUT; fSqlType = sqlType passed as parameter
setBoolean		
setByte		
setShort		
setInt		
setLong		
setFloat		
setDouble		
setNumeric		
setString		
setBytes		
setDate		
setTime		
setTimestamp	SQLBindParameter	fParamType = SQL_PARAM_INPUT;

		fSqlType is derived by the type of get method
setAsciiStream		
setUnicodeStream		
setBinaryStream	SQLBindParameter	fParamType = SQL_PARAM_INPUT, pcbValue = SQL_DATA_AT_EXEC
May return SQL_NEED_DATA (because of setAsciiStream, setUnicodeStream, or setBinary Stream); in this case, the Bridge will call SQLParamData and SQLPutData until no more data is needed		

Table 5.6 CallableStatement ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
fParamType = SQL_PARAM_OUTPUT; rgbValue is a buffer that has been allocated in Java; when using the getXXX methods, this buffer is used to retrieve the data		

Table 5.7 ResultSet ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
next	SQLFetch	Fetches a row of data from a ResultSet
close	SqlFreeStmt	fOption = SQL_CLOSE
getString		
getBoolean		
getByte		
getShort		
getInt		

getLong		
getFloat		
getDouble		
getNumeric		
getBytes		
getTime		
getTimestamp	SQLGetData	fCType is derived by the type of get method
getAsciiStream		
getUnicodeStream		
getBinaryStream	SQLGetData	An InputStream object is created to provide a wrapper around the SQLGetData call; data is read from the data source as needed

Returns the cursor name associated with the statement

Table 5.4 Statement ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
close	SQLFreeStmt	fOption = SQL_CLOSE
getMaxFieldSize	SQLGetStmtOption	fOption = SQL_MAX_LENGTH
setMaxFieldSize	SQLSetStmtOption	fOption = SQL_MAX_LENGTH
getMaxRows	SQLGetStmtOption	fOption = SQL_MAX_ROWS
setMaxRows	SQLSetStmtOption	fOption = SQL_MAX_ROWS
setEscapeProcessing	SQLSetStmtOption	fOption = SQL_NOSCAN
getQueryTimeout	SQLGetStmtOption	fOption = SQL_QUERY_TIMEOUT

setQueryTimeout	SQLSetStmtOption	fOption = SQL_QUERY_TIMEOUT
cancel	SQLCancel	Cancels the processing on a statement
setCursorName	SQLSetCursorName	Associates a cursor name with a statement
execute	SQLExecDirect	The Bridge checks for a SQL statement containing a 'FOR UPDATE' clause; if present, the cursor concurrency level for the statement is changed to SQL_CONCUR_LOCK
getUpdateCount	SQLRowCount	Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement

Determines whether there are more results available on a statement and, if so, initializes processing for those results

Table 5.5 PreparedStatement ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
setNull	SQLBindParameter	fParamType = SQL_PARAM_INPUT; fSqlType = sqlType passed as parameter
setBoolean		
setByte		
setShort		
setInt		
setLong		
setFloat		

setDouble		
setNumeric		
setString		
setBytes		
setDate		
setTime		
setTimestamp	SQLBindParameter	fParamType = SQL_PARAM_INPUT; fSqlType is derived by the type of get method
setAsciiStream		
setUnicodeStream		
setBinaryStream	SQLBindParameter	fParamType = SQL_PARAM_INPUT, pcbValue = SQL_DATA_AT_EXEC
May return SQL_NEED_DATA (because of setAsciiStream, setUnicodeStream, or setBinary Stream); in this case, the Bridge will call SQLParamData and SQLPutData until no more data is needed		

Table 5.6 CallableStatement ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**
fParamType = SQL_PARAM_OUTPUT; rgbValue is a buffer that has been allocated in Java; when using the getXXX methods, this buffer is used to retrieve the data		

Table 5.7 ResultSet ODBC calls. **JDBC Interface Method**	**ODBC Call**	**Comments**

next	SQLFetch	Fetches a row of data from a ResultSet
close	SqlFreeStmt	fOption = SQL_CLOSE
getString		
getBoolean		
getByte		
getShort		
getInt		
getLong		
getFloat		
getDouble		
getNumeric		
getBytes		
getTime		
getTimestamp	SQLGetData	fCType is derived by the type of get method
getAsciiStream		
getUnicodeStream		
getBinaryStream	SQLGetData	An InputStream object is created to provide a wrapper around the SQLGetData call; data is read from the data source as needed

Returns the cursor name associated with the statement

Chapter 6
SQL Data Types In Java And ORM

Many of the standard SQL-92 data types, such as Date, do not have a native Java equivalent. To overcome this deficiency, you must map SQL data types into Java. This process involves using JDBC classes to access SQL data types. In this chapter, we'll take a look at the classes in the JDBC that are used to access SQL data types. In addition, we'll briefly discuss the Object Relation Model (ORM), an interesting area in database development that attempts to map relational models into objects.

You need to know how to properly retrieve equivalent Java data types—like int, long, and String—from their SQL counterparts and store them in your database. This can be especially important if you are working with numeric data (which requires careful handling of decimal precision) and SQL timestamps (which have a well-defined format). The mechanism for handling raw binary data is touched on in this chapter, but it is covered in more detail in Chapter 8.

Mapping SQL Data To Java

Mapping Java data types into SQL is really quite simple. Table 6.1 shows how Java data types map into equivalent SQL data types. Note that the types beginning with *java.sql.* are not elemental data types, but are classes that have methods for translating the data into usable formats.

Table 6.1 Java data type mapping into SQL data types. **Java Type**	**SQL Type**
string	VARCHAR or LONGVARCHAR
java.sql.Numeric	NUMERIC
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	VARBINARY or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
TIMESTAMP	

The **byte[]** data type is a byte array of variable size. This data structure is used to store binary data; binary data is manifest in SQL as VARBINARY and LONGVARBINARY. These types are used to store images, raw document files, and so on. To store or retrieve this data from the database, you would use the stream methods available in the JDBC: **setBinaryStream** and **getBinaryStream**. In Chapter 8, we'll use these methods to build a multimedia Java/JDBC application.

Table 6.2 shows the mapping of SQL data types into Java. You will find that both tables will come in handy when you're attempting to decide which types need special treatment.

You can also use the tables as a quick reference to make sure that you're properly casting data that you want to store or retrieve.

Table 6.2 SQL data type mapping into Java and JDBC. **Java Type**	**SQL Type**
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.sql.Nueric
DECIMAL	java.sql.Numeric
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	souble
BINARY	byte[]
VARBINARY	byte[]
LONGBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
java.sql.Timestamp	

Now that you've seen how these data types translate from Java to SQL and vice versa, let's look at some of the methods that you'll use to retrieve data from a database. These methods, shown in Table 6.3, are contained in the **ResultSet** class, which is the class that is passed back when you invoke a **Statement.executeQuery** function. You'll find a complete reference of the **ResultSet** class methods in Chapter 12.

The parameters **int** and **String** allow you to specify the column you want by column number or column name.

Table 6.3 A few ResultSet methods for getting data. **Method**	**Description**

getAsciiStream(String), getAsciiStream(int)	Retrieves a column value as a stream of ASCII characters and then reads in chunks from the stream
getBinaryStream(int), getBinaryStream(String)	Retrieves a column value as a stream of uninterpreted bytes and then reads in chunks from the stream
getBoolean(int), getBoolean(String)	Returns the value of a column in the current row as a Java boolean
getDate(int), getDate(String)	Returns the value of a column in the current row as a java.sql.Date object
Returns the value of a column as a Java object	

ResultSetMetaData

One of the most useful classes you can use to retrieve data from a ResultSet is the **ResultSetMetaData** class. This class contains methods that allow you to obtain vital information about the query's result. After a query has been executed, you can call the **ResultSet.getMetaData** method to fetch a **ResultSetMetaData** object for the resulting data. Table 6.4 shows some of the methods that you will most likely use. Again, more **ResultSetMetaData** methods are listed in Chapter 12.

Table 6.4 Handy methods in the ResultSetMetaData class. **Method**	**Description**
getColumnCount()	Indicates the number of columns in the ResultSet
getColumnLabel(int)	Returns the database-assigned Label for the column at position int in the ResultSet
getColumnName(int)	Returns the column's name (for query reference)
getColumnType(int)	Returns the specified column's SQL type
isNullable(int)	Tells you if the specified column can

	contain NULLs
Indicates whether the specified column is searchable via a WHERE clause	

Understanding The Object Relation Model

The Object Relation Model (ORM) attempts to fuse object-orientation with the relational database model. Because many of today's programming languages, such as Java, are object-oriented, a tighter integration between the two would provide easier abstraction for developers who program in object-oriented languages and also are required to "program" in SQL. Such an integration would also relieve the necessity of constant translation between database tables and object-oriented data structures, which can be an arduous task.

Mapping A Table Into A Java Object

Let's look at a simple example to demonstrate the basics of ORM. Suppose we create the following table in a database:

First_Name	Last_Name	Phone_Number	Employee_Number
Pratik	Patel	800-555-1212	30122
Karl	Moss	800-555-1213	30124
Keith	Weiskamp	800-555-1214	09249
Ron	Pronk	800-555-1215	10464

You can easily map this table into a Java object. Here's the Java code you would write to encapsulate the data contained in the table:

```
class Employee {
int Key;
String First_Name;
String Last_Name;
String Phone_Number;
int Employee_Number;
Key=Employee_Number;
}
```

To retrieve this table from the database into Java, we simply assign the respective columns to the **Employee** object we created previously for each row we retrieve, as shown here:

```
Employee emp_object = new Employee();
emp_object.First_Name= resultset.getString("First_Name");
emp_object.Last_Name= resultset.getString("Last_Name");
emp_object.Phone_Number=resultset.getString("Phone_Number");
emp_object.Employee_Number=resultset.getInt("Employee_Number");
```

With a larger database model (with links between tables), a number of problems can arise, including scalability due to multiple JOINs in the data model and cross-linking of table keys. Fortunately, a number of products are already available that allow you to create these kinds of object-oriented/relational bridges. Moreover, there are several solutions being developed to work specifically with Java.

I've given you an idea of what ORM is all about. If you would like to investigate this topic further, check out The Coriolis Group Web site (http://www.coriolis.com/jdbc-book) for links to ORM vendors and some really informative ORM documents. The ODMG (Object Database Management Group) is a consortium that is working on a revised standard for object database technology and the incorporation of this concept into programming languages such as Java. A link to the consortium's Web site can be found on The Coriolis Group Web site as well.

Summary

As you can see from this brief chapter, mapping SQL data types to Java is truly a snap. We covered a few of the more important methods you will use to retrieve data from a database. For a complete reference, see Chapter 12 and have a look at the Date, Time, TimeStamp, Types, and Numeric classes.

The next chapter steps back from the JDBC to look at ways of presenting your data in Java. Using Java packages available on the Net, we'll cover graphs, tables, and more. We'll also discuss some nifty methods in the JDBC that will help streamline your code for retrieving data from your database.

Chapter 7
Working With Query Results

So far, we've been concentrating on how to use the classes in the JDBC to perform SQL queries. That's great, but now we have to do something with the data we've retrieved. The end user of your JDBC applets or applications will want to see more than just rows and rows of data. In this chapter, we'll learn how to package the raw table data that is returned after a successful SQL query into a Java object, and then how to use this packaged data to produce easy-to-read graphs.

The first issue we'll look at is creating a Java object to store the results of a query. This object will provide a usable interface to the actual query results so they can be plugged into a Java graphics library. We'll create a simple data structure to hold the column results in a formatted way so that we can easily parse them and prepare them for display. Second, we'll look at taking these results in the Java object and setting up the necessary code to plug the data into a pie chart and bar graph Java package.

In the next chapter, we'll go one step further and work with BLOB data types (like images). Between these chapters, I will be providing plenty of examples, complete with code, to help you work up your own JDBC programs. At the very least, these chapters will give you some ideas for dealing with raw table data and displaying it in an effective manner.

A Basic Java Object For Storing Results

Although the JDBC provides you with the **ResultSet** class to get the data from an SQL query, you will still need to store and format within your program the results for display. The smart way to do this is in a re-usable fashion (by implementing a generic object or class) which allows you to re-use the same class you develop to retrieve data from a query in any of your JDBC programs. The code snippet in Listing 7.1 is a method that will keep your results in a Java object until you are ready to parse and display it.

Let's begin by defining the data we will be getting from the source, and determining how we want to structure it within our Java applet. Remember that the **ResultSet** allows us to retrieve data in a row-by-row, column-by-column fashion; it simply gives us sequential access to the resulting table data. Table 7.1 shows the example table we will be using in this chapter.

Table 7.1 Example table. **emp_no**	**first_name**	**last_name**	**salary**
01234	Pratik	Patel	8000
1235	Karl	Moss	23000
0002	Keith	Weiskamp	90000
0045	Ron	Pronk	59999
53000			

The optimal way to store this data in our Java program is to put each column's data in its own structure and then link the different columns by using an index; this will allow us to keep the columnar relationship of the table intact. We will put each column's data in an array. To simplify matters, we'll use the **getString** method, which translates the different data types returned by a query into a String type. Then, we'll take the data in a column and delimit the instances with commas. We'll use an array of String to do this; each place in the array will represent a different column. The data object we will create is shown here:

```
table_data[0] => 01234,1235,0002,0045,0067
table_data[1] => Pratik,Karl,Keith,Ron,David
table_data[2] => Patel,Moss,Weiskamp,Pronk,Friedel
table_data[3] => 8000,23000,90000,59999,53000
```

Listing 7.1 shows the method we'll use to query the database and return a String array that contains the resulting table data.

Listing 7.1 The getData method.

```
public String[] getData( String QueryLine ) {
// Run the QueryLine SQL query, and return the resulting columns in an
// array of String. The first column is at index [0], the second at [1],
// etc.
```

```
  int columns, pos;
  String column[]=new String[4];
// We have to initialize the column String variable even though we re-
// declare it below. The reason is because the declaration below is in a
// try{} statement, and the compiler will complain that the variable may
// not be initialized.

  boolean more;

 try {

      Statement stmt = con.createStatement();
        // Create a Statement object from the
        // Connection.createStatement method.

      ResultSet rs = stmt.executeQuery(QueryLine);
        // Execute the passed in query, and get
        // the ResultSet for the query.

      columns=(rs.getMetaData()).getColumnCount();
        // Get the number of columns in the resulting table so we can
        // declare the column String array, and so we can loop
        // through the results and retrieve them.

      column = new String[columns];
        // Create the column variable to be the exact number of
        // columns that are in the result table.
        // Initialize the column array to be blank since we'll be adding
        // directly to them later.

for(pos=1; pos<=columns; pos++) {
        column[pos-1]="";
      }

      more=rs.next();
        // Get the first row of the ResultSet. Loop through the
ResultSet
        // and get the data, row-by-row, column-by-column.
      while(more) {

        for (pos=1; pos<=columns; pos++) {
          column[pos-1]+=(rs.getString(pos));
            // Add each column to the respective column[] String array.
        }

        more=rs.next();
          // Get the next row of the result if it exists.

          // Now add a comma to each array element to delimit this row
is
          // done.
        for (pos=1; pos<=columns; pos++) {
          if(more) {
          // We only want to do this if this isn't the last row of the
          // table!
            column[pos-1]+=(",");
          }
        }
      }
      stmt.close();
        // All done. Close the statement object.
    }
    catch( Exception e ) {
      e.printStackTrace();
      System.out.println(e.getMessage());
    }
return column;
// Finally, return the entire column[] array.
}
```

Showing The Results

Now that we have the data nicely packaged into our Java object, how do we show it? The code in Listing 7.2 dumps the data in the object to the screen. We simply loop through the array and print the data.

Listing 7.2 Code to print retrieved data to the console.

```
public void ShowFormattedData(String[] columnD ) {

int i;

for ( i=0; i< columnD.length; i++) {
      System.out.println(columnD[i]+"\n");
  }
}
```

Charting Your Data

Now that we've covered the preliminaries, we are ready to get to the fun stuff! Instead of creating a package that has graphics utilities, we're going to use the NetCharts library, which is stored on the accompanying CD-ROM. The package on the CD is only an evaluation copy. Stop by http://www.netcharts.com to pick up the latest version (and some helpful documentation). We'll use the table in Table 7.1 and a bar chart to display the salary information for our fictional company. Figure 7.1 shows the applet that is generated from the code in Listing 7.3. Remember, the code for this example can be found on the accompanying CD-ROM, or at The Coriolis Group Web site at http://www.coriolis.com/jdbc-book.

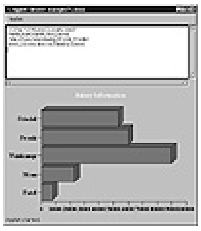


Figure 7.1 The bar chart applet.

Listing 7.3 Dynamically generating a bar chart from a database query—Part I.

```
/*
```

```
Example 7-1
*/
import java.awt.*;
import java.applet.Applet;
import java.sql.*;

public class example71 extends java.applet.Applet {
   String url;
  String Name;
  Connection con;
  TextArea OutputField = new TextArea(10,35);
  NFBarchartApp bar;
  // This is the bar chart class from the NetCharts package

public void init() {
  setLayout(new BorderLayout());
    url="jdbc:msql://elanor/jdbctest";
  // The URL for the database we wish to connect to

 ConnectToDB();
 // Connect to the database.

  add("North", OutputField);
 // Add the TextArea for showing the data to the user
  String columnData[] = getData("select * from Employee");
 // Run a query that goes and gets the complete table listing; we can
put
 // any query here and would optimally want to get only the columns we
 // need.

  ShowFormattedData(columnData);
 // Show the data in the TextArea
  ShowChartData(columnData[3],columnData[2]);

// Now, pass the two data sets and create a bar chart
  add("Center", bar);
// And add the bar chart to the applet's panel
}

public void ShowFormattedData(String[] columnD ) {

int i;

for ( i=0; i< columnD.length; i++) {
  OutputField.appendText(columnD[i]+"\n");
}

}

public void ConnectToDB() {


  try {
    new imaginary.sql.iMsqlDriver();
    con = DriverManager.getConnection(url, "prpatel", "");
  }
  catch( Exception e ) {
    e.printStackTrace();
    System.out.println(e.getMessage());
  }

}

public void ShowChartData(String Data1, String Data2) {
try {
                bar = new NFBarchartApp(this);
                  // Instantiate the bar chart class

                bar.init();
                bar.start();
                  // Initialize it, and start it running.

                  // Below is where we load the parameters for the chart.
                  // See the documentation at the NetCharts Web site, or
                  // the CD-ROM for details.

                bar.loadParams(
                        "Header     = ('Salary Information');"+
                        "DataSets   = ('Salary', red);"+
                        "DataSet1   = "+ Data1 + ";"+
                        "BarLabels  = "+ Data2 + ";"+
                        "GraphLayout= HORIZONTAL;"+
                        "BottomAxis = (black, 'TimesRoman', 14, 0,
                        0,100000)"
                        );

                bar.loadParams ("Update");
                  // Tell the bar chart class we've put
                  // some new parameters in.

        } catch (Exception e) {
                System.out.println (e.getMessage());
        }

} // More to come following some comments…
```

The bar chart class from the NetCharts package uses a method to load the values for the chart. We have to define the labels and corresponding values, but this is generally straightforward. Because our data is formatted in a comma-delimited fashion, we don't have to parse the data again to prepare it for use. In the next example (the pie chart example), we do have to parse it to put it in the proper format for the charting class to recognize it. Listing 7.4 picks up the code where we left off in Listing 7.3.

Listing 7.4 Dynamically generating a bar chart from a database query—Part II.

```
public String[] getData( String QueryLine ) {

  int columns, pos;
  String column[]=new String[4];
  boolean more;

 try {
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(QueryLine);
      columns=(rs.getMetaData()).getColumnCount();
      column = new String[columns];

      // Initialize the columns to be blank
      for(pos=1; pos<=columns; pos++) {
        column[pos-1]="";
      }

      more=rs.next();

      while(more) {


        for (pos=1; pos<=columns; pos++) {
          column[pos-1]+=(rs.getString(pos));
        }

        more=rs.next();
        for (pos=1; pos<=columns; pos++) {
          if(more) {
            column[pos-1]+=(",");
          }
        }
      }
      stmt.close();

    }
    catch( Exception e ) {
      e.printStackTrace();
      System.out.println(e.getMessage());
    }

return column;
}
```

That’s it! We’ve successfully queried a database, formatted the resulting data, and created a bar chart to present a visual representation of the results. Listing 7.5 shows the generation of a pie chart, and Figure 7.2 shows the pie chart applet.

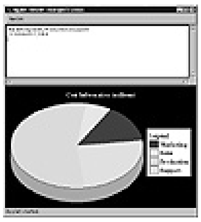


Figure 7.2 The pie chart applet.

Listing 7.5 Dynamically generating a pie chart from a database query.

```
/*
Example 7-2: Pie chart
*/
```

```
import java.awt.*;
import java.applet.Applet;
import java.sql.*;
import java.util.StringTokenizer;

public class example72 extends java.applet.Applet {
   String url;
  String Name;
  Connection con;
  TextArea OutputField = new TextArea(10,35);
  NFPiechartApp pie;

public void init() {
  setLayout(new BorderLayout());
  url="jdbc:msql://elanor/jdbctest";
  pie = new NFPiechartApp(this);

  ConnectToDB();

  add("North", OutputField);
  String columnData[] = getData("select * from Cost");

  ShowFormattedData(columnData);
  ShowChartData(columnData[1],columnData[0]);
  add("Center", pie);

}

public void ConnectToDB() {


  try {
    new imaginary.sql.iMsqlDriver();
    con = DriverManager.getConnection(url, "prpatel", "");
  }
  catch( Exception e ) {
    e.printStackTrace();
    System.out.println(e.getMessage());
  }

}

public void ShowFormattedData(String[] columnD ) {

int i;

for ( i=0; i< columnD.length; i++) {
  OutputField.appendText(columnD[i]+"\n");
  }

}

public void ShowChartData(String dataNumber, String dataLabel) {

StringTokenizer nData, lData;
String SliceData = "";
ColorGenerator colorGen = new ColorGenerator();
```

```
// We need to assign colors to the pie slices automatically, so we use a
// class that cycles through colors. See this class defined below.

nData = new StringTokenizer(dataNumber, ",");
lData = new StringTokenizer(dataLabel, ",");
// We used our preformatted column data, and need to break it down to
the
// elements. We use the StringTokenizer to break the column string data
// individual down by commas we inserted when we created the data.

// We assume that dataNumber and dataLabel have the same number of
// elements since we just generated them from the getData method.

while(nData.hasMoreTokens()) {
// Loop through the dataNumber and dataLabel and build the slice data:
// ( 1234, darkBlue, "Label" ). This is what the pie chart class expects,
// so we must parse our data and put it in this format.

SliceData += "("+nData.nextToken() + ", "
                + colorGen.next() + ", '"
                + lData.nextToken() + "', green)";

System.out.println(SliceData);
if (nData.hasMoreTokens()) {SliceData += ", ";}
}

try {
                  // We already instantiated the pie chart
                  // class(NFPieChartAPP) at the top of the applet.
                pie.init();
                pie.start();
                  // Initialize and start the pie chart class.

                pie.loadParams(
                "Background=(black, RAISED, 4);"+
                "Header=('Cost Information (millions)');"+
                "LabelPos=0.7;"+
                "DwellLabel      = ('', black, 'TimesRoman', 16);"+
                "Legend        = ('Legend', black);"+
                "LegendBox     = (white, RAISED, 4);"+
                "Slices=(12.3, blue, 'Marketing', cyan), (4.6,
                antiquewhite, 'Sales'), (40.1, aqua, 'Production'),
                (18.4, aquamarine, 'Support');");

                  // Above, we set the parameters for the pie chart,
                  // including the data and  labels which we generated
                  // in the loop above ( SliceData ), and the Legend,
                  // label position, header, and other properties.
                  // Again, have a look at the NetCharts documentation
                  // for all of the possible parameters.

                pie.loadParams ("Update");
                  // Tell the pie chart class we've sent it new
                  // parameters to display.
        } catch (Exception e) {
                System.out.println (e.getMessage());
        }
}

// Below is the same as before except for the new ColorGenerator class
// that we needed to produce distinct colors.

public String[] getData( String QueryLine ) {

  int columns, pos;
  String column[]=new String[4];
  boolean more;

 try {

      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(QueryLine);
      columns=(rs.getMetaData()).getColumnCount();

      column = new String[columns];

      // Initialize the columns to be blank
      for(pos=1; pos<=columns; pos++) {
        column[pos-1]="";
      }

      more=rs.next();

      while(more) {for (pos=1; pos<=columns; pos++) {
          column[pos-1]+=(rs.getString(pos));
        }

        more=rs.next();
        for (pos=1; pos<=columns; pos++) {
          if(more) {
            column[pos-1]+=(",");
          }
        }
      }
      stmt.close();
      // con.close();
    }
    catch( Exception e ) {
      e.printStackTrace();
      System.out.println(e.getMessage());
    }

return column;
}
public void destroy() {

  try {con.close();}
  catch( Exception e ) {
      e.printStackTrace();
      System.out.println(e.getMessage());
    }
}
}
```

```
class ColorGenerator {
// This class is needed to produce colors that the pie chart can use to
// color the slices properly. They are taken from the NetCharts color
// class, NFColor.
public ColorGenerator() {

}

int color_count = -1;
// Keep a running count of the colors we have used. We'll simply index
// the colors in a String array, and call up the incremented counter to
// get a new color. If you need more colors than are added below, you
can
// add more by pulling them from the NFColor class found in the
NetCharts
// package on the CD-ROM or Web site.

String colors[] =
{"aliceblue","antiquewhite","aqua","aquamarine","azure","beige",
"bisque","black","blanchedalmond","blue","blueviolet","brown","chocolate
",
"cadetblue","chartreuse","cornsilk","crimson","cyan"};

public String next() {

// Increment the color counter, and return a String which contains the
// color at this index.
  color_count += 1;
  return colors[color_count];

}

} // end example72.java
```

Summary

This chapter has shown you how to generate meaningful charts to represent data obtained from a query. We've seen how to create both bar and pie charts. You can use the properties of the NetCharts package to customize your charts as you wish, and there are many more options in the package that haven't been shown in the examples here.

In the next chapter, we will continue to discuss working with database query results, and we will provide a complete code example for showing SQL BLOB data types. It shows you how to get an image from the **ResultSet**, as well as how to add images or binary data to a table in a database.

Chapter 8
The IconStore Multimedia JDBC Application

In the previous chapter, we learned how to process query results with JDBC. In this chapter, we'll take these query results and put them to use in a multimedia application. The application we'll be developing, IconStore, will connect to a database, query for image data stored in database tables, and display the images on a canvas. It's all very simple, and it puts the JDBC to good use by building a dynamic application totally driven by data stored in tables.

