18

Connecting to MySQL from a Java/J2EE Application

Most users access data in a MySQL database by using an application that interfaces with that database. In many cases, the application is built with Web pages that reside on a application server or Web server such as Apache or Internet Information Services. Chapter 17 provides an example of a PHP application that access data in the DVDRentals database. However, Web-based applications are by no means limited to PHP. Another popular programming language that you can use to build Web-based applications is Java. Java supports the same functionality as PHP, and much more. In fact, Java can also be used to build client-server and multi-tiered systems that are not limited to Web-based applications — and all those systems can be built to access a MySQL database.

In this chapter, you learn how to build a Java Web-based application that accesses a MySQL database. The application is based on the Java 2 Enterprise Edition (J2EE) specification, which defines a collection of libraries that contain classes for server-side tasks, including support for the JavaServer Pages (JSP) technology. By using Java and JSP technology, you can create Web pages that deliver dynamic content to your users, similar to the way the PHP delivers content to users. In fact, if you're familiar with PHP, you'll find that creating JSP files is quite similar to creating PHP files. To demonstrate how to create a JSP application that connects to a MySQL database, this chapter covers the following topics:

- ❑ Introduces you to Java and how it communicates with a MySQL server and its databases
- ❑ Explains how to build a JSP application that connects to a MySQL database, retrieves data from that database, inserts data into the database, modifies the data, and then deletes the data

Introduction to Java/J2EE

Java is a robust application language based on the principles of object-oriented programming (OOP). As the name indicates, OOP refers to a programming model that is based on the concept of encapsulating code and the data that it manipulates into defined objects. Each object is based on an object class that specifies how that object can be built. An *object*, then, is an instance of the class from which it is derived.

A class is made up of fields and methods. The fields are variables that can contain data that describes the object, which is why the fields are sometimes referred to as data members. The variable values are what distinguish multiple object instances that are based on the same class. The methods defined on a class can perform some type of action on the data stored in the variables. A method is similar to a function in the way that it performs a predefined task. For example, suppose that you have a class named BookInventory. The class might include three fields, one to identify the book, one to identify the number in stock, and one to identify the number on order. The class also includes a method that calculates the total number of books that will be available by adding the values in the last two fields. As you can see, the three fields contain data about the book and the method performs an action on that data. Together, these elements make up the BookInventory class. You can then create an object based on the BookInventory class and assign unique values to the fields.

This example is, of course, an over-simplification of how objects work within Java, but it does help to distinguish between fields and methods. Although a thorough discussion of OOP is beyond the scope of this book, it's important that you recognize that Java is based on these concepts and that everything you do in Java is within the context of objects. For example, suppose that you retrieve data from a MySQL database and you now want to process that data so that you can display each row in your application. When you retrieve that data, it is stored in a ResultSet object. You can then use methods defined on that object to access the data in that result set.

In order for a Java application to access data in MySQL database, it must have some way to interface with that database. Two components are needed to provide that connectivity: JDBC and Connector/J. JDBC is a call-level API that allows a Java application to connect with a wide range of SQL databases. By using JDBC, a Java application can connect to the database, send SQL statements, and retrieve data. Because JDBC can communicate with different database products, Java applications are very portable, which means that you don't have to modify a lot of code if you were to switch the database product used to support the application. However, it does mean that you need a product-specific driver that completes the JDBC connection from the Java application to the database.

JDBC is sometimes said to mean Java Database Connectivity, but the makers of JDBC, Sun Microsystems, make no such claim. In fact, they list JDBC as a registered trademark, but not Java Database Connectivity.

To facilitate your ability to use JDBC to connect to a MySQL database, MySQL AB provides Connector/J, a JDBC-specific driver that implements most of the functionality supported by JDBC. The JDBC functionality that is not supported pertains to those functions that are not currently implemented in MySQL. However, for the most part, you will be able to perform nearly any SQL-related task from within a Java application that you would expect to perform from any application.

To implement a JSP-based application, you must run it under a Web server or application server that implements the J2EE Servlet container specification. For some application servers, such as JBoss, you must also create special build or configuration files that allow the JSP application to be implemented in that environment. However the JSP files themselves, which are the focus of this chapter, can be created with any text editor, as long as the files are saved with the .jsp extension. Now take a look at how you actually create those files.

This chapter was written based on the following configuration: the Java 2 SDK 1.4.2, MySQL 4.1.6, Connector/J 3.0.15, JBoss 4.0.0RC1 (an application server), and Apache Ant 1.6.2 (a Java-based build tool) installed on a Windows XP computer. Details about the configuration of your operating system, *applications server, the Java SDK, and Apache Ant are beyond the scope of the book. Be sure to view the necessary product documentation when setting up your application environment. For more information about the Java 2 SDK, go to* `www.sun.com/j2se`*. For information about Connector/J and MySQL, go to* `www.mysql.com`*. For information about JBoss, go to* `www.jboss.org`*. For information about Apache Ant, go to* `http://ant.apache.org`*.*

Building a Java/J2EE Web Application

When you create a JSP file, the first step that you must take is to tell the Java compiler where to find the language classes that will be referenced within the application. From there, you must establish your connection to the MySQL server and its database, and then you can select or retrieve data. Java supports numerous objects and statements that allow you to perform each of these tasks. In the remaining part of the chapter, you learn about each aspect of database access, from importing classes to closing connections. You also create a JSP application that accesses data in the DVDRentals database. If you created the PHP application shown in Chapter 17, you'll find that the application in this chapter achieves the same results.

The examples in this chapter focus on the JSP code used to create a basic Web-based application. However, the principles behind connecting to a database and accessing data can apply to any Web or non-Web application written in other programming languages.

Importing Java Classes

When creating a JSP file, you must specify the classes that will be used in your application. In Java, classes are organized into packages. In other words, a package is a collection of classes.) You can specify that all classes within a package be made available to the application, or you can specify individual classes. To specify the classes that your application will access, you should use a page directive that specifies the package or class name.

When specifying a directive in the JSP file, you must enclose the directive in an opening directive tag (`<%@`) and a closing directive tag (`%>`). For a page directive, include the `page` keyword. To specify a class or package, you must also use the `import` keyword. For example, the following statement imports the `java.lang` class package:

```
<%@ page import "java.lang.*" %>
```

As you can see, the statement is enclosed in the directive tags and includes the `page` and the `import` keywords, with the name of the package enclosed in double quotes. Notice that the package name ends with the asterisk (*) wildcard, which indicates that all classes within the package should be included in the directive. If you want to specify a specific class within that package, such as `Integer`, you would use the following statement:

```
<%@ page import "java.lang.Integer" %>
```

As you can see, the wildcard has been replaced with the class name, but everything else is the same as in the previous example. If you plan to use a lot of the classes within a package, you'll probably want to specify the entire package; otherwise, specify only those classes that you'll need so you minimize the impact on the Java compiler.

Declaring and Initializing Variables

Java is a strongly typed application language, which means that each variable must be explicitly declared and a data type must be assigned to that variable. In Java, a variable is either a primitive variable or an object variable. A *primitive* variable is a basic variable that is simply a specified placeholder that holds a value in memory, similar to the variables you see in MySQL. Primitive variables include such types as `boolean`, `int`, `char`, and `long`. They are unique within Java because, unlike other variables, they do not require that an object be created when you initialize them.

To declare a primary variable, you must specify the type and the variable name. For example, the following statement declares the `cdId` variable as an `int` type:

```
int cdId;
```

Once a variable is declared, you need to assign a value to it in order for that variable to be used within the code. The process of assigning a value is referred to as initializing the variable and you can do that simply by specifying a single equal sign (=) plus the value that should be assigned, as shown in the following example:

```
cdId = 42;
```

In this case, the value 42 is assigned to `cdId`. You can also declare and initialize a variable within one statement:

```
int cdId = 42;
```

This, of course, makes your code simpler; however, you might not always be able to initialize a variable at the time that you declare it, depending on the scope that you want the variable to have. Scope determines the lifetime of the variable, and scope is determined when the variable is declared. If you declare a variable within a block of code, the variable is available only to that block and to any blocks nested within the outer block. The outer block defines the scope of that variable. (A block of code is a set of Java statements that are enclosed in curly brackets.) As a result, if you want a variable to be available to all code, but you cannot assign a value to the variable until a specific operation within a block of code is completed, you should declare the variable outside any blocks of code and initialize the variable when the value is available. You see examples of this later in the chapter.

The second type of variable is the object variable. An *object* variable is always based on a class and is usually associated with an object that is based on the same class. In some cases, the object references a null value and is not associated with an object until initialized to an object. The variable references that object, which contains the value that is associated with the variable. For example, suppose that you want to create a variable that holds a string value. To do so, you must use the `String` class to declare the variable and initialize it, as shown in the following example:

```
String strName;
strName = new String("Strauss");
```

In the first statement, you have simply declared a variable named `strName` that is based on the `String` class. At this point, no object has been created. However, when you initialize the variable, a `String` object is created and configured with the value `Strauss`. The `strName` variable then references the new object. As a result, whenever you want to use `Strauss` in your code, you can use the `strName` variable, which in turn references the new `String` object, which contains the value `Strauss`.

Notice that the second statement includes the new keyword. Whenever you explicitly initialize a variable in this way, you must use the new keyword. This tells the Java compiler to create an object based on the specified class. The new keyword is followed by a constructor, which in this case is String(). A *constructor* is a special type of method defined within the class. A constructor always has the same name as the class and is used to create objects based on that class.

As with primary variables, you can also declare and initialize an object variable in one statement, as shown in the following statement:

```
String strName = new String("Strauss");
```

As you can see, all the same elements are there, only now condensed into one statement. You declare the variable and then initialize it by creating a String object that is assigned to the variable.

The String class allows you assign a string directly to a variable, and the object is created by the compiler and you are referencing it. In this way, a String variable appears similar to a primitive variable, which allows you to initialize a variable simple by assigning a value to that variable, as the following statement demonstrates:

```
String strName = "Strauss";
```

For some classes, you can create variables that use yet another approach to create the object that is assigned to the variable. In these cases, you use a method in one class to create an object in a different class. One example of this approach is when you establish a connection to a database, so next you review that process to see how it works.

Connecting to a MySQL Database

To establish a connection to a MySQL database, you must take two steps. The first is to dynamically load a Java class at runtime, specifically, the Driver class in the com.mysql.jdbc class package. This step is necessary to locate and instantiate the Connector/J driver. To load the Java class, you must use the forName() method in the Class object, and then you must use the newInstance() method to create a new class, as shown in the following statement:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

When the Driver class is located, the newInstance() method creates a new instance of the Driver class and names that instance DriverManager. The DriverManager class manages all currently installed JDBC drivers.

To use the MySQL Connector/J driver, it must be properly installed on your system. For information about installing the driver, see the Connector/J documentation that comes with the installation files.

Once the DriverManager class has been created, you can use the getConnection() method in that class to create a Connection object. The getConnection() method takes three arguments, the connection URL, the user account name, and the password for that account. For example, the following statement uses the getConnection() method to connect to the MySQL server on the local computer:

```
conn = DriverManager.getConnection("jdbc:mysql://localhost/CDs", "cduser", "pw1");
```

The statement includes a number of components, so first take a look at the getConnection() method. The first argument includes the jdbc prefix, then mysql, and then a URL that specifies the location of the MySQL server and the name of the database. The location is localhost and the database is CDs. The second and third arguments specify the username and then the password.

Notice that the DriverManager.getConnection() structure is assigned to the conn variable. Normally this variable is declared prior to the running the statement above and is based on the Connection object, as shown in the following declaration:

```
   Connection conn;
```

Once you declare this variable, you can initialize it with the DriverManager.getConnection() method, as demonstrated previously. In establishing a connection, the getConnection() method returns a Connection object, which is in turn assigned to the conn variable. The variable can then be used to reference the Connection object, which contains the connection to the database. As this demonstrates, you do not always have to initialize an object variable directly by using the new command and a class constructor, but you can instead use a method in another object, as has been done previously.

Retrieving Data from a MySQL Database

Once you have established your connection to the MySQL server database, you can retrieve data that can then be displayed by your Java application. The primary mechanism that you use to retrieve data is the SELECT statement, written as it is written when you access data interactively. However, the SELECT statement alone is not enough. You need additional mechanisms that pass that SELECT statement to the MySQL server and that then process the results returned by the statement.

When you're retrieving data from a MySQL database from within your JSP application, you generally follow a specific set of steps:

1. Declare and initialize a String variable that creates the SELECT statement.
2. Declare and initialize a Statement variable that creates a Statement object.
3. Declare and initialize a ResultSet variable that uses the Statement object to execute the query and to create a ResultSet object that contains the query results.
4. Use the ResultSet variable to process the results, normally within some sort of conditional structure.

Now take a closer look at each step. The first step creates a SELECT statement that is assigned to a String variable. For example, in the following statement, a variable named selectSql is defined:

```
   String selectSql = "SELECT CDID, CDName, InStock, OnOrder, " +
                      "Reserved, Department, Category from CDs";
```

Notice that the value that is assigned to the variable is a SELECT statement. If you want to split the statement into multiple lines, as has been done here, you enclose each portion of the statement line in double quotes. In addition, for each line other than the last line, you end the line with a plus sign. This indicates that the two lines are to be concatenated when executed. Only the final line is terminated with a semi-colon.