IconStore Requirements

The IconStore application will utilize two database tables: ICONCATEGORY and ICONSTORE. The ICONCATEGORY table contains information about image categories, which can be items like printers, sports, and tools. The ICONSTORE table contains information about each image. Tables 8.1 and 8.2 show the database tables' underlying data structures.

Note that the CATEGORY column in the ICONSTORE is a foreign key into the ICONCATEGORY table. If the category ID for sports is "1", you can obtain a result set containing all of the sports images by using this statement:

```
SELECT ID, DESCRIPTION, ICON FROM ICONSTORE WHERE CATEGORY = 1
```

Table 8.1 The ICONCATEGORY table. **Column Name**	**SQL Type**	**Description**
CATEGORY	INTEGER	Category ID
Description of the image category		

Table 8.2 The ICONSTORE table. **Column Name**	**SQL Type**	**Description**
ID	INTEGER	Image ID
DESCRIPTION	VARCHAR	Description of the image
CATEGORY	INTEGER	Category ID
Binary image		

Now, let's take a look at what's going on in the application:

- An Icons menu, which is dynamically created by the ICONCATEGORY table, contains each of the image categories as an option. The user can select an image category from this menu to display the proper list of image descriptions in a list box. The ICONSTORE table is used to dynamically build the list.

- The user can select an image description from the list box to display the corresponding image.
- Once an image has been displayed, the user can select the Save As menu option to save the image to disk.

As you can see, IconStore will not be too complicated, but it will serve as a very good foundation for developing database-driven applications.

Building The Database

Now that we've established the application's requirements, we need to build the underlying database. We'll look at a simple JDBC application to accomplish this, although it may be created by any number of methods. Listing 8.1 shows the BuildDB.java source code. This application uses the SimpleText JDBC driver (covered in great detail in Chapter 10) to create the ICONCATEGORY and ICONSTORE tables, but any JDBC driver can be used in its place.

Listing 8.1 Building the IconStore database.

```
import java.sql.*;
import java.io.*;

class BuildDB {
//————————————————————————————————————
// main
//————————————————————————————————————
public static void main(String args[]) {
    try {
        // Create an instance of the driver
        java.sql.Driver d = (java.sql.Driver) Class.forName (
                    "jdbc.SimpleText.SimpleTextDriver").newInstance();

        // Properties for the driver
        java.util.Properties prop = new java.util.Properties();

      // URL to use to connect
        String url = "jdbc:SimpleText";

        // The only property supported by the SimpleText driver
        // is "Directory."
        prop.put("Directory", "/java/IconStore");

        // Connect to the SimpleText driver
        Connection con = DriverManager.getConnection(url, prop);

        // Create the category table
        buildCategory(con, "IconCategory");

        // Create the IconStore table
        buildIconStore(con, "IconStore");
        // Close the connection
        con.close();
    }
    catch (SQLException ex) {
        System.out.println("\n*** SQLException caught ***\n");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException ();
        }
        System.out.println("");
    }
    catch (java.lang.Exception ex) {
        ex.printStackTrace ();
    }
}
//————————————————————————————————————
// BuildCategory
// Given a connection object and a table name, create the IconStore
// category database table.
//————————————————————————————————————
protected static void buildCategory(
    Connection con,
    String table)
    throws SQLException
{
    System.out.println("Creating " + table);
    Statement stmt = con.createStatement();
    // Create the SQL statement
    String sql = "create table " + table +
            " (CATEGORY NUMBER, DESCRIPTION VARCHAR)";

    // Create the table
    stmt.executeUpdate(sql);

    // Create some data using the statement
    stmt.executeUpdate("INSERT INTO " + table + " VALUES (1,
      'Printers')");
    stmt.executeUpdate("INSERT INTO " + table + " VALUES (2,
'Sports')");
    stmt.executeUpdate("INSERT INTO " + table + " VALUES (3, 'Tools')");
}
//————————————————————————————————————
// BuildIconStore
// Given a connection object and a table name, create the IconStore
// icon database table.
//————————————————————————————————————
protected static void buildIconStore(
    Connection con,
    String table)
    throws SQLException
{

    System.out.println("Creating " + table);

    Statement stmt = con.createStatement();
    // Create the SQL statement
    String sql = "create table " + table +
            " (ID NUMBER, DESCRIPTION VARCHAR, CATEGORY NUMBER, ICON
              BINARY)";

    // Create the table
    stmt.executeUpdate(sql);
    stmt.close();

    // Create some data using a prepared statement
    sql = "insert into " + table + " values(?,?,?,?)";
    FileInputStream file;
    PreparedStatement ps = con.prepareStatement(sql);

    int category;
    int id = 1;

    // Add the printer icons
    category = 1;

    addIconRecord(ps, id++, "Printer 1", category,
"printers/print.gif");
    addIconRecord(ps, id++, "Printer 2", category,
"printers/print0.gif");

    // Add the sports icons
    category = 2;

    addIconRecord(ps, id++, "Archery", category, "sports/
      sport_archery.gif");
    addIconRecord(ps, id++, "Baseball", category, "sports/
      sport_baseball.gif");

    // Add the tools
    category = 3;

    addIconRecord(ps, id++, "Toolbox 1", category, "tools/toolbox.gif");
    addIconRecord(ps, id++, "Toolbox 2", category,
"tools/toolbox1.gif");
    ps.close();
}

//————————————————————————————————————
// AddIconRecord
// Helper method to add an IconStore record. A PreparedStatement is
// provided to which this method binds input parameters. Returns
// true if the record was added.
//————————————————————————————————————
protected static boolean addIconRecord(
    PreparedStatement ps,
    int id,
    String desc,
    int category,
    String filename)
    throws SQLException
{
    // Create a file object for the icon
    File file = new File(filename);
    if (!file.exists()) {
        return false;
    }

    // Get the length of the file. This will be used when binding
    // the InputStream to the PreparedStatement.
    int len = (int) file.length();

    FileInputStream inputStream;

    try {

        // Attempt to create an InputStream from the File object
        inputStream = new FileInputStream (filename);
    }
    catch (Exception ex) {

            // Some type of failure. Convert it into a SQLException.
            throw new SQLException (ex.getMessage ());
        }

        // Set the parameters
        ps.setInt(1, id);
        ps.setString(2, desc);
        ps.setInt(3,category);
        ps.setBinaryStream(4, inputStream, len);

        // Now execute
        int rows = ps.executeUpdate();
        return (rows == 0) ? false : true;
    }
}
```

The BuildDB application connects to the SimpleText JDBC driver, creates the ICONCATEGORY table, adds some image category records, creates the ICONSTORE table, and adds some image records. Note that when the image records are added to the ICONSTORE table, a **PreparedStatement** object is used. We'll take a closer look at **PreparedStatements** in Chapter 11; for now, just realize that this is an efficient way to execute the same SQL statement multiple times with different parameters values. Also note that the image data is coming out of GIF files stored on disk. An **InputStream** is created using these files, which is then passed to the JDBC driver for input. The JDBC driver reads the **InputStream** and stores the binary data in the database table. Simple, isn't it? Now that we've created the database, we can start writing our IconStore application.

Application Essentials

The source code for the IconStore application is shown throughout the rest of this chapter, broken across the various sections. As always, you can pick up a complete copy of the source code on the CD-ROM. Remember, you need to have the SimpleText JDBC driver installed before using the IconStore application. See Chapter 3, if you have trouble getting the application to run.

Writing The main Method

Every JDBC application must have an entry point, or a place at which to start execution. This entry point is the **main** method, which is shown in Listing 8.2. For the IconStore application, **main** simply processes any command line arguments, creates a new instance of the **IconStore** class (which extends **Frame**, a top-level window class), and sets up the window attributes. The IconStore application accepts one command line argument: the location of the IconStore database. The default location is /IconStore.

Listing 8.2 IconStore main method.

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

public class IconStore
        extends Frame
{
    IconCanvas      imageCanvas;
     List            iconList;
     Panel           iconListPanel;
    MenuBar         menuBar;
     Menu            fileMenu;
    Menu            sectionMenu;
     List            lists[];

     static String myHome = "/IconStore";
    Connection      connection;
    Hashtable       categories;
    Hashtable       iconDesc[];
    String          currentList;
    String          currentFile = null;
     FileDialog      fileDialog;
    //————————————————————————————————————
    // main
    //————————————————————————————————————
    public static void main (String[] args) {

        // If an argument was given, assume it is the location of the
      // database.
        if (args.length > 0) {
            myHome = args[0].trim();

            // If there is a trailing separator, remove it
            if (myHome.endsWith("/") ||
                myHome.endsWith("\\")) {
                myHome = myHome.substring(0, myHome.length() - 1);
            }
        }

        // Create our IconStore object
        IconStore frame = new IconStore();

        // Setup and display
         frame.setTitle("The IconStore");
        frame.init();

        frame.pack();
         frame.resize(300, 400);
        frame.show();
    }
```

A lot of work is being performed in **IconStore.init**, such as establishing the database connection, reading the icon categories, creating the menus, and reading the icon descriptions. We'll take a look at each of these in greater detail in the following sections.

Establishing The Database Connection

Listing 8.3 shows the code used by the IconStore application to connect to the SimpleText JDBC driver.

Listing 8.3 Establishing the database connection.

```
public Connection establishConnection()
    {

         Connection con = null;
         try {
              // Create an instance of the driver
                 java.sql.Driver d = (java.sql.Driver) Class.forName (

"jdbc.SimpleText.SimpleTextDriver").newInstance();

             // Properties for the driver
                java.util.Properties prop = new java.util.Properties();

          // URL to use to connect
             String url = "jdbc:SimpleText";

             // Set the location of the database tables
              prop.put("Directory", myHome);

             // Connect to the SimpleText driver
              con = DriverManager.getConnection(url, prop);
    }
    catch (SQLException ex) {

         // An SQLException was generated. Dump the exception
             // contents. Note that there may be multiple SQLExceptions
             // chainedtogether.

             System.out.println("\n*** SQLException caught ***\n");
            while (ex != null) {
                   System.out.println("SQLState: " + ex.getSQLState());
                 System.out.println("Message:  " + ex.getMessage());
                 System.out.println("Vendor:   " + ex.getErrorCode());
               ex = ex.getNextException();
         }
```

```
            System.exit(1);
    }
        catch (java.lang.Exception ex) {
            ex.printStackTrace();
           System.exit(1);
    }
      return con;
}
```

Note that we need to set a property for the SimpleText driver to specify the location of the database tables. In reality, the SimpleText driver stores each database table as a file, and the **Directory** property specifies the directory in which these files are kept. As I mentioned in the previous section, the default location is /IconStore (the IconStore directory of your current drive), but this can be overridden to be any location.

If successful, a JDBC **Connection** object is returned to the caller. If there is any reason a database connection cannot be established, the pertinent information will be displayed and the application will be terminated.

Creating The Menu

One of the requirements for the IconStore application is the ability to dynamically build the Icons menu. To do this, we'll need to query the ICONCATEGORY table and build the menu from the results. First, we need to read the database table and store the query results, as shown in Listing 8.4.

Listing 8.4 Reading the ICONCATEGORY table.

```
//————————————————————————————————————
// getCategories
// Read the IconStore CATEGORY table and create a Hashtable containing
// a list of all the categories. The key is the category description and
// the data value is the category ID.
//————————————————————————————————————
public Hashtable getCategories(
    Connection con)
{
    Hashtable table = new Hashtable();

    try {
        // Create a Statement object
        Statement stmt = con.createStatement();

        // Execute the query and process the results
        ResultSet rs = stmt.executeQuery(
                "SELECT DESCRIPTION,CATEGORY FROM ICONCATEGORY");

        // Loop while more rows exist
          while (rs.next()) {
              // Put the description and id in the Hashtable
                   table.put(rs.getString(1), rs.getString(2));
        }
        // Close the statement
```

```
            stmt.close();
      }
      catch (SQLException ex) {

          // An SQLException was generated. Dump the exception contents.
          // Note that there may be multiple SQLExceptions chained
          // together.

           System.out.println("\n*** SQLException caught ***\n");
           while (ex != null) {
                  System.out.println("SQLState: " + ex.getSQLState());
                 System.out.println("Message:  " + ex.getMessage());
              System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException();
       }
            System.exit(1);
   }

      return table;
}
```

The flow of this routine is very basic, and we'll be using it throughout our IconStore application. First, we create a **Statement** object; then, we submit an SQL statement to query the database; next, we process each of the resulting rows; and finally, we close the **Statement**. Note that a **Hashtable** object containing a list of all the categories is returned; the category description is the key and the category ID is the element. In this way, we can easily cross-reference a category description to an ID. We'll see why this is necessary a bit later.

Now that all of the category information has been loaded, we can create our menu. Listing 8.5 shows how this is done.

Listing 8.5 Creating the Icons menu.

```
// Get a Hashtable containing an entry for each icon category.
// The key is the description and the data value is the
// category number.

categories = getCategories(connection);

// File menu
fileMenu = new Menu("File");
fileMenu.add(new MenuItem("Save As"));
fileMenu.add(new MenuItem("Exit"));
menuBar.add(fileMenu);

// Icons menu
sectionMenu = new Menu("Icons");

Enumeration e = categories.keys();
    int listNo = 0;
    String desc;

// Loop while there are more keys (category descriptions)
while (e.hasMoreElements()) {
    desc = (String) e.nextElement();
    // Add the description to the Icons menu
    sectionMenu.add(new MenuItem(desc));
}

// Add the Icons menu to the menu bar
menuBar.add(sectionMenu);

// Set the menu bar
setMenuBar(menuBar);
```

Notice that the **Hashtable** containing a list of the image categories is used to create our menu. The only way to examine the contents of a **Hashtable** without knowing each of the keys is to create an **Enumeration** object, which can be used to get the next key value of the **Hashtable**. Figure 8.1 shows our database-driven menu.

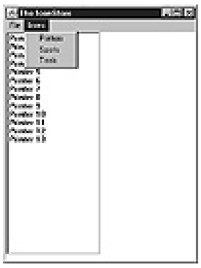


Figure 8.1 The IconStore menu.

Creating The Lists

Next on our agenda: creating the list boxes containing the image descriptions. We'll create a list for each category, so when the user selects a category from the Icons menu, only a list of the images for the selected category will be shown. We'll use a **CardLayout** to do this, which is a nifty way to set up any number of lists and switch between them effortlessly. For each of the categories that we read from the ICONCATEGORY table, we also read each of the image descriptions for that category from the ICONSTORE table and store those descriptions in a **Hashtable** for use later. At the same time, we add each description to a list for the category. Listing 8.6 shows the code used to read the ICONSTORE table.

Listing 8.6 Reading the ICONSTORE table.

```
//————————————————————————————————————
// getIconDesc
// Read the IconStore ICONSTORE table and create a Hashtable
// a list of all the icons for the given category. The key is the
// icon containing description and the data value is the icon ID. The
// description is also added to the List object given.
//————————————————————————————————————
public Hashtable getIconDesc(
    Connection con,
    String category,
    List list)
{
    Hashtable table = new Hashtable();
    String desc;

    try {
        // Create a Statement object
         Statement stmt = con.createStatement();

        // Execute the query and process the results
         ResultSet rs = stmt.executeQuery(
                  "SELECT DESCRIPTION,ID FROM ICONSTORE WHERE CATEGORY="
+

                  category);

        // Loop while more rows exist
          while (rs.next()) {
            desc = rs.getString(1);

           // Put the description and ID in the Hashtable
             table.put(desc, rs.getString(2));

           // Put the description in the list
            list.addItem(desc);

    }
    // Close the statement
      stmt.close();

}
catch (SQLException ex) {

          // An SQLException was generated. Dump the exception contents.
          // Note that there may be multiple SQLExceptions chained
          // together.
            System.out.println("\n*** SQLException caught ***\n");
           while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
           System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException();
        }
        System.exit(1);
   }

   return table;
}
```

The process we used here is the same as we have seen before—creating a **Statement**, executing a query, processing the results, and closing the **Statement**. Listing 8.7 shows the entire code for the **IconStore.init** method. In addition to building the menu, we also build the **CardLayout**. It is important to note that the IconStore application is totally database-driven; no code will have to be modified to add or remove categories or images.

Listing 8.7 IconStore init method.

```
//————————————————————————————————————————
// init
// Initialize the IconStore object. This includes reading the
// IconStore database for the icon descriptions.
//------------------------------------
public void init()
{
    // Create our canvas that will be used to display the icons
    imageCanvas = new IconCanvas();

    // Establish a connection to the JDBC driver
     connection = establishConnection();

    // Get a Hashtable containing an entry for each icon category.
    // The key is the description and the data value is the
   // category number.
    categories = getCategories(connection);

    // Setup the menu bar
    menuBar = new MenuBar();

    // File menu
     fileMenu = new Menu("File");
     fileMenu.add(new MenuItem("Save As"));
      fileMenu.add(new MenuItem("Exit"));
     menuBar.add(fileMenu);

    // Icons menu
    sectionMenu = new Menu("Icons");

     // Setup our category lists, list panel (using a CardLayout), and
   // icon menu.
    iconListPanel = new Panel();
      iconListPanel.setLayout(new CardLayout());

    lists = new List[categories.size()];
   iconDesc = new Hashtable[categories.size()];
  Enumeration e = categories.keys();
 int listNo = 0;
 String desc;

     // Loop while there are more keys (category descriptions)
      while (e.hasMoreElements()) {
               desc = (String) e.nextElement();

        // The first item in the list will be our default
      if (listNo == 0) {
           currentList = desc;
     }

        // Create a new list, with a display size of 20
          lists[listNo] = new List(20, false);

        // Create a new CardLayout panel
             iconListPanel.add(desc, lists[listNo]);

        // Add the description to the Icons menu
        sectionMenu.add(new MenuItem(desc));
        // Get a Hashtable containing an entry for each row found
         // for this category. The key is the icon description and
         // the data value is the ID.

        iconDesc[listNo] = getIconDesc(connection,
                      (String) categories.get(desc), lists[listNo]);
     listNo++;
}
    // Add the Icons menu to the menu bar
     menuBar.add(sectionMenu);

    // Set the menu bar
    setMenuBar(menuBar);

    // Create a Save As file dialog box
     fileDialog = new FileDialog(this, "Save File", FileDialog.SAVE);

    // Setup our layout
     setLayout(new GridLayout(1,2));
     add(iconListPanel);
    add(imageCanvas);
}
```

It is very important to note how the **CardLayout** has been set up. Each of the lists is added to the **CardLayout** with a description as a title, which, in our case, is the name of the category. When the user selects a category from the Icons menu, we can use the category description to set the new **CardLayout** list. Figure 8.2 shows the initial screen after loading the database tables.

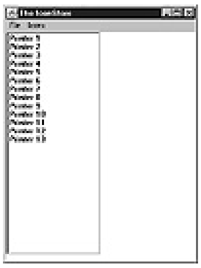


Figure 8.2 The IconStore main screen.

Handling Events

There are two types of events that we need to be aware of in the IconStore application: selecting menu options and clicking on the image list to select an icon. As with the Interactive SQL applet we discussed in Chapter 4, the event handling code is contained in the **handleEvent** method, as shown in Listing 8.8.

Listing 8.8 IconStore handleEvent.

```
//————————————————————————————————————
// handleEvent
// Handle an event by the user.
//————————————————————————————————————
public boolean handleEvent(
    Event evt)
{
    switch (evt.id) {
  case Event.ACTION_EVENT:

        // Determine the type of event that just occurred
        if (evt.target instanceof MenuItem) {

            // The user selected a menu item. Figure out what action
            // should be taken.
            String selection = (String) evt.arg;

            // 'Save As' - Save the currently displayed icon to a file
            if (selection.equals("Save As")) {
                if (currentFile != null) {
                      fileDialog.setFile("");
                    fileDialog.pack();
                    fileDialog.show();

                       String saveFile = fileDialog.getFile();

                    if (saveFile == null) {
                        return true;
                }

                    // If this is a new file, it will end with .*.*
                    if (saveFile.endsWith(".*.*")) {
                        saveFile = saveFile.substring(0,
                                saveFile.length() - 4);
                        // If no extension is given, append .GIF
                        if (saveFile.indexOf(".") < 0) {
                            saveFile += ".gif";
                     }
                }
                    // Copy the file. Returns true if successful.
                     boolean rc = copyFile (currentFile, saveFile);
           }
             return true;
       }
          // 'Exit' - Exit the application
           else if (selection.equals("Exit")) {
             // If there was an image file, delete it
              if (currentFile != null) {
                    (new File(currentFile)).delete();
          }

           System.exit(0);
       }

         // The user must have selected a different set of icons;
         // Display the proper list.
       else {
              currentList = selection;
               ((CardLayout) iconListPanel.getLayout()).show(
                                     iconListPanel, currentList);

             // Display the icon, if one was previously selected
              displayIcon(connection);
             return true;
```

```
            }
        }
        break;

    case Event.LIST_SELECT:
            displayIcon(connection);
          break;
    }

        return false;
}
```

Most of the code is very straightforward. Of interest here is how the **CardLayout** is managed. When a user makes a selection from the Icons menu, the selected item (which is the category description) is used to change the **CardLayout**. Remember that when the **CardLayout** was created, the title of each list was the category description. Also note that when the user selects an item from the list box (LIST_SELECT), the corresponding image can be displayed. Listing 8.9 shows how this is done.

When the user selects Exit from the menu, the temporary image file (which is discussed later) is deleted from disk, and the application is terminated. This is the perfect time to close the **Connection** that was in use. I purposefully omitted this step to illustrate a point: The JDBC specification states that all close operations are purely optional. It is up to the JDBC driver to perform any necessary clean-up in the finalize methods for each object. I strongly recommend, though, that all JDBC applications close objects when it is proper to do so.

Listing 8.9 Loading and displaying the selected image.

```
//————————————————————————————————————————
// displayIcon
// Display the currently selected icon.
//————————————————————————————————————————
public void displayIcon(
    Connection con)
{
    // Get the proper list element
     int n = getCategoryElement(currentList);

    // Get the item selected
      String item = lists[n].getSelectedItem();

    // Only continue if an item was selected
     if (item == null) {
          return;
    }

    // Get the ID
      String id = (String) iconDesc[n].get(item);

     try {
        // Create a Statement object
        Statement stmt = con.createStatement();

        // Execute the query and process the results
    ResultSet rs = stmt.executeQuery(
         "SELECT ICON FROM ICONSTORE WHERE ID=" + id);
   // If no rows are returned, the icon was not found
    if (!rs.next()) {
       stmt.close();
       return;
   }

   // Get the data as an InputStream
   InputStream inputStream = rs.getBinaryStream(1);

   if (inputStream == null) {
        stmt.close();
       return;
   }

   // Here's where things get ugly. Currently, there is no way
   // to display an image from an InputStream. We'll create a
   // new file from the InputStream and load the Image from the
   // newly created file. We need to create a unique name for
   // each icon; the Java VM caches the image file.

 String name = myHome + "/IconStoreImageFile" + id + ".gif";

   FileOutputStream outputStream = new FileOutputStream(name);
   // Write the data
   int bytes = 0;
   byte b[] = new byte[1024];

   while (true) {
       // Read from the input. The number of bytes read is returned.
       bytes = inputStream.read(b);

       if (bytes == -1) {
           break;
     }

       // Write the data
         outputStream.write(b, 0, bytes);
 }
   outputStream.close();
   inputStream.close();

   // Close the statement
   stmt.close();

   // Now, display the icon
   loadFile(name);

   // If there was an image file, delete it
    if (currentFile != null) {
       if (!currentFile.equals(name)) {
           (new File(currentFile)).delete();
     }
 }

   // Save our current file name
```

```
        currentFile = name;
    }
    catch (SQLException ex) {

        // An SQLException was generated. Dump the exception contents.
        // Note that there may be multiple SQLExceptions chained
        // together.

        System.out.println("\n*** SQLException caught ***\n");
       while (ex != null) {
              System.out.println("SQLState: " + ex.getSQLState());
             System.out.println("Message:  " + ex.getMessage());
              System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException();
      }
        System.out.exit(1);
   }
    catch (java.lang.Exception ex) {
         ex.printStackTrace();
        System.exit(1);
   }

}
```

Notice that each time an image is selected from the list, the image is read from the database. It could be very costly in terms of memory resources to save all of the images, so we'll just get the image from the database when needed. When the user selects an item from the list, we can get the image description. This description is used to get the icon ID from the image **Hashtable**. For the most part, we follow the same steps we have seen several times before in getting results from a database. Unfortunately, we've had to use a very nasty workaround here. The image is retrieved from the database as a binary **InputStream**, and it is from this **InputStream** that we need to draw the image on our canvas. This technique seems like it should be a simple matter, but it turns out to be impossible as of the writing of this book. To get around this problem, the IconStore application uses the **InputStream** to create a temporary file on disk, from which an image can be loaded and drawn on the canvas. Hopefully, a method to draw images from an **InputStream** will be part of Java in the future.

Figure 8.3 shows the IconStore screen after the user has selected an image from the initial category list. Figure 8.4 shows the IconStore screen after the user has changed the category (from the Icons menu) to sports and has made a selection.

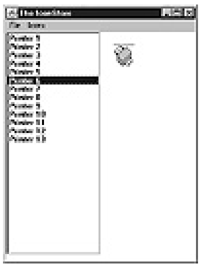


Figure 8.3 Selecting on image from the category list box.

Figure 8.4 Changing the image category.

Saving The Image

All that's left is to add the ability to save the image to disk. We saw previously how to handle the Save As menu event, so we just need to be able to create the disk file. Our workaround approach for drawing an image from an **InputStream** will be used to our advantage. Because an image file has already been created, we can simply make a copy of the temporary file. Listing 8.10 shows the code to copy a file.

Listing 8.10 Copying a file.

```
//————————————————————————————————————
// copyFile
// Copy the source file to the target file.
//————————————————————————————————————
public boolean copyFile(
    String source,
    String target)
{
    boolean rc = false;

    try {
           FileInputStream in = new FileInputStream(source);
          FileOutputStream out = new FileOutputStream(target);

        int bytes;
       byte b[] = new byte[1024];

        // Read chunks from the input stream and write to the output
        // stream.
        while (true) {
            bytes = in.read(b);
            if (bytes == -1) {
                break;
          }
             out.write(b, 0, bytes);
        }
        in.close();
        out.close();
        rc = true;
   }
     catch (java.lang.Exception ex) {
          ex.printStackTrace();
   }

    return rc;
```

}

Figure 8.5 shows the IconStore screen after the user has selected the Save As menu option.

Figure 8.5 The IconStore Save As dialog box.

That’s all there is to it.

Summary

Let’s recap the important details that we have covered in this chapter:

- Creating a basic GUI Java application
- Opening a connection to a data source
- Using database data to create dynamic GUI components (menus and lists)
- Handling user events
- Handling JDBC **InputStreams**

If you would like to take the IconStore application further, one obvious enhancement would be to allow the user to add images to the database. I’ll leave this as an exercise for you.

Chapter 9
Java And Database Security

Security is at the top of the list of concerns for people sharing databases on the Internet and large intranets. In this chapter, we’ll have a look at security in Java and how Java/JDBC security relates to database security. We’ll also have a peek at the new security features planned for Java, which will incorporate encryption and authentication into the JDBC.

Database Server Security

The first issue I’d like to tackle, and the first one you need to consider, is the security of your actual database server. If you are allowing direct connections to your database server from your Java/JDBC programs, you need to prepare for a number of potential security pitfalls. Although security breaks are few and far between, I advise you to cover all the angles so you don’t get caught off-guard.

Rooting Out The Packet Sniffers

Information is sent over networks in packets, and packet sniffing happens because a computer's network adapter is configured to read all of the packets that are sent over the network, instead of just packets meant for that computer. Therefore, anyone with access to a computer attached to your LAN can check out all transactions as they occur. Of course, a well-managed network and users you can trust are the best methods of preventing an inside job. Unfortunately, you must also consider another possibility: the outside threat. The possibility that someone from outside your LAN might break into a computer inside your LAN is another issue altogether; you must make sure that the other computers on your LAN are properly secured. To prevent such a situation, a firewall is often the best remedy. Though not completely foolproof, it does not allow indiscriminate access to any computers that are behind the firewall from outside. There are several good books on basic Internet security, and this book's Website contains a list of URLs that highlight several books on firewalls.

Packet sniffing doesn't necessarily involve only your local network; it can occur on the route the packet takes from the remote client machine somewhere on the Internet to your server machine. Along one of the many "hops" a packet takes as it travels across the Internet, a hacker who has gained entry into one of these hop points could be monitoring the packets sent to and from your server. Although this is a remote possibility, it's still a possibility. One solution is to limit the IP addresses from which connections to the database server can be made. However, IP authorization isn't bulletproof either—IP spoofing is a workaround for this method. For more information on these basic security issues, please see this book's Web site for references to security material.

Web Server CGI Holes

If you only allow local direct access to your database server via pre-written software, like CGI scripts run from Web pages, you'll still find yourself with a possible security hole. Some folks with too much time on their hands take great pleasure in hacking through CGI scripts to seek out unauthorized information. Are you vulnerable to this type of attack? Consider this situation: You have a CGI script that searches a table. The HTML form that gives the CGI its search information uses a field containing a table name; if a hacker realizes that you are directly patching in the table name from the HTML page, it would be easy to modify the CGI parameters to point to a different table. Of course, the easy solution to this scenario is to check in the CGI script that only the table you intend to allow to be queried can be accessed.

For in-house distribution of Java programs that access database servers, many of these security considerations are minimal. But for Internet applications, such as a merchandising applet where a user enters a credit card number to purchase some goods, you not only want to send this data encrypted to the Web server, but you want to protect the actual database server that this sensitive data is stored on.

Finding A Solution

So how do we deal with these security holes? The most straightforward way is to use a database server that implements secure login encryption. Some database servers do this already, and with the proliferation of "Web databases," login encryption is likely to be incorporated into more popular database servers in the future. The other solution, which is more viable, is to use an application server in a three-tier system. First, the Java program uses encryption to send login information to the application server. Then, the application server decodes the information. And finally, the application server sends the decoded information to the database server, which is either running on the same machine or on a machine attached to a secure local network. We'll discuss application servers in more detail in Chapter 11.

Another solution involves using the Java Security API, currently under development at Javasoft. This API, which provides classes that perform encryption and authentication, will be a standard part of the Java API and will allow you to use plug-in classes to perform encryption on a remote connection.

As a user, how do you know if the Java applet you're getting is part of a front for an illegitimate business? The Java Commerce API addresses the security issue of determining whether an applet is from a legitimate source by using digital signatures, authorization, and certification. Both the Java Commerce API and Java Security API will likely be incorporated into Web browsers' Java interpreters, and will also be linked in heavily with the security features of the Web browser itself. At the time this manuscript was written, however, these APIs were still under construction.

Applet Security: Can I Trust You?

As we've seen, setting up safe connections is quite possible. However, applet security is an entirely different issue. This aspect of security, where an applet that has been downloaded to your computer is running in your Web browser, has been under scrutiny since Java-enabled Web browsers appeared.

The Applet Security Manager

Every Web browser's Java interpreter includes a security manager to determine what an applet can and can't do. For instance, the security mangager does not allow applets downloaded from remote Web pages to access the local disk; it restricts network connections attempted by the applet to only the machine from which the applet came from; and it restricts applets from gaining control of local system devices. These restrictions are in place to protect users from rogue applets (or should I say rogue applet programmers) attempting to break into your computer. The user does not need to worry about the applet formatting the hard disk or reading password files. Of course, I'm simplifying the applet security scheme, but I want to point out the care that is taken to protect the user, and the restrictions that developers are faced with when programming applets. So how does this relate to the JDBC? The immediate concern for you as the developer is that your JDBC applet can only connect to the same machine that served the applet initially (i.e. your Web server). This means that you must run a Web server on the same machine as your database server. However, if you choose the application server route that we will discuss in Chapter 11, you must run the application server alongside the Web server, but then you are free to run the database server on another machine. If the user installs the applet locally and runs it, these security restrictions do not apply. But unfortunately, that defeats the purpose behind an applet: a program that comes over the network and begins running locally without installation.

I'm A Certified Applet

To account for these tight security restrictions, the Java Commerce API addresses easing security *if the applet comes from a "trusted" source*. This means that if the Web browser recognizes as genuine the certification of the Web page, applets on the page may also be considered "certified." To obtain such a status, you must apply for certification from the proper authority. When you receive certification, simply attach it to applets that are served from your Web site. The Commerce and Security APIs allow for the fetching of trusted applets, so if the user uses a Java interpreter that incorporates the Java Commerce API and Security API, you (the developer) can serve applets that can connect to an application server or database server running on a different machine than the Web server. In fact, you can even attach to different database servers simultaneously if necessary. In addition, this approach may allow the applet to save the contents of a database session on the user's disk, or read data from the user's disk to load previous session data.

The exact security restrictions of trusted applets are not set in stone, and they may differ depending on the Web browser the applet is run on. Also, the Java Commerce and Security specifications and related APIs have not been finalized as of the writing of this book, so much may change from the preliminary details of the security scheme by the time the APIs are released and implemented.

Summary

Security in data transactions is a top priority in the Internet community. In this chapter, we've discussed possible security holes and techniques to sew them up. We also took a look at Javasoft's approach to easing security restrictions for applets that come from a certified trusted source.

In the next chapter, we jump back into the meat of the JDBC when we explore writing JDBC drivers. We'll explore the heart of the JDBC's implementation details, and we'll also develop a real JDBC driver that can serve as the basis for drivers you write in the future.

Chapter 10
Writing Database Drivers

We've covered a lot of territory so far in this book. Now we can put some of your newly gained knowledge to use. In this chapter, we will explore what it takes to develop a JDBC driver. In doing so, we will also touch on some of the finer points of the JDBC specification. Throughout this chapter, I will use excerpts from the SimpleText JDBC driver that is included on the CD-ROM. This driver allows you to manipulate simple text files; you will be able to create and drop files, as well as insert and select data within a file. The SimpleText driver is not fully JDBC-compliant, but it provides a strong starting point for developing a driver. We'll cover what the JDBC components provide, how to implement the JDBC API interfaces, how to write native code to bridge to an existing non-Java API, some finer points of driver writing, and the major JDBC API interfaces that must be implemented.

The JDBC Driver Project: SimpleText

The SimpleText JDBC driver is just that—a JDBC driver that manipulates simple text files, with a few added twists. It is not a full-blown relational database system, so I would not recommend attempting to use it as one. If you are looking for a good way to prototype a system, or need a very lightweight database system to drive a simplistic application or applet, then SimpleText is for you. More importantly, though, the SimpleText driver can serve as a starting point for your own JDBC driver. Before continuing, let's take a look at the SimpleText driver specifications.

SimpleText SQL Grammar

The SimpleText JDBC driver supports a very limited SQL grammar. This is one reason that the driver is not JDBC compliant; a JDBC-compliant driver must support ANSI92 entry level SQL grammar. The following SQL statements define the base SimpleText grammar:

```
create-table-statement ::= CREATE TABLE table-name

                                        (column-element [, column-
element]...)

drop-table-statement ::= DROP TABLE table-name

insert-statement ::= INSERT INTO table-name

                            [(column-identifier [, column-
identifier]...)] VALUES

                               (insert-value [, insert-value]...)

select-statement ::= SELECT select-list FROM table-name [WHERE search-
                            condition]
```

The following elements are used in these SQL statements:

```
column-element ::= column-identifier data-type

column-identifier ::= user-defined-name

comparison-operator ::= < | > | = | <>

data-type ::= VARCHAR | NUMBER | BINARY

dynamic-parameter ::= ?

insert-value ::= dynamic-parameter | literal

search-condition ::= column-identifier comparison-operator literal
```

```
select-list ::= * | column-identifier [, column-identifier]...

table-name ::= user-defined-name

user-defined-name ::= letter [digit | letter]
```

What all this grammar means is that the SimpleText driver supports a **CREATE TABLE** statement, a **DROP TABLE** statement, an **INSERT** statement (with parameters), and a very simple **SELECT** statement (with a **WHERE** clause). It may not seem like much, but this grammar is the foundation that will allow us to create a table, insert some data, and select it back.

SimpleText File Format

The format of the files used by the SimpleText driver is, of course, very simple. The first line contains a signature, followed by each one of the column names (and optional data types). Any subsequent lines in the text file are assumed to be comma-separated data. There is no size limit to the text file, but the larger the file, the longer it takes to retrieve data (the entire file is read when selecting data; there is no index support). The data file extension is hard coded to be .SDF (Simple Data File). For example, the statement

```
CREATE TABLE TEST (COL1 VARCHAR, COL2 NUMBER, COL3 BINARY)
```

creates a file named TEST.SDF, with the following initial data:

```
.SDFCOL1,#COL2,@COL3
```

Note that none of the SQL grammar is case-sensitive. The .SDF is the file signature (this is how the SimpleText driver validates whether the text file can be used), followed by a comma-separated list of column names. The first character of the column name can specify the data type of the column. A column name starting with a # indicates a numeric column, while a column name starting with an @ indicates a binary column. What's that? Binary data in a text file? Well, not quite. A binary column actually contains an offset pointer into a sister file. This file, with an extension of .SBF (Simple Binary File), contains any binary data for columns in the text file, as well as the length of the data (maximum length of 1048576 bytes). Any other column name is considered to be character data (with a maximum length of 5120 bytes). The following statement shows how data is inserted into the TEST table:

```
INSERT INTO TEST VALUES ('FOO', 123, '0123456789ABCDEF')
```

After the INSERT, TEST.SDF will contain the following data:

```
.SDFCOL1,#COL2,@COL3
FOO,123,0
```

COL3 contains an offset of zero since this is the first row in the file. This is the offset from within the TEST.SBF table in which the binary data resides. Starting at the given offset, the first four bytes will be the length indicator, followed by the actual binary data that was inserted. Note that any character or binary data must be enclosed in single quotation marks.

We'll be looking at plenty of code from the SimpleText driver throughout this chapter. But first, let's start by exploring what is provided by the JDBC developer's kit.

The DriverManager

The JDBC **DriverManager** is a static class that provides services to connect to JDBC drivers. The **DriverManager** is provided by JavaSoft and does not require the driver developer to perform any implementation. Its main purpose is to assist in loading and initializing a requested JDBC driver. Other than using the **DriverManager** to register a JDBC driver (**registerDriver**) to make itself known and to provide the logging facility (which is covered in detail later), a driver does not interface with the **DriverManager**. In fact, once a JDBC driver is loaded, the **DriverManager** drops out of the picture all together, and the application or applet interfaces with the driver directly.

JDBC Exception Types

JDBC provides special types of exceptions to be used by a driver: **SQLException**, **SQLWarning**, and **DataTruncation**. The **SQLException** class is the foundation for the other types of JDBC exceptions, and extends **java.lang.Exceptn**. When created, an **SQLException** can have three pieces of information: a **String** describing the error, a **String** containing the XOPEN SQLstate (as described in the XOPEN SQL specification), and an **int** containing an additional vendor or database-specific error code. Also note that **SQLExceptions** can be chained together; that is, multiple **SQLExceptions** can be thrown for a single operation. The following code shows how an **SQLException** is thrown:

```
//----------------------------------------------------------------------
-----
// fooBar
// Demonstrates how to throw an SQLException
//----------------------------------------------------------------------
----
public void fooBar()
    throws SQLException
{
    throw new SQLException("I just threw a SQLException");
}
```

Here’s how you call fooBar and catch the **SQLException**:

```
try {
     fooBar();
}
catch (SQLException ex) {

    // If an SQLException is thrown, we'll end up here. Output the error
   // message, SQLstate, and vendor code.
    System.out.println("A SQLException was caught!");
    System.out.println("Message: " + ex.getMessage());
     System.out.println("SQLState: " + ex.getSQLState());
     System.out.println("Vendor Code: " + ex.getErrorCode());
}
```

An **SQLWarning** is similar to an **SQLException** (it extends **SQLException**). The main difference is in semantics. If an **SQLException** is thrown, it is considered to be a critical error (one that needs attention). If an **SQLWarning** is thrown, it is considered to be a non-critical error (a warning or informational message). For this reason, JDBC treats **SQLWarnings** much differently than **SQLExceptions**. **SQLExceptions** are thrown just like any other type of exception; **SQLWarnings** are not thrown, but put on a list of warnings on an owning object type (for instance, **Connection, Statement**, or **ResultSet**, which we'll cover later). Because they are put on a list, it is up to the application to poll for warnings after the completion of an operation. Listing 10.1 shows a method that accepts an **SQLWarning** and places it on a list.

Listing 10.1 Placing an SQL Warning on a list.

```
//-------------------------------------------------------------------------
----
// setWarning
// Sets the given SQLWarning in the warning chain. If null, the
// chain is reset. The local attribute lastWarning is used
// as the head of the chain.
//------------------------------------------------------------------ ------
---
protected void setWarning(
    SQLWarning warning)
{

    // A null warning can be used to clear the warning stack
     if (warning == null) {
          lastWarning = null;
    }
    else {
        // Set the head of the chain. We'll use this to walk through the
        // chain to find the end.
        SQLWarning chain = lastWarning;

        // Find the end of the chain. When the current warning does
        // not have a next pointer, it must be the end of the chain.
        while (chain.getNextWarning() != null) {
            chain = chain.getNextWarning();
        }

        // We're at the end of the chain. Add the new warning
        chain.setNextWarning(warning);
        }
}
```

Listing 10.2 uses this method to create two **SQLWarnings** and chain them together.

Listing 10.2 Chaining SQLWarnings together.

```
//-------------------------------------------------------------------------
-
// fooBar
// Do nothing but put two SQLWarnings on our local
// warning stack (lastWarning).
//-------------------------------------------------------------------------
--
protected void fooBar()
{
    // First step should always be to clear the stack. If a warning
     // is lingering, it will be discarded. It is up to the application
to
     // check and clear the stack.
    setWarning(null);

     // Now create our warnings
    setWarning(new SQLWarning("Warning 1"));
    setWarning(new SQLWarning("Warning 2"));
}
```

Now we’ll call the method that puts two **SQLWarnings** on our warning stack, then poll for the warning using the JDBC method **getWarnings**, as shown in Listing 10.3.

Listing 10.3 Polling for warnings.

```
// Call fooBar to create a warning chain
fooBar();

// Now, poll for the warning chain. We'll simply dump any warning
// messages to standard output.
SQLWarning chain = getWarnings();

if (chain  != null) {
         System.out.println("Warning(s):");

    // Display the chain until no more entries exist
    while (chain != null) {
          System.out.println("Message: " + chain.getMessage());

        // Advance to the next warning in the chain. null will be
        // returned if no more entries exist.
        chain = chain.getNextWarning();

    }
}
```

DataTruncation objects work in the same manner as **SQLWarnings**. A **DataTruncation** object indicates that a data value that was being read or written was truncated, resulting in a loss of data. The **DataTruncation** class has attributes that can be set to specify the column or parameter number, whether a truncation occurred on a read or a write, the size of the data that should have been transferred, and the number of bytes that were actually transferred. We can modify our code from Listing 10.2 to include the handling of **DataTruncation** objects, as shown in Listing 10.4.

Listing 10.4 Creating dDataTruncation warnings.

```
//--------------------------------------------------------------------
--
// fooBar
// Do nothing but put two SQLWarnings on our local
// warning stack (lastWarning) and a DataTruncation
// warning.
//--------------------------------------------------------------------
--
protected void fooBar()
{
    // First step should always be to clear the stack. If a warning
     // is lingering, it will be discarded. It is up to the application
to
    // check and clear the stack.
    setWarning(null);

    // Now create our warnings
     setWarning(new SQLWarning("Warning 1"));
     setWarning(new SQLWarning("Warning 2"));

    // And create a DataTruncation indicating that a truncation
   // occurred on column 1, 1000 bytes were requested to
    // read, and only 999 bytes were read.
     setWarning(new DataTruncation(1, false, true, 1000, 999);
}
```

Listing 10.5 shows the modified code to handle the **DataTruncation**.

Listing 10.5 Processing DataTruncation warnings.

```
// Call fooBar to create a warning chain
fooBar();

// Now, poll for the warning chain. We'll simply dump any warning
// messages to standard output.
SQLWarning chain = getWarnings();

if (chain   != null) {
         System.out.println("Warning(s):");

    // Display the chain until no more entries exist
    while (chain != null) {
        // The only way we can tell if this warning is a DataTruncation
         // is to attempt to cast it. This may fail, indicating that
        // it is just an SQLWarning.
        try {
              DataTruncation trunc = (DataTruncation) chain;
              System.out.println("Data Truncation on column: " +
                  trunc.getIndex());
       }
        catch (Exception ex) {
              System.out.println("Message: " + chain.getMessage());
       }

        // Advance to the next warning in the chain. null will be
        // returned if no more entries exist.
        chain = chain.getNextWarning();
    }
}
```

JDBC Data Types

The JDBC specification provides definitions for all of the SQL data types that can be supported by a JDBC driver. Only a few of these data types may be natively supported by a given database system, which is why data coercion becomes such a vital service (we'll discuss data coercion a little later in this chapter). The data types are defined in **Types.class**:

```
public class Types
{

    public final static int BIT = -7;
    public final static int TINYINT = -6;
    public final static int SMALLINT = 5;
    public final static int INTEGER = 4;
    public final static int BIGINT = -5;
    public final static int FLOAT = 6;
    public final static int REAL = 7;
    public final static int DOUBLE = 8;
    public final static int NUMERIC = 2;
    public final static int DECIMAL = 3;
    public final static int CHAR = 1;
    public final static int VARCHAR = 12;
    public final static int LONGVARCHAR = -1;
    public final static int DATE = 91;
    public final static int TIME = 92;
    public final static int TIMESTAMP = 93;
    public final static int BINARY = -2;
    public final static int VARBINARY = -3;
    public final static int LONGVARBINARY = -4;
    public final static int OTHER = 1111;
}
```

At a minimum, a JDBC driver must support one (if not all) of the character data types (**CHAR**, **VARCHAR**, and **LONGVARCHAR**). A driver may also support driver-specific data types (**OTHER**) which can only be accessed in a JDBC application as an **Object**. In other words, you can get data as some type of object and put it back into a database as that same type of object, but the application has no idea what type of data is actually contained within. Let's take a look at each of the data types more closely.

Character Data: CHAR, VARCHAR, And LONGVARCHAR

CHAR, **VARCHAR**, and **LONGVARCHAR** data types are used to express character data. These data types are represented in JDBC as Java **String** objects. Data of type **CHAR** is represented as a fixed-length **String**, and may include some padding spaces to ensure that it is the proper length. If data is being written to a database, the driver must ensure that the data is properly padded. Data of type **VARCHAR** is represented as a variable-length **String**, and is trimmed to the actual length of the data. **LONGVARCHAR** data can be either a variable-length **String** or returned by the driver as a Java **InputStream**, allowing the data to be read in chunks of whatever size the application desires.

Exact Numeric Data: NUMERIC And DECIMAL

The **NUMERIC** and **DECIMAL** data types are used to express signed, exact numeric values with a fixed number of decimal places. These data types are often used to represent currency values. **NUMERIC** and **DECIMAL** data are both represented in

JDBC as **Numeric** objects. The **Numeric** class is new with JDBC, and we'll be discussing it shortly.

Binary Data: BINARY, VARBINARY, And LONGVARBINARY

The **BINARY**, **VARBINARY**, and **LONGVARBINARY** data types are used to express binary (non-character) data. These data types are represented in JDBC as Java byte arrays. Data of type **BINARY** is represented as a fixed-length byte array, and may include some padding zeros to ensure that it is the proper length. If data is being written to a database, the driver must ensure that the data is properly padded. Data of type **VARBINARY** is represented as a variable-length byte array, and is trimmed to the actual length of the data. **LONGVARBINARY** data can either be a variable-length byte array or returned by the driver as a Java **InputStream**, allowing the data to be read in chunks of whatever size the application desires.

Boolean Data: BIT

The **BIT** data type is used to represent a boolean value—either true or false—and is represented in JDBC as a **Boolean** object or **boolean** data type.

Integer Data: TINYINT, SMALLINT, INTEGER, And BIGINT

The **TINYINT**, **SMALLINT**, **INTEGER**, and **BIGINT** data types are used to represent signed integer data. Data of type **TINYINT** is represented in JDBC as a Java *byte* data type (1 byte), with a minimum value of -128 and a maximum value of 127. Data of type **SMALLINT** is represented in JDBC as a Java *short* data type (2 bytes), with a minimum value of -32,768 and a maximum value of 32,767. Data of type **INTEGER** is represented as a Java *int* data type (4 bytes), with a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647. Data of type **BIGINT** is represented as a Java *long* data type (8 bytes), with a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807.

Floating-Point Data: REAL, FLOAT, And DOUBLE

The **REAL**, **FLOAT**, and **DOUBLE** data types are used to represent signed, approximate values. Data of type **REAL** supports seven digits of mantissa precision, and is represented as a Java *float* data type. Data of types **FLOAT** and **DOUBLE** support 15 digits of mantissa precision, and are represented as Java *double* data types.

Time Data: DATE, TIME, And TIMESTAMP

The **DATE**, **TIME**, and **TIMESTAMP** data types are used to represent dates and times. Data of type **DATE** supports specification of the month, day, and year, and is represented as a JDBC **Date** object. Data of type **TIME** supports specification of the hour, minutes, seconds, and milliseconds, and is represented as a JDBC **Time** object. Data of type **TIMESTAMP** supports specification of the month, day, year, hour, minutes, seconds, and milliseconds, and is represented as a JDBC **Timestamp** object. The **Date**, **Time,** and **Timestamp** objects, which we'll get into a bit later, are new with JDBC.

Tip: Be aware of date limitations.
One important note about **Date** and **Timestamp** objects: The Java calendar starts at January 1, 1970, which means that you cannot represent dates prior to 1970.

New Data Classes

The JDBC API introduced several new data classes. These classes were developed to solve specific data-representation problems like how to accurately represent fixed-precision numeric values (such as currency values) for **NUMERIC** and **DECIMAL** data types, and how to represent time data for **DATE**, **TIME**, and **TIMESTAMP** data types.

Numeric

As mentioned before, the **Numeric** class was introduced with the JDBC API to represent signed, exact numeric values with a fixed number of decimal places. This class is ideal for representing monetary values, allowing accurate arithmetic operations and comparisons. Another aspect is the ability to change the rounding value. Rounding is performed if the value of the scale (the number of fixed decimal places) plus one digit to the right of the decimal point is greater than the rounding value. By default, the rounding value is 4. For example, if the result of an arithmetic operation is 2.495, and the scale is 2, the number is rounded to 2.50. Listing 10.6 provides an example of changing the rounding value. Imagine that you are a devious retailer investigating ways to maximize your profit by adjusting the rounding value.

Listing 10.6 Changing the rounding value.