Putting the SELECT statement in a variable in this way makes it easier to work with the statement in other places in the code. Once the variable has been declared and initialized, you should then create a

Statement object and assign that to a variable. The Statement object contains the methods that you need to execute your SELECT statement. The following variable is declared and then initialized by creating a Statement object:

```
Statement stmt = conn.createStatement();
```

The createStatement() method is part of the Connection class, as represented by the conn variable. You can access an object's method by calling it through the associated variable, as has been done here. The createStatement() method creates a Statement object, which is then associated with the stmt variable. Notice that the stmt variable has been declared and initialized within one statement.

Once you have created the Statement object and the variable that references that object, you can use the executeQuery() method in the Statement object to execute your SELECT statement, as shown in the following example:

```
ResultSet rs = stmt.executeQuery(selectSql);
```

In addition to executing the SELECT statement, the executeQuery() method also creates a ResultSet object that in turn is assigned to a ResultSet variable, which is named rs. You can now use the rs variable to call the methods in the ResultSet object to process your result set.

Processing the Result Set

Once you have a result set to work with, you must process the rows so that they can be displayed in a usable format. However, Java, like other procedural application languages, cannot easily process sets of data, so you must set up your code to be able to process one row at a time. You can do this by setting up a loop condition and by using a function that supports the row-by-row processing. For example, you can use the while command to set up the loop and the next() method of the ResultSet object to process the rows, as shown in the following example:

```
while(rs.next())
{
   int cdId = rs.getInt("CDID");
   String cdName = rs.getString("CDName");
   int inStock = rs.getInt("InStock");
   int onOrder = rs.getInt("OnOrder");
   int reserved = rs.getInt("Reserved");
   String department = rs.getString("Department");
   String category = rs.getString("Category");
   System.out.println("cdId=" + cdId + ", cdName=" + cdName + ", inStock=" +
                      inStock, " + "onOrder=" + onOrder + ", reserved=" +
                      reserved + ", " + "department=" + department +
                      ", category=" + category);
}
```

The while command tells Java to continue to execute the block of statements as long as the current condition equals true. The condition in this case is defined by rs.next(). The rs variable (which is associated with a ResultSet object) allows you to call the next() method. The method references each row in the results set, one row at a time, starting with the first row. Every time the while loop is executed, next() points to the next row.

Each value is retrieved from the ResultSet object by using the name of column, as it was returned by the SELECT statement. In this query, you retrieve MySQL integer and string values, so you use the getInt() and getString() methods of the ResultSet object to retrieve the applicable values. (The ResultSet object includes methods for other types of data as well.) The getInt() method returns an integer value, which is then assigned to the variable that is being declared for that particular column. The getString() method creates a String object to store the retrieved value and associates that object with the applicable variable.

To help illustrate this, suppose that the first row returned by your query contains the following values:

- CDID = 101
- CDName = Bloodshot
- InStock = 10
- OnOrder = 5
- Reserved = 3
- Department = Popular
- Category = Rock

When the while loop is executed the first time, the first call to the next() method points to the first row in the result set, which is stored in the ResultSet object that was created specifically to hold this result set. Because the rs variable is associated with that ResultSet object, you can use that variable to access the data stored in the object as well as the methods defined on that object (as inherited from the ResultSet class). You can then use the necessary method for each column within each row to return that particular value. For example, you can use the getInt() method to return the CDID value in the first row. As a result, the cdId variable is set to a value of 101. For each column returned by the next() method for a particular row, the column value is assigned to the applicable variable. The column-related variables can then be used to display the actual values returned by the SELECT statement.

The next step then is to use some sort of mechanism to print those values that have been returned. One such mechanism is to use the System.out.println() method to print the variable values. In addition, you can also print literal values by enclosing those values in double quotes, but you do not enclose the variable names in quotes, otherwise they will be treated as literal values, rather than printing the values that they represent.

Each time the while command is executed, the values for a specific row (taken from the variables) are displayed on your screen. By the time the while command has looped through the entire result set, all rows returned by the query are displayed. (Later, in the Try It Out sections, you'll see the results of using the while command to create a loop construction.)

Manipulating String Data

After you have retrieved string data from a database, you'll often find that you want to manipulate that data in some way. Because all string data is associated with String objects, you can use the methods defined in the String class to take some sort of action on that data. Two particularly useful methods are length() and equals(). To demonstrate how both these methods work, first take a look at the following statement:

```
String compName = "Strauss";
```

The statement declares a String variable named compName, whose purpose is to represent a composer's name. Initially, a String object that contains the value Strauss is assigned to the variable. Suppose now, that you want to determine the number of characters in the string value that the variable currently references. To do so, you can invoke the length() method to determine the number of characters, as shown in the following statement:

```
return compName.length();
```

The return command returns the value determined by length(). Because the compName variable is currently associated with the value Strauss, length() returns a value of 7.

The equals() method compares two strings and returns a value of true if the strings are equal and a value of false if they are not. To use this method, you must specify the first string, add the equals() method, and specify the second string as an argument in that method. For example, the following statement compares the Chopin string to the string in the compName variable:

```
return "Chopin".equals(compName);
```

The reason that the equals() method can be used with the string Chopin is because Java automatically treats a literal string value as an object, even when used as it is used in this example. As a result, you can call methods from the String class simply by specifying the string value, followed by a period, and then followed by the method. You can then pass the second string as an argument to the method. In this case, because the variable is currently associated with the value Strauss, the statement returns a value of false.

When working with string data, there might be times when you want to concatenate the string value in a variable with another string value. In that case, you would use the plus/equal signs (+=) to indicate that the value to the right of these signs should be concatenated to the value on the left. For example, the following statement adds Ricard to the value in the compName variable:

```
compName += ", Ricard";
```

This statement results in one value: Strauss, Ricard. If you were to use the equal sign without the plus sign, a new value would be assigned to the variable, resulting in the value Ricard (preceded by a comma) replacing the value Strauss.

Converting Values

In many cases when working with Java, you will find that you want to convert one type of value to another type. For example, when creating a Web application, numerical values are often passed as strings. However, you might find that you want to work with those values as integers, so you need to convert them. The way to do this is to use static methods in the Integer class to retrieve and convert the value to an integer. A *static* method is one that can be called directly from a class, without having to first create an object. The static methods that you use to convert numerical string values to integers are the valueOf() method and the intValue() method. The valueOf() method takes the numerical string as a parameter and returns it as an Integer object. The inValue() method then retrieves the value from the object and returns it as a primitive int type value. For example, the following statement returns the int type value of 100:

```
return Integer.valueOf("100").intValue();
```

As you can see, the string value is specified as an argument of the valueOf() method. From the Integer object created by this method, the intValue() method extracts the value and returns it as an int type.

In addition to converting numerical string values to integer values, you might find that you want to convert a string value to a date value and display that value in a specific format. This process often involves a couple of steps. The first step is to create a variable that is associated with a date-related object that allows you to convert and format a date value. For example, the following statement declares and initializes the dateFormat variable:

```
   SimpleDateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");
```

The dateFormat variable is based on the SimpleDateFormat class. To initialize the variable, you must create a SimpleDateFormat object by using the new keyword along with the class constructor. As an argument in the constructor, you should include the format of the date that you want to convert.

Once you've created the necessary variable, you can use the parse() method of the SimpleDateFormat object associated with the variable to convert the string value to a date value, as shown in the following statement:

```
   Date newDate = dateFormat.parse("01-31-2000");
```

The parse() method converts the string value to a Date object, which is then assigned to the newDate variable. Now suppose that you want to convert the date value back to a string. You can use the format() method in the SimpleDateFormat object (associated with the dateFormat variable) to format the date as a string value, according to the format specified in the SimpleDateFormat object. For example, the following statement returns the value associated with the date variable as a string value:

```
   return dateFormat.format(newDate);
```

When you want to insert a Java date value into a MySQL database, data conversion gets a little trickier. To convert date values, you must use two different objects: one is the java.util.Date class and other is java.sql.Date class. The java.sql.Date class is actually an extension of the java.util.Date class that allows you to enter date values into an SQL database from your Java application.

When you want to insert a date value in an SQL database, you should first create an object based on the java.util.Date class and assign it to a variable, as shown in the following statement:

```
   java.util.Date utilDate = new java.util.Date();
```

You can now use the utilDate variable to access methods in the java.util.Date object. Specifically, you want to use the getTime() method to return the number of milliseconds that have passed since January 1, 1970. The value is then used by the java.sql.Date class to create an object to store the data value to be inserted into the SQL database, as shown in the following statement:

```
   java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
```

As the statement indicates, a new java.sql.Date object is created, based on the value returned by the getTime() method. The new object is then assigned to the sqlDate variable. By using this approach, you can convert two different date objects back and forth. In this example, the date value returned by the getTime() method is converted to a java.sql.Date value, which can then be used to insert the date value into the MySQL database.

As you can see in the last two examples, the `Date` *class name is fully qualified, which means that the entire package and class name are provided. If you import both classes into your JSP page and then simply call the* `Date` *class, Java does not know which class to use. In addition, if you import only one package and you use a* `Date` *object that has not been fully qualified, Java always uses the* `Date` *object from the imported class, even if you want to use a class that has not been imported. To use a class that has not been explicitly imported, you must fully qualify the class name.*

Working with HTML Forms

When working with Web-related applications, you often need to pass data from the client browser to the server. This is often done through the use of a *form*, which is an element on an HTML page that allows a user submit data that can be passed on to the server. For example, a form can be used to allow a user to log on to an application. The user can submit an account name and password, which the form then sends to a specified location.

An HTML form supports two types of posting methods: `POST` and `GET`. The `POST` method sends the data through an HTTP header. An HTTP header is a mechanism that allows commands and data to be sent to and from the browser and server. The `GET` method adds the data to a URL. Because the data is added to the URL, it is visible when the URL is displayed, which can create some security issues The `POST` method is often preferred because the `POST` data is hidden.

A thorough discussion about HTML forms and HTTP headers is beyond the scope of this book. However, there are plenty of resources that describe forms, headers, and all other aspects of HTML and HTTP extensively. If you're not familiar with how to work with forms and headers, be sure to read the appropriate documentation.

Like many other server-side programming languages, Java's JSP has built-in mechanisms that automatically process the data submitted by a form. In Java/J2EE, all JSP files are actually converted to Java classes and compiled. The classes that they are converted into are derived from a class called `Servlet`. This underlying process is fairly involved and beyond the scope of this book; however, the end result is that all pages have some pre-built classes available to them, including the `HttpServletResponse` and `HttpServletRequest` classes. From these classes, two objects are created: `response` and `request`. The `request` object abstracts the form parameters. Java then merges the `GET` and `POST` methods so that the server-side application can simply access the parameters via the `getParameter()` method of the request object. The `getParameter()` method takes as an argument the name of the parameter that was passed through the form, and then returns a string value or a null if no value is found.

To give you a better sense of how this works, the following example demonstrates this process. Suppose that your application includes an HTML form that contains an input element named department, as shown in the following code:

```
<input type="text" name="department">
```

Because the input element is a text type, the user enters a value into a text box and that value is assigned to the parameter named department. The `response` object abstracts this parameter, and Java makes it available to the application. You can then use the `getParameter()` method of the `request` object to retrieve that value, as the following statement demonstrates:

```
String strDept = request.getParameter("department");
```

The value entered into the form is returned as a string value that is assigned to a `String` variable named `strDept`. From there, you can use the `strDept` variable in other Java statements as you would any other string value.

Redirecting Browsers

Web application pages can sometimes decide that the user should be redirected to another page. This means that the current page stops processing and the server loads a new page and processes it. This process is usually part of a condition that specifies that, if a certain result is received, an action must be taken based on that result. For example, suppose that you want to redirect a user if that user enters Classical in the department `<input>` element (which is then assigned to the `strDept` variable). You can use the `sendRedirect()` method in the `response` object to redirect the user to another page, similar to the following statement:

```
if("Classical".equals(strDept))
{
   response.sendRedirect("ClassicalSpecials.jsp");
}
```

First, an `if` statement is created to compare the `strDept` value to Classical. If the two string values are equal, the condition evaluates to true and the `if` statement is executed. In this case, the `sendRedirect()` method in the response object is called and the user is redirected to the page specified as an argument to the method. As a result, the `ClassicalSpecials.jsp` page is displayed.

Including JSP Files

There might be times when you want to execute Java code that is in a file separate from your current file. For example, suppose that your application includes code that is repeated often. You can put that code in a file separate from your primary file and then call that file from the primary file. The second file is referred to as a include file, and you can simply reference it from your primary file to execute the code in the include file.

To reference an include file from within your current JSP file, you must use the `include` command, followed by the name of the include file. (If the file is located someplace other than the local directory, you must also specify the path.) The following if statement uses the `include` command to execute the Java code in the ClassicalSpecials.jsp file:

```
if("Classical".equals(strDept))
{
   %>
  <%@ include "ClassicalSpecials.jsp " %>
  <%
}
```

The first thing to notice is that the `include` command is enclosed in a opening (`<%@`) and closing (`%>`) directive tags. Most Java code in a JSP page is enclosed in opening (`<%`) and closing (`%>`) scriptlet tags. As a result, you must close a scriptlet before executing a directive, and then re-open the scriptlet, as necessary.

The example above also includes an `if` statement. The `if` condition specifies that the `strDept` value must equal Classical. If the condition is true, the include file is accessed and the script in the file is executed as though it were part of the current file. It would be the same as if the code in the include file actually existed within the primary file.

Managing Exceptions

To handle errors that occur when a statement is executed, Java uses a system based on the `Exception` class. When an error occurs, an exception is generated, which is a special class that can contain an error message. If your code includes ways to handle that exception, some sort of action is taken. For example, if an exception is generated, you might have it logged to a file or displayed to the user.