```
import java.sql.*;

class NumericRoundingValueTest {

    public static void main(String args[]) {

        // Set our price and discount amounts
        Numeric price = new Numeric(4.91, 2);
        Numeric discount = new Numeric(0.15, 2);
        Numeric newPrice;

        // Give the item a discount
        newPrice = discountItem(price, discount);

            System.out.println("discounted price="+newPrice.toString());

        // Now, give the item a discount with a higher rounding value.
        // This will lessen the discount amount in many cases.
        discount.setRoundingValue(9);

       newPrice = discountItem(price, discount);

        System.out.println("discounted price with high rounding="+
                newPrice.toString());
    }
    // Perform the calculation to discount a price
    public static Numeric discountItem(
        Numeric price,
        Numeric discount)
    {
        return price.subtract(price.multiply(discount));
    }
}
```

Listing 10.6 produces the following output:

```
discounted price=004.17
discounted price with high rounding=004.18
```

Date

The **Date** class is used to represent dates in the ANSI SQL format YYYY-MM-DD, where YYYY is a four-digit year, MM is a two-digit month, and DD is a two-digit day. The JDBC **Date** class extends the existing **java.util.Date** class (setting the hour, minutes, and seconds to zero) and, most importantly, adds two methods to convert Strings into dates, and vice-versa:

```
// Create a Date object with a date of June 30th, 1996
Date d = Date.valueOf("1996-06-30");

// Print the date
System.out.println("Date=" + d.toString());

// Same thing, without leading zeros
Date d2 = Date.valueOf("1996-6-30");
System.out.println("Date=" + d2.toString());
```

The **Date** class also serves very well in validating date values. If an invalid date string is passed to the **valueOf** method, a **java.lang.IllegalArgument-Exception** is thrown:

```
String s;

// Get the date from the user
.
.
.
//   Validate the date
try  {
      Date d = Date.valueOf(s);
}
catch  (java.lang.IllegalArgumentException ex) {
       // Invalid date, notify the application
    .
    .
    .
}
```

It is worth mentioning again that the Java date epoch is January 1, 1970; therefore, you cannot represent any date values prior to January 1, 1970, with a **Date** object.

Time

The **Time** class is used to represent times in the ANSI SQL format HH:MM:SS, where HH is a two-digit hour, MM is a two-digit minute, and SS is a two-digit second. The JDBC **Time** class extends the existing **java.util.Date** class (setting the year, month, and day to zero) and, most importantly, adds two methods to convert Strings into times, and vice-versa:

```
// Create a Time object with a time of 2:30:08 pm
Time t = Time.valueOf("14:30:08");

// Print the time
System.out.println("Time=" + t.toString());

// Same thing, without leading zeros
Time t2 = Time.valueOf("14:30:8");
System.out.println("Time=" + t2.toString());
```

The **Time** class also serves very well in validating time values. If an invalid time string is passed to the **valueOf** method, a **java.lang.IllegalArgument-Exception** is thrown:

```
String s;

// Get the time from the user
.
.
.
// Validate the time
try {
     Time t = Time.valueOf(s);
}
catch  (java.lang.IllegalArgumentException ex) {
       // Invalid time, notify the application
    .
    .
    .
}
```

Timestamp

The **Timestamp** class is used to represent a combination of date and time values in the ANSI SQL format YYYY-MM-DD HH:MM:SS.F..., where YYYY is a four-digit year, MM is a two-digit month, DD is a two-digit day, HH is a two-digit hour, MM is a two-digit minute, SS is a two-digit second, and F is an optional fractional second up to nine digits in length. The JDBC **Timestamp** class extends the existing **java.util.Date** class (adding the fraction seconds) and, most importantly, adds two methods to convert Strings into timestamps, and vice-versa:

```
// Create a Timestamp object with a date of 1996-06-30 and a time of
// 2:30:08 pm.
Timestamp t = Timestamp.valueOf("1996-06-30 14:30:08");

// Print the timestamp

System.out.println("Timestamp=" + t.toString());
```

```
// Same thing, without leading zeros
Timestamp t2 = Timestamp.valueOf("1996-6-30 14:30:8");
System.out.println("Timestamp=" + t2.toString());
```

The **Timestamp** class also serves very well in validating timestamp values. If an invalid time string is passed to the **valueOf** method, a **java.lang.Illegal-ArgumentException** is thrown:

```
String s;

// Get the timestamp from the user
.
.
.
// Validate the timestamp
try {
    Timestamp t = Timestamp.valueOf(s);
}
catch  (java.lang.IllegalArgumentException ex) {
      // Invalid timestamp, notify the application
    .
    .
    .
}
```

As is the case with the **Date** class, the Java date epoch is January 1, 1970; therefore, you cannot represent any date values prior to January 1, 1970, with a **Timestamp** object.

Native Drivers: You're Not From Around Here, Are Ya?

Before beginning to implement a JDBC driver, the first question that must be answered is: Will this driver be written completely in Java, or will it contain native (machine dependent) code? You may be forced to use native code because many major database systems—such as Oracle, Sybase, and SQLServer—do not provide Java client software. In this case, you will need to write a small library containing C code to bridge from Java to the database client API (the JDBC to ODBC Bridge is a perfect example). The obvious drawback is that the JDBC driver is not portable and cannot be automatically downloaded by today's browsers.

If a native bridge is required for your JDBC driver, you should keep a few things in mind. First, do as little as possible in the C bridge code; you will want to keep the bridge as small as possible, ideally creating just a Java wrapper around the C API. Most importantly, avoid the temptation of performing memory management in C (i.e. malloc). This is best left in Java code, since the Java Virtual Machine so nicely takes care of garbage collection. Secondly, keep all of the native method declarations in one Java class. By doing so, all of the bridge routines will be localized and much easier to maintain. Finally, don't make any assumptions about data representation. An integer value may be 2 bytes on one system, and 4 bytes on another. If you are planning to port the native bridge code to a different system (which is highly likely), you should provide native methods that provide the size and interpretation of data.

Listing 10.7 illustrates these suggestions. This module contains all of the native method declarations, as well as the code to load our library. The library will be loaded when the class is instantiated.

Listing 10.7 Java native methods.

```
//----------------------------------------------------------------------
--
// MyBridge.java
//
// Sample code to demonstrate the use of native methods
//----------------------------------------------------------------------
--
package jdbc.test;

import java.sql.*;

public class MyBridge
     extends Object
{
    //------------------------------------------------------------------
--
    // Constructor
     // Attempt to load our library. If it can't be loaded, an
     // SQLException will be thrown.
    //------------------------------------------------------------------
--
    public MyBridge()
        throws SQLException
    {
        try {
            // Attempt to load our library. For Win95/NT, this will
             // be myBridge.dll. For Unix systems, this will be
            // libmyBridge.so.
              System.loadLibrary("myBridge");
        }
          catch (UnsatisfiedLinkError e) {
              throw new SQLException("Unable to load myBridge library");
        }
    }
    //------------------------------------------------------------------
--
    // Native method declarations
//--------------------------------------------------------------------

    // Get the size of an int
    public native int getINTSize();

    // Given a byte array, convert it to an integer value
    public native int getINTValue(byte intValue[]);

    // Call some C function that does something with a String, and
    // returns an integer value.
    public native void callSomeFunction(String stringValue, byte
    intValue[]);
}
```

Once this module has been compiled (javac), a Java generated header file and C file must be created:

```
javah   jdbc.test.MyBridge
javah   -stubs  jdbc.test.MyBridge
```

These files provide the mechanism for the Java and C worlds to communicate with each other. Listing 10.8 shows the generated header file (jdbc_test_MyBridge.h, in this case), which will be included in our C bridge code.

Listing 10.8 Machine-generated header file for native methods.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class jdbc_test_MyBridge */

#ifndef  _Included_jdbc_test_MyBridge
#define  _Included_jdbc_test_MyBridge

typedef struct Classjdbc_test_MyBridge {
     char PAD; /* ANSI C requires structures to have at least one member
*/
}  Classjdbc_test_MyBridge;
HandleTo(jdbc_test_MyBridge);

#ifdef __cplusplus
extern "C" {
#endif
__declspec(dllexport) long jdbc_test_MyBridge_getINTSize(struct
Hjdbc_test_MyBridge *);
__declspec(dllexport) long jdbc_test_MyBridge_getINTValue(struct
Hjdbc_test_MyBridge *,HArrayOfByte *);
struct Hjava_lang_String;
__declspec(dllexport) void jdbc_test_MyBridge_callSomeFunction(struct
Hjdbc_test_MyBridge *,struct Hjava_lang_String *,HArrayOfByte *);
#ifdef __cplusplus
}
#endif
#endif
```

The generated C file (shown in Listing 10.9) must be compiled and linked with the bridge.

Listing 10.9 Machine-generated C file for native methods.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <StubPreamble.h>

/* Stubs for class jdbc/test/MyBridge */
/* SYMBOL: "jdbc/test/MyBridge/getINTSize()I",
Java_jdbc_test_MyBridge_getINTSize_stub  */
__declspec(dllexport)  stack_item
*Java_jdbc_test_MyBridge_getINTSize_stub(stack_item  *_P_,struct
execenv
*_EE_) {
      extern long jdbc_test_MyBridge_getINTSize(void *);
      _P_[0].i = jdbc_test_MyBridge_getINTSize(_P_[0].p);
```

```
    return _P_ + 1;
    }
    /* SYMBOL: "jdbc/test/MyBridge/getINTValue([B)I",
     Java_jdbc_test_MyBridge_getINTValue_stub */
      __declspec(dllexport) stack_item
      *Java_jdbc_test_MyBridge_getINTValue_stub(stack_item *_P_,struct
execenv *_EE_) {
     extern long jdbc_test_MyBridge_getINTValue(void *,void *);
       _P_[0].i = jdbc_test_MyBridge_getINTValue(_P_[0].p,((_P_[1].p)));
    return _P_ + 1;
}
/*  SYMBOL: "jdbc/test/MyBridge/callSomeFunction(Ljava/lang/String;[B)V",
Java_jdbc_test_MyBridge_callSomeFunction_stub  */
__declspec(dllexport)   stack_item
*Java_jdbc_test_MyBridge_callSomeFunction_stub(stack_item    *_P_,struct
execenv *_EE_) {
          extern void jdbc_test_MyBridge_callSomeFunction(void *,void
*,void
    *);
            (void)
jdbc_test_MyBridge_callSomeFunction(_P_[0].p,((_P_[1].p)),
          ((_P_[2].p)));return _P_;
}
```

The bridge code is shown in Listing 10.10. The function prototypes were taken from the generated header file.

Listing 10.10 Bridge code.

```
//------------------------------------------------------------------------
--
// MyBridge.c
//
// Sample code to demonstrate the use of native methods
//------------------------------------------------------------------------
--
#include <stdio.h>
#include <ctype.h>
#include <string.h>

// Java internal header files
#include "StubPreamble.h"
#include "javaString.h"

// Our header file generated by JAVAH
#include "jdbc_test_MyBridge.h"

//------------------------------------------------------------------------
--
// getINTSize
// Return the size of an int
//------------------------------------------------------------------------
--
long   jdbc_test_MyBridge_getINTSize(
      struct Hjdbc_test_MyBridge *caller)
{
    return sizeof(int);
}

//----------------------------------------------------------------------
--
// getINTValue
// Given a buffer, return the value as an int
//----------------------------------------------------------------------
--
long jdbc_test_MyBridge_getINTValue(
    struct Hjdbc_test_MyBridge *caller,
    HArrayOfByte *buf)
{
    // Cast our array of bytes to an integer pointer
     int* pInt = (int*) unhand (buf)->body;

    // Return the value
     return (long) *pInt;
}

//----------------------------------------------------------------------
--
// callSomeFunction
// Call some function that takes a String and an int pointer as
arguments
//----------------------------------------------------------------------
--
void jdbc_test_MyBridge_callSomeFunction(
    struct Hjdbc_test_MyBridge *caller,
    struct Hjava_lang_String *stringValue,
    HArrayOfByte *buf)
{

    // Cast the string into a char pointer
    char* pString = (char*) makeCString (stringValue);

    // Cast our array of bytes to an integer pointer
    int* pInt = (int*) unhand (buf)->body;

     // This fictitious function will print the string, then return the
    // length of the string in the int pointer.
      printf("String value=%s\n", pString);
     *pInt = strlen(pString);
}
```

Now, create a library (DLL or Shared Object) by compiling this module and linking it with the jdbc_test_MyDriver compiled object and the one required Java library, **javai.lib**. Here's the command line I used to build it for Win95/NT:

```
cl -DWIN32 mybridge.c jdbc_test_mybridge.c -FeMyBridge.dll -MD -LD
javai.lib
```

Now we can use our native bridge, as shown in Listing 10.11.

Listing 10.11 Implementing the bridge.

```
import jdbc.test.*;
import java.sql.*;
```

```
class Test {

     public static void main (String args[]) {

        MyBridge myBridge = null;
         boolean loaded = false;

        try {

            // Create a new bridge object. If it is unable to load our
            // native library, an SQLException will be thrown.
            myBridge = new MyBridge();
            loaded = true;
        }
        catch (SQLException ex) {
            System.out.println("SQLException: " + ex.getMessage());
        }

        // If the bridge was loaded, use the native methods
         if (loaded) {

            // Allocate storage for an int
             byte intValue[] = new byte[myBridge.getINTSize()];

            // Call the bridge to perform some function with a string,
            // returning a value in the int buffer.
             myBridge.callSomeFunction("Hello, World.", intValue);

            // Get the value out of the buffer.
             int n = myBridge.getINTValue(intValue);

            System.out.println("INT value=" + n);
        }
    }
}
```

Listing 10.11 produces the following output:

```
String value=Hello,  World.
INT value=13
```

As you can see, using native methods is very straightforward. Developing a JDBC driver using a native bridge is a natural progression for existing database systems that provide a C API. The real power and ultimate solution, though, is to develop non-native JDBC drivers—those consisting of 100 percent Java code.

Implementing Interfaces

The JDBC API specification provides a series of *interfaces* that must be implemented by the JDBC driver developer. An interface declaration creates a new reference type consisting of constants and abstract methods. An interface cannot contain any implementations (that is, executable code). What does all of this mean? The JDBC API specification dictates the methods and method interfaces for the API, and a driver must fully implement these interfaces. A JDBC application makes method calls to the JDBC interface, not a specific driver. Because all JDBC drivers must implement the same interface, they are interchangeable.

There are a few rules that you must follow when implementing interfaces. First, you must implement the interface exactly as specified. This includes the name, return value, parameters, and **throws** clause. Secondly, you must be sure to implement all interfaces as **public** methods. Remember, this is the interface that other classes will see; if it isn't **public**, it can't be seen. Finally, all methods in the interface must be implemented. If you forget, the Java compiler will kindly remind you.

Take a look at Listing 10.12 for an example of how interfaces are used. The code defines an interface, implements the interface, and then uses the interface.

Listing 10.12 Working with interfaces.

```
//------------------------------------------------------------------------
--
// MyInterface.java
//
// Sample code to demonstrate the use of interfaces
//------------------------------------------------------------------------
--
package jdbc.test;

public interface MyInterface
{
    //--------------------------------------------------------------------
--
    // Define 3 methods in this interface
//----------------------------------------------------------------------
    void method1();
     int method2(int x);
      String method3(String y);
}
//------------------------------------------------------------------------
--
// MyImplementation.java
//
// Sample code to demonstrate the use of interfaces
//------------------------------------------------------------------------
--

package   jdbc.test;

public  class MyImplementation
       implements jdbc.test.MyInterface
{
//----------------------------------------------------------------------
    // Implement the 3 methods in the interface
//----------------------------------------------------------------------
       public void method1()
    {
    }
    public int method2(int x)
    {
       return addOne(x);
    }

    public String method3(String y)
    {
       return y;
    }
    //------------------------------------------------------------------
--
    // Note that you are free to add methods and attributes to this
    // new class that were not in the interface, but they cannot be
    // seen from the interface.
//----------------------------------------------------------------------
    protected int addOne(int x)
    {
        return x + 1;
    }
}
//----------------------------------------------------------------------
--
//   TestInterface.java
//
//  Sample code to demonstrate the use of interfaces
//----------------------------------------------------------------------
--
import jdbc.test.*;

class TestInterface {

    public static void main (String args[])
   {
        // Create a new MyImplementation object. We are assigning the
        // new object to a MyInterface variable, thus we will only be
        // able to use the interface methods.
        MyInterface myInterface = new MyImplementation();

        // Call the methods
         myInterface.method1();
         int x = myInterface.method2(1);
           String y = myInterface.method3("Hello, World.");

    }
}
```

As you can see, implementing interfaces is easy. We'll go into more detail with the major JDBC interfaces later in this chapter. But first, we need to cover some basic foundations that should be a part of every good JDBC driver.

Tracing

One detail that is often overlooked by software developers is providing a facility to enable debugging. The JDBC API does provide methods to enable and disable tracing, but it is ultimately up to the driver developer to provide tracing information in the driver.

It becomes even more critical to provide a detailed level of tracing when you consider the possible wide-spread distribution of your driver. People from all over the world may be using your software, and they will expect a certain level of support if problems arise. For this reason, I consider it a must to trace all of the JDBC API method calls (so that a problem can be re-created using the output from a trace).

Turning On Tracing

The **DriverManager** provides a method to set the tracing **PrintStream** to be used for all of the drivers; not only those that are currently active, but any drivers that are subsequently loaded. Note that if two applications are using JDBC, and both have turned tracing on, the **PrintStream** that is set last will be shared by both applications. The following code snippet shows how to turn tracing on, sending any trace messages to a local file:

```
try {
    // Create a new OuputStream using a file. This may fail if the
     // calling application/applet does not have the proper security
     // to write to a local disk.
     java.io.OutputStream outFile = new
        java.io.FileOutputStream("jdbc.out");

     // Create a PrintStream object using our newly created OuputStream
      // object. The second parameter indicates to flush all output with
      // each write. This ensures that all trace information gets
written
      // into the file.
       java.io.PrintStream outStream = new java.io.PrintStream(outFile,
     true);

      // Enable the JDBC tracing, using the PrintStream
       DriverManager.setLogStream(outStream);
}
catch  (Exception ex) {
      // Something failed during enabling JDBC tracing. Notify the
      // application that tracing is not available.
    .
    .
    .
}
```

Using this code, a new file named jdbc.out will be created (if an existing file already exists, it will be overwritten), and any tracing information will be saved in the file.

Writing Tracing Information

The **DriverManager** also provides a method to write information to the tracing **OutputStream**. The **println** method will first check to ensure that a trace **OutputStream** has been registered, and if so, the **println** method of the **OutputStream** will be called. Here's an example of writing trace information:

```
// Send some information to the JDBC trace OutputStream
String a = "The quick brown fox ";
String b = "jumped over the ";
String c = "lazy dog";
```

```
DriverManager.println("Trace=" + a + b + c);
```

In this example, a **String** message of “Trace=The quick brown fox jumped over the lazy dog” will be constructed, the message will be provided as a parameter to the **DriverManager.println** method, and the message will be written to the **OutputStream** being used for tracing (if one has been registered).

Some of the JDBC components are also nice enough to provide tracing information. The **DriverManager** object traces most of its method calls. **SQLException** also sends trace information whenever an exception is thrown. If you were to use the previous code example and enable tracing to a file, the following example output will be created when attempting to connect to the SimpleText driver:

```
DriverManager.initialize: jdbc.drivers = null
JDBC DriverManager initialized
registerDriver:
driver[className=jdbc.SimpleText.SimpleTextDriver,context=null,
jdbc.SimpleText.SimpleTextDriver@1393860]
DriverManager.getConnection("jdbc:SimpleText")
trying
driver[className=jdbc.SimpleText.SimpleTextDriver,context=null,
jdbc.SimpleText.SimpleTextDriver@1393860]
driver[className=jdbc.SimpleText.SimpleTextDriver,context=null,j
dbc.SimpleText.SimpleTextDriver@1393860]
```

Checking For Tracing

I have found it quite useful for both the application and the driver to be able to test for the presence of a tracing **PrintStream**. The JDBC API provides us with a method to determine if tracing is enabled, as shown here:

```
//-----------------------------------------------------------------------
--
// traceOn
// Returns true if tracing (logging) is currently enabled
//-----------------------------------------------------------------------
--
public static boolean traceOn()
{

    // If the DriverManager log stream is not null, tracing
    // must be currently enabled.
    return (DriverManager.getLogStream() != null);

}
```

From an application, you can use this method to check if tracing has been previously enabled before blindly setting it:

```
// Before setting tracing on, check to make sure that tracing is not
// already turned on. If it is, notify the application.
if (traceOn()) {
    // Issue a warning that tracing is already enabled
   .
```

```
    .
    .
}
```

From the driver, I use this method to check for tracing before attempting to send information to the **PrintStream**. In the example where we traced the message text of “Trace=The quick brown fox jumped over the lazy dog,” a lot had to happen before the message was sent to the **DriverManager.println** method. All of the given **String** objects had to be concatenated, and a new **String** had to be constructed. That’s a lot of overhead to go through before even making the **println** call, especially if tracing is not enabled (which will probably be the majority of the time). So, for performance reasons, I prefer to ensure that tracing has been enabled before assembling my trace message:

```
// Send some information to the JDBC trace OutputStream
String a = "The quick brown fox ";
String b = "jumped over the ";
String c = "lazy dog";

// Make sure tracing has been enabled
if (traceOn()) {
    DriverManager.println("Trace=" + a + b + c);
}
```

Data Coercion

At the heart of every JDBC driver is data. That is the whole purpose of the driver: providing data. Not only providing it, but providing it in a requested format. This is what data coercion is all about—converting data from one format to another. As Figure 10.1 shows, JDBC specifies the necessary conversions.

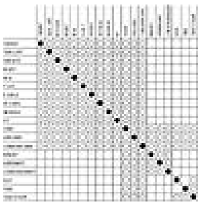


Figure 10.1 JDBC data conversion table.

In order to provide reliable data coercion, a data wrapper class should be used. This class contains a data value in some known format and provides methods to convert it to a specific type. As an example, I have included the **CommonValue** class from the SimpleText driver in Listing 10.13. This class has several overloaded constructors that accept different types of data values. The data value is stored within the class, along with the type of data (String, Integer, etc.). A series of methods are then provided to get the data in different formats. This class greatly reduces the burden of the JDBC driver developer, and can serve as a fundamental class for any number of drivers.

Listing 10.13 The CommonValue class.

```
package jdbc.SimpleText;

import java.sql.*;

public class CommonValue
```

```
    extends         Object
{
    //------------------------------------------------------------------
------
    // Constructors
//----------------------------------------------------------------------
--
    public CommonValue()
    {
        data = null;
    }

    public CommonValue(String s)
    {
        data = (Object) s;
        internalType = Types.VARCHAR;
    }

    public CommonValue(int i)
    {
        data = (Object) new Integer(i);
        internalType = Types.INTEGER;
    }

    public CommonValue(Integer i)
    {
        data = (Object) i;
        internalType = Types.INTEGER;
    }

    public CommonValue(byte b[])
    {
        data = (Object) b;
        internalType = Types.VARBINARY;
    }

//----------------------------------------------------------------------
-
    // isNull
    // returns true if the value is null
//----------------------------------------------------------------------
--
     public boolean isNull()
    {
         return (data == null);
    }
    //------------------------------------------------------------------
------
    // getMethods
//----------------------------------------------------------------------
--

    // Attempt to convert the data into a String. All data types
    // should be able to be converted.
    public String getString()
        throws SQLException
    {
```

```
    String s;

    // A null value always returns null
    if (data == null) {
        return null;
   }

    switch(internalType) {

  case Types.VARCHAR:
         s = (String) data;
        break;

   case Types.INTEGER:
          s = ((Integer) data).toString();
        break;

  case Types.VARBINARY:
        {
            // Convert a byte array into a String of hex digits
            byte b[] = (byte[]) data;
            int len = b.length;
          String digits = "0123456789ABCDEF";
           char c[] = new char[len * 2];

            for (int i = 0; i < len; i++) {
                c[i * 2] = digits.charAt((b[i] >> 4) & 0x0F);
                c[(i * 2) + 1] = digits.charAt(b[i] & 0x0F);
           }
            s = new String(c);
        }
        break;

    default:
        throw new SQLException("Unable to convert data type to
          String: " +
                            internalType);
    }

    return s;
}

// Attempt to convert the data into an int
public int getInt()
    throws SQLException
{
    int i = 0;

    // A null value always returns zero
    if (data == null) {
        return 0;
    }

    switch(internalType) {

    case Types.VARCHAR:
              i = (Integer.valueOf((String) data)).intValue();
        break;

    case Types.INTEGER:
           i = ((Integer) data).intValue();
        break;

    default:
        throw new SQLException("Unable to convert data type to
          String: " +
                             internalType);
    }

    return i;
}

// Attempt to convert the data into a byte array
 public byte[] getBytes()
    throws SQLException
{
    byte b[] = null;

    // A null value always returns null
    if (data == null) {
        return null;
   }

    switch(internalType) {

 case Types.VARCHAR:
      {

            // Convert the String into a byte array. The String must
           // contain an even number of hex digits.
            String s = ((String) data).toUpperCase();
           String digits = "0123456789ABCDEF";
            int len = s.length();
           int index;

            if ((len % 2) != 0) {
                throw new SQLException(
                        "Data must have an even number of hex
                          digits");
            }

            b = new byte[len / 2];

            for (int i = 0; i < (len / 2); i++) {
                   index = digits.indexOf(s.charAt(i * 2));

               if (index < 0) {
                      throw new SQLException("Invalid hex digit");
              }

                b[i] = (byte) (index << 4);
                 index = digits.indexOf(s.charAt((i * 2) + 1));

                if (index < 0) {
```

```
                              throw new SQLException("Invalid hex digit");
                     }
                       b[i] += (byte) index;
                   }
               }
               break;

           case Types.VARBINARY:
                  b = (byte[]) data;
                 break;

           default:
               throw new SQLException("Unable to convert data type to
                 byte[]: " +
                                     internalType);
          }
           return b;
       }

       protected Object data;
        protected int internalType;
}
```

Note that the SimpleText driver supports only character, integer, and binary data; thus, **CommonValue** only accepts these data types, and only attempts to convert data to these same types. A more robust driver would need to further implement this class to include more (if not all) data types.

Escape Clauses

Another consideration when implementing a JDBC driver is processing escape clauses. Escape clauses are used as extensions to SQL and provide a method to perform DBMS-specific extensions, which are interoperable among DBMSes. The JDBC driver must accept escape clauses and expand them into the native DBMS format before processing the SQL statement. While this sounds simple enough on the surface, this process may turn out to be an enormous task. If you are developing a driver that uses an existing DBMS, and the JDBC driver simply passes SQL statements to the DBMS, you may have to develop a parser to scan for escape clauses.

The following types of SQL extensions are defined:

- Date, time, and timestamp data
- Scalar functions such as numeric, string, and data type conversion
- LIKE predicate escape characters
- Outer joins
- Procedures

The JDBC specification does not directly address escape clauses; they are inherited from the ODBC specification. The syntax defined by ODBC uses the escape clause provided by the X/OPEN and SQL Access Group SQL CAE specification (1992). The general syntax for an escape clause is:

```
{escape}
```

We’ll cover the specific syntax for each type of escape clause in the following sections.

Date, Time, And Timestamp

The **date**, **time**, and **timestamp** escape clauses allow an application to specify date, time, and timestamp data in a uniform manner, without concern to the native DBMS format (for which the JDBC driver is responsible). The syntax for each (respectively) is

```
{d 'value'}
{t 'value'}
{ts 'value'}
```

where **d** indicates *value* is a date in the format yyyy-mm-dd, **t** indicates *value* is a time in the format hh:mm:ss, and **ts** indicates *value* is a timestamp in the format yyyy-mm-dd hh:mm:ss[.f...]. The following SQL statements illustrate the use of each:

```
UPDATE EMPLOYEE SET HIREDATE={d '1992-04-01'}
UPDATE EMPLOYEE SET LAST_IN={ts '1996-07-03 08:00:00'}
UPDATE EMPLOYEE SET BREAK_DUE={t '10:00:00'}
```

Scalar Functions

The five types of scalar functions—string, numeric, time and date, system, and data type conversion—all use the syntax:

```
{fn    scalar-function}
```

To determine what type of string functions a JDBC driver supports, an application can use the DatabaseMetaData method **getStringFunctions**. This method returns a comma-separated list of string functions, possibly containing ASCII, CHAR, CONCAT, DIFFERENCE, INSERT, LCASE, LEFT, LENGTH, LOCATE, LTRIM, REPEAT, REPLACE, RIGHT, RTRIM, SOUNDEX, SPACE, SUBSTRING, and/or UCASE.

To determine what type of numeric functions a JDBC driver supports, an application can use the DatabaseMetaData method **getNumericFunctions**. This method returns a comma-separated list of numeric functions, possibly containing ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, MOD, PI, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, TAN, and/or TRUNCATE.

To determine what type of system functions a JDBC driver supports, an application can use the **DatabaseMetaData** method **getSystemFunctions**. This method returns a comma-separated list of system functions, possibly containing DATABASE, IFNULL, and/or USER.

To determine what type of time and date functions a JDBC driver supports, an application can use the **DatabaseMetaData** method **getTimeDateFunctions**. This method returns a comma-separated list of time and date functions, possibly containing CURDATE, CURTIME, DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, HOUR, MINUTE, MONTH, MONTHNAME, NOW, QUARTER, SECOND, TIMESTAMPADD, TIMESTAMPDIFF, WEEK, and/or YEAR.

To determine what type of explicit data type conversions a JDBC driver supports, an application can use the **DatabaseMetaData** method **supportsConvert**. This method has two parameters: a *from* SQL data type and a *to* SQL data type. If the explicit data conversion between the two SQL types is supported, the method returns true. The syntax for the CONVERT function is

```
{fn  CONVERT(value,  data_type)}
```

where *value* is a column name, the result of another scalar function, or a literal, and *data_type* is one of the JDBC SQL types listed in the **Types** class.

LIKE Predicate Escape Characters

In a LIKE predicate, the "%" (percent character) matches zero or more of any character, and the "_" (underscore character) matches any one character. In some instances, an SQL query may have the need to search for one of these special matching characters. In such cases, you can use the "%" and "_" characters as literals in a LIKE predicate by preceding them with an escape character. The **DatabaseMetaData** method **getSearchStringEscape** returns the default escape character (which for most DBMSes will be the backslash character " \"). To override the escape character, use the following syntax:

```
{escape 'escape-character'}
```

The following SQL statement uses the LIKE predicate escape clause to search for any columns that start with the "%" character:

```
SELECT * FROM EMPLOYEE WHERE NAME LIKE '\%' {escape '\'}
```

Outer Joins

JDBC supports the ANSI SQL-92 LEFT OUTER JOIN syntax. The escape clause syntax is

```
{oj  outer-join}
```

where *outer-join* is the table-reference LEFT OUTER JOIN {table-reference | outer-join} ON search-condition.

Procedures

A JDBC application can call a procedure in place of an SQL statement. The escape clause used for calling a procedure is

```
{[?=]  call procedure-name[(param[, param]...)]}
```

where *procedure-name* specifies the name of a procedure stored on the data source, and *param* specifies procedure parameters. A procedure can have zero or more parameters, and may return a value.

The JDBC Interfaces

Now let's take a look at each of the JDBC interfaces, which are shown in Figure 10.2. We'll go over the major aspects of each interface and use code examples from our

SimpleText project whenever applicable. You should understand the JDBC API specification before attempting to create a JDBC driver; this section is meant to enhance the specification, not to replace it.

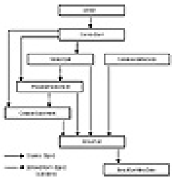


Figure 10.2 The JDBC interfaces.

Driver

The **Driver** class is the entry point for all JDBC drivers. From here, a connection to the database can be made in order to perform work. This class is intentionally very small; the intent is that JDBC drivers can be pre-registered with the system, enabling the **DriverManager** to select an appropriate driver given only a *URL* (Universal Resource Locator). The only way to determine which driver can service the given URL is to load the **Driver** class and let each driver respond via the **acceptsURL** method. To keep the amount of time required to find an appropriate driver to a minimum, each **Driver** class should be as small as possible so it can be loaded quickly.

Register Thyself

The very first thing that a driver should do is register itself with the **DriverManager**. The reason is simple: You need to tell the **DriverManager** that you exist; otherwise you may not be loaded. The following code illustrates one way of loading a JDBC driver:

```
java.sql.Driver d = (java.sql.Driver)
            Class.forName
("jdbc.SimpleText.SimpleTextDriver").newInstance();

Connection con = DriverManager.getConnection("jdbc:SimpleText", "", "");
```

The class loader will create a new instance of the SimpleText JDBC driver. The application then asks the **DriverManager** to create a connection using the given URL. If the SimpleText driver does not register itself, the **DriverManager** will not attempt to load it, which will result in a nasty “No capable driver” error.

The best place to register a driver is in the **Driver** constructor:

```
public  SimpleTextDriver()
     throws SQLException
{

     // Attempt to register this driver with the JDBC DriverManager.
     // If it fails, an exception will be thrown.
     DriverManager.registerDriver(this);

}
```

URL Processing

As I mentioned a moment ago, the **acceptsURL** method informs the **DriverManager** whether a given URL is supported by the driver. The general format for a JDBC URL is

jdbc:subprotocol:subname

where *subprotocol* is the particular database connectivity mechanism supported (note that this mechanism may be supported by multiple drivers) and the *subname* is defined by the JDBC driver. For example, the format for the JDBC-ODBC Bridge URL is:

jdbc:odbc:data source name

Thus, if an application requests a JDBC driver to service the URL of

```
jdbc:odbc:foobar
```

the only driver that will respond that the URL is supported is the JDBC-ODBC Bridge; all others will ignore the request.

Listing 10.14 shows the **acceptsURL** method for the SimpleText driver. The SimpleText driver will accept the following URL syntax:

```
jdbc:SimpleText
```

Note that no subname is required; if a subname is provided, it will be ignored.

Listing 10.14 The acceptsURL method.

```
//----------------------------------------------------------------------
--
// acceptsURL - JDBC API
//
// Returns true if the driver thinks that it can open a connection
// to the given URL. Typically, drivers will return true if they
// understand the subprotocol specified in the URL, and false if
// they don't.
//
//    url        The URL of the database.
//
// Returns true if this driver can connect to the given URL.
//----------------------------------------------------------------------
--
public  boolean acceptsURL(
       String url)
      throws SQLException
{
      if (traceOn()) {
           trace("@acceptsURL (url=" + url + ")");
    }

     boolean rc = false;
     // Get the subname from the url. If the url is not valid for
     // this driver, a null will be returned.

     if (getSubname(url) != null) {
         rc = true;
    }
     if (traceOn()) {
          trace(" " + rc);
    }
    return rc;
}

//------------------------------------------------------------------------
--
// getSubname
// Given a URL, return the subname. Returns null if the protocol is
// not "jdbc" or the subprotocol is not "simpletext."
//------------------------------------------------------------------------
--
public String getSubname(
      String url)
{
     String subname = null;
     String protocol = "JDBC";
    String subProtocol = "SIMPLETEXT";

    // Convert to uppercase and trim all leading and trailing
   // blanks.
     url = (url.toUpperCase()).trim();

    // Make sure the protocol is jdbc:
      if (url.startsWith(protocol)) {

         // Strip off the protocol
           url = url.substring (protocol.length());

        // Look for the colon
           if (url.startsWith(":")) {
               url = url.substring(1);

           // Check the subprotocol
              if (url.startsWith(subProtocol)) {

                // Strip off the subprotocol, leaving the subname
                   url = url.substring(subProtocol.length());

                // Look for the colon that separates the subname
                 // from the subprotocol (or the fact that there
                 // is no subprotocol at all).
                   if (url.startsWith(":")) {
                    subname = url.substring(subProtocol.length());
              }
                else if (url.length() == 0) {
                 subname = "";
              }
           }
        }
    }
    return subname;
}
```

Driver Properties

Connecting to a JDBC driver with only a URL specification is great, but the vast majority of the time, a driver will require additional information in order to properly connect to a database. The JDBC specification has addressed this issue with the **getPropertyInfo** method. Once a **Driver** has been instantiated, an application can use this method to find out what required and optional properties can be used to connect to the database. You may be tempted to require the application to embed properties within the URL subname, but by returning them from the **getPropertyInfo** method, you can identify the properties at runtime, giving a much more robust solution. Listing 10.15 shows an application that loads the SimpleText driver and gets the property information.

Listing 10.15 Using the getPropertyInfo method to identify properties at runtime.

```
import java.sql.*;

class PropertyTest {

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

            // Quick way to create a driver object
              java.sql.Driver d = new
jdbc.SimpleText.SimpleTextDriver();

            String url = "jdbc:SimpleText";
            // Make sure we have the proper URL
               if (!d.acceptsURL(url)) {
                throw new SQLException("Unknown URL: " + url);
            }

            // Setup a Properties object. This should contain an entry
             // for all known properties to this point. Properties that
             // have already been specified in the Properties object
will
             // not be returned by getPropertyInfo.
            java.util.Properties props = new java.util.Properties();

            // Get the property information
            DriverPropertyInfo info[] = d.getPropertyInfo(url, props);

          // Just dump them out
              System.out.println("Number of properties: " +
info.length);

            for (int i=0; i < info.length; i++) {
                 System.out.println("\nProperty " + (i + 1));
                System.out.println("Name:        " + info[i].name);
                 System.out.println("Description: " +
                   info[i].description);
                  System.out.println("Required:    " +
info[i].required);
                System.out.println("Value:       " + info[i].value);
                 System.out.println("Choices:     " + info[i].choices);
            }

        }
        catch (SQLException ex) {
               System.out.println ("\nSQLException(s) caught\n");

            // Remember that SQLExceptions may be chained together
              while (ex != null) {
                     System.out.println("SQLState: " +
ex.getSQLState());
                    System.out.println("Message:  " + ex.getMessage());
                     System.out.println ("");
                   ex = ex.getNextException ();
            }
        }
    }
}
```

Listing 10.15 produces the following output:

```
Number of properties: 1

Property 1
Name:        Directory
Description: Initial text file directory
Required:    false
Value:       null
Choices:     null
```

It doesn't take a lot of imagination to envision an application or applet that gathers the property information and prompts the user in order to connect to the database. The actual code to implement the **getPropertyInfo** method for the SimpleText driver is very simple, as shown in Listing 10.16.

Listing 10.16 Implementing the getPropertyInfo method.

```
//------------------------------------------------------------------------
--
// getPropertyInfo - JDBC API
//
// The getPropertyInfo method is intended to allow a generic GUI tool to
// discover what properties it should prompt a human for in order to get
// enough information to connect to a database. Note that depending on
// the values the human has supplied so far, additional values may
become
// necessary, so it may be necessary to iterate though several calls.
// to getPropertyInfo.
//
//    url        The URL of the database to connect to.
//
//    info    A proposed list of tag/value pairs that will be sent on
//            connect open.
//
// Returns an array of DriverPropertyInfo objects describing possible
//            properties. This array may be an empty array if no
//            properties are required.
//------------------------------------------------------------------------
--
```

```
public   DriverPropertyInfo[] getPropertyInfo(
      String url,
       java.util.Properties info)
     throws SQLException
{
      DriverPropertyInfo prop[];
     // Only one property required for the SimpleText driver, the
     // directory. Check the property list coming in. If the
     // directory is specified, return an empty list.
      if (info.getProperty("Directory") == null) {

        // Setup the DriverPropertyInfo entry
        prop = new DriverPropertyInfo[1];
          prop[0] = new DriverPropertyInfo("Directory", null);
            prop[0].description = "Initial text file directory";
        prop[0].required = false;

    }
    else {
        // Create an empty list
         prop = new DriverPropertyInfo[0];
    }

    return prop;

}
```

Let’s Get Connected

Now that we can identify a driver to provide services for a given URL and get a list of the required and optional parameters necessary, it’s time to establish a connection to the database. The **connect** method does just that, as shown in Listing 10.17, by taking a URL and connection property list and attempting to make a connection to the database. The first thing that **connect** should do is verify the URL (by making a call to **acceptsURL**). If the URL is not supported by the driver, a null value will be returned. This is the only reason that a null value should be returned. Any other errors during the **connect** should throw an **SQLException**.

Listing 10.17 Connecting to the database.

```
//---------------------------------------------------------------------
--
// connect - JDBC API
//
// Try to make a database connection to the given URL.
// The driver should return "null" if it realizes it is the wrong kind
// of driver to connect to the given URL. This will be common, as when
// the JDBC driver manager is asked to connect to a given URL, it passes
// the URL to each loaded driver in turn.
//
// The driver should raise an SQLException if it is the right
// driver to connect to the given URL, but has trouble connecting to
// the database.
//
// The java.util.Properties argument can be used to pass arbitrary
// string tag/value pairs as connection arguments.
// Normally, at least "user" and "password" properties should be
// included in the Properties.
//
//    url    The URL of the database to connect to.
//
//     info    a list of arbitrary string tag/value pairs as
//             connection arguments; normally, at least a "user" and
//             "password" property should be included.
//
// Returns a Connection to the URL.
//---------------------------------------------------------------------
--
public Connection connect(
      String url,
       java.util.Properties info)
    throws SQLException
{
    if (traceOn()) {
          trace("@connect (url=" + url + ")");
    }

    // Ensure that we can understand the given URL
     if (!acceptsURL(url)) {
          return null;
    }

    // For typical JDBC drivers, it would be appropriate to check
   // for a secure environment before connecting, and deny access
   // to the driver if it is deemed to be unsecure. For the
    // SimpleText driver, if the environment is not secure, we will
    // turn it into a read-only driver.

   // Create a new SimpleTextConnection object
    SimpleTextConnection con = new SimpleTextConnection();

     // Initialize the new object. This is where all of the
    // connection work is done.
    con.initialize(this, info);

   return con;
}
```

As you can see, there isn't a lot going on here for the SimpleText driver; remember that we need to keep the size of the **Driver** class implementation as small as possible. To aid in this, all of the code required to perform the database connection resides in the **Connection** class, which we'll discuss next.

Connection

The **Connection** class represents a session with the data source. From here, you can create **Statement** objects to execute SQL statements and gather database statistics.

Depending upon the database that you are using, multiple connections may be allowed for each driver.

For the SimpleText driver, we don't need to do anything more than actually connect to the database. In fact, there really isn't a database at all—just a bunch of text files. For typical database drivers, some type of connection context will be established, and default information will be set and gathered. During the SimpleText connection initialization, all that we need to do is check for a read-only condition (which can only occur within untrusted applets) and any properties that are supplied by the application, as shown in Listing 10.18.

Listing 10.18 SimpleText connection initialization.

```
public   void   initialize(
       Driver driver,
         java.util.Properties info)
      throws SQLException
{
    // Save the owning driver object
     ownerDriver = driver;

    // Get the security manager and see if we can write to a file.
    // If no security manager is present, assume that we are a trusted
     // application and have read/write privileges.
    canWrite = false;

    SecurityManager securityManager = System.getSecurityManager ();

    if (securityManager != null) {
        try {
            // Use some arbitrary file to check for file write
privileges
            securityManager.checkWrite ("SimpleText_Foo");
            // Flag is set if no exception is thrown
            canWrite = true;
        }

       // If we can't write, an exception is thrown. We'll catch
        // it and do nothing.
        catch (SecurityException ex) {
        }
    }
    else {
        canWrite = true;
    }

    // Set our initial read-only flag
    setReadOnly(!canWrite);

    // Get the directory. It will either be supplied in the property
    // list, or we'll use our current default.
    String s = info.getProperty("Directory");

    if (s == null) {
        s = System.getProperty("user.dir");
    }

    setCatalog(s);

}
```

Creating Statements

From the **Connection** object, an application can create three types of **Statement** objects. The base **Statement** object is used for executing SQL statements directly. The **PreparedStatement** object (which extends **Statement**) is used for pre-compiling SQL statements that may contain input parameters. The **CallableStatement** object (which extends **PreparedStatement**) is used to execute stored procedures that may contain both input and output parameters.

For the SimpleText driver, the **createStatement** method does nothing more than create a new **Statement** object. For most database systems, some type of statement context, or handle, will be created. One thing to note whenever an object is created in a JDBC driver: Save a reference to the owning object because you will need to obtain information (such as the connection context from within a **Statement** object) from the owning object.

Consider the **createStatement** method within the **Connection** class:

```
public Statement createStatement()
    throws SQLException
{
    if (traceOn()) {
        trace("Creating new SimpleTextStatement");

    }

    // Create a new Statement object
    SimpleTextStatement stmt = new SimpleTextStatement();

     // Initialize the statement
    stmt.initialize(this);

   return stmt;

}
```

Now consider the corresponding **initialize** method in the **Statement** class:

```
public void initialize(
    SimpleTextConnection con)
    throws SQLException
{
    // Save the owning connection object
    ownerConnection = con;
}
```

Which module will you compile first? You can't compile the **Connection** class until the **Statement** class has been compiled, and you can't compile the **Statement** class until the **Connection** class has been compiled. This is a circular dependency. Of course, the Java compiler does allow multiple files to be compiled at once, but some build environments do not support circular dependency. I have solved this problem in the SimpleText driver by defining some simple interface classes. In this way, the **Statement** class knows only about the general interface of the **Connection** class; the implementation of the interface does not need to be present. Our modified **initialize** method looks like this:

```
public void initialize(
    SimpleTextIConnection con)
    throws SQLException
{
    // Save the owning connection object
    ownerConnection = con;
}
```

Note that the only difference is the introduction of a new class, **SimpleTextIConnection**, which replaces **SimpleTextConnection**. I have chosen to preface the JDBC class name with an "I" to signify an interface. Here's the interface class:

```
public interface SimpleTextIConnection
    extends java.sql.Connection
{
    String[] parseSQL(String sql);
    Hashtable getTables(String directory, String table);
    Hashtable getColumns(String directory, String table);
    String getDirectory(String directory);
}
```

Note that our interface class extends the JDBC class, and our **Connection** class implements this new interface. This allows us to compile the interface first, then the **Statement**, followed by the **Connection**. Say good-bye to your circular dependency woes.

Now, back to the **Statement** objects. The **prepareStatement** and **prepareCall** methods of the **Connection** object both require an SQL statement to be provided. This SQL statement should be pre-compiled and stored with the **Statement** object. If any errors are present in the SQL statement, an exception should be raised, and the **Statement** object should not be created.

Tell Me About Yourself

One of the most powerful aspects of the JDBC specification (which was inherited from X/Open) is the ability for introspection. This is the process of asking a driver for information about what is supported, how it behaves, and what type of information exists in the database. The **getMetaData** method creates a **DatabaseMetaData** object which provides us with this wealth of information.

DatabaseMetaData

At over 130 methods, the **DatabaseMetaData** class is by far the largest. It supplies information about what is supported and how things are supported. It also supplies catalog information such as listing tables, columns, indexes, procedures, and so on. Because the JDBC API specification does an adequate job of explaining the methods contained in this class, and most of them are quite straightforward, we'll just take a look at how the SimpleText driver implements the **getTables** catalog method. But first, let's review the basic steps needed to implement each of the catalog methods (that is, those methods that return a **ResultSet**):

1. Create the result columns, which includes the column name, type, and other information about each of the columns. You should perform this step regardless of whether the database supports a given catalog function (such as stored procedures). I believe that it is much better to return an empty result set with only the column information than to raise an exception indicating that the database does not support the function. The JDBC specification does not currently address this issue, so it is open for interpretation.
2. Retrieve the catalog information from the database.
3. Perform any filtering necessary. The application may have specified the return of only a subset of the catalog information. You may need to filter the information in the JDBC driver if the database system doesn't.
4. Sort the result data per the JDBC API specification. If you are lucky, the database you are using will sort the data in the proper sequence. Most likely, it will not. In this case, you will need to ensure that the data is returned in the proper order.
5. Return a **ResultSet** containing the requested information.

The SimpleText **getTables** method will return a list of all of the text files in the catalog (directory) given. If no catalog is supplied, the default directory is used. Note that the SimpleText driver does not perform all of the steps shown previously; it does not provide any filtering, nor does it sort the data in the proper sequence. You are more than welcome to add this functionality. In fact, I encourage it. One note about column information: I prefer to use a **Hashtable** containing the column number as the key, and a class containing all of the information about the column as the data value. So, for all **ResultSets** that are generated, I create a **Hashtable** of column information that is then used by the **ResultSet** object and the **ResultSetMetaData** object to describe each column. Listing 10.19 shows the **SimpleTextColumn** class that is used to hold this information for each column.

Listing 10.19 The SimpleTextColumn class.

```
package jdbc.SimpleText;

public class SimpleTextColumn
    extends        Object
{
//--------------------------------------------------------------------
--
```

```
    // Constructor
//----------------------------------------------------------------------
--
    public SimpleTextColumn(
        String name,
        int type,
        int precision)
    {
        this.name = name;
        this.type = type;
        this.precision = precision;
    }

    public SimpleTextColumn(
        String name,
        int type)
    {
        this.name = name;
        this.type = type;
        this.precision = 0;
    }
    public SimpleTextColumn(
        String name)
    {
        this.name = name;
        this.type = 0;
        this.precision = 0;
    }

    public String name;
    public int type;
    public int precision;
    public boolean searchable;
    public int colNo;
    public int displaySize;
    public String typeName;
}
```

Note that I have used several constructors to set up various default information, and that all of the attributes are **public**. To follow object-oriented design, I should have provided a **get** and **set** method to encapsulate each attribute, but I chose to let each consumer of this object access them directly. Listing 10.20 shows the code for the **getTables** method.

Listing 10.20 The getTables method.

```
//----------------------------------------------------------------------
// getTables - JDBC API
// Get a description of tables available in a catalog
//
// Only table descriptions matching the catalog, schema, table
// name and type criteria are returned. They are ordered by
// TABLE_TYPE, TABLE_SCHEM, and TABLE_NAME.
//
// Each table description has the following columns:
//
//     (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
//              Typical types are "TABLE", "VIEW", "SYSTEM TABLE",
//           "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM"
//    (5) REMARKS       String => explanatory comment on the table
//
// Note: Some databases may not return information for
// all tables.
//
//    catalog            a catalog name; "" retrieves those without a
//                       catalog.
//    schemaPattern    a schema name pattern; "" retrieves those
//                     without a schema.
//    tableNamePattern    a table name pattern.
//    types            a list of table types to include; null returns
all
//                     types.
//
// Returns a ResultSet. Each row is a table description.
//-----------------------------------------------------------------------
public ResultSet getTables(
     String catalog,
     String schemaPattern,
     String tableNamePattern,
     String types[])
    throws SQLException
{
    if (traceOn()) {
        trace("@getTables(" + catalog + ", " + schemaPattern +
                ", " + tableNamePattern + ")");
    }

    // Create a statement object
    SimpleTextStatement stmt =
                (SimpleTextStatement) ownerConnection.createStatement();

    // Create a Hashtable for all of the columns
    Hashtable columns = new Hashtable();

    add(columns, 1, "TABLE_CAT", Types.VARCHAR);
    add(columns, 2, "TABLE_SCHEM", Types.VARCHAR);
    add(columns, 3, "TABLE_NAME", Types.VARCHAR);
    add(columns, 4, "TABLE_TYPE", Types.VARCHAR);
    add(columns, 5, "REMARKS", Types.VARCHAR);

    // Create an empty Hashtable for the rows
    Hashtable rows = new Hashtable();

    // If any of the parameters will return an empty result set, do so
    boolean willBeEmpty = false;

    // If table types are specified, make sure that 'TABLE' is
    // included. If not, no rows will be returned.

    if (types != null) {
        willBeEmpty = true;
        for (int ii = 0; ii < types.length; ii++) {
            if (types[ii].equalsIgnoreCase("TABLE")) {
                willBeEmpty = false;
                break;
            }
        }
    }
    if (!willBeEmpty) {
        // Get a Hashtable with all tables
        Hashtable tables = ownerConnection.getTables(
                    ownerConnection.getDirectory(catalog),
                      tableNamePattern);

        Hashtable singleRow;
        SimpleTextTable table;

        // Create a row for each table in the Hashtable
        for (int i = 0; i < tables.size(); i++) {
            table = (SimpleTextTable) tables.get(new Integer(i));

            // Create a new Hashtable for a single row
            singleRow = new Hashtable();

            // Build the row
            singleRow.put(new Integer(1), new CommonValue(table.dir));
            singleRow.put(new Integer(3), new CommonValue(table.name));
           singleRow.put(new Integer(4), new CommonValue("TABLE"));

            // Add it to the row list
            rows.put(new Integer(i + 1), singleRow);
        }
    }

    // Create the ResultSet object and return it
    SimpleTextResultSet rs = new SimpleTextResultSet();

    rs.initialize(stmt, columns, rows);

    return rs;
}
```

Let's take a closer look at what's going on here. The first thing we do is create a **Statement** object to "fake out" the **ResultSet** object that we will be creating to return back to the application. The **ResultSet** object is dependent upon a **Statement** object, so we'll give it one. The next thing we do is create all of the column information. Note that all of the required columns are given in the JDBC API specification. The **add** method simply adds a **SimpleTextColumn** object to the **Hashtable** of columns:

```
protected  void  add(
    Hashtable h,
     int col,
    String name,
    int type)
{
     h.put(new Integer(col), new SimpleTextColumn(name,type));
}
```

Next, we create another **Hashtable** to hold all of the data for all of the catalog rows. The **Hashtable** contains an entry for each row of data. The entry contains the key, which is the row number, and the data value, which is yet another **Hashtable** whose key is the column number and whose data value is a **CommonValue** object containing the actual data. Remember that the **CommonValue** class provides us with the mechanism to store data and coerce it as requested by the application. If a column is null, we simply cannot store any information in the **Hashtable** for that column number.

After some sanity checking to ensure that we really need to look for the catalog information, we get a list of all of the tables. The **getTables** method in the **Connection** class provides us with a list of all of the SimpleText data files:

```
public  Hashtable  getTables(
      String dir,
     String table)
{
     Hashtable list = new Hashtable();

     // Create a FilenameFilter object. This object will only allow
    // files with the .SDF extension to be seen.
     FilenameFilter  filter = new SimpleTextEndsWith(
                    SimpleTextDefine.DATA_FILE_EXT);

     File file = new File(dir);

      if (file.isDirectory()) {

        // List all of the files in the directory with the .SDF
extension
           String entries[] = file.list(filter);
       SimpleTextTable tableEntry;

         // Create a SimpleTextTable entry for each, and put in
        // the Hashtable.
          for (int i = 0; i < entries.length; i++) {

             // A complete driver needs to further filter the table
          // name here.
              tableEntry = new SimpleTextTable(dir, entries[i]);
                   list.put(new Integer(i), tableEntry);

        }
    }

     return list;
}
```

Again, I use a **Hashtable** for each table (or file in our case) that is found. By now, you will have realized that I really like using **Hashtables**; they can grow in size dynamically and provide quick access to data. And because a **Hashtable** stores data as an abstract **Object**, I can store whatever is necessary. In this case, each **Hashtable** entry for a table contains a SimpleTextTable object:

```
public   class   SimpleTextTable
```

```
     extends        Object
{
//----------------------------------------------------------------------
--
     // Constructor
//----------------------------------------------------------------------
--
      public SimpleTextTable(
           String dir,
           String file)
    {
           this.dir = dir;
           this.file = file;

         // If the filename has the .SDF extension, get rid of it
          if (file.endsWith(SimpleTextDefine.DATA_FILE_EXT)) {
            name = file.substring(0, file.length() -
                      SimpleTextDefine.DATA_FILE_EXT.length());
        }
        else {
            name = file;
        }
    }

     public String dir;
     public String file;
    public String name;
}
```

Notice that the constructor strips the file extension from the given file name, creating the table name.

Now, back to the **getTables** method for **DatabaseMetaData**. Once a list of all of the tables has been retrieved, the **Hashtable** used for storing all of the rows is generated. If you were to add additional filtering, this is the place that it should be done. Finally, a new **ResultSet** object is created and initialized. One of the constructors for the **ResultSet** class accepts two **Hashtables**: one for the column information (**SimpleTextColumn** objects), and the other for row data (**CommonValue** objects). We'll see later how these are handled by the **ResultSet** class. For now, just note that it can handle both in-memory results (in the form of a **Hashtable**) and results read directly from the data file.

Statement

The **Statement** class contains methods to execute SQL statements directly against the database and to obtain the results. A **Statement** object is created using the **createStatement** method from the **Connection** object. Of note in Listing 10.21 are the three methods used to execute SQL statements: **executeUpdate**, **executeQuery**, and **execute**. In actuality, you only need to worry about implementing the **execute** method; the other methods use it to perform their work. In fact, the code provided in the SimpleText driver should be identical for all JDBC drivers.

Listing 10.21 Executing SQL statements.