To work with exceptions, you must enclose your Java code in a try/catch block. The try part of this block includes all the primary application code, and the catch part of the block includes the code necessary to handle the exceptions. At its very basic, a try/catch block looks like the following:

```
try
{
   <Java application code>
}
catch(Exception ex)
{
   throw ex;
}
```

As you can see, two blocks have actually been created: the try block and the catch block. The try block includes all your basic program code, and the catch block processes the exception. The `catch()` method takes two arguments. The first is the name of the exception that is being caught, and the second is a variable that references the exception. The variable can then be used within the catch block.

In reality, you can include multiple catch blocks after the try block. In the example above, the `Exception` argument represents all exceptions. However, you can specify individual exceptions, rather than all exceptions. For example, you can specify the exception `ClassNotFoundException` if you want to catch the exception that is thrown if an object class cannot be found. In which case, your catch block would begin with the following:

```
catch(ClassNotFoundException ce)
```

If within the catch block you specify a throw statement, as shown in the preceding example, the applicable caller within the application server handles the exception. In some cases, the exception message is displayed to the user, depending on where the exception is occurs. If you want other action to be taken, you would create the necessary statements in the catch block.

Now that you've been introduced to many of the basic Java elements that you can use when retrieving and displaying data from a MySQL database, you're ready to build an application.

Creating a Basic Java/J2EE Web Application

In the Try It Out sections in this chapter, you build a simple Web application that allows you to view the transactions in the DVDRentals database. You will also be able to add a transaction, edit that transaction, and then delete it from the database. As you'll recall when you designed the DVDRentals database, transactions are actually a subset of orders. Each order is made up of one or more transaction, and each transaction is associated with exactly one order. In addition, each transaction represents exactly one DVD rental.

For example, if someone were to rent three DVDs at the same time, that rental would represent one order that includes three transactions.

The application that you build in this chapter is very basic and does not represent a complete application, in the sense that you would probably want your application to allow you to create new orders, add DVDs to the database, as well as add and edit other information. However, the purpose of this application is merely to demonstrate how you connect to a MySQL database from within Java, how you retrieve data, and how you manipulate data. The principles that you learn here can then be applied to any Java application that must access data in a MySQL database.

When creating a Web application such as a JSP application, you will usually find that you are actually programming in three or four different languages. For example, you might use Java for the dynamic portions of your application, HTML for the static portions, SQL for data access and manipulation, and JavaScript to perform basic page-related functions. The application that you create in this chapter uses all four languages. At first this might seem somewhat confusing; however, the trick is to think about each piece separately. If you are new to these technologies, try doing each piece separately and then integrating them. The application is fully integrated and can be run and examined to see how these technologies work. Keep in mind, however, that the focus of the Try It Out sections is to demonstrate Java and SQL, so you will not find detailed explanations about the JavaScript and HTML. However, you cannot develop a JSP application without including some HTML, and JavaScript is commonly implemented in Web-based applications. As a result, in order to show you a realistic application, HTML and Java Script are included, but a discussion of these two technologies is well beyond the scope of this book. Fortunately, there are ample resources available for both of them, so be sure to consult the appropriate documentation if there is an HTML or JavaScript concept that you do not understand.

To support the application that you will create, you need two include files, one that contains HTML styles and one that contains the Java script necessary to support several page-related functions. You can download the files from the Wrox Web site at `www.wrox.com`, or you can copy them from here. The first of these files is dvdstyle.css, which controls the HTML styles that define the look and feel of the application's Web pages. The styles control the formatting of various HTML attributes that can be applied to text and other objects. The following code shows the contents of the dvdstyle.css file.

```
table.title{background-color:#eeeeee}

td.title{background-color:#bed8e1;color:#1a056b;font-family:sans-serif;font-weight:
bold;font-size: 12pt}

td.heading{background-color:#486abc;color:#ffffff;font-family:sans-serif;font-
weight: bold;font-size: 9pt}

td.item{background-color:#99ff99;color:#486abc;font-family:sans-serif;font-weight:
normal;font-size: 8pt}

input.delete{background-color:#990000;color:#99ffff;font-family:sans-serif;font-
weight: normal;font-size: 8pt}

input.edit{background-color:#000099;color:#99ffff;font-family:sans-serif;font-
weight: normal;font-size: 8pt}

input.add{background-color:#000099;color:#99ffff;font-family:sans-serif;font-
weight: normal;font-size: 8pt}

td.error{background-color:#ff9999;color:#990000;font-family:sans-serif;font-weight:
bold;font-size: 9pt}
```

When you create an HTML element in your code, you can reference a particular style in the dvdstyle.css file, and then that style is applied. For example, the dvdstyle.css file includes the following style definition:

```
td.item{background-color:#99ff99;color:#486abc;font-family:sans-serif;font-weight:
normal;font-size: 8pt}
```

The `td.item` keywords identify the style definition. The `td` refers to the type of style definition, which in this case is a cell within a table, and item is the unique name given to this particular definition. The options defined within the paragraph are the various styles that apply to this definition. You can then reference this style definition in your HTML code. For example, if you are creating a table and you want a cell within that table to use this style, you would reference the item style name.

Whether you copy the file from the Web site or create the file yourself, you should save the file to the same directory where your JSP pages are stored. You can then modify the styles to meet your own needs.

The second file that you need to support the application is the dvdrentals.js file, which contains the JavaScript support functions for the web form submission. These functions allow the program to manipulate the command values and also the values of the form's action parameter. By using this technique, a user button-click can redirect the form to a different page. The following code shows the contents of the dvdrentals.js file:

```
function doEdit(button, transactionId)
{
   button.form.transaction_id.value = transactionId;
   button.form.command.value = "edit";
   button.form.action = "edit.jsp";
   button.form.submit();
}

function doAdd(button)
{
   button.form.transaction_id.value = -1;
   button.form.command.value = "add";
   button.form.action = "edit.jsp";
   button.form.submit();
}

function doDelete(button, transactionId)
{
   var message = "Deleting this record will permanently remove it.\r\n" +
                 "Are you sure you want to proceed?";

   var proceed = confirm(message);

   if(proceed)
   {
      button.form.transaction_id.value = transactionId;
      button.form.command.value = "delete";
      button.form.submit();
   }
}

function doCancel(button)
```

```
{
   button.form.command.value = "view";
   button.form.action = "index.jsp";
   button.form.submit();
}

function doSave(button, command)
{
   button.form.command.value = command;
   button.form.submit();
}
```

The dvdrentals.js file includes numerous function definitions. For example, the following JavaScript statement defines the `doEdit()` function:

```
function doEdit(button, transactionId)
{
   button.form.transaction_id.value = transactionId;
   button.form.command.value = "edit";
   button.form.action = "edit.jsp";
   button.form.submit();
}
```

The `doEdit()` function can be called from within your HTML code, usually through an `<input>` element that uses a button-click to initiate the function. The `doEdit()` function takes two parameters: `button` and `transaction`. The `button` parameter is used to pass the HTML button object that references the form element into the JavaScript function, and the `transactionId` parameter holds the transaction ID for the current record. The `transactionId` value, along with a command value of `edit`, is submitted to the form in the `edit.jsp` file when that file is launched. Again, whether you copy the file from the Web site or create the file yourself, you should save the dvdrentals.js file to the same directory where your JSP files are stored in your Web server.

Once you've ensured that the dvdstyle.css and dvdrentals.js file have been created and added to the appropriate directory, you're ready to begin creating your application. The first file that you create — index.jsp — provides the basic structure for the application. The file contains the Java statements necessary to establish the connection to the DVDRentals database, retrieve data from the database, and then display that data. The page lists all the transactions that currently exist in the DVDRentals database. In addition, the index.jsp file provides the foundation on which additional application functionality is built in later Try It Out sections. You can download any of the files used for the DVDRentals application created in this chapter at the Wrox Web site at `www.wrox.com`. The Web site also includes the build files used to implement the application on the JBoss application server.

Try It Out Creating the index.jsp File

The following steps describe how to create the index.jsp file, which establishes a connection to the DVDRentals database and retrieves transaction-related data:

1. The first part of the index.jsp file specifies the classes that will be used by the JSP page. Open a text editor and enter the following code:

```
<%@ page import="java.sql.*,
java.text.SimpleDateFormat,
java.lang.Integer" %>
```

2. After you specify the classes, set up the basic HTML elements that provide the structure for the rest of the page. This includes the page header, links to the dvdstyle.css and dvdrentals.js files, and the initial table structure in which to display the data retrieved from the DVDRentals database. Enter the following code:

```
<html>
<head>
   <title>DVD - Listing</title>
   <link rel="stylesheet" href="dvdstyle.css" type="text/css">
   <script language="JavaScript" src="dvdrentals.js"></script>
   </script>
</head>

<body>
<p></p>

<table cellSpacing=0 cellPadding=0 width=619 border=0>
<tr>
   <td>
      <table height=20 cellSpacing=0 cellPadding=0 width=619 bgcolor=#bed8e1
border=0>
      <tr align=left>
         <td valign="bottom" width="400" class="title">
            DVD Transaction Listing
         </td>
      </tr>
      </table>
      <br>
      <table cellSpacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="250" class="heading">Order Number</td>
         <td width="250" class="heading">Customer</td>
         <td width="250" class="heading">DVDName</td>
         <td width="185" class="heading">DateOut</td>
         <td width="185" class="heading">DateDue</td>
         <td width="185" class="heading">DateIn</td>
      </tr>
```

3. Next, you must declare two variables used to connect to the database and to retrieve data. Add the following statements to your file:

```
<%
// Variables for database operations
   Connection connection;
   Statement stmt;
```

4. The next section of the file first initiates a try/catch block to catch any exception that might have been thrown. From there, you can create the connection to the MySQL server and the DVDRentals database. Add the following code after the code you added in Step 2:

```
// Wrap database-related code in try/catch block to handle errors
   try
   {

// Create a DriverManager object to connect to the database
      Class.forName("com.mysql.jdbc.Driver").newInstance();

      connection = DriverManager.getConnection("jdbc:mysql://localhost/DVDRentals",
                                          "mysqlapp",
                                          "pw1");
```

The user account specified in this section of code — mysqlapp — is an account that you created in Chapter 14. The account is set up to allow you to connect from the local computer. If you did not set up this account or will be connecting to a host other than local host, you must create the correct account now. If you want to connect to the MySQL server with a hostname or username other than the ones shown here, be sure to enter the correct details. (For information about creating user accounts, see Chapter 14.)

You might decide that, for reasons of security, not to store the user account name and password in the JSP file, as is done here. Java provides other methods for accessing a MySQL database. For example, you can add the user account information to the application server's configuration file, and then use a DataSource object to reference the file.

5. In the following section you create the query that will retrieve data from the DVDRentals database and assign the results of that query to a variable. Add the following code to your file:

```
// Construct the SQL statement
      String selectSql = "SELECT " +
                         "Transactions.TransID, " +
                         "Transactions.OrderID, " +
                         "Transactions.DVDID, " +
                         "Transactions.DateOut, " +
                         "Transactions.DateDue, " +
                         "Transactions.DateIn, " +
                         "Customers.CustFN, " +
                         "Customers.CustLN, " +
                         "DVDs.DVDName " +
                         "FROM Transactions, Orders, Customers, DVDs " +
                         "WHERE Orders.OrderID = Transactions.OrderID " +
                         "AND Customers.CustID = Orders.CustID " +
                         "AND DVDs.DVDID = Transactions.DVDID " +
                         "ORDER BY Transactions.OrderID DESC, " +
                         "Customers.CustLN ASC, Customers.CustFN ASC, " +
                         "Transactions.DateDue DESC, DVDs.DVDName ASC";

// Create Statement object and execute the SQL Statement
      stmt = connection.createStatement();

      ResultSet rs = stmt.executeQuery(selectSql);
```

6. The next step is to loop through the results returned by your query. Add the following code to your application file:

```
// Loop through the result set
      while(rs.next())
      {
// Retrieve the columns from the result set into variables
         int transId = rs.getInt("TransID");
         int orderId = rs.getInt("OrderID");
         int dvdId = rs.getInt("DVDID");
         Date dateOut = rs.getDate("DateOut");
         Date dateDue = rs.getDate("DateDue");
         Date dateIn = rs.getDate("DateIn");
         String custFirstName = rs.getString("CustFN");
         String custLastName = rs.getString("CustLN");
         String dvdName = rs.getString("DVDName");
```

7. Now put the customer names and dates into a more readable format. Add the following code to your page:

```
// Format the result set data into a readable form and assign to variables
         SimpleDateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");

         String customerName = "";
         String dateOutPrint = "";
         String dateDuePrint = "";
         String dateInPrint = "";

         if(custFirstName != null)
            customerName += custFirstName + " ";

         if(custLastName != null)
            customerName += custLastName;

         if(dvdName == null)
            dvdName = "";

         if(dateOut != null)
            dateOutPrint = dateFormat.format(dateOut);

         if(dateDue != null)
            dateDuePrint = dateFormat.format(dateDue);

         if(dateIn != null)
            dateInPrint = dateFormat.format(dateIn);
```

8. Next, insert the values retrieved from the database into an HTML table structure. Add the following code to the JSP file:

```
// Print each value in each row in the HTML table
%>
      <tr height="35" valign="top">
         <td class="item">
            <nobr>
            <%=orderId%>
            </nobr>
         </td>
```

```
            <td class="item">
               <nobr>
               <%=customerName%>
               </nobr>
            </td>
            <td class="item">
               <nobr>
               <%=dvdName%>
               </nobr>
            </td>
            <td class="item">
               <nobr>
               <%=dateOutPrint%>
               </nobr>
            </td>
            <td class="item">
               <nobr>
               <%=dateDuePrint%>
               </nobr>
            </td>
            <td class="item">
               <nobr>
               <%=dateInPrint%>
               </nobr>
            </td>
         </tr>
```

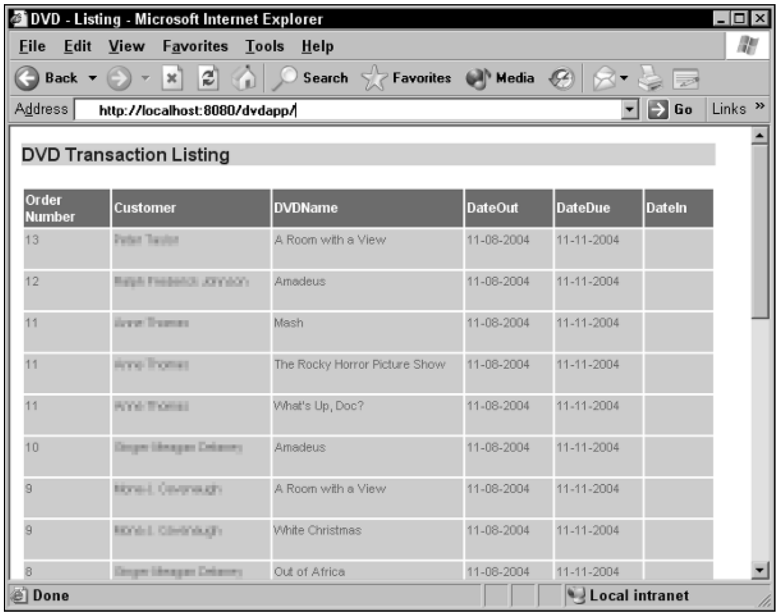
9. The final section of the file closes the connection and the Java code. It also closes the `<table>`, `<body>`, and `<html>` elements on the Web page. Add the following code to the end of the JSP file:

```
<%
      }
// Close the database objects and the HTML elements
      rs.close();
      stmt.close();
      connection.close();
   }
   catch(Exception exception)
   {
      throw exception;
   }
%>
      </table>
    </td>
</tr>
</table>
</body>
</html>
```

10. Save the index.jsp file to the appropriate Web application directory.

You might need to build your JSP application if you are using an application server. If you need to build the application in order to implement it, you should build it now and set up the application files according to the guidelines of the application server that you're using.

11. Open your browser and go to the address `http://localhost:8080/dvdapp`. The value `8080` represents the port number. If your application server or Web server use a different port number, use that one instead of `8080`. Your browser should display a page similar to the one shown in Figure 18-1.


DVD - Listing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/dvdapp/ Go Links
Done
Local intranet

DVD Transaction Listing

Order Number	Customer	DVDName	DateOut	DateDue	DateIn
13	[redacted]	A Room with a View	11-08-2004	11-11-2004	
12	[redacted]	Amadeus	11-08-2004	11-11-2004	
11	[redacted]	Mash	11-08-2004	11-11-2004	
11	[redacted]	The Rocky Horror Picture Show	11-08-2004	11-11-2004	
11	[redacted]	What's Up, Doc?	11-08-2004	11-11-2004	
10	[redacted]	Amadeus	11-08-2004	11-11-2004	
9	[redacted]	A Room with a View	11-08-2004	11-11-2004	
9	[redacted]	White Christmas	11-08-2004	11-11-2004	
8	[redacted]	Out of Africa	11-08-2004	11-11-2004	

Figure 18-1

If you find that you cannot connect to the MySQL server when trying to open the JSP file, it might be because of the password encryption method used for the MySQL user account. Beginning with MySQL 4.1, a different method is used to encrypt passwords than was used in previous versions. However, some client applications have not yet implemented this new encryption method. As a result, when you try to pass the password from the Java application to MySQL, there is an encryption mismatch. You can test whether this is a problem by using the mysql client utility — logging in with the mysqlapp user account name and the pw1 password — to access the MySQL server. If you're able to log on to the server with mysql utility, then you know that the account is working fine; in which case, encryption mismatch is probably the problem. To remedy this, open the mysql client utility as the root user and execute the following SQL statement:

```
SET PASSWORD FOR 'mysqlapp'@'localhost' = OLD_PASSWORD('pw1');
```

The `OLD_PASSWORD()` function saves that password using the old encryption method, which makes the password compatible with the way your Java application has been implemented.

How It Works

The first step that you took in this exercise was to create a page directive that specified the classes that you will be using for the application, as shown in the following statement:

```
<%@ page import="java.sql.*,
java.text.SimpleDateFormat,
java.lang.Integer" %>
```

The page directive is enclosed in the opening and closing directive tags and includes the `page` keyword and the `import` command, which specifies the classes that you plan to use. In this case, you've specified that all classes in the `java.sql` package be used, plus the `java.text.SimpleDateFormat` and the `java.lang.Integer` classes. The `java.sql` package includes all the classes that you need to interact with a SQL database. The `SimpleDateFormat` class allows you to format date values, and the `Integer` class allows you to convert string values into primary `int` values.

After your page directive, you set up the opening HTML section of your index.jsp file. The `<head>` section establishes the necessary links to the dvdstyle.css and dvdrentals.js files. You then added a `<body>` section that includes two HTML `<table>` elements. The first table provides a structure for the page title — DVD Transaction Listing — and the second table provides the structure for the data displayed on the page. The data includes the order number, customer name, DVD name, and dates that the DVD was checked out, when it is due back, and, if applicable, when it was returned. As a result, the initial table structure that is created in this section sets up a row for the table headings and a cell for each heading.

For more information about HTML, file linking from within HTML, style sheets, and JavaScript functions, refer to the appropriate documentation.

Once you set up your HTML structure, you began the main Java section of the page, which you indicated by including an opening scriptlet tag. Following the tag, you declared two variables:

```
Connection connection;
Statement stmt;
```

The `connection` variable is based on the `Connection` class and will be used to establish a connection to the database. The `stmt` variable is based on the `Statement` class and will be used to access methods in that class. Once you declared the variables, you initiated a try/catch block by adding the following code:

```
try
{
```

The try/catch block is used to catch any exceptions that might be generated if a Java statement does not execute properly. After you set up the block, you created the statements necessary to establish a connection to the database:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();

connection = DriverManager.getConnection("jdbc:mysql://localhost/DVDRentals",
                                         "mysqlapp",
                                         "pw1");
```

The first statement uses the `forName()` method of the class named `Class` to load the `Driver` class in the `com.mysql.jdbc` class package. This process locates and instantiates the Connector/J driver. The `newInstance()` method then creates a new class instance named `DriverManager`, which manages JDBC drivers.

In the next statement, you used the `getConnection()` method of the `DriverManager` class to connect to the MySQL computer on the local computer. The connection is specific to the DVDRentals database and the mysqlapp account in MySQL. The `getConnection()` method creates a `Connection` object, which is then assigned to the `connection` variable that you declared earlier in the code.

Once you established your connection, you declared a String variable named selectSql and assigned the necessary SELECT statement to that variable:

```
String selectSql = "SELECT " +
                   "Transactions.TransID, " +
                   "Transactions.OrderID, " +
                   "Transactions.DVDID, " +
                   "Transactions.DateOut, " +
                   "Transactions.DateDue, " +
                   "Transactions.DateIn, " +
                   "Customers.CustFN, " +
                   "Customers.CustLN, " +
                   "DVDs.DVDName " +
                   "FROM Transactions, Orders, Customers, DVDs " +
                   "WHERE Orders.OrderID = Transactions.OrderID " +
                   "AND Customers.CustID = Orders.CustID " +
                   "AND DVDs.DVDID = Transactions.DVDID " +
                   "ORDER BY  Transactions.OrderID DESC, Customers.CustLN ASC, " +
                   "Customers.CustFN ASC, Customers.CustMN ASC, " +
                   "Transactions.DateDue DESC, DVDs.DVDName ASC";
```

As you can see, this is a basic SELECT statement that joins the Transactions, Orders, Customers, and DVDs tables in the DVDRentals database. You then used the createStatement() method in the Connection object to create a Statement object, as shown in the following statement:

```
stmt = connection.createStatement();
```

The new Connection object is then assigned to the stmt variable, which you declared earlier in the code. You can now use the methods in the Statement object to execute the SELECT statement. To do so, you used the executeQuery() method and specified the selectSql variable as an argument to that method:

```
ResultSet rs = stmt.executeQuery(selectSql);
```

The executeQuery() method executes the SELECT statement and creates a ResultSet object that contains the query results returned by the statement. The ResultSet object is then assigned to the rs variable. From there, you created a while loop that uses ResultSet methods to process the query results:

```
while(rs.next())
{
```

The while loop begins with the while keyword and a condition in parentheses. Each time the condition evaluates to true, the while loop is executed. The condition uses the next() method of the ResultSet object to point to each row in the result set. Each time the while loop is executed, the next() method points to the next row. That row is then used by the statements within the while loop to retrieve values from the result set. For example, the first statement within the while loop retrieves a value from the TransID column and assigns that value to the transId variable:

```
int transId = rs.getInt("TransID");
```

To retrieve the value from the column, you used the getInt() method of the ResultSet object. The method returns a value that is a primary int type. That value is then assigned to the variable, which is the same type as the value. This process is repeated for each value in the row. The `ResultSet` method used depends on the type of data that is retrieved. The `getDate()` method is used for dates, and the `getString()` method is used for strings.

After you populate the variables for a particular row, you then started the process of formatting some of the values into a more readable form. The first step you took to prepare for this process was to declare a variable that creates a new `SimpleDateFormat` object:

```
SimpleDateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");
```

The new object specifies that dates should be formatted as MM-dd-yyyy. For example, dates should have the format of 10-31-2004. The new `SimpleDateFormat` object is then assigned to the `dateFormat` variable, which is used in later code to format the dates that are retrieved from the database. Next, you declared several new variables that will be used to hold the reformatted values, at the same time, you initiated them as empty strings. For example, the following statement declares and initializes the `customerName` variable:

```
String customerName = "";
```

The `customerName` variable has been set as an empty string. The reason you did this is to ensure that if any part of a customer name is null, that null value is not displayed. That way, when you begin formatting the names, you can test for the existence of null values and assign names to the `customerName` variable only if the values are not null. If they are null, then only an empty string will be displayed or only one name, with no null values appended to the name. As a result, after you initiated the variables, you then began concatenating the names, starting with the first name, as shown in the following if statement:

```
if(custFirstName != null)
   customerName += custFirstName + " ";
```

The statement first checks whether the customer's first name is not null. If the condition evaluates to true, the value in the `custFirstName` variable, plus a space (enclosed in the pair of double quotes), is added to the `customerName` variable. Note that when a plus sign precedes an equal sign, the existing variable value is added to the new values, rather than being replaced by those values. This is better illustrated by the next if statement, which then adds the last name to the first name:

```
if(custLastName != null)
   customerName += custLastName;
```

In the case, unless the first name was null, the `customerName` variable currently holds the first name value, along with a space, which is then added to the last name value. As a result, the `customerName` variable now holds the customer's full name, displayed as first name, space, then last name.

You also used an `if` statement to ensure that the `dvdName` variable contains a string, rather than a null, as shown in the following statement:

```
if(dvdName == null)
   dvdName = "";
```

The reason for this is again to ensure that a null value is not displayed on the Web page, but rather a blank value if the DVD name is null.

Next, you formatted the dates so that they're displayed in a more readable format. To format the dates, you used the `format()` method of the `SimpleDateFormat` object assigned to the `dateFormat` variable. For example, the following statement takes the `dateOut` value and converts it into a more readable format:

```
if(dateOut != null)
   dateOutPrint = dateFormat.format(dateOut);
```

The `format()` method produces a formatted date string that is then assigned to the `dateOutPrint` variable, which, as you'll recall, is a `String` variable.

Once you have formatted the values in the way that you want, you can use the variables to display those values in an HTML table structure. This structure follows the same structure that is defined at the beginning of the file, thus providing the column heads for the rows that you now add. Keep in mind that you are still working within the while loop created earlier. So every step that you take at this point still applies to each individual row that is returned by the `next()` method.

To create the necessary row in the table, you used the scriptlet closing tag (`%>`) to get out of Java mode and back into HTML mode. You then created a cell definition for each value that is returned by the result set (and subsequently assigned to a variable). For example, you used the following definition in for first cell in the first row of the table (not counting the heading):

```
<td class="item">
   <nobr>
   <%=orderId%>
   </nobr>
</td>
```

You used the `<td>` and `</td>` elements to enclose the individual cell within the row. The `<nobr>` and `</nobr>` elements indicate that there should be no line break between the two tags. Notice that squeezed between all that is a Java variable that is enclosed by opening (`<%=`) and closing (`%>`) expression tags. As a result, the value of the `orderId` variable is displayed in that cell.

This process is repeated for each value in the row and repeated for each row until the `while` statement loops through all the rows in the result set. After the while loop, you closed the `ResultSet` object, the `Statement` object, and the `Connection` object:

```
rs.close();
stmt.close();
connection.close();
```

This process releases the application and database resources related to these objects. You should always close any objects that you no longer need. After you closed the objects, you closed the try block (by using a closing curly bracket), and then you created a catch block to catch all exceptions:

```
catch(Exception exception)
{
   throw exception;
}
```

Any exception thrown from within the try block will now be printed to a Web page. After you set up the catch block, you closed out the HTML elements and saved the file. You then opened the file in your browser and viewed the transactions in the DVDRentals database. As you discovered, you can view the transactions, but you cannot modify them. So now you can explore what steps you can take to allow your application to support data modification operations.

Inserting and Updating Data in a MySQL Database

Earlier in the chapter, you looked at how to retrieve data from a MySQL database. In this section, you look at how to insert and update data. The process you use for adding either insert or update functionality is nearly identical, except that, as you would expect, you use an INSERT statement to add data and an UPDATE statement to modify data.

Inserting and updating data is different from retrieving data in a couple ways. For one thing, when you execute a SELECT statement, you use the executeQuery() method of the Statement class to execute the query and create a ResultSet object. The data returned by the SELECT statement is contained in that new object. However, when you execute a data modification statement, you use a different approach, such as calling the executeUpdate() method of the PreparedStatement class, which executes the SQL statement and then returns an integer that represents the number of rows affected by the query. You can then use the value as necessary to display messages or take other actions.

To better understand how this works, take a look at this process one step at a time. First, as you did with the SELECT statement, you should declare a String variable and assign your SQL statement to that variable, as shown in the following example:

```
String insertSql = "INSERT INTO CDs (CDName, InStock, OnOrder, Category) VALUES
(?,?,?,?)";
```

What you probably notice immediately is that question marks are used in place of each value to be inserted. Later in the process, you will create statements that insert values in place of the question marks. However, you must first create a PreparedStatement object and assign it to a variable, as shown in the following statement:

```
PreparedStatement ps = conn.prepareStatement(insertSql);
```

As the statement shows, you create the PreparedStatement object by using the prepareStatement() method in a Connection object (which has been assigned to the conn variable). When calling the prepareStatement() method, you include the insertSql variable as an argument. The method then uses this INSERT statement to create a PreparedStatement object, which is then assigned to the ps variable. From there, you can use the variable to access methods in the PreparedStatement object that allow you to assign values to the question mark placeholders in the INSERT statement above. The following statements each assign a value to the placeholders:

```
ps.setString(1, "Beethoven Symphony No. 6 Pastoral");
ps.setInt(2, 10);
ps.setInt(3, 10);
ps.setString(4, "Classical");
```

The PreparedStatement class includes methods that are specific to the type of data that you're going to insert into the table. For example, the first statement uses the setString() method to insert a value in the CDName column of the CDs table. Notice that PreparedStatement method takes two arguments. The first argument is in integer that corresponds to the question mark placeholders in the INSERT statement. The numbers are assigned to placeholders in the order in which they appear in the SQL statement. In other words, 1 is associated with the first question mark, 2 is associated with the second question mark, and so on. The second argument in each method is the actual value to be inserted into the column. When the INSERT statement is executed, these values are used in place of the question marks.

Once you've assigned values to the placeholders, you can use the executeUpdate() method in the PreparedStatement object to execute the INSERT statement, as shown in the following statement:

```
    int rowCount = ps.executeUpdate();
```

Because the INSERT statement is already associated with the PreparedStatement object (through the ps variable), you do not need to specify the query here. When the executeUpdate() method is called, the INSERT statement is executed and an integer is returned, indicating the number of rows that have been affected by the statement. This integer is then assigned to a primitive int variable. You can then use this variable to display the number of rows that have been returned or to notify users if the statement has affected no rows, depending on the needs of your users and the application that you're designing.

You could have just as easily executed an UPDATE statement in place of the INSERT statement, and the process would have been the same. It should also be noted that you can use a PreparedStatement object to execute a SELECT statement if you want to pass values into the statement as you did with the INSERT statement above. If you use this method, you must still use the executeQuery() method as you saw with other SELECT statements, and you can still process the result set. In the following Try It Out sections, you will see how all these processes work.

Adding Insert and Update Functionality to Your Application

So far, your DVDRentals application displays only transaction-related information. In this section, you add to the application so that it also allows you to add a transaction to an existing order and to modify transactions. To support the added functionality, you must create three more files — edit.jsp, insert.jsp, and update.jsp — and you must modify the index.jsp file. Keep in mind that your index.jsp file acts as a foundation for the rest of the application. You should be able to add a transaction and edit a transaction by first opening the index.jsp file and then maneuvering to wherever you need to in order to accomplish these tasks.

The first additional file that you create is the edit.jsp file. The file serves two roles: adding transactions to existing orders and editing existing transactions. These two operations share much of the same functionality, so combining them into one file saves duplicating code. If you're adding a new transaction, the Web page will include a drop-down list that displays each order number and the customer associated with that order, a drop-down list that displays DVD titles, a text box for the date the DVD is rented, and a text box for the date the DVD should be returned. The default value for the date rented text box is the current date. The default value for the date due text box is three days from the current date.

If you're editing a transaction, the Web page will display the current order number and customer, the rented DVD, the date the DVD was rented, the date it's due back, and, if applicable, the date that the DVD was returned.

The edit.jsp Web page will also contain two buttons: Save and Cancel. The Save button saves the new or updated record and returns the user to the index.jsp page. The Close button cancels the operation, without making any changes, and returns the user to the index.jsp page.

After you create the edit.jsp page, you then create the insert.jsp file and the update.jsp file in Try It Out sections that follow this one. From there, you modify the index.jsp page to link together the different functionality. Now take a look at how to create the edit.jsp file.

Try It Out Creating the edit.jsp File

The following steps describe how to create the edit.jsp file, which supports adding new transactions to a record and editing existing transactions:

1. As with the insert.jsp file, you must import the necessary classes into your application. Use your text editor to start a new file, and enter the following code:

```
<%@ page import="java.sql.*,
java.text.SimpleDateFormat,
java.lang.Integer" %>
```

2. Next, you must declare and initialize a number of variables. Later in the code, you use these variables to perform different tasks, such as processing and verifying the data retrieved by a form. Add the following statements your file:

```
<%
// Initialize variables with parameters retrieved from the form
String command = request.getParameter("command");
String transactionIdString = request.getParameter("transaction_id");
String dateDueString = request.getParameter("date_due");
String orderIdString = request.getParameter("order_id");
String dvdIdString = request.getParameter("dvd_id");
String dateOutString = request.getParameter("date_out");
String dateInString = request.getParameter("date_in");

// Initialize variables with default values
java.util.Date dateDue = null;
int transId = -1;
int orderId = -1;
int dvdId = -1;
java.util.Date dateOut = null;
java.util.Date dateIn = null;

String error = "";

SimpleDateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");

// Declare variables for database-related operations
Connection connection;
Statement statement;
```

3. As you did with the index.jsp file, you must establish a connection to the MySQL server and select the database. Add the following code to the edit.jsp file:

```
// Wrap database-access code in try/catch block to handle errors
try
{
// Create a DriverManager object to connect to the database
   Class.forName("com.mysql.jdbc.Driver").newInstance();
```

```
      connection = DriverManager.getConnection("jdbc:mysql://localhost/DVDRentals",
                                               "mysqlapp",
                                               "pw1");
```

4. Next, the file should process the new or edited transactions when the user clicks the Save button. The first step in doing this is to check for missing parameters and reformat the date information. Enter the following Java code:

```
  // Process the save and savenew commands
     if("save".equals(command) || "savenew".equals(command))
     {
  // Check for missing parameters and reformat the dates for MySQL
        if(transactionIdString != null)
           transId = Integer.valueOf(transactionIdString).intValue();

        if(orderIdString != null)
           orderId = Integer.valueOf(orderIdString).intValue();

        if(orderId == -1)
           error += "Please select an \"Order\"<br>";

        if(dvdIdString != null)
           dvdId = Integer.valueOf(dvdIdString).intValue();

        if(dvdId == -1)
           error += "Please select a \"DVD\"<br>";

        if((dateDueString != null) && (dateDueString.length() > 0))
           dateDue = dateFormat.parse(dateDueString);
        else
           error += "Please enter a \"Date Due\"<br>";

        if((dateOutString != null) && (dateOutString.length() > 0))
           dateOut = dateFormat.parse(dateOutString);
        else
           error += "Please enter a \"Date Out\"<br>";

        if((dateInString != null) && (dateInString.length() > 0))
           dateIn = dateFormat.parse(dateInString);
```

Note that the application does not check the format of the date submitted by users. Normally, an application would include some type of mechanism to ensure that submitted dates are in a usable format.

5. Then you can carry out the update or insert by calling the applicable include files. (These files are created in later Try It Out sections.) Once the code in the applicable include file runs, you should redirect users back to the index.jsp page. Enter the following code in your file:

```
        if(error.length() == 0)
        {
           if("save".equals(command))
           {
  // Run the update in update.jsp
```

```
%>
            <%@ include file="update.jsp" %>
<%
         }
         else
         {
// Run the insert in insert.jsp
%>
            <%@ include file="insert.jsp" %>
<%
         }

// Redirect the application to the listing page
         response.sendRedirect("index.jsp");
      }
   }
```

6. The next step is to set up the file to support adding or updating a record when the user has been redirected to this page from the index.jsp page. This is done as part of the else statement in an if...else structure. This particular section sets up the default values for a new record. Add the following code to your file:

```
   else
   {
// If it is a new record, initialize the variables to default values
      if("add".equals(command))
      {
         transId = 0;
         orderId = 0;
         dvdId = 0;
         dateOutString = dateFormat.format(new java.util.Date());
         dateDueString = dateFormat.format(new Date((new
java.util.Date()).getTime() + 3L*24L*60L*60000L));
         dateInString = "";
      }
```

7. Next, you must set up the file with the values necessary to support editing a record. This involves retrieving records to set an initial value for a number of variables. Add the following code to your JSP file:

```
      else
      {
// If it is an existing record, read from database

         if(transactionIdString != null)
         {

// Build query from transactionId value passed down from form
            transId = Integer.valueOf(transactionIdString).intValue();

            String selectSql = "SELECT " +
                               "OrderID, " +
                               "DVDID, " +
                               "DateOut, " +
                               "DateDue, " +
                               "DateIn " +
                              "FROM Transactions " +
                               "WHERE TransID = ?";
// Execute query
            PreparedStatement preparedStatement =
connection.prepareStatement(selectSql);

            preparedStatement.setInt(1, transId);

            ResultSet rs = preparedStatement.executeQuery();

// Populate the variables for display into the form
            if(rs.next())
            {
               orderId = rs.getInt("OrderID");
               dvdId = rs.getInt("DVDID");
               dateOut = (java.util.Date) rs.getDate("DateOut");
               dateDue = (java.util.Date) rs.getDate("DateDue");
               dateIn = (java.util.Date) rs.getDate("DateIn");

// Reformat the dates into a more readable form
               if(dateOut != null)
                  dateOutString = dateFormat.format(dateOut);
               else
                  dateOutString = "";

               if(dateDue != null)
                  dateDueString = dateFormat.format(dateDue);
               else
                  dateDueString = "";

               if(dateIn != null)
                  dateInString = dateFormat.format(dateIn);
               else
                  dateInString = "";
            }

// Close the database objects
            rs.close();
            preparedStatement.close();
         }
      }
   }
%>
```

8. Now you must create the HTML section of your form to allow users to view and enter data. This section includes a form to pass data to Java and the table structure to display the form. Add the following code to your JSP file:

```
<html>
<head>
   <title>DVD - Listing</title>
   <link rel="stylesheet" href="dvdstyle.css" type="text/css">
   <script language="JavaScript" src="dvdrentals.js"></script>
```

```
</head>

<body>

<form name="mainForm" method="post" action="edit.jsp">
<input type="hidden" name="command" value="view">
<input type="hidden" name="transaction_id" value="<%=transId%>">

<p></p>

<table cellspacing="0" cellPadding="0" width="619" border="0">
<tr>
   <td>
      <table height="20" cellspacing="0" cellPadding="0" width="619" border="0">
      <tr align=left>
         <td valign="bottom" width="400" class="title">
            DVD Transaction
         </td>
         <td align="right" width="219" class="title">&nbsp;</td>
      </tr>
      </table>
      <br>
      <%if(error.length() > 0){%>
      <table cellspacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="619" class="error"><%=error%></td>
      </tr>
      </table>
      <%}%>
```

9. Now create the first row of your form, which allows users to view and select an order ID. Enter the following code in your file:

```
      <table cellspacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="250" class="heading">Order</td>
         <td class="item">
            <select name="order_id">
               <option value="-1">Select Order</option>
<%
// Retrieve data to populate drop-down list
   String selectSql = "SELECT " +
                      "Orders.OrderID, " +
                      "Orders.CustID, " +
                      "Customers.CustFN, " +
                      "Customers.CustLN " +
                      "FROM Orders, Customers " +
                      "WHERE " +
                      "Customers.CustID = Orders.CustID " +
                      "ORDER BY Orders.OrderID DESC, Customers.CustLN ASC, " +
                      "Customers.CustFN ASC";

// Execute the query
   statement = connection.createStatement();
   ResultSet rs = statement.executeQuery(selectSql);
```

```
// Loop through the results
   while(rs.next())
   {
// Assign returned values to the variables
      int orderId1 = rs.getInt("OrderID");
      String custFirstName = rs.getString("CustFN");
      String custLastName = rs.getString("CustLN");

// Format the data for display
      String customerName = "";

      if(custFirstName != null)
         customerName += custFirstName + " ";

      if(custLastName != null)
         customerName += custLastName;

// If the order ID matches the existing value, mark it as selected
      if(orderId1 != orderId)
      {
%>
               <option value="<%=orderId1%>"><%=orderId1%> -
<%=customerName%></option>
<%
      }
      else
      {
%>
               <option selected value="<%=orderId1%>"><%=orderId1%> -
<%=customerName%></option>
<%
      }
   }

// Close database objects
   rs.close();
   statement.close();
%>
            </select>
         </td>
      </tr>
```

10. The second row of your form allows users to view and select a DVD to associate with your transaction. Add the following code to your edit.jsp file:

```
      <tr>
         <td class="heading">DVD</td>
         <td class="item">
            <select name="dvd_id">
               <option value="-1">Select DVD</option>
<%
// Retrieve data to populate the drop-down list
    selectSql = "SELECT DVDID, DVDName FROM DVDs ORDER BY DVDName";

    statement = connection.createStatement();
```

```
        rs = statement.executeQuery(selectSql);

        while(rs.next())
        {
           int dvdId1 = rs.getInt("DVDID");
           String dvdName = rs.getString("DVDName");

           if(dvdName == null) dvdName = "";

           if(dvdId1 != dvdId)
           {
    %>
                   <option value="<%=dvdId1%>"><%=dvdName%></option>
    <%
           }
           else
           {
    %>
                   <option selected value="<%=dvdId1%>"><%=dvdName%></option>
    <%
           }
        }

    // Close the database objects
        rs.close();
        statement.close();
    %>
                </select>
             </td>
          </tr>
```

11. Next, create three more rows in your table, one for each date-related value. Enter the following code:

```
          <tr>
             <td class="heading">Date Out</td>
             <td class="item">
                <input type="text" name="date_out" value="<%=dateOutString%>"
    size="50">
             </td>
          </tr>
          <tr>
             <td class="heading">Date Due</td>
             <td class="item">
                <input type="text" name="date_due" value="<%=dateDueString%>"
    size="50">
             </td>
          </tr>
          <%if((!"add".equals(command)) && (!"savenew".equals(command))){%>
          <tr>
             <td class="heading">Date In</td>
             <td class="item">
                <input type="text" name="date_in" value="<%=dateInString%>" size="50">
             </td>
          </tr>
          <%}%>
```

12. Now add the Save and Cancel buttons to your form by appending the following code to your file:

```
      <tr>
         <td colspan="2" class="item" align="center">
            <table cellspacing="2" cellPadding="2" width="619" border="0">
            <tr>
               <td align="center">
                  <%if(("add".equals(command)) || ("savenew".equals(command))){%>
                  <input type="button" value="Save" class="add"
onclick="doSave(this, 'savenew')">
                  <%}else{%>
                  <input type="button" value="Save" class="add"
onclick="doSave(this, 'save')">
                  <%}%>
               </td>
               <td align="center">
                  <input type="button" value="Cancel" class="add"
onclick="doCancel(this)">
               </td>
            </tr>
            </table>
         </td>
      </tr>
      </table>
   </td>
</tr>
```

13. Close the various HTML elements and close your connection to the MySQL server by entering the following code:

```
</table>
</form>
</body>
</html>

<%
// Close connection and throw exceptions
   connection.close();
}
catch(Exception exception)
{
throw exception;
}
%>
```

14. Save the edit.jsp file to the appropriate Web application directory.

How It Works

In this exercise, you created the edit.jsp file, which supports the insert and update functionality in your DVDRentals application. The first step you took to set up the file was to add the page directive necessary to import the Java classes into your page. These are the same classes that you import for the index.jsp file. Once you set up the page directive, you declared and initialized a number of variables. The first set of variables is associated with values returned by the `request` object, which contains values submitted to the page by forms. For example, the following Java statement retrieves the `command` value returned by a form:

```
String command = request.getParameter("command");
```

The statement uses the `getParameter()` method of the `request` object to retrieve the `command` value. The value is returned as a string. As a result, a `String` object is automatically created. That object is then assigned to the `command` variable, which has been declared as a `String` type.

Once you assigned the form parameter values to the necessary variables, you then declared and initialized several variables used later in the code to display initial values in form. For example, the first variable that you declared was `dateDue`:

```
java.util.Date dateDue = null;
```

Notice that the `dateDue` variable is declared as a `java.util.Date` type. Because date values will need to be displayed, entered into a form, and subsequently added to a database, you need to create both `java.util.Date` objects and `java.sql.Date` objects. Keep in mind, however, that only the `java.sql.Date` class is actually imported into the page, not the `java.util.Date` class. So if you simply reference `Date` in your Java code, you're referring to the `java.sql.Date` class. However, whenever you're converting or manipulating date values in this page and using both types of `Date` objects, it's generally a good idea to use the fully qualified name in either case to ensure that you're referring to the correct `Date` class.

Also notice in the statement variable declaration above that an initial value of `null` is assigned to the `dateDue` variable. This is done to prepare the variable for processing later in the code, as you see in that section of the file.

The next variable that you declared and initialized was the `transID` variable, as shown in the following statement:

```
int transId = -1;
```

This is a straightforward primitive `int` type variable. You initialized the variable by assigning a `-1` to the variable. The `-1` value is used simply to ensure that no value is assigned that might conflict with a value retrieved by the database. As you see later in the code, the `transId` variable is associated with the transaction ID as it is stored in the Transactions table of the DVDRentals database. The IDs are stored in the TransID column, which is configured as a primary key. As a primary key, only positive integer values are stored in the column. As a result, by assigning an integer other than a positive integer, you're ensuring that the initial variable value will not conflict with an actual value.

Once you declared these variables, you then went on to declare the `error` variable and initiated it by setting its value to an empty string, as shown in the following statement:

```
String error = "";
```

The variable is used to display error messages if a user does not fill out the form properly. You set the value to an empty string so that you can use that to verify whether any error messages have been received, as you see in later in the page. After you initiated the `error` variable, you declared and initiated the following `dateFormat` variable:

```
SimpleDateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");
```

In initiating the variable, you created a SimpleDateFormat object that specifies the format MM-dd-yyyy. As a result, you can use the variable to call methods in the SimpleDateFormat class. This variable is necessary to convert and format date values later in the page.

Next you set up a Connection variable and a Statement variable, as you did for the index.jsp file. You use these variables in the same way that you used the variables in index.jsp. After declaring the variables, you set up if...else statements that begin with the following if condition:

```
if("save".equals(command) || "savenew".equals(command))
```

The if statement specifies two conditions. The first condition uses the equals() method to compare the string save to the current value in the command variable. If the values are equal, the condition evaluates to true. The second condition also uses the equals() method to compare the string savenew to the command variable. If the values are equal, the condition evaluates to true. Because the conditions are connected by the or (||) operator, either condition can be true in order for the if statement to be executed. If neither condition is true, the else statement is executed.

The save and savenew command values are issued when you click the Save button. You learn more about that button shortly.

The if...else statements contain a number of if...else statements embedded in them. Before getting deeper into the code that makes up all these statements, first take a look at a high overview of the logic behind these statements. It gives you a bigger picture of what's going on and should make understanding the individual components a little easier. The following pseudo-code provides an abbreviated statement structure starting with the outer if statement described previously:

```
if command = save or savenew, continue (if !=, go to else)
{
   if transactionIdString != null, assign to transID
   if OrderIdString != null, assign to orderId
   if OrderIdString = -1, return error message
   if dvdIdString != null, assign to dvdId
   if dvdIdString = -1, return error message
   if dateDueString != null and length > 0, assign to dateDue
      else return error message
   if dateOutString != null and length > 0, assign to dateOut
      else return error message
   if dateInString != null and length > 0, assign to dateIn
   if no error, continue
   {
      if command = save, include update.jsp
         else include insert.jsp
      redirect to index.jsp
   }
}
else (if command != save or savenew)
{
   if command = add, continue (if !=, go to else)
   {
      initialize variables (x 6)
   }
   else (if command != add)
```

```
      {
         if transactionIdString != null
         {
            process query
            if query results exist, fetch results
            {
               assign variables (x 5)
               if date != null, format date and assign to String variable
                  else set date to empty string (x 3)
            }
         }
      }
   }
```

As you can see, the action taken depends on the values in the `command` and `transactionIdString` variables. The outer `if` statement basically determines what happens when you try to save a record, and the outer `else` statement determines what happens when you first link to the page from the index.jsp page. Embedded in the else statement is another set of `if...else` statements. The embedded `if` statement determines what happens if you want to add a transaction, and the embedded `else` statement determines what happens if you want to update a transaction.

Now take a closer look at these statements. You've already seen the opening `if` statement. A number of embedded `if` statements follow the outer `if` statement. For example, the following statements convert a string value to an integer value and set up an error condition:

```
if(orderIdString != null)
   orderId = Integer.valueOf(orderIdString).intValue();

if(orderId == -1)
   error += "Please select an \"Order\"<br>";
```

The first `if` statement defines the condition that the `orderIdString` value should not be null. If the condition evaluates to true, the `valueOf()` and `intValue()` methods of the `Integer` object are used to assign a value to the `orderId` variable. The `valueOf()` method creates an `Integer` object based on the string value assigned to the `orderIdString` variable. The `intValue()` method then returns that value as an primitive `int` type, which is then assigned to the `orderId` variable.

The next `if` statement specifies the condition that `orderId` should equal a value of `-1`. If the condition evaluates to true, an error message is returned, telling the user to select a value. (You assigned the value -1 to the variable earlier in the page, and the default value that you set up in the order ID and DVD drop-down lists is -1.)

This section of code also includes embedded `if...else` statements. The `if` statement includes two conditions, as shown in the following code:

```
if((dateDueString != null) && (dateDueString.length() > 0))
   dateDue = dateFormat.parse(dateDueString);
else
   error += "Please enter a \"Date Due\"<br>";
```

The `if` condition specifies that the `dateDueString` value cannot be null *and* the value must have a length greater than zero characters. If both these conditions are met, the `dateFormat` variable is used to call the `parse()` method of the `SimpleDateFormat` class. The `parse()` method converts a string value to a date value that is used to create a Date object. That object is then assigned to the dateDue variable. If either of the two if conditions evaluate to false, the else statement is executed and Java returns an error to the user.

The next step you took was to use the length() method to determine whether the error variable contained any characters. If the error variable is empty, the condition evaluates to true, which means that it contains no error messages and the rest of the code should be processed. This means that a file should be included in the page, as shown in the following code:

```
         if(error.length() == 0)
         {
            if("save".equals(command))
            {
%>
               <%@ include file="update.jsp" %>
<%
            }
            else
            {
%>
               <%@ include file="insert.jsp" %>
<%
            }
```

If no error messages are returned by the previous code, the embedded if...else statements are then applied. If the command value is save, the update.jsp file code is executed; otherwise, the insert.jsp file code is executed. (You create these files in later Try It Out sections.) Once the statements in the applicable include file are executed, the following statement is executed:

```
response.sendRedirect("index.jsp");
```

The statement uses the sendRedirect() method of the response object to redirect the user to index.jsp, which is specified as an argument in the method. This completes the outer if statement that initiates this section of code. However, if the if statement is not applicable (command does not equal save or savenew), Java executes the outer else statement.

The outer else statement is made up of its own embedded if...else statements. The if statement applies if the command variable currently holds the value add, as shown in the following code:

```
if("add".equals(command))
{
   transId = 0;
   orderId = 0;
   dvdId = 0;
   dateOutString = dateFormat.format(new java.util.Date());
   dateDueString = dateFormat.format(new Date((new java.util.Date()).getTime() +
3L*24L*60L*60000L));
   dateInString = "";
}
```

If command is set to add, this means that you are creating a new transaction. To prepare the Web page with the variables it needs to properly display the form when you open the page, you set a number of variables to specific values. For example, you set transId to 0, which is an unused number and so does not match any current transaction IDs.

Now take a look at the `dateOutString` variable. The goal here is to assign the current date to the variable and to assign that date as a string value. The first step is to create a new `java.util.Date` object based on the current date. This object serves as an argument to the `format()` method of the `SimpleDateFormat` class, which creates a new `String` object based on that date value. The `String` object is then assigned to the `dateOutString` variable, which can then be used to access the current date as a string value.

Next, take a look at the `dateDueString` variable. This is similar to the `dateOutString` variable except that it contains an additional element: it adds three days to the current date. It accomplishes this by using the `getTime()` method of the `Date` object to return the number of milliseconds that have passed since January 1, 1970. It then adds the equivalent of three days in milliseconds to that amount (3L days * 24L hours * 60L minutes * 60000L milliseconds). The L is included after the numbers to indicate that Java should use the long form of the `int` values to calculate the total milliseconds.

The new `java.util.Date()` object and its calculations are then passed as an argument to the `Date()` constructor in order to create a new `Date()` object. Because name of the `Date()` constructor is not fully qualified, the `java.sql.Date` class is assumed because that is the class that has been imported into the JSP page. As a result, a `Date` object is created that contains a date three days from the current date. Now you can use the `format()` method of the `SimpleDateFormat` class to convert the date value to a string, which is then assigned to the `dateDueString` variable.

After you set up the variables for the embedded `if` statement, you then added the necessary statements to the embedded `else` statement. You began the `else` block with another `if` statement that verifies that the `transactionIdString` variable is not null, as shown in the following code:

```
if(transactionIdString != null)
```

If the value is not null (a value does exist), the remaining part of the `if` statement is executed. This means that you are editing an existing transaction, in which case, the `transactionIdString` variable identifies that transaction. However, the value is returned as a string, but transaction IDs are stored in the database as integers. As a result, you used `Integer` class methods to convert the string value to an integer value. From there, you created a `SELECT` statement and assigned it to the `selectSql` variable:

```
String selectSql = "SELECT " +
                   "OrderID, " +
                   "DVDID, " +
                   "DateOut, " +
                   "DateDue, " +
                   "DateIn " +
                   "FROM Transactions " +
                   "WHERE TransID = ?";
```

As you can see, the `SELECT` statement includes a question mark placeholder. To assign a variable to the placeholder, you first created a `PreparedStatement` object, assigned a value to the placeholder, and executed the statement, as shown in the following code:

```
PreparedStatement preparedStatement = connection.prepareStatement(selectSql);
preparedStatement.setInt(1, transId);
ResultSet rs = preparedStatement.executeQuery();
```

The first statement uses the `prepareStatement()` method of the `Connection` class to create a `PreparedStatement` object based on the `SELECT` statement. The object is then assigned to the `preparedStatement` variable. You then used the `setInt()` method of the new `PreparedStatement` object to set the question mark placeholder to the value associated with the transId variable. Next, you used the executeQuery() method to execute the SELECT statement and create a ResultSet object, which is assigned to the rs variable. Once you set up the query, you used an if statement and next() method to process the result set. Because only one row is returned by the query, you do not need to use a while loop.

When you processed the result set, you used ResultSet methods to assign values to variables. This is the same method that you use in the index.jsp file to process query results. However, there is one element that you didn't see in the index.jsp file, as shown in the following statement:

```
dateOut = (java.util.Date) rs.getDate("DateOut");
```

Notice that, following the equal sign, the statement includes the java.util.Date class enclosed in parentheses. This is known as *casting* and is used to convert an object from a subclass to a parent class type. (The java.sql.Date class inherits from java.util.Date class.) The date value retrieved from the DateOut column is converted from a java.sql.Date value to a java.util.Date value when it is assigned to the dateOut variable, which is how it is retrieved from the database. You did this so that you can then convert the date values to strings, as shown in the following code:

```
if(dateOut != null)
   dateOutString = dateFormat.format(dateOut);
else
   dateOutString = "";
```

If the dateOut value is not null, the value is converted to a string and assigned to the dateOutString variable, otherwise the variable is set to an empty string. The first step (the if statement) is necessary to display the value on the form, and the second step (the else statement) is necessary to ensure that null is not displayed, should the value be null.

Once you set up the data to populate the form, you have completed the if...else block of code. If necessary, refer to the summary code provided at the beginning of this description. This provides a handy overview of what is going on.

The next section of code that you created set up the HTML for the Web page. As with the index.jsp file, the HTML section includes header information that links to the dvdstyle.css file and the dvdrentals.js file. The section also includes a <form> element and two <input> elements:

```
<form name="mainForm" method="post" action="edit.jsp">
<input type="hidden" name="command" value="view">
<input type="hidden" name="transaction_id" value="<%=transId%>">
```

A form is an HTML structure that allows a user to enter or select data and then submit that data. The data can then be passed on to other Web pages or to the Java code. This particular form uses the post method to send data (method="post") and sends that data to the current page (action="edit.jsp"). Beneath the form, you added two <input> elements that create the initial values to be inserted into the command and transaction_id parameters. The command parameter is set to view, and the transaction_id parameter is set to the value contained in the transId variable. To use the variable to set the transaction_id value, you must enclose the variable in Java opening and closing expression tags so the value can be used by the form. Also note that the input type for both <input> elements is hidden, which means that the user does not actually see these two elements. Instead, they serve only as a way to pass the command and transaction_id values, which is done in the background. The user does not enter these values.

Forms are a common method used in HTML to pass data between pages. It is also useful for passing values between HTML and Java. For more information about forms, consult the applicable HTML documentation.

After you defined the form, you then set up the table to display the heading DVD Transaction at the top of the page. From there, you added another table whose purpose is to display error messages, as shown in the following code:

```
<%if(error.length() > 0){%>
<table cellspacing="2" cellPadding="2" width="619" border="0">
<tr>
   <td width="619" class="error"><%=error%></td>
</tr>
</table>
<%}%>
```

The HTML table structure is preceded by Java opening and closing scriptlet tags so that you can use an `if` statement to specify a condition in which the table will be displayed. The `if` statement specifies that the `error` variable must contain a string whose length is greater than zero characters. This is done by using the `length()` method to determine the length of the error value and then comparing the length to zero. If the condition evaluates to true, the table is created and the error is printed. At the end of the table, you again used a pair of opening and closing scriptlet tags to add the closing bracket of the `if` statement.

Once you have established a way for error messages to be displayed, you set up the table that will be used to display the part of the form that they user sees. The first row of the form will contain a drop-down list of order IDs — along with the customer names associated with those orders — that the user will be able to select from when adding a transaction. To populate this drop-down list, you retrieved data from the database, processed the data, and formatted the customer name. The methods used for retrieving and processing the data are the same methods that you've already used in the application. Once you retrieved the value, you added another form element to your Web page, only this form element is visible to the user:

```
      if(orderId1 != orderId)
      {
%>
               <option value="<%=orderId1%>"><%=orderId1%> -
<%=customerName%></option>
<%
      }
      else
      {
%>
               <option selected value="<%=orderId1%>"><%=orderId1%> -
<%=customerName%></option>
<%
      }
```

The form element shown here is an `<option>` element. An `<option>` element allows a user to select from a list of options in order to submit data to the form. There are actually two `<option>` elements here, but only one is used. This is because Java `if...else` statements enclose the `<option>` elements.

The if statement condition specifies that orderId1 should not equal orderId. If they are not equal, the if statement is executed and the first <option> element is used, otherwise the second <option> element is used. The second element includes the selected option, which means that the current order ID is the selected option when the options are displayed.

The orderId1 variable receives its value from the results returned by the SELECT statement used to populate the <option> element. The orderId variable receives its value from the SELECT statement that is used to assign values to variables once it has been determined that the user is editing an existing transaction. (This occurs when the if...else statements earlier in the code are processed.) If the two values are equal, the second <option> element is used, which means that the current order ID is displayed when this page is loaded. If the two values are not equal, which is the condition specified in the if statement, no order ID is displayed, which you would expect when creating a new record.

The next row that you created for your form table allows users to select from a list of DVD names. The same logic is used to create the drop-down list available to the users. The only difference is that, because only DVD names are displayed, no special formatting or concatenation is required to display the values.

After you created your two rows that display the drop-down lists to the users, you created three date-related rows. Each row provides a text box in which users can enter the appropriate dates. For example, the first of these rows includes the following form element:

```
<input type="text" name="date_out" value="<%=dateOutString%>" size="50">
```

As you can see, this is an <input> element, similar to the ones that you created when you first defined the form. Only the input type on this one is not hidden, but instead is text, which means that a text box will be displayed. The name of the text box is date_out. This is actually the name of the parameter that will hold the value that is submitted by the user. The initial value displayed in the text box depends on the value of the dateOutString variable. For new records, this value is the current date, and for existing records, this is the value as it currently exists in the database. (Both these values are determined in the earlier Java code.)

Once you completed setting up the various form elements, you added two more elements: one for the Save button and one for the Cancel button. For example, your code for the Save button is as follows:

```
<td align="center">
   <%if(("add".equals(command)) || ("savenew".equals(command))){%>
   <input type="button" value="Save" class="add" onclick="doSave(this, 'savenew')">
   <%}else{%>
   <input type="button" value="Save" class="add" onclick="doSave(this, 'save')">
   <%}%>
</td>
```

A button is also an <input> element on a form, but the type is specified as button. The value for this element determines the name that appears on the button, which in this case is Save. The class option specifies the style that should be used in the button, as defined in the dvdstyle.css file, and the onclick option specifies the action to be taken. In this case, the action is to execute the doSave() function, which is defined in the dvdrentals.js file.

Notice that there are again two <input> elements, but only one is used. If the command value equals add or savenew, the first <input> element is used, otherwise the second <input> element is used.

When you click the Save button, the `doSave()` function is called. The function takes one argument, `this`, which is a self-referencing value that indicates that the action is related to the current HTML input button. When the function is executed, it submits the form to the edit.jsp file and sets the `command` parameter value to `savenew` or `save`, depending on which `<input>` option is used. Based on the `command` value, the Java script is processed once again, only this time, the first `if` statement (in the large `if...else` construction) evaluates to true and that statement is executed. Assuming that there are no errors, the date values are reformatted for MySQL, the Java code in the update.jsp or insert.jsp include file is executed, and the user is redirected to the index.jsp page.

As you can see, the edit.jsp page provides the main logic that is used to insert and update data. However, as the code in this page indicates, you must also create the include files necessary to support the actual insertion and deletion of data. In the next Try It Out section, you create the insert.jsp file. The file contains only that script that is necessary to insert a record, based on the values provided by the user in the edit.jsp form.

Try It Out Creating the insert.jsp file

The following steps describe how to create the insert.jsp file:

1. In your text editor, create a new file and enter the following code:

```
<%
// Build the INSERT statement with parameter references
String insertSql = "INSERT INTO Transactions (OrderID, DVDID, DateOut, DateDue)
VALUES (?, ?, ?, ?)";

PreparedStatement preparedStatement = connection.prepareStatement(insertSql);

// Set the parameters
preparedStatement.setInt(1, orderId);
preparedStatement.setInt(2, dvdId);
preparedStatement.setDate(3, new java.sql.Date(dateOut.getTime()));
preparedStatement.setDate(4, new java.sql.Date(dateDue.getTime()));

// Execute the INSERT statement
int rowCount = preparedStatement.executeUpdate();

preparedStatement.close();
%>
```

2. Save the insert.jsp file to the appropriate Web application directory.

How It Works

In this exercise, you created the insert.jsp file, which is an include file for edit.jsp. The first step you took in creating the insert.jsp file was to assign an `INSERT` statement to the `insertSql` variable:

```
String insertSql = "INSERT INTO Transactions (OrderID, DVDID, DateOut, DateDue)
VALUES (?, ?, ?, ?)";
```

Instead of including the values to be inserted into the MySQL database, the statement includes four question mark placeholders. Values for the placeholders are defined later in the file.

After you created the INSERT statement, you assigned a PreparedStatement object to a variable:

```
PreparedStatement preparedStatement = connection.prepareStatement(insertSql);
```

First, you used the prepareStatement() method in the Connection class (accessed through the connection variable) to create a PreparedStatement object based on the INSERT statement. You then used methods in that object to assign values to the question mark placeholders. For example, the following statement uses the setInt() method to assign the value in the orderId variable to the first placeholder:

```
preparedStatement.setInt(1, orderId);
```

When you insert a date value in an SQL database, you must be certain that the value conforms to SQL standard. To do this, you must first use the setDate() method to specify the value to be inserted, as shown in the following statement:

```
preparedStatement.setDate(3, new java.sql.Date(dateOut.getTime()));
```

In this case, the setDate() method assigns the current date to the question mark placeholder. This is done by first using the getTime() method of the Date() object (accessed through the dateOut variable) to specify the date in the dateOut variable in milliseconds. Next, you created a java.sql.Date object based on dateOut.getTime(), which ensures that the date is passed to the in the format correct for SQL.

After you assigned values to the placeholders, you used the executeUpdate() method of the PreparedStatement class to execute the INSERT statement and to return the number of rows in the database that were affected by the statement. This number is a primitive int value and is assigned to the rowCount variable, as shown in the following statement.

```
int rowCount = preparedStatement.executeUpdate();
```

If you want, you can use this variable to take some sort of action, such as return a message to a user or log the results.

In addition to creating the insert.jsp file, you must also create the update.jsp file. This file works just like the insert.jsp file in that it is included in the edit.jsp file. This has the same effect as including the statements directly into the edit.jsp file. In the following Try it Out section, you create the update.jsp file.

Try It Out Creating the update.jsp file

The following steps describe how to create the update.jsp file:

1. Open a new file in your text editor, and enter the following code:

```
<%
// Build the UPDATE statement with parameters references
String updateSql = "UPDATE Transactions SET OrderID = ?, DVDID = ?, DateOut = ?,
DateDue = ?, DateIn = ? WHERE TransID = ?";

PreparedStatement preparedStatement = connection.prepareStatement(updateSql);

// Set the parameters
```

```
preparedStatement.setInt(1, orderId);
preparedStatement.setInt(2, dvdId);
preparedStatement.setDate(3, new java.sql.Date(dateOut.getTime()));
preparedStatement.setDate(4, new java.sql.Date(dateDue.getTime()));

// Provide a default value for the DateIn column if no value is provided
if(dateIn != null)
   preparedStatement.setDate(5, new java.sql.Date(dateIn.getTime()));
else
   preparedStatement.setString(5, "0000-00-00");

preparedStatement.setInt(6, transId);

// Execute the UPDATE statement
int rowCount = preparedStatement.executeUpdate();

preparedStatement.close();
%>
```

2. Save the update.jsp file to the appropriate Web application directory.

How It Works

In this exercise, you created the update.jsp file. The file uses the same types of objects and methods that you used in the insert.jsp file. First, you created an update statement that you assigned to the `updateSql` variable. Then you used the `prepareStatement()` method of the `Connection` class to create a `PreparedStatement` object and assign that object to the `preparedStatement` variable. From there, you used `PreparedStatement` methods to assign values to the placeholders. You then used the `executeUpdate()` method to execute the SQL statement and return an integer to the `rowCount` variable. The only new element added to the update.jsp file is the following statement:

```
if(dateIn != null)
   preparedStatement.setDate(5, new java.sql.Date(dateIn.getTime()));
else
   preparedStatement.setString(5, "0000-00-00");
```

This statement is a little different from what you've seen so far, although the elements that make up the statement should be familiar to you. First, you used the `getTime()` method retrieved through the `dateIn` variable to return that date in milliseconds. You passed this value as an argument in the `java.sql.Date()` constructor in order to create a new `Date` object based on that value. You then assigned that date value to the fifth question mark placeholder in your `UPDATE` statement.

The steps that you took to assign the value to the fifth placeholder apply only if the condition in the `if` statement evaluates to true, otherwise the else statement is executed. If that occurs, the value is set to 0000-00-00. This is done because the `DateIn` column in the Transaction table does not permit null values. As a result, if a value is not known, MySQL uses 0000-00-00 as the default.

Now that you have created the insert.jsp file and the update.jsp file, only one step remains to set up your application to insert and update data. You must modify the index.jsp file so that it includes the functionality necessary to link the user to the edit.jsp page. The following Try It Out section explains how to modify the index.jsp file. It then walks you through the process of inserting a transaction and then modifying that transaction.

Try It Out Modifying the index.jsp File

The following steps describe how to modify the index.jsp file to support the insert and update operations:

1. In your text editor, open the index.jsp file. Add a form, an `<input>` element, and a cell definition to your HTML code. Add the following code (shown with the gray screen background) to your file:

```
<html>
<head>
   <title>DVD - Listing</title>
   <link rel="stylesheet" href="dvdstyle.css" type="text/css">
   <script language="JavaScript" src="dvdrentals.js"></script>
   </script>
</head>

<body>

<form name="mainForm" method="post" action="index.jsp">
<input type="hidden" name="command" value="view">
<input type="hidden" name="transaction_id" value="">

<p></p>

<table cellSpacing=0 cellPadding=0 width=619 border=0>
<tr>
   <td>
      <table height=20 cellSpacing=0 cellPadding=0 width=619 bgcolor=#bed8e1
border=0><TBODY>
      <tr align=left>
         <td valign="bottom" width="400" class="title">
            DVD Transaction Listing
         </td>
         <td align="right" width="219" class="title">
            <input type="button" value="New Transaction" class="add"
onclick="doAdd(this)">
         </td>
      </tr>
      </table>
      <br>
      <table cellSpacing="2" cellPadding="2" width="619" border="0">
      <tr>
         <td width="250" class="heading">Order Number</td>
         <td width="250" class="heading">Customer</td>
         <td width="250" class="heading">DVDName</td>
         <td width="185" class="heading">DateOut</td>
         <td width="185" class="heading">DateDue</td>
         <td width="185" class="heading">DateIn</td>
         <td width="99" class="heading">&nbsp;</td>
      </tr>
```

When adding code to you file, be sure to add it in the position shown here.

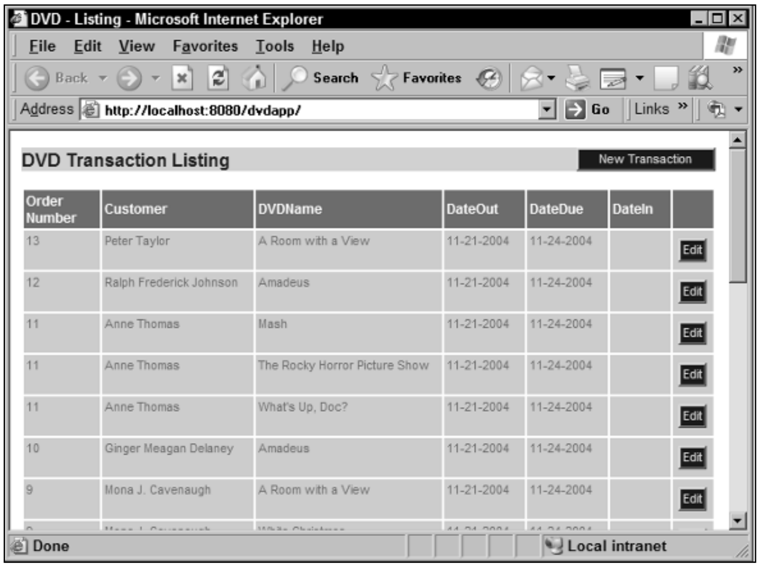
2. Next, add an HTML table cell and an `<input>` element to the area of code that prints out the values returned by the database. Add the following code (shown with the gray screen background) to your file:

```
            <td class="item">
               <nobr>
               <%=dateInPrint%>
               </nobr>
            </td>
            <td class="item" valign="center" align="center">
               <input type="button" value="Edit" class="edit" onclick="doEdit(this,
<%=transId%>)">
            </td>
         </tr>
```

3. Now you must close the form, which you do near the end of the file. To close the form, you must use a `</form>` element. Add the following code (shown with the gray screen background) to the end of the JSP file:

```
         </table>
       </td>
</tr>
</table>
</form>
</body>
</html>
```

4. Save the index.jsp file.
5. Open your browser and go to the address `http://localhost:8080/dvdapp` (or to whichever address you're using for the application). The value `8080` represents the port number. If your application server or Web server use a different port number, use that one instead of `8080`. Your browser should display a page similar to the one shown in the Figure 18-2.

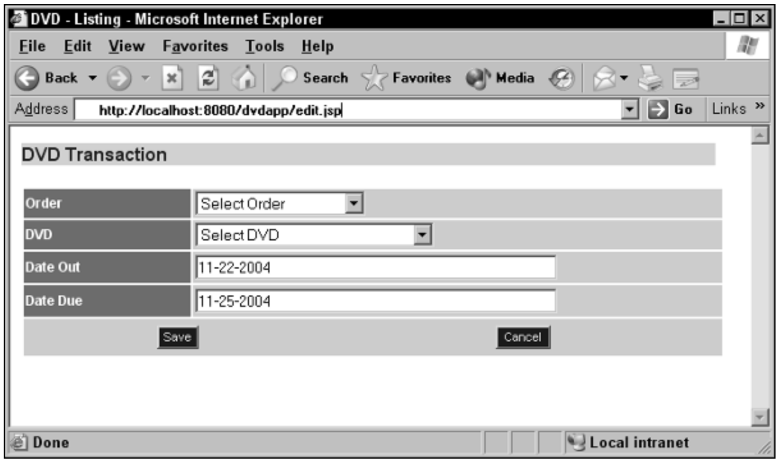

DVD - Listing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/dvdapp/
Go Links
New Transaction
Done
Local intranet

DVD Transaction Listing

Order Number	Customer	DVDName	DateOut	DateDue	DateIn	
13	Peter Taylor	A Room with a View	11-21-2004	11-24-2004		Edit
12	Ralph Frederick Johnson	Amadeus	11-21-2004	11-24-2004		Edit
11	Anne Thomas	Mash	11-21-2004	11-24-2004		Edit
11	Anne Thomas	The Rocky Horror Picture Show	11-21-2004	11-24-2004		Edit
11	Anne Thomas	What's Up, Doc?	11-21-2004	11-24-2004		Edit
10	Ginger Meagan Delaney	Amadeus	11-21-2004	11-24-2004		Edit
9	Mona J. Cavenaugh	A Room with a View	11-21-2004	11-24-2004		Edit

Figure 18-2

6. Click the New Transaction button at the top of the page. Your browser should display a page similar to the one shown in the Figure 18-3.


DVD - Listing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/dvdapp/edit.jsp Go Links
Save
Cancel
Done
Local intranet

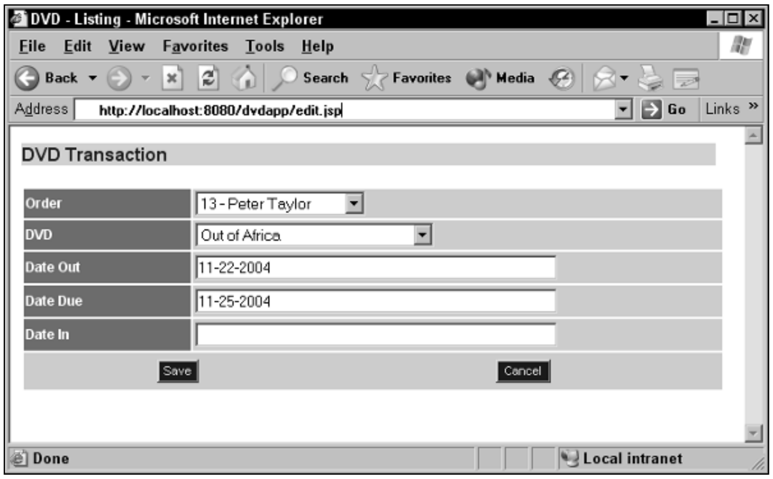
DVD Transaction

Order	Select Order
DVD	Select DVD
Date Out	11-22-2004
Date Due	11-25-2004

Figure 18-3

7. Now add a transaction to an existing order. In the Order drop-down list, select 13 - Peter Taylor. In the DVD drop-down list, select *Out of Africa*. Click Save. You're returned to the index.jsp page. The new transaction should now be displayed at the top of the list.

8. Now edit the new transaction. In the row of the transaction that you just added, click the Edit button. Your browser should display a page similar to the one shown in the Figure 18-4.


DVD - Listing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/dvdapp/edit.jsp Go Links
Save
Cancel
Done
Local intranet

DVD Transaction

Order	13 - Peter Taylor
DVD	Out of Africa
Date Out	11-22-2004
Date Due	11-25-2004
Date In	

Figure 18-4

9. In the Date In text box, type the same date that is in the Date Due text box. Be certain to type the date in the same format that is used for the other date-related text boxes. Click the Save button. You should be returned to the index.jsp page.

How It Works

In this exercise, you added a form to your index.jsp file. This is similar to the form that you added to the edit.jsp file. The main difference between the two is that, in this form, the `transaction_id` value is set to an empty string. This is because no ID is necessary initially, but you want the parameter to exist so that a vehicle has been provided to pass that ID through the form when you submit the form.

Once you created the form, you added the following HTML cell definition and `<input>` element at the top of the page:

```
<td align="right" width="219" class="title">
   <input type="button" value="New Transaction" class="add" onclick="doAdd(this)">
</td>
```

As you can see, the input type is `button` and it calls the JavaScript `doAdd()` function. The function links the user to the edit.jsp page, allowing the user to create a new transaction. At the same time, the function passes a `command` value of `add` to the edit.jsp page. That way, the edit.jsp page knows that a new transaction is being created and responds accordingly. You next added the following code to the initial table structure that is created in the HTML code:

```
<td width="99" class="heading">&nbsp;</td>
```

This creates an additional column head in the table to provide a column for the Edit button that will be added to each row returned by your query results. Finally, you added the actual cell and button to your table definition, as shown in the following code:

```
<td class="item" valign="center" align="center">
   <input type="button" value="Edit" class="edit" onclick="doEdit(this,
<%=transId%>)">
</td>
```

The Edit button calls the `doEdit()` function, which passes the transaction ID to the form and links the user to the edit.jsp page. At the same time, the function passes the `command` value of `edit` so that when the edit.jsp page opens, it has the information it needs to allow the user to edit a current record.

Once you modified and saved the file, you opened the index.jsp page, created a transaction, and then edited that transaction. However, the application still does not allow you to delete a transaction. As a result, the next section describes how you can set up Java statements to delete data.

Deleting Data from a MySQL Database

Deleting MySQL data from within your Java application is just like inserting and updating data. You must use the same Java statement elements. For example, suppose that you want to delete a CD listing from a table name CDs. The ID for the specific CD is store in the `cdId` variable. You can use the following statements to set up your application to delete the data:

```
String deleteSQL = "DELETE FROM CDs WHERE CDID = ?";
PreparedStatement preparedStatement = connection.prepareStatement(deleteSQL);
preparedStatement.setInt(1, cdId);
int rowCount = preparedStatement.executeUpdate();
```

As with inserting and updating data, you first assign the SQL statement to a variable (`deleteSql`). You then create a `PreparedStatement` object based on the `DELETE` statement and assign that object to a variable (`preparedStatement`). Finally you use a method in the `PreparedStatement` class to assign a value to the variable. In this case, you use the `setInt()` method to assign the value in the `cdId` variable to the placeholder in the `DELETE` statement. Finally, you use the `executeUpdate()` method in the `PreparedStatement` class to execute the `DELETE` statement and return a row count, which is assigned to the `rowCount` variable.

As you can see, deleting data is no more difficult than updating or inserting data. The key to any of these types of statements is to make sure that you set up your variables in such a way that the correct information can be passed to the SQL statement when it is being executed. In the next Try It Out section, you see how you can delete a transaction from your database. To do so, you modify the index.jsp file and then create a delete.jsp include file.

Try It Out Modifying the index.jsp File and Creating the delete.jsp File

The following steps describe how to set up delete capabilities in your DVDRentals application:

1. First, add a column head to your table so that you can include a Delete button for each row. The button will be added next to the Edit button you added in the previous Try It Out section. Add the following code (shown with the gray screen background) to your file:

```
<tr>
   <td width="250" class="heading">Order Number</td>
   <td width="250" class="heading">Customer</td>
   <td width="250" class="heading">DVDName</td>
   <td width="185" class="heading">DateOut</td>
   <td width="185" class="heading">DateDue</td>
   <td width="185" class="heading">DateIn</td>
   <td width="99" class