```
//----------------------------------------------------------------------
--
// executeQuery - JDBC API
// Execute an SQL statement that returns a single ResultSet.
//
//    sql    Typically this is a static SQL SELECT statement.
//
// Returns the table of data produced by the SQL statement.
//----------------------------------------------------------------------
--
public  ResultSet  executeQuery(
      String sql)
     throws SQLException
{
      if (traceOn()) {
           trace("@executeQuery(" + sql + ")");
    }

      java.sql.ResultSet rs = null;

    // Execute the query. If execute returns true, then a result set
    // exists.
     if (execute(sql)) {
          rs = getResultSet();
    }
    else {          // If the statement does not create a ResultSet, the
                    // specification indicates that an SQLException
should
                    // be raised.
        throw new SQLException("Statement did not create a ResultSet");
    }
     return rs;
}

//----------------------------------------------------------------------
--
// executeUpdate - JDBC API
// Execute an SQL INSERT, UPDATE, or DELETE statement. In addition,
// SQL statements that return nothing, such as SQL DDL statements,
// can be executed.
//
//    sql    an SQL INSERT, UPDATE, or DELETE statement, or an SQL
//        statement that returns nothing.
//
// Returns either the row count for INSERT, UPDATE, or DELETE; or 0
// for SQL statements that return nothing.
//----------------------------------------------------------------------
--
public   int   executeUpdate(
      String sql)
     throws SQLException
{
      if (traceOn()) {
           trace("@executeUpdate(" + sql + ")");
    }
     int count = -1;
    // Execute the query. If execute returns false, then an update
    // count exists.
     if (execute(sql) == false) {
         count = getUpdateCount();
    }
    else {
        // If the statement does not create an update count, the
         // specification indicates that an SQLException should be
raised.
        throw new SQLException("Statement did not create an update
          count");
    }

    return count;
}
```

As you can see, **executeQuery** and **executeUpdate** are simply helper methods for an application; they are built completely upon other methods contained within the class. The **execute** method accepts an SQL statement as its only parameter, and will be implemented differently, depending upon the underlying database system. For the SimpleText driver, the SQL statement will be parsed, prepared, and executed. Note that parameter markers are not allowed when executing an SQL statement directly. If the SQL statement created results containing columnar data, **execute** will return true; if the statement created a count of rows affected, **execute** will return false. If **execute** returns true, the application then uses **getResultSet** to return the current result information; otherwise, **getUpdateCount** will return the number of rows affected.

Warnings

As opposed to **SQLException**, which indicates a critical error, an **SQLWarning** can be issued to provide additional information to the application. Even though **SQLWarning** is derived from **SQLException**, warnings are not thrown. Instead, if a warning is issued, it is placed on a warning stack with the **Statement** object (the same holds true for the **Connection** and **ResultSet** objects). The application must then check for warnings after every operation using the **getWarnings** method. At first, this may seem a bit cumbersome, but when you consider the alternative of wrapping **try**...**catch** statements around each operation, this seems like a better solution. Note also that warnings can be chained together, just like **SQLExceptions** (for more information on chaining, see the *JDBC Exception Types* section earlier in this chapter).

Two (Or More) For The Price Of One

Some database systems allow SQL statements that return multiple results (columnar data or an update count) to be executed. If you are unfortunate enough to be developing a JDBC driver using one of these database systems, take heart. The JDBC specification does address this issue. The **getMoreResults** method is intended to move through the results. Figuring out when you have reached the end of the results, however, is a bit convoluted. To do so, you first call **getMoreResults**. If it returns true, there is another **ResultSet** present and you can use **getResultSet** to retrieve it. If **getMoreResults** returns false, you have either reached the end of the results, or an update count exists; you must call **getUpdateCount** to determine which situation exists. If **getUpdateCount** returns -1, you have reached the end of the results; otherwise, it will return the number of rows affected by the statement.

The SimpleText driver does not support multiple result sets, so I don't have any example code to present to you. The only DBMS that I am aware of that supports this is Sybase. Because there are already multiple JDBC drivers available for Sybase (one of which I have developed), I doubt you will have to be concerned with **getMoreResults**. Consider yourself lucky.

PreparedStatement

The **PreparedStatement** is used for pre-compiling an SQL statement, typically in conjunction with parameters, and can be efficiently executed multiple times with just a change in a parameter value; the SQL statement does not have to be parsed and compiled each time. Because the **PreparedStatement** class extends the **Statement** class, you will have already implemented a majority of the methods. The **executeQuery**, **executeUpdate**, and **execute** methods are very similar to the **Statement** methods of the same name, but they do not take an SQL statement as a parameter. The SQL statement for the **PreparedStatement** was provided when the object was created with the **prepareStatement** method from the **Connection** object. One danger to note here: Because **PreparedStatement** is derived from the **Statement** class, all of the methods in **Statement** are also in **PreparedStatement**. The three execute methods from the **Statement** class that accept SQL statements are not valid for the **PreparedStatement** class. To prevent an application from invoking these methods, the driver should also implement them in **PreparedStatement**, as shown here:

```
// The overloaded executeQuery on the Statement object (which we
// extend) is not valid for PreparedStatement or CallableStatement
// objects.
public  ResultSet executeQuery(
      String sql)
     throws SQLException
{
      throw new SQLException("Method is not valid");
}

// The overloaded executeUpdate on the Statement object (which we
// extend) is not valid for PreparedStatement or CallableStatement
// objects.
public  int  executeUpdate(
      String sql)
     throws SQLException
{
      throw new SQLException("Method is not valid");
}

// The overloaded execute on the Statement object (which we
// extend) is not valid for PreparedStatement or CallableStatement
// objects.
public   boolean   execute(
      String sql)
     throws SQLException
{
      throw new SQLException("Method is not valid");
}
```

Setting Parameter Values

The **PreparedStatement** class introduces a series of "set" methods to set the value of a specified parameter. Take the following SQL statement:

```
INSERT INTO FOO VALUES (?, ?, ?)
```

If this statement was used in creating a **PreparedStatement** object, you would need to set the value of each parameter before executing it. In the SimpleText driver, parameter values are kept in a **Hashtable**. The **Hashtable** contains the parameter number as the key, and a **CommonValue** object as the data object. By using a **CommonValue** object, the application can set the parameter using any one of the supported data types, and we can coerce the data into the format that we need in order to bind the parameter. Here's the code for the **setString** method:

```
public   void   setString(
      int parameterIndex,
     String x)
    throws SQLException
{
    // Validate the parameter index
      verify(parameterIndex);

    // Put the parameter into the boundParams Hashtable
      boundParams.put(new Integer(parameterIndex), x);
}
```

The **verify** method validates that the given parameter index is valid for the current prepared statement, and also clears any previously bound value for that parameter index:

```
protected   void   verify(
      int parameterIndex)
     throws SQLException
{

      clearWarnings();

      // The paramCount was set when the statement was prepared
       if ((parameterIndex <= 0) ||
           (parameterIndex > paramCount)) {
           throw new SQLException("Invalid parameter number: " +
                                            parameterIndex);
    }

      // If the parameter has already been set, clear it
       if (boundParams.get(new Integer(parameterIndex)) != null) {
            boundParams.remove(new Integer(parameterIndex));
    }
}
```

Because the **CommonValue** class does not yet support all of the JDBC data types, not all of the set methods have been implemented in the SimpleText driver. You can see, however, how easy it would be to fully implement these methods once **CommonValue** supported all of the necessary data coercion.

What Is It?

Another way to set parameter values is by using the **setObject** method. This method can easily be built upon the other set methods. Of interest here is the ability to set an **Object** without giving the JDBC driver the type of driver being set. The SimpleText driver implements a simple method to determine the type of object, given only the object itself:

```
protected   int   getObjectType(
     Object x)
     throws SQLException
{

    // Determine the data type of the Object by attempting to cast
     // the object. An exception will be thrown if an invalid casting
     // is attempted.
    try {
           if ((String) x != null) {
             return Types.VARCHAR;
        }
    }
    catch (Exception ex) {
    }

   try {
           if ((Integer) x != null) {
              return Types.INTEGER;
        }
    }
    catch (Exception ex) {
    }

    try {
           if ((byte[]) x != null) {
              return Types.VARBINARY;
        }
    }
    catch (Exception ex) {
    }

    throw new SQLException("Unknown object type");
}
```

Setting InputStreams

As we'll see with **ResultSet** later, using **InputStreams** is the recommended way to work with long data (blobs). There are two ways to treat **InputStreams** when using them as input parameters: Read the entire **InputStream** when the parameter is set and treat it as a large data object, or defer the read until the statement is executed and read it in chunks at a time. The latter approach is the preferred method because the contents of an

InputStream may be too large to fit into memory. Here's what the SimpleText driver does with **InputStreams**:

```
public   void   setBinaryStream(
      int parameterIndex,
      java.io.InputStream x,
     int length)
    throws SQLException
{

    // Validate the parameter index
      verify(parameterIndex);

    // Read in the entire InputStream all at once. A more optimal
    // way of handling this would be to defer the read until execute
    // time, and only read in chunks at a time.
    byte b[] = new byte[length];

    try {
        x.read(b);
    }
    catch (Exception ex) {
        throw new SQLException("Unable to read InputStream: " +
                                 ex.getMessage());
    }

    // Set the data as a byte array
      setBytes(parameterIndex, b);
}
```

But wait, this isn't the preferred way! You are correct, it isn't. The SimpleText driver simply reads in the entire **InputStream** and then sets the parameter as a byte array. I'll leave it up to you to modify the driver to defer the read until execute time.

ResultSet

The **ResultSet** class provides methods to access data generated by a table query. This includes a series of get methods which retrieve data in any one of the JDBC SQL type formats, either by column number or by column name. When the issue of providing get methods was first introduced by JavaSoft, some disgruntled programmers argued that they were not necessary; if an application wanted to get data in this manner, then the application could provide a routine to cross reference the column name to a column number. Unfortunately (in my opinion), JavaSoft chose to keep these methods in the API and provide the implementation of the cross reference method in an appendix. Because it is part of the API, all drivers must implement the methods. Implementing the methods is not all that difficult, but it is tedious and adds overhead to the driver. The driver simply takes the column name that is given, gets the corresponding column number for the column name, and invokes the same get method using the column number:

```
public   String   getString(
     String columnName)
     throws SQLException
{
       return getString(findColumn(columnName));
}
```

And here’s the **findColumn** routine:

```
public int findColumn(
    String columnName)
    throws SQLException
{
    // Make a mapping cache if we don't already have one
     if (md == null) {
         md = getMetaData();
         s2c = new Hashtable();
    }
    // Look for the mapping in our cache
    Integer x = (Integer) s2c.get(columnName);

    if (x != null) {
        return (x.intValue());
    }

    // OK, we'll have to use metadata
     for (int i = 1; i < md.getColumnCount(); i++) {
          if (md.getColumnName(i).equalsIgnoreCase(columnName)) {

              // Success! Add an entry to the cache
               s2c.put(columnName, new Integer(i));
               return (i);
        }
    }

    throw new SQLException("Column name not found: " + columnName,
                             "S0022");
}
```

This method uses a **Hashtable** to cache the column number and column names.

It’s Your Way, Right Away

An application can request column data in any one of the supported JDBC data types. As we have discussed before, the driver should coerce the data into the proper format. The SimpleText driver accomplishes this by using a **CommonValue** object for all data values. Therefore, the data can be served in any format, stored as a **CommonValue** object, and the application can request it in any other supported format. Let’s take a look at the **getString** method:

```
public String getString(
    int columnIndex)
    throws SQLException
{
    // Verify the column and get the absolute column number for the
    // table.
     int colNo = verify(columnIndex);

    String s = null;

    if (inMemoryRows != null) {
          s = (getColumn(rowNum, columnIndex)).getString();
    }
    else {
        CommonValue value = getValue(colNo);

        if (value != null) {
            s = value.getString();
        }
    }
    if (s == null) {
        lastNull = true;
    }

    return s;
}
```

The method starts out by verifying that the given column number is valid. If it is not, an exception is thrown. Some other types of initialization are also performed. Remember that all **ResultSet** objects are provided with a **Hashtable** of **SimpleTextColumn** objects describing each column:

```
protected int verify(
     int column)
    throws SQLException
{
     clearWarnings();
    lastNull = false;

   SimpleTextColumn col = (SimpleTextColumn) inMemoryColumns.get(
                                    new Integer(column));

    if (col == null) {
         throw new SQLException("Invalid column number: " + column);
  }
    return col.colNo;
}
```

Next, if the row data is stored in an in-memory **Hashtable** (as with the **DatabaseMetaData** catalog methods), the data is retrieved from the **Hashtable**. Otherwise, the driver gets the data from the data file. In both instances, the data is retrieved as a **CommonValue** object, and the **getString** method is used to format the data into the requested data type. Null values are handled specially; the JDBC API has a **wasNull** method that will return true if the last column that was retrieved was null:

```
public boolean wasNull()
    throws SQLException
{
    return lastNull;
}
```

The SimpleText driver also supports **InputStreams**. In our case, the **SimpleTextInputStream** class is just a simple wrapper around a **CommonValue** object. Thus, if an application requests the data for a column as an **InputStream**, the SimpleText driver will get the data as a **CommonValue** object (as it always does) and create an **InputStream** that fetches the data from the **CommonValue**.

The **getMetaData** method returns a **ResultSetMetaData** object, which is our last class to cover.

ResultSetMetaData

The **ResultSetMetaData** class provides methods that describe each one of the columns in a result set. This includes the column count, column attributes, and the column name. **ResultSetMetaData** will typically be the smallest class in a JDBC driver, and is usually very straightforward to implement. For the SimpleText driver, all of the necessary information is retrieved from the **Hashtable** of column information that is required for all result sets. Thus, to retrieve the column name:

```
public   String getColumnLabel(
       int column)
      throws SQLException
{
     // Use the column name
      return getColumnName(column);
}

protected SimpleTextColumn getColumn(
    int col)
   throws SQLException
{
    SimpleTextColumn column = (SimpleTextColumn)
                        inMemoryColumns.get(new Integer(col));

    if (column == null) {
        throw new SQLException("Invalid column number: " + col);
    }

    return column;
}
```

Summary

We have covered a lot of material in this chapter, including the JDBC **DriverManager** and the services that it provides, implementing Java interfaces, creating native JDBC drivers, tracing, data coercion, escape sequence processing, and each one of the major JDBC interfaces. This information, in conjunction with the SimpleText driver, should help you to create your own JDBC driver without too much difficulty.

Chapter 11
Internet Database Issues: Middleware

The JDBC specification says that the JDBC API should serve as a platform for building so-called “three-tier” client/server systems, often called *middleware*. As you might imagine, these systems have three basic components: the client, the server, and the application server. Figure 11.1 shows the basic structure of a three-tier system.

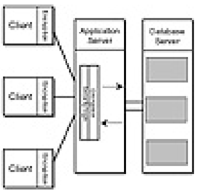


Figure 11.1 Three-tier system structure.

In this chapter, I'll provide you with the code necessary to implement a simple application server of your own. We'll also take a look at building a client for our home-grown application server. But before we get to the coding, we first need to discuss why we would want to go to such lengths to build a three-tier system instead of allowing direct database access.

Several middleware solutions based on the JDBC are already available, and although you may ultimately decide to buy one from a vendor instead of coding one yourself, I feel that it's important to learn the issues involved with middleware. Knowing the advantages and disadvantages that go along with inserting a middle tier to a system can help you decide if you need one.

Connectivity Issues Involved With Database Access

Let's begin by examining some issues of database scalabilty that you are likely to encounter. The Internet and large intranet scenarios pose interesting dilemmas for databases that serve a large number of users:

- **Concurrency**—Suppose a user receives some data from the database server, and while the user is looking at it, the data on the database server is changed in some way. For the user to see the updated material, both the database server and the client need to be able to handle the change. While some database servers can handle the necessary coding (and the increased load on the server) for updating, some cannot.
- **Legacy Databases**—Some legacy database systems may not support simultaneous connections, or even direct connections using TCP/IP.
- **Security**—Most database servers do not support encrypted connections, which means that certain transactions, such as the login using a password, will not be secure. Over the Internet, such a lack of security is a major hole.
- **Simultaneous Connections**—Database servers have a limit on the number of active connections. Unfortunately, exceeding this predefined limit on the Internet is easy.

Advantages Of Middleware

Let's now have a look at how a middle tier can address the issues presented in the previous section, while adding extra capability to a client/server system:

- **Concurrency**—You can program the application server to handle concurrency issues, off-loading the task from the database server. Of course, you would also need to program the clients to respond to update broadcasts. You can implement concurrency checking entirely on the application server, if necessary. This process involves checking to see if a specific data object requested by a client has changed since the current request, asking the client to update the previously retrieved data, and alerting the user.
- **Legacy Databases**—Databases that operate on older network protocols can be piped through an application server running on a machine that can communicate with the database server, as well as with remote Internet clients. A JDBC driver that can speak to a non-networked legacy database can be used to provide Internet access to its data, even using an ODBC driver, courtesy of the JDBC-ODBC Bridge. The application server can reside on the same machine as the non-networked database, and provide network access using a client that communicates to the application server.
- **Security**—You can program/obtain an application server that supports a secure connection to the remote clients. If you keep the local connection between the database server and the application server restricted to each other, you can create a fairly secure system. In this type of setup, your database server can only talk to the application server, so the threat of someone connecting directly to the database server and causing damage is greatly limited. However, you must be sure that there are no loopholes in your application server.
- **Simultaneous Connections**—The application server, in theory, can maintain only one active connection to the database server. On the other side, it can allow as many connections to itself from clients as it wants. In practice, however, significant speed problems will arise as more users attempt to use one connection. Managing a number of fixed connections to the database server is possible, though, so this speed degradation is not noticeable.

Disadvantages Of Middleware

Of course, middleware is not without its own pitfalls. Let's take a brief look at some disadvantages you may encounter if you choose to implement an application server:

- **Speed**—As I've hinted, speed is the main drawback to running an application server, especially if the application server is running on a slow machine. If the application server does not run on the same machine as the database server, there may be additional speed loss as the two communicate with each other.
- **Security**—If your application server is not properly secured, additional security holes could easily crop up. For example, a rogue user could break into the application server, then break into the database server using the application server's functions. Again, you must take great care to make sure that unauthorized access to the database server via the application server is not possible.
- **Reliability**—Adding an application server to the system introduces potential problems that may not be present in a two-tier system, where the clients are communicating directly with the database server.

The Application Server: A Complete Example With Code

I've shown you the advantages and disadvantages of implementing an application server; it's up to you to weigh these points and other relating factors when it comes time to make a decision on your own system. Let's look at a fully functional application server. The application server shown in Listing 11.1 uses JDBC to interact with data sources, so any JDBC driver could be used. I used the mSQL driver in this example, but you can easily modify the code to use the JDBC-ODBC Bridge, and then use the ODBC drivers for Access 95 to allow applets to query an Access 95 database. (This is an interesting scenario, because Access does not provide direct network connectivity in the form of a true "database server.") This application server is truly multithreaded—it spawns each client connection into its own thread. Each client connection also make a new instance of the JDBC driver, so each client has its own virtual connection to the data source via the application server.

The application server only allows two real functions:

- Connect to a predefined data source
- Make Select queries against the data source

The query is processed against the data source, and the result is piped directly back to the client in pre-formatted text. You can easily extend this approach so that a ResultSet can be encapsulated and sent unprocessed to the client by using the upcoming remote objects specification from JavaSoft. For the purposes of this example, I won't make it too elaborate and instead just send over the results in a delimited String format. The client is not a true JDBC client in that it does not implement a JDBC driver; it uses the two functions defined earlier to make queries. The results can be parsed by the applet calling the client, but for the purpose of this simple example, we'll just show them to the user (you'll see this when we show the code for the client).

You can find the source file for Listing 11.1 on the CD-ROM or on The Coriolis Group's Web site at http://www.coriolis.com/jdbc-book. Figure 11.2 shows the application server's window.

Figure 11.2 The application server console.

Listing 11.1 Application server.

```
import java.awt.List;
import java.awt.Frame;
import java.net.*;
import java.io.*;
import java.util.*;
import java.sql.*;
// Remember that we are using the JDBC driver on the _server_ to connect
// to a data source, so we need the JDBC API classes!

public class ApplicationServer extends Thread {
  public final static int DEFAULT_PORT = 6001;
  protected int port;
  protected ServerSocket server_port;
  protected ThreadGroup CurrentConnections;
  protected List connection_list;
  protected Vector connections;
  protected ConnectionWatcher watcher;
  public Frame f;
  // We plan on showing the connections to the server, so we need a
frame

  // Exit with an error message if there's an exception
  public static void fail(Exception e, String msg) {
    System.err.println(msg + ": " +  e);
    System.exit(1);
   }

  // Create a ServerSocket to listen for connections and start its
thread.
  public ApplicationServer(int port) {
    // Create our server thread with a name
    super("Server");
    if (port == 0) port = DEFAULT_PORT;
    this.port = port;
    try { server_port = new ServerSocket(port); }
    catch (IOException e) {fail(e, "Exception creating server socket");}
    // Create a threadgroup for our connections
    CurrentConnections = new ThreadGroup("Server Connections");
    // Create a window to display our connections in
    f = new Frame("Server Status");
    connection_list = new List();
    f.add("Center", connection_list);
    f.resize(400, 200);
    f.show();

    // Initialize a vector to store our connections in
    connections = new Vector();
    // Create a ConnectionWatcher thread to wait for other threads to
die
    // and to perform clean-up.
    watcher = new ConnectionWatcher(this);
    // Start the server listening for connections
    this.start();
    }

    public void run() {
      // this is where new connections are listened for
        try {
           while(true) {
               Socket client_socket = server_port.accept();
            ServerConnection c = new ServerConnection(client_socket,
              CurrentConnections, 3, watcher);
              // Prevent simultaneous access
              synchronized (connections) {
                connections.addElement(c);
                connection_list.addItem(c.getInfo());
              }
          }
    }
    catch (IOException e) {fail(e, "Exception while listening for
      connections");}
      f.dispose();
      System.exit(0);
  }

  // Start the server up, get a port number if specified
  public static void main(String[] args) {
      int port = 0;
      if (args.length == 1) {
        try {port = Integer.parseInt(args[0]);}
        catch (NumberFormatException e) {port = 0;}
      }
      new ApplicationServer(port);
  }
}

// This class is the thread that handles all communication with a client.
// It also notifies the ConnectionWatcher when the connection is dropped.
class ServerConnection extends Thread {
    static int numberOfConnections = 0;
    protected Socket client;
    protected ConnectionWatcher watcher;
    protected DataInputStream in;
    protected PrintStream out;
  Connection con;
```

```
    // Initialize the streams and start the thread
    public ServerConnection(Socket client_socket, ThreadGroup
      CurrentConnections,
              int priority, ConnectionWatcher watcher) {
        // Give the thread a group, a name, and a priority
        super(CurrentConnections, "Connection number" +
        numberOfConnections++);
        this.setPriority(priority);

        // We'll need this data later, so store it in local objects
        client = client_socket;
        this.watcher = watcher;

        // Create the streams for talking with client
        try {
            in = new DataInputStream(client.getInputStream());
            out = new PrintStream(client.getOutputStream());
        }
        catch (IOException e) {
          try {client.close();} catch (IOException e2) {
            System.err.println("Exception while getting socket streams:
"
            + e); return;}
        }
        // And start the thread up
        this.start();
    }

// This is where the real "functionality" of the server takes place.
// This is where the input and output is done to the client.
    public void run() {
        String inline;
        try {
        // Loop forever, or until the connection is broken!
            while(true) {
                // Read in a line
                inline = in.readLine();
                if (inline == null) break;
                      // If the client has broken connection, get out of
                      // the loop

                   inline=inline.trim();
                   // Get rid of leading and trailing whitespace

                   // These are the two functions implemented, connect
                   // and query. The client sends one of these commands,
                   // and if it's query ("S") then the server expects
the
                   // next line sent to be the query.
                   switch(inline.toCharArray()[0]) {
                   case `L': out.println("Connected to datasource");
                           out.println("DONE");
           ConnectToDatasource("jdbc:msql://elanor:1112/bcancer",
           "prpatel");
               // See this method next... it starts up the driver and
               // connects to the data source.
```

```
                      break;
               case `S': out.println("Run query: send SQL Query");
                       out.println("DONE");
                       inline = in.readLine();
                               inline=inline.trim();
               // This line gets the query sent here, runs its against
               // the connected data source, and returns the results in
               // formatted text
                            out.print(RunQuery(inline));
               // RunQuery is the method that runs the passed in
               // query using the initialized driver and connection.
               out.println("DONE");
                            break;
               default: out.println("ERROR - Invalid Request");
                     out.println("DONE");
               }

                     out.flush();
            }
        }
        catch (IOException e) {}

        // If the client broke off the connection, notify the
        // ConnectionWatcher
        // (watcher) which will close the connection.
        finally {
          try {client.close();}
           catch (IOException e2) {
           synchronized (watcher) {watcher.notify();}
          }       }
    }

// This sends info back to the connection starter so that it can
// be displayed in the frame.
  public String getInfo() {
     return ("Client connected from:"+client.getInetAddress().
     getHostName());
     }

  // DB specific stuff follows
private void ConnectToDatasource(String url, String Name) {
try {
     new imaginary.sql.iMsqlDriver();
     con = DriverManager.getConnection(url, Name, "");
// Create an instance of the driver and connect to the DB server
    }
    catch( Exception e ) {
      e.printStackTrace(); System.out.println(e.getMessage());
    }
}

private String RunQuery(String QueryLine) {
// Run the passed in query and return the Stringified results
  String Output="";
  int columns;
  int pos;
  try {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(QueryLine);
     columns=(rs.getMetaData()).getColumnCount();

      while(rs.next()) {

        for( pos=1; pos<=columns; pos++) {

          Output+=rs.getObject(pos)+" ";
        }
        Output+="\n";

      }
      stmt.close();
      //      con.close();
    }
    catch( Exception e ) {
      e.printStackTrace();
      Output=e.getMessage();
    }
    return Output;
  }
  // End DB specific stuff

} // End class Connection

// This class cleans up closed connections and updates the displayed
// list of connected clients.
class ConnectionWatcher extends Thread {
    protected ApplicationServer server;
    protected ConnectionWatcher(ApplicationServer s) {
        super(s.CurrentConnections, "ConnectionWatcher");
        server = s;
        this.start();
    }

    public synchronized void run() {
        while(true) {
          try {this.wait(10000);}
            catch (InterruptedException e){
              System.out.println("Caught an Interrupted Exception");
              }
              // Prevent simultaneous access
            synchronized(server.connections) {
              // Loop through the connections
              for(int i = 0; i < server.connections.size(); i++) {
                  ServerConnection c;
                  c = (ServerConnection)server.connections.elementAt(i);
                  // If the connection thread isn't alive anymore,
                  // remove it from the Vector and List.
                  if (!c.isAlive()) {
                      server.connections.removeElementAt(i);
                              server.connection_list.delItem(i);
                     i--;
                  }
              }
          }
                             }
       }
```

The Client: A Complete Example With Code

Now that we have the server code, let’s look at the client class, which is shown in Listing 11.2. This client class is not self-standing; we’ll need an applet to call this class and make use of the methods we define in it. The code for a sample applet that calls this client class is shown in Listing 11.3. Note that the client is specially coded to communicate with the application server in Listing 11.1, and that it does not require the Web browser it is run on to have the JDBC API classes. For our simple example, we don’t need to implement all of the functionality that is demanded of a JDBC driver, so I didn’t write one; a JDBC driver that can talk to our application server would not be difficult to write at this point, however, because we have a simple “command set” and simple functionality. Figure 11.3 shows the client applet in Listing 11.3, which uses the Dbclient class.

Figure 11.3 Sample applet that uses our client.

Listing 11.2 Client class.

```
import java.io.*;
import java.net.*;
import java.applet.*;

public class DBClient {
public  Socket socket;
public PrintStream out;
public String Name;
public Reader reader;

public DBClient (String ServerName, int ServerPort) {
  try {  socket = new Socket(ServerName, ServerPort);
       // We put the reading of the inputStream from the application
       // server in its own thread, Reader.
         reader = new Reader(this);
           out = new PrintStream(socket.getOutputStream());
         }
         catch (IOException e) {System.err.println(e);}
    }
public String ProcessCommand(String InLine) {
System.out.println("FROM DBCLIENT:"+InLine);
  out.println(InLine);

  out.flush();
// tell the reader we've sent some data/command
  synchronized(reader){reader.notify();reader.notifyOn=false;}
  while(true) {
      // We have to wait until the Reader has finished reading, so we
set
      // this notifyOn flag in the reader when it has finished reading.
    if (reader.notifyOn) {break;}
  }

// Return the results of the command/query
  return(reader.getResult());
}
}

class  Reader extends Thread {
// This class reads data in from the application server
protected DBClient client;
public String Result="original";
public  boolean notifyOn=true;

public Reader(DBClient c) {
  super("DBclient Reader");
  this.client = c;
  this.start();
}

public synchronized void run() {
  String line="";

  DataInputStream in=null;
  try {
    in = new DataInputStream(client.socket.getInputStream());
    while(true) {
       // We start reading when we are notified from the main thread
       // and we stop when we have finished reading the stream for
       // this command/query.
    try {if (notifyOn) {this.wait(); notifyOn=false; Result="";}}
      catch (InterruptedException e){
       System.out.println("Caught an Interrupted Exception");
      }
      // Prevent simultaneous access
      line = in.readLine();
      if (line.equalsIgnoreCase("DONE")) {
        notifyOn=true;
      } else
        {
        if (line == null) {
          System.out.println("Server closed connection.");
          break;
        } // if NOT null
        else {Result+=line+"\n";}
        System.out.println("Read from server: "+Result);
        } // if NOT done..
    } //while loop
  }
  catch (IOException e) {System.out.println("Reader: " + e);}
  finally {
    try {if (in != null) in.close();}
    catch (IOException e) {
      System.exit(0);
      }
    }
}
public String getResult() {
  return (Result);
  }
}
```

The client class needs to be instantiated in a Java program, and the connection needs to be started before any queries can be made. If you remember our Interactive Query Applet from Chapter 4, this sample applet will certainly look familiar to you.

Listing 11.3 Applet to call our client class.

```
import java.net.URL;
import java.awt.*;
import java.applet.Applet;
import DBClient;

public class IQ extends java.applet.Applet {
  Button ConnectBtn = new Button("Connect to Database");
protected  DBClient DataConnection;

  TextField QueryField = new TextField(40);
  TextArea OutputField = new TextArea(10,75);

public void init() {
  QueryField.setEditable(true);
  OutputField.setEditable(false);
  DataConnection  = new DBClient(getDocumentBase().getHost(), 6001);

 GridBagLayout gridbag = new GridBagLayout();
 GridBagConstraints Con = new GridBagConstraints();
 setLayout(gridbag);
 setFont(new Font("Helvetica", Font.PLAIN, 12));
 setBackground(Color.gray);
 Con.weightx=1.0;
 Con.weighty=0.0;
 Con.anchor = GridBagConstraints.CENTER;
 Con.fill = GridBagConstraints.NONE;
 Con.gridwidth = GridBagConstraints.REMAINDER;
  gridbag.setConstraints(ConnectBtn, Con);
  add(ConnectBtn);

  add(new Label("SQL Query"));
  gridbag.setConstraints(QueryField, Con);
  add(QueryField);

  Label result_label = new Label("Result");
  result_label.setFont(new Font("Helvetica", Font.PLAIN, 16));
  result_label.setForeground(Color.blue);
  gridbag.setConstraints(result_label, Con);
  Con.weighty=1.0;
  add(result_label);

  gridbag.setConstraints(OutputField, Con);
  OutputField.setForeground(Color.white);
  OutputField.setBackground(Color.black);
  add(OutputField);

  show();
    } //init

public boolean handleEvent(Event evt) {

if ((evt.target == QueryField) & (evt.id == Event.KEY_PRESS))
    {char c=(char)evt.key;
     if (c  == '\n')
        {
        // When a user enters q query and hits "return," we send the
        // query to be processed and get the results to show in the
        // OutputField.
          DataConnection.ProcessCommand("S");

OutputField.setText(DataConnection.ProcessCommand(QueryField.getText()))
;
        return true;
      }
  }

if ((evt.target == ConnectBtn) & (evt.id == Event.ACTION_EVENT))
  {
        // This is the first command the application server expects,
        // connect to the data source.
        OutputField.setText(DataConnection.ProcessCommand("L"));
    return true;
  }
return false;
} // handleEvent()
}
```

You’ll need a Web page to call this applet from:

```
<HTML>
<HEAD>
<TITLE>
JDBC Client Applet - Interactive SQL Command Util via application server
</TITLE>
</HEAD>
<BODY>
<H1>Interactive JDBC SQL command interpreter via application server</H1>
<hr>
<applet code=IQ.class width=450 height=350>
</applet>
<hr>
</BODY>
</HTML>
```

Summary

In this chapter, we took a brief look at middleware. You saw the advantages and disadvantages of implementing a three-tier system, and we created a simple application server and a client server which you can easily extend to fit your needs.

We're almost at the end of this journey through the JDBC. The next chapter is a reference chapter of the JDBC API. It contains documentation on the JDBC methods used in the writing of this book, as well as methods that we didn't explicitly cover. You may want to browse through the package tree to get an idea of how the various classes and methods fit together, as well as their relation to one another.

Chapter 12
The JDBC API

This chapter ends our journey through the JDBC. I've provided a summary of the class interfaces and exceptions that are available in the JDBC API version 1.01, which was the most current version at the time of this writing. Although this chapter's primary purpose is to serve as a reference, you should still read through the sections completely so that you are aware of all the constructors, variables, and methods available.

Classes

We'll begin with the class listings. Each class listing includes a description and the class' constructors, methods, and variables.

public class Date

This class extends the java.util.Date object. But unlike the java util.Date, which stores time, this class stores the day, year, and month. This is for strict matching with the SQL date type.

Constructors

Constructor	Additional Description
Date(int Year, int Month, int day)	Construct a java.sql.Date object with the appropriate parameters

Methods

Method Name	Additional Description
public String toString()	Formats a Date object as YYYY-MM-DD
public static Date valueOf (String str)	Converts a String str to an sql.Date object

public class DriverManager

This class is used to load a JDBC driver and establish it as an available driver. It is usually not instantiated, but is called by the JDBC driver.

Constructors

DriverManager()

Methods

Method Name	Additional Description
public static void deregisterDriver(Driver-JDBCdriver) throws SQLException	Drops a driver from the available drivers list
public static synchronized Connection getConnection(String URL) throws SQLException	
public static synchronized Connection getConnection(String URL, String LoginName, String LoginPassword) throws SQLException	
public static synchronized Connection getConnection(String URL, Properties LoginInfo) throws SQLException	Establishes a connection to the given database URL, with the given parameters
public static Driver getDriver(String URL) throws SQLException	Finds a driver that understands the JDBC URL from the registered driver list
public static Enumeration getDrivers()	Gets an Enumeration of the available JDBC drivers
public static int getLoginTimeout()	Indicates the maximum time (seconds) that a driver will wait when logging into a database
public static PrintStream getLogStream()	Gets the logging PrintStream used by the DriverManager and JDBC drivers
public static void println(String msg)	Sends msg to the current JDBC logging stream (fetched from above method)
public static synchronized	Specifies that a new driver

void register Driver(Driver JDBCdriver) throws SQLException	class should call registerDriver when loading to "register" with the DriverManager
public static void setLoginTimeout(int sec)	Indicates the time (in seconds) that all drivers will wait when logging into a database
public static void setLogStream (PrintStream log)	Define the PrintStream that logging messages are sent to via the println method above

public class DriverPropertyInfo

This class is for developers who want to obtain and set properties for a loaded JDBC driver. It's not necessary to use this class, but it is useful for debugging JDBC drivers and advanced development.

Constructors

Constructor	**Additional Description**
public DriverPropertyInfo (String propName, String propValue)	The propName is the name of the property, and propValue is the current value; if it's not been set, it may be null

Variables

Variable Name	**Additional Description**
choices	If the property value is part of a set of values, then choices is an array of the possible values
description	The property's description
name	The property's name
required	This is true if this property is required to be set during Driver.connect
value	The current value of the property

public final class Numeric

This special fixed-point, high precision number class is used to store the SQL data types NUMERIC and DECIMAL.

Constructors

Constructor	Additional Description
public Numeric(String strNum)	Produces a Numeric object from a string; strNum can be in one of two formats: “1234.32” or “3.1E8”
public Numeric(String strNum, int scale)	Produces a Numeric, and scale is the number of digits right of the decimal
public Numeric(int intNum)	Produces a Numeric object from an int Java type parameter
public Numeric(int intNum, int scale)	Produces a Numeric object from an int, and scale gives the desired number of places right of the decimal
public Numeric(long x)	Produces a Numeric object from a long Java type parameter
public Numeric(long x, int scale)	Produces a Numeric object from a long parameter, and scale gives the desired number of places right of the decimal
public Numeric(double x, int scale)	Produces a Numeric object from a double Java type parameter, and scale gives the desired number of places right of the decimal
public Numeric(Numeric num)	Produces a Numeric object from a Numeric
public Numeric(Numeric num, int scale)	Produces a Numeric object from a Numeric, and scale gives the desired number of places right of the decimal

Methods

Method Name	**Additional Description**
public Numeric add(Numeric n)	Performs arithmetic addition on the reference Numeric object and the Numeric argument
public static Numeric createFromByteArray(byte byteArray[])	Produces a Numeric object from the byte array parameter
public static Numeric createFromIntegerArray(int intArray[])	Produces a Numeric object from the int array parameter
public static Numeric createFromRadixString(String str, int radix)	Produces a Numeric object from the String and int radix parameters
public static Numeric createFromScaled(long longNum, int power)	Produces a Numeric object by taking the longNum to the 10^power
public Numeric divide(Numeric q)	Divides the Numeric by the Numeric parameter q and returns the result
public double doubleValue()	Returns the Numeric as a Java type double
public boolean equals(Object objct)	Returns true if the Numeric object equals the objct parameter
public float floatValue()	Returns the Numeric as a Java type float
public static int getRoundingValue()	Returns the roundingValue used in rounding operations in the Numeric object
public int getScale()	Returns the number of places to the right of the decimal
public long getScaled()	Returns the Numeric object as a long, but removes the decimal (1234.567 -> 1234567); precision may be lost
public boolean greaterThan(Numeric num)	Returns true if the Numeric object is greater than the

	Numeric num argument
public boolean greaterThanOrEquals(Numeric num)	Returns true if the Numeric object is greater than or equal to the Numeric num argument
public int hashCode()	Returns an integer hashcode for the Numeric object
public Numeric[] integerDivide(Numeric x)	Returns an array with two Numeric objects: the first one is the quotient, the second is the remainder
public int intValue()	Returns the Numeric as a Java type int, digits after the decimal are dropped
public boolean isProbablePrime()	Returns true if the number is prime; it divides the Numeric object by several small primes, and then uses the Rabin probabilistic primality test to test if the number is prime—the failure rate is less than (1/(4^N))
public boolean lessThan(Numeric num)	Returns true if the Numeric object is less than the Numeric num argument
public boolean lessThanOrEquals(Numeric num)	Returns true if the Numeric object is less than or equal to the Numeric num argument
public long longValue()	Returns the Numeric as a Java type long
public Numeric modExp (Numeric numExp, Numeric numMod)	The two parameters are used to do a numMod modulus to the numExp exponent calculation; returns the result as a Numeric
public Numeric modInverse(Numeric numMod)	The modular multiplicative inverse is returned using numMod as the modulus
public Numeric multiply(Numeric num)	Returns the product of the Numeric object and the

	Numeric num parameter
public static Numeric pi(int places)	Returns pi to the number of decimal places
public Numeric pow(int exp)	Returns a Numeric object using the current Numeric object taken to the power of the given exponent exp
public static Numeric random(int bits, Random randSeed)	Returns a Numeric object that is a random number using randSeed as a seed, having size in bits equal to the bits parameter
public Numeric remainder(Numeric num)	Returns the remainder resulting from dividing this Numeric object by the Numeric num parameter
public static void setRoundingValue(int val)	Sets the rounding value used in rounding operations for the Numeric object
public Numeric setScale(int scale)	Returns a Numeric object from the current object with the specified scale parameter
public Numeric shiftLeft(int numberOfBits)	Returns the Numeric object with the specified numberOfBits shifted left
public Numeric shiftRight(int numberOfBits)	Returns the Numeric object with the specified numberOfBits shifted right
public int significantBits()	Returns the number of significant bits in the Numeric object
public Numeric sqrt()	Returns the square root of this Numeric object
public Numeric subtract(Numeric num)	Returns the difference between the Numeric object and the Numeric num parameter
public String toString()	Returns a String type that is the String representation of the Numeric object

public String toString(int radix)	Returns a String type that is the String representation of the Numeric object, in the specified radix

Variables

Variable Name	Additional Description
public final static Numeric ZERO	A Numeric equivalent to the value of 0
public final static Numeric ONE	A Numeric equivalent to the value of 1

public class Time

The public class Time is another SQL-JDBC data coversion class. This class extends java.util.Date, and basically implements the time-storing functions that are not present in the java.sql.Date class shown earlier.

Constructors

Constructor	Additional Description
public Time(int hour, int minute, int second)	Makes a Time object with the specified hour, minute, and second

Methods

Method Name	Additional Description
public String toString()	Returns a String with the Time formatted this way: HH:MM:SS
public static Time valueOf(String numStr)	Returns a Numeric object from the String numStr parameter that is in the format: HH:MM:SS

public class TimeStamp

This class is used to map the SQL data type TIMESTAMP. It extends java.util.Date, and has nanosecond precision for time-stamping purposes.

Constructors

Constructor	Additional Description
public Timestamp(int year, int	Builds a Timestamp object

month, int date, int hour, int minute, int second, int nano)	using the int parameters: year, month, date, hour, minute, second, and nano

Methods

Method Name	Additional Description
public boolean equals(Timestamp tstamp)	Compares the Timestamp object with the Timestamp parameter tstamp; returns true if they match
public int getNanos()	Returns the Timestamp object's nanoseconds
public void setNanos(int n)	Sets the Timestamp object's nanosecond value
public String toString()	Returns a formatted String object with the value of the Timestamp object in the format: YYYY-MM-DD HH:MM:SS.F
public static Timestamp valueOf(String strts)	Returns a Timestamp object converted from the strts parameter that is in the previous format

public class Types

This class contains the SQL data types as constants. It is used by other classes as the standard constant for the data types.

Constructors

Constructor	Additional Description
public Types()	Builds a Types object; not usually necessary as they can be accessed as so: Types.BIGINT

Variables

BIGINT
BINARY
BIT
CHAR
DATE

DECIMAL
DOUBLE
FLOAT
INTEGER
LONGVARBINARY
LONGVARCHAR
NULL
NUMERIC
OTHER (for a database specific data type, not a standard SQL-92 data type)
REAL
SMALLINT
TIME
TIMESTAMP
TINYINT
VARBINARY
VARCHAR

Interfaces

Next are the interface listings. As with the class listings, each interface listing includes a description and the interface's methods and variables.

public interface CallableStatement

This is the primary interface to access stored procedures on a database. If OUT parameters are specified and a query is executed via this class, its results are fetched from this class and not the **ResultSet** class. This class extends the **PreparedStatement** class, thus inheriting many of its methods.

The first 15 methods (the get methods) are identical in functionality to those in the **ResultSet** class, but they are necessary if OUT parameters are used. See the **ResultSet** class for a description of the methods.

Methods

Method Name	Additional Description
public abstract boolean getBoolean(int parameterIndex) throws SQLException	
public abstract byte getByte(int parameterIndex) throws SQLException	
public abstract byte[] getBytes(int parameterIndex)	

throws SQLException	
public abstract Date getDate(int parameterIndex) throws SQLException	
public abstract double getDouble(int parameterIndex) throws SQLException	
public abstract float getFloat(int parameterIndex) throws SQLException	
public abstract int getInt(int parameterIndex) throws SQLException	
public abstract long getLong(int parameterIndex) throws SQLException	
public abstract Numeric getNumeric(int parameterIndex, int scale) throws SQLException	
public abstract Object getObject(int parameterIndex) throws SQLException	
public abstract short getShort(int parameterIndex) throws SQLException	
public abstract String getString(int parameterIndex) throws SQLException	
public abstract Time getTime(int parameterIndex) throws SQLException	
public abstract Timestamp getTimestamp(int parameterIndex) throws SQLException	
public abstract void registerOutParameter(int	Each parameter of the stored procedure must be registered

paramIndex, int sqlDataType) throws SQLException	before the query is run; paramIndex is the stored proc’s parameter location in the output sequence, and sqlDataType is the data type of the parameter at the specified location (sqlDataType should be set from the Type class using one of its variables, for example, Types.BIGINT)
public abstract void registerOutParameter(int parameterIndex, int sqlDataType, int scale) throws SQLException	Specifies the number of places to the right of the decimal desired when getting Numeric data objects
public abstract boolean wasNull() throws SQLException	Returns true if the stored proc parameter was value NULL

public interface Connection

This is the high-level class used to interact with a database. The object is established from the **DriverManager.getConnection** method, which returns this object (Connection). This class obtains information about the specific database connection via the instantiated JDBC driver, and its primary use is to perform queries via the **createStatement**, **prepareCall**, and **prepareStatement** methods, which return **Statement**, **PreparedCall**, and **PreparedStatement** objects, respectively.

Methods

Method Name	**Additional Description**
public abstract void clearWarnings() throws SQLException	Clears the warnings for the connection
public abstract void close() throws SQLException	Closes the connection to the database
public abstract void commit() throws SQLException	Functions as the JDBC equivalent of the standard database commit command; it applies all commands and changes made since the last

	commit or rollback, including releasing database locks; results from queries are closed when commit is invoked
public abstract Statement createStatement() throws SQLException	Returns a Statement object, which can then be used to perform actual queries
public abstract boolean getAutoClose() throws SQLException	Returns true if automatic closing of the connection is enabled; automatic closing results in the closing of the connection when commit or rollback is performed
public abstract boolean getAutoCommit() throws SQLException	Returns true if automatic committing of the connection is on; automatic commit is on by default and means that the connection is committed on individual transactions; the actual commit occurs when the last row of a result set is fetched, or when the ResultSet is closed
public abstract String getCatalog() throws SQLException	Returns the current catalog name for the connection
public abstract DatabaseMetaData getMetaData() throws SQLException	Returns a DatabaseMetaData object for the current connection
public abstract int getTransactionIsolation() throws SQLException	Returns the transaction isolation mode of the connection
public abstract SQLWarning getWarnings() throws SQLException	Returns the SQLWarning object with the warnings for the connection
public abstract boolean isClosed() throws SQLException	Returns true if the connection has been closed
public abstract boolean	Returns true if the connection

isReadOnly() throws SQLException	is a read only connection
public abstract String nativeSQL(String throws SQLException	Returns the native SQL that the JDBC driver sqlQuery) would send to the database for the specified sqlQuery parameter
public abstract CallableStatement prepareCall(String sqlQuery) throws SQLException	Returns a CallableStatement object used to perform stored procedures; note that the SQL query must be passed in as the sqlQuery parameter here
public abstract PreparedStatement prepareStatement(String sqlQuery) throws SQLException	Returns a PreparedStatement object used to perform the specified sqlQuery; this query can be executed repeatedly if desired by using the PreparedStatement.execute method
public abstract void rollback() throws SQLException	Drops changes made since the last commit or rollback, and closes respective results; database locks are also released
public abstract void setAutoClose (boolean throws SQLException	Sets the connection to auto close mode if the auto) auto parameter is true
public abstract void throws SQLException	Sets the connection to auto commit mode if setAutoCommit(boolean auto) the auto parameter is true
public abstract void setCatalog (String catalog) throws SQLException	The catalog may be changed by specifying the catalog
public abstract void setReadOnly(boolean readOnly) throws SQLException	Sets the connection to read only mode
public abstract void setTransactionIsolation(int level) throws SQLException	Sets translation isolation to the specified level

Variables

The following constants are used in the **setTransactionIsolation** method as the level parameter:

```
TRANSACTION_NONE
TRANSACTION_READ_COMMITTED
TRANSACTION_READ_UNCOMMITTED
TRANSACTION_REPEATABLE_READ
TRANSACTION_SERIALIZABLE
```

public interface DatabaseMetaData

This class contains useful information about the open connection to the database. The **Connection.getMetaData** method returns a **Database-MetaData** object that is specific to the opened connection.

Methods

Method Name	**Additional Description**
public abstract boolean allProceduresAreCallable() throwsk SQLException	Returns true if all the procedures available to the user are callable
public abstract boolean allTablesAreSelectable() throws SQLException	Returns true if all of the tables are accessible to the user on the open connection
public abstract boolean dataDefinitionCausesTransactionCommit() throws SQLException	Returns true if data defintion causes the transaction to commit
public abstract boolean dataDefinitionIgnoredInTransactions() throws SQLException	Returns true if data defintion is ignored in the transaction
public abstract boolean doesMaxRowSizeIncludeBlobs() throws SQLException	Returns true if the getMaxSize method does not account for the size of LONGVARCHAR and LONGVARBINARY SQL data types
public abstract ResultSet getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean	Returns a ResultSet object for the specified parameters that gets the specified table's key or

nullok) throws SQLException	the attributes that can be used to uniquely identify a row, which may be composite; the scope parameter is one of the constants: bestRowTemporary, bestRowTransaction, or betRowSession; the nullok parameter allows columns that may be null; the ResultSet is composed of the following columns: scope (of the same types as above scope parameter), column name, SQL data type, name of the data type dependent on the database, precision, buffer length, significant places if a Numeric type, and pseudo column (one of the constants bestRowUnknown, bestRowNotPseudo, or bestRowPseudo)
public abstract ResultSet getCatalogs() throws SQLException	Returns a ResultSet object that contains a column for the catalog names that are in the database
public abstract String getCatalogSeparator() throws SQLException	Returns the separator between the catalog String and the table name
public abstract String getCatalogTerm() throws SQLException	Returns the database-specific term for "catalog"
public abstract ResultSet getColumnPrivileges(String catalog, String schemaString table, String	Returns a ResultSet object that contains information about the

columnNamePattern) throws SQLException	specified table's matching columnNamePattern; the returned ResultSet object contains the following columns: the catalog name that the table is in, the schema the table is in, the table name, the column name, owner of the table, grantee, type of access (SELECT, UPDATE, etc.), and if the grantee can grant access to others, "YES," "NO," or null (if unknown)
public abstract ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern) throws SQLException	Returns a ResultSet object that contains information about the matching columns for the matching tables and schemas; the ResultSet contains the following columns: catalog name, schema name, table name, column name, SQL data type, name of the type specific to the database, the maximum number of characters or precision depending on the data type, buffer length (not used), the number of digits (if applicable), radix (if applicable), null-ability (one of the constants columnNoNulls, columnNullable, columnNullableUnknown), comments for the column, default value (if it exists, else null),

	empty column, empty column, maximum number of bytes in the column of type CHAR (if applicable), index number of column; the last column is set to "YES" if it can contain NULLS if not "NO" else it's empty if the status is unknown
public abstract ResultSet get CrossReference(String primaryCatalog, String primarySchema, String primaryTable, String foreignCatalog, String foreignSchema, String foreignTable) throws SQLException	Returns a ResultSet object that describes the way a table imports foreign keys; the ResultSet object returned by this method contains these columns: primary key's table catalog, primary key's table schema, primary key's table, primary key's column name, foreign key's table catalog, foreign key's table schema, foreign key's table, foreign key's column name, sequence number within foreign key, action to foreign key when primary key is updated (one of the constants importedKeyCascade, importedKeyRestrict, importedKeySetNull), action to foreign key when primary key is deleted (one of the constants importedKeyCascade, importedKeyRestrict, importedKeySetNull),

	foreign key identifier, and primary key indentifier
public abstract String getDatabaseProductName() throws SQLException	Returns the database product name
public abstract String getDatabaseProductVersion() throws SQLException	Returns the database product number
public abstract int getDefaultTransactionIsolation() throws SQLException	Returns the default transaction isolation level as defined by the applicable constants in the Connection class
public abstract int getDriverMajorVersion()	Gets the driver's major version
public abstract int getDriverMinorVersion()	Gets the driver's minor version
public abstract String getDriverName() throws SQLException	Returns the name of the JDBC driver
public abstract String getDriverVersion() throws SQLException	Returns the version of the JDBC driver
public abstract ResultSet getExportedKeys(String catalog, String schema, String table) throws SQLException	Returns a ResultSet object that describes the foreign key attributes that reference the specified table's primary key; the ResultSet object returns the following columns: primary key's table catalog, primary key's table schema, primary key's table, primary key's column name, foreign key's table catalog, foreign key's table schema, foreign key's table, foreign key's column name, sequence number within foreign key, action to foreign key

	when primary key is updated (one of the constants importedKeyCascade, importedKeyRestrict, importedKeySetNull), action to foreign key when primary key is deleted (one of the constants importedKeyCascade, importedKeyRestrict, importedKeySetNull), foreign key identifier, and primary key indentifier
public abstract String getExtraNameCharacters() throws SQLException	Returns characters that can be used in unquoted identifier names besides the standard A through Z, 0 through 9, and _
public abstract String getIdentifierQuoteString() throws SQLException	Returns the String used to quote SQL identifiers
public abstract ResultSet getImportedKeys(String String schema, String table) throws SQLException	Returns a ResultSet object that describes the primary key attributes that are referenced by the specified table’s foreign key attributes; the ResultSet object contains the following columns: primary key’s table catalog, primary key’s table schema, primary key’s table, primary key’s column name, foreign key’s table catalog, foreign key’s table schema, foreign key’s table, foreign key’s column name, sequence number within foreign

	key, action to foreign key when primary key is updated (one of the constants importedKeyCascade, importedKeyRestrict, importedKeySetNull), action to foreign key when primary key is deleted (one of the constants importedKeyCascade, importedKeyRestrict, importedKeySetNull), foreign key identifier, and primary key indentifier
public abstract ResultSet getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate) throws SQLException	Returns a ResultSet object that describes the specified table's indices and statistics; the ResultSet object contains the following columns: catalog name, schema name, table name, "false" boolean (if tableIndexStatic is the type), index catalog (or null if type is tableIndexStatic), index type, sequence number, column name, column sort sequence, number of unique values in the table or number of rows (if tableIndexStatic), number of pages used for the index (or the number of pages used for the table if tableIndexStatic), and filter condition (if it exists)

public abstract int getMaxBinaryLiteralLength() throws SQLException	Returns the number of hex characters allowed in an inline binary literal
public abstract int getMaxCatalogNameLength() throws SQLException	The maximum length for a catalog name
public abstract int getMaxCharLiteralLength() throws SQLException	Returns the maximum length for a character literal
public abstract int getMaxColumnNameLength() throws SQLException	Indicates the maximum length for a column name
public abstract int getMaxColumnsInGroupBy() throws SQLException	Indicates the maximum number of columns in a GROUP BY clause
public abstract int getMaxColumnsInIndex() throws SQLException	Indicates the maximum number of columns in an index
public abstract int getMaxColumnsInOrderBy() throws SQLException	Indicates the maximum number of columns allowed in a ORDER BY clause
public abstract int getMaxColumnsInSelect() throws SQLException	Indicates the maximum number of columns in a SELECT statement
public abstract int getMaxColumnsInTable() throws SQLException	Indicates the maximum number of columns allowed in a table
public abstract int getMaxConnections() throws SQLException	Indicates the maximum number of simultaneous connections allowed to the database
public abstract int getMaxCursorNameLength() throws SQLException	Returns the maximum allowed length of a cursor name
public abstract int getMaxIndexLength() throws SQLException	Returns the maximum length of an index in bytes
public abstract int getMaxProcedureNameLength() throws	Returns the maximum allowed length of a

SQLException	procedure name
public abstract int getMaxRowSize() throws SQLException	Indicates the maximum row size
public abstract int getMaxSchemaNameLength() throws SQLException	Returns the maximum allowed length of a schema name
public abstract int getMaxStatementLength() throws SQLException	Returns the maximum allowed length of a SQL statement
public abstract int getMaxStatements() throws SQLException	Returns the maximum number of statements allowed at one time
public abstract int getMaxTableNameLength() throws SQLException	Returns the maximum allowed length of a table name
public abstract int getMaxTablesInSelect() throws SQLException	Indicates the maximum number of tables allowed in a SELECT statement
public abstract int getMaxUserNameLength() throws SQLException	Returns the maximum allowed length of a user name
public abstract String getNumericFunctions() throws SQLException	Returns a comma-separated list of the math functions available
public abstract ResultSet getPrimaryKeys(String catalog, String schema, String table) throws SQLException	Returns a ResultSet object that contains the primary key’s description for the specified table; the ResultSet object contains the following columns: catalog name, schema name, table name, column name, sequence number, primary key name, and, possibly, NULL
public abstract ResultSet getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String	Returns a ResultSet object that describes the catalog’s stored procedures and result

columnNamePattern) throws SQLException	columns matching the specified procedureNamePatten and columnNamePattern; the ResultSet object contains the following columns: catalog name, schema name, procedure name, column or parameter name, column type, data type, data name, precision, length in bytes, scale, radix, nullability, and comments
public abstract ResultSet getProcedures(String catalogString String procedureNamePattern) throws SQLException	Returns a ResultSet object that describes the catalog’s procedures; the ResultSet object contains the following columns: catalog name, schema name, procedure name, empty column, empty column, empty column, comments about the procedure, and kind of procedure
public abstract String getProcedureTerm() throws SQLException	Return the database-specific term for procedure
public abstract ResultSet getSchemas() throws SQLException	Returns a ResultSet object that describes the schemas in a database; the ResultSet object contains one column that contains the schema names
public abstract String getSchemaTerm() throws SQLException	Returns the database-specific term for schema
public abstract String getSearchStringEscape() throws	Returns the escape characters for pattern

SQLException	searching
public abstract String getSQLKeywords() throws SQLException	Returns a comma-separated list of keywords that the database recognizes, but the keywords are not SQL-92 keywords
public abstract String getStringFunctions() throws SQLException	Returns a comma-separated list of string functions in the database
public abstract String getSystemFunctions() throws SQLException	Returns a comma-separated list of system functions in the database
public abstract ResultSet getTablePrivileges(String catalog, String schemaPattern schemaPattern, String tableNamePattern) throws SQLException	Returns a ResultSet object that describes the privileges for the matching and tableNamePattern; the ResultSet object contains the following columns: catalog name, schema name, table name, grantor, grantee, type of access, and “YES” if a grantee can grant other access
public abstract ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[]) throws SQLException	Returns a ResultSet object that describes tables matching the schemaPattern and tableNamePattern; the ResultSet object contains the following columns: catalog name, schema name, table name, table type, and comments
public abstract ResultSet getTableTypes() throws SQLException	Returns a ResultSet object that describes the table types available in the database; the ResultSet object contains the column that is a list

	of the table types
public abstract String getTimeDateFunctions() throws SQLException	Returns the date and time functions for the database
public abstract ResultSet getTypeInfo() throws SQLException	Returns a ResultSet object that describes the SQL data types supported by the database; the ResultSet object contains the columns: type name, SQL data type constants in the Types class, maximum precision, prefix used to quote a literal, suffix used to quote a literal, parameters used to create the type, nullability, case sensitivity, searchability, signed or unsigned (boolean), is it a currency, auto incrementable or not, local version of data type, minimum scale, maximum scale, empty column, empty column, and radix
public abstract String getURL() throws SQLException	The URL for the database
public abstract String getUserName() throws SQLException	Returns the user name as known by the database
public abstract ResultSet getVersionColumns(String catalog, String String table) throws SQLException	Returns a ResultSet object that describes the specified table’s columns that are updated when any column is updated in the table; the ResultSet object contains the following columns: empty

	columns, column name, SQL datatype, type name, precision, column value length in bytes, scale, and pseudoColumn or not
public abstract boolean isCatalogAtStart() throws SQLException	Returns true if the catalog name appears at the start of a qualified table name
public abstract boolean isReadOnly() throws SQLException	Returns true if the database is in read only mode
public abstract boolean nullPlusNonNullIsNull() throws SQLException	Returns true if a concatenation between a NULL and non-NULL is NULL
public abstract boolean nullsAreSortedAtEnd() throws SQLException	
public abstract boolean nullsAreSortedAtStart() throws SQLException	
public abstract boolean nullsAreSortedHigh() throws SQLException	
public abstract boolean nullsAreSortedLow() throws SQLException	
public abstract boolean storesLowerCaseIdentifiers() throws SQLException	
public abstract boolean storesLowerCaseQuotedIdentifiers() throws SQLException	
public abstract boolean storesMixedCaseIdentifiers() throws SQLException	
public abstract boolean storesMixedCaseQuotedIdentifiers()	

throws SQLException	
public abstract boolean storesUpperCaseIdentifiers() throws SQLException	
public abstract boolean storesUpperCaseQuotedIdentifiers() throws SQLException	
public abstract boolean supportsAlterTableWithAddColumn() throws SQLException	
public abstract boolean supportsAlterTableWithDropColumn() throws SQLException	
public abstract boolean supportsAlterTableWithDropColumn() throws SQLException	
public abstract boolean supportsANSI92EntryLevelSQL() throws SQLException	
public abstract boolean supportsANSI92FullSQL() throws SQLException	
public abstract boolean supportsANSI92IntermediateSQL() throws SQLException	
public abstract boolean supportsANSI92FullSQL() throws SQLException	
public abstract boolean supportsCatalogsInDataManipulation() throws SQLException	
public abstract boolean supportsCatalogsInIndexDefinitions() throws SQLException	
public abstract boolean supportsCatalogsInPrivilegeDefinitions() throws SQLException	
public abstract boolean supportsCatalogsInProcedureCalls() throws SQLException	

public abstract boolean supportsCatalogsInTableDefinitions() throws SQLException	
public abstract boolean supportsColumnAliasing() throws SQLException	
public abstract boolean supportsConvert() throws SQLException	
public abstract boolean supportsConvert(int fromType, int toType) throws SQLException	
public abstract boolean supportsCoreSQLGrammar() throws SQLException	
public abstract boolean supportsCorrelatedSubqueries() throws SQLException	
public abstract boolean supportsDataDefinitionAnd DataManipulationTransactions() throws SQLException	
public abstract boolean supportsDataManipulation TransactionsOnly() throws SQLException	
public abstract boolean supportsDifferentTableCorrelationNames() throws SQLException	
public abstract boolean supportsExpressionsInOrderBy() throws SQLException	
public abstract boolean supportsExtendedSQLGrammar() throws SQLException	
public abstract boolean supportsFullOuterJoins() throws SQLException	
public abstract boolean supportsGroupBy() throws SQLException	
public abstract boolean supportsGroupByBeyondSelect() throws	

SQLException	
public abstract boolean supportsGroupByUnrelated() throws SQLException	
public abstract boolean supportsIntegrityEnhancementFacility() throws SQLException	
public abstract boolean supportsLikeEscapeClause() throws SQLException	
public abstract boolean supportsLimitedOuterJoins() throws SQLException	
public abstract boolean supportsMinimumSQLGrammar() throws SQLException	
public abstract boolean supportsMixedCaseIdentifiers() throws SQLException	
public abstract boolean supportsMixedCaseQuotedIdentifiers() throws SQLException	
public abstract boolean supportsMultipleResultSets() throws SQLException	
public abstract boolean supportsMultipleTransactions() throws SQLException	
public abstract boolean supportsNonNullableColumns() throws SQLException	
public abstract boolean supportsOpenCursorsAcrossCommit() throws SQLException	
public abstract boolean supportsOpenCursorsAcrossRollback() throws SQLException	
public abstract boolean supportsOpenStatementsAcrossCommit() throws SQLException	

public abstract boolean supportsOpenStatementsAcrossRollback() throws SQLException	
public abstract boolean supportsOrderByUnrelated() throws SQLException	
public abstract boolean supportsOuterJoins() throws SQLException	
public abstract boolean supportsPositionedDelete() throws SQLException	
public abstract boolean supportsPositionedUpdate() throws SQLException	
public abstract boolean supportsSchemasInDataManipulation() throws SQLException	
public abstract boolean supportsSchemasInProcedureCalls() throws SQLException	
public abstract boolean supportsSchemasInProcedureCalls() throws SQLException	
public abstract boolean supportsSchemasInTableDefinitions() throws SQLException	
public abstract boolean supportsSelectForUpdate() throws SQLException	
public abstract boolean supportsStoredProcedures() throws SQLException	
public abstract boolean supportsSubqueriesInComparisons() throws SQLException	
public abstract boolean supportsSubqueriesInExists() throws SQLException	
public abstract boolean	

supportsSubqueriesInIns() throws SQLException	
public abstract boolean supportsSubqueriesInQuantifieds() throws SQLException	
public abstract boolean supportsTableCorrelationNames() throws SQLException	
public abstract boolean supportsTransactionIsolationLevel(int level) throws SQLException	
public abstract boolean supportsTransactions() throws SQLException	
public abstract boolean supportsUnion() throws SQLException	
public abstract boolean supportsUnionAll() throws SQLException	
public abstract boolean usesLocalFilePerTable() throws SQLException	
public abstract boolean usesLocalFiles() throws SQLException	

Variables

```
public final static int bestRowNotPseudo
public final static int bestRowPseudo
public final static int versionColumnUnknown
public final static int versionColumnNotPseudo
public final static int versionColumnPseudo
public final static int importedKeyCascade
public final static int importedKeyRestrict
public final static int importedKeySetNull
primary key has been updated or deleted
public final static int typeNoNulls
public final static int typeNullable
public final static int typeNullableUnknown
public final static int typePredNone
public final static int typePredChar
public final static int typePredBasic
public final static int typeSearchable
public final static short tableIndexStatistic
public final static short tableIndexClustered
public final static short tableIndexHashed
public final static short tableIndexOther
```

public interface Driver

The JDBC driver implements this interface. The JDBC driver must create an instance of itself and then register with the DriverManager.

Methods

Method Name	Additional Description
public abstract boolean acceptsURL(String URL) throws SQLException	Returns true if the driver can connect to the specified database in the URL
public abstract Connection connect(String url, Properties props) throws SQLException	Connects to the database specified in the URL with the specified Properties props
public abstract int getMajorVersion()	Returns the JDBC driver's major version number
public abstract int getMinorVersion()	Returns the JDBC driver's minor version number
public abstract DriverPropertyInfo[] getPropertyInfo(String URL, Properties props) throws SQLException	Returns an array of DriverPropertyInfo that contains possible properties based on the supplied URL and props
public abstract boolean jdbcCompliant()	Returns true if the JDBC driver can pass the JDBC compliance suite

public interface PreparedStatement

This object extends Statement, and it is used to perform queries that will be repeated. This class exists primarily to optimize queries that will be executed repeatedly.

Methods

Note: The set methods set the parameter at the paramIndex location in the prepared query to the specified paramType object.

Method Name	Additional Description

public abstract void clearParameters() throws SQLException	Resets all of the PreparedStatment’s query parameters
public abstract boolean execute() throws SQLException	Runs the prepared query against the database; this method is used primarily if multiple ResultSets are expected
public abstract ResultSet executeQuery() throws SQLException	Executes the prepared query
public abstract int executeUpdate() throws SQLException	Executes the prepared query; this method is used for queries that do not produce a ResultSet (such as Update); returns the number or rows affected or 0 if nothing is returned by the SQL command
public abstract void setAsciiStream(int paramIndex, InputStream paramType, int length) throws SQLException	
public abstract void setBinaryStream(int paramIndex, InputStream paramType, int length) throws SQLException	
public abstract void setBoolean(int paramIndex, boolean paramType) throws SQLException	
public abstract void setByte(int paramIndex, byte paramType) throws SQLException	
public abstract void setBytes(int paramIndex, byte paramType[]) throws SQLException	
public abstract void	

setDate(int paramIndex, Date paramType) throws SQLException	
public abstract void setDouble(int double paramType) throws SQLException	
public abstract void setFloat(int paramIndex, float paramType) throws SQLException	
public abstract void setInt(int paramIndex, int paramType) throws SQLException	
public abstract void setLong(int paramIndex, long paramType) throws SQLException	
public abstract void setNull(int paramIndex, int sqlType) throws SQLException	
public abstract void setNumeric(int paramIndex, Numeric paramType) throws SQLException	
public abstract void setObject(int paramIndex, Object paramType) throws SQLException	
public abstract void setObject(int paramIndex, Object paramType, int targetSqlType) throws SQLException	
public abstract void setObject(int paramIndex, Object paramType, int targetSqlType, int scale) throws SQLException	
public abstract void	

setShort(int paramIndex, short paramType) throws SQLException	
public abstract void setString(int paramIndex, String paramType) throws SQLException	
public abstract void setTime(int paramIndex, Time paramType) throws SQLException	
public abstract void setTimestamp(int TimestampparamType) throws SQLException	
public abstract void setUnicodeStream(int paramIndexInputStream paramType, int length) throws SQLException	

public interface ResultSet

The results of a query are stored in this object, which is returned when the respective query execute method is run for the **Statement**, **PreparedStatement**, and **CallableStatement** methods. The get methods in this class fetch the result for the specified column, but the proper data type must be matched for the column. The **getMetaData** method in this class can facilitate the process of checking the data type in each column of the result set.

Methods

Method Name	**Additional Description**
public abstract void clearWarnings() throws SQLException	Clears the warnings for the ResultSet
public abstract void close() throws SQLException	Closes the ResultSet
public abstract int findColumn(String columnName) throws	Gets the column number for the specified columnName in the ResultSet

SQLException	
public abstract ResultSetMetaData getMetaData() throws SQLException	Returns a ResultSetMetaData object that contains information about the query’s resulting table
public abstract InputStream getAsciiStream(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract InputStream getAsciiStream(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract InputStream getBinaryStream(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract InputStream getBinaryStream(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract boolean getBoolean(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract boolean getBoolean(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract byte getByte(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract byte	Fetches the result from the

getByte(String columnName) throws SQLException	current row in the specified column (the column name - columnName) in the resulting table
public abstract byte[] getBytes(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract byte[] getBytes(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract String getCursorName() throws SQLException	This returns a String with this ResultSet's cursor name
public abstract Date getDate(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract Date getDate(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract double getDouble(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract double getDouble(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract float getFloat(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting

	table
public abstract float getFloat(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract int getInt(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract int getInt(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract long getLong(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract long getLong(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract Numeric getNumeric(int columnIndex, int scale) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract Numeric getNumeric(String columnName, int scale) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract Object getObject(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table

public abstract Object getObject(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract short getShort(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract short getShort(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract String getString(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract String getString(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract Time getTime(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract Time getTime(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract Timestamp getTimestamp (int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract Timestamp	Fetches the result from the

getTimestamp(String columnName) throws SQLException	current row in the specified column (the column name - columnName) in the resulting table
public abstract InputStream getUnicodeStream(int columnIndex) throws SQLException	Fetches the result from the current row in the specified column (the column number - columnIndex) in the resulting table
public abstract InputStream getUnicodeStream(String columnName) throws SQLException	Fetches the result from the current row in the specified column (the column name - columnName) in the resulting table
public abstract SQLWarning getWarnings() throws SQLException	Returns the warnings for the ResultSet
public abstract boolean next() throws SQLException	Retrieves the next row of the resulting table
public abstract boolean wasNull() throws SQLException	Returns true if the last column read by one of the get methods was NULL

public interface ResultSetMetaData

This methods allows access to information about a query's results, but not the results themselves. This object is created by the **ResultSet.getMetaData** method.

Methods

Method Name	**Additional Description**
public abstract String getCatalogName(int column) throws SQLException	Returns the name of the catalog hit by the query
public abstract int getColumnCount() throws SQLException	Returns the number of columns in the resulting table
public abstract int getColumnDisplaySize(int column) throws SQLException	Returns the specified column's maximum size
public abstract String	Gets a label, if it exists, for

getColumnLabel(int column) throws SQLException	the specified column in the result set
public abstract String getColumnName(int column) throws SQLException	Gets a name for the specific column number in the resulting table
public abstract int getColumnType(int column) throws SQLException	Returns a constant in the Type class that is the JDBC type of the specified column in the result set
public abstract String getColumnTypeName(int column) throws SQLException	Gets the name of the type of the specified column in the result set
public abstract int getPrecision(int column) throws SQLException	Returns the precision of the data in the specified column, if applicable
public abstract int getScale(int column) throws SQLException	Returns the scale of the data in the specified column, if applicable
public abstract String getSchemaName(int column) throws SQLException	Returns the name of the schema that was accessed in the query to produce the result set for the specific column
public abstract String getTableName(int column) throws SQLException	Returns the name of the table from which the specified column in the result set came from
public abstract boolean isAutoIncrement (int column) throws SQLException	Returns true if the specified column is automatically numbered
public abstract boolean isCaseSensitive (int column) throws SQLException	Returns true if the specified column’s contents are case sensitive, if applicable
public abstract boolean isCurrency(int column) throws SQLException	Returns true if the content of the specific column in the result set was a currency
public abstract boolean isDefinitelyWritable(int column) throws SQLException	Returns true if a write operation in the specified column can be done for certain

public abstract int isNullable(int column) throws SQLException	Returns true if the specified column accepts NULL entries
public abstract boolean isReadOnly(int column) throws SQLException	Returns true if the specified column is read only
public abstract boolean isSearchable(int column) throws SQLException	Returns true if the WHERE clause can be a part of the SQL query performed on the specified column
public abstract boolean isSigned(int column) throws SQLException	Returns true if the data contained in the specified column in the result set is signed, if applicable
public abstract boolean isWritable(int column) throws SQLException	Returns true if a write on the specified column is possible

Variables

Variable Name	**Additional Description**
public final static int columnNoNulls	NULL values not allowed
public final static int columnNullable	NULL values allowed
public final static int columnNullableUnknown	NULL values may or may not be allowed, uncertain

public interface Statement

This class is used to execute a SQL query against the database via the **Connection** object. The **Connection.createStatement** returns a **Statement** object. Methods in the **Statement** class produce **ResultSet** objects which are used to fetch the result of a query executed in this class.

Methods

Method Name	**Additional Description**
public abstract void cancel() throws SQLException	If a query is running in another thread, a foreign thread can cancel it by calling this method on the local Statement object’s

	instantiation
public abstract void clearWarnings() throws SQLException	Clears the warnings for the Statement
public abstract void close() throws SQLException	Closes the Statement and frees its associated resources, including any ResultSets
public abstract boolean execute(String sql) throws SQLException	Executes the parameter sql, which is an SQL query; this method accounts for multiple ResultSets
public abstract ResultSet executeQuery(String sql) throws SQLException	Executes a query that returns a ResultSet object (produces some results) using the sql parameter as the SQL query
public abstract int executeUpdate(String sql) throws SQLException	Executes a query that does not produce a resulting table; the method returns the number of rows affected or 0 if no result is produced
public abstract int getMaxFieldSize() throws SQLException	Returns the maximum amount of data returned for a resulting column; applies only to the following SQL datatypes: BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR
public abstract int getMaxRows() throws SQLException	Returns the maximum number of rows a ResultSet can contain
public abstract boolean getMoreResults() throws SQLException	Returns true if the next ResultSet of the query is present, and moves the ResultSet into the current result space
public abstract int getQueryTimeout() throws SQLException	Returns the number of seconds that the JDBC driver will wait for a query to execute

public abstract ResultSet getResultSet() throws SQLException	Returns a ResultSet object that is the current result of the query; only one of these is returned if only one ResultSet is the result of the query; if more ResultSets are present, the getMoreResults method is used to move to the next ResultSet
public abstract int getUpdateCount() throws SQLException	Returns the update count; if the result is a ResultSet, -1 is returned
public abstract SQLWarning getWarnings() throws SQLException	Returns the warnings encountered for the query of this Statement object
public abstract void setCursorName(String name) throws SQLException	Sets the name of a cursor for future reference, and uses it in update statements
public abstract void setEscapeProcessing(boolean enable) throws SQLException	Sets escape substitution processing
public abstract void setMaxFieldSize(int max) throws SQLException	Sets the maximum amount of data that can be returned for a column of type BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR
public abstract void setMaxRows(int max) throws SQLException	Sets the maximum number of rows that can be retrieved in a ResultSet
public abstract void setQueryTimeout(int seconds) throws SQLException	Sets the time a driver will wait for a query to execute

Exceptions

Finally, we get to the exceptions. As with the other sections, the exception listings include a description and the class' constructors and methods.

public class DataTruncation

This class extends **SQLWarning**. An exception is produced when data transfer is prematurely terminated on a write operation, and a warning is generated when data transfer is prematurely terminated on a read operation. You can use the methods contained here to provide debugging information because the JDBC driver should throw this exception when a data transfer problem is encountered.

Constructors

Constructor	Additional Description
public DataTruncation(int index, boolean parameter, boolean read, int dataSize, int transferSize)	Builds a Throwable DataTruncation object with the specified properties

Methods

Method Name	Additional Description
public int getDataSize()	Returns the number of bytes that should have been transferred
public int getIndex()	Returns the index of the column or parameter that was interrupted
public boolean getParameter()	Returns true if the truncated value was a parameter, or false if it was a column
public boolean getRead()	Returns true if truncation occurred on a read; false means truncation occurred on a write
public int getTransferSize()	Returns the number of bytes actually transferred

public class SQLException

This class extends java.lang.Exception. It is the responsibility of the JDBC driver to throw this class when a problem occurs during an operation.

Constructors

These constructors are used to create an **SQLException** with the specified information. It is normally not necessary to create an exception unless the developer is working on creating a driver or higher level JDBC interface:

```
public SQLException()
public SQLException(String problem)
public SQLException(String problem, String SQLState)
public SQLException(String problem, String SQLState,
int vendorCode)
```

Methods

Method Name	Additional Description
public int getErrorCode()	Returns the error code that was part of the thrown exception
public SQLException getNextException()	Returns the next exception as an SQLException object
public String getSQLState()	Returns the SQL state that was part of the thrown exception
public synchronized void setNextException (SQLException excp)	Sets the next exception as excp for the SQLException object

public class SQLWarning

This class extends **SQLException**. It is the responsibility of the JDBC driver to throw this class when a problem occurs during an operation.

Constructors

These constructors build an **SQLWarning** object with the specified information. It is normally not necessary to create an **SQLWarning** unless the developer is working on creating a driver or higher level JDBC interface:

```
public SQLWarning()
public SQLWarning(String problem)
public SQLWarning(String problem, String SQLstate)
public SQLWarning(String problem, String SQLstate, int
vendorCode)
```

Methods

Method Name	Additional Description
public SQLWarning getNextWarning()	Returns an SQLWarning object that contains the next warning
public void setNextWarning(SQLWarning warn)	Sets the next SQLWarning warning warn for the SQLWarning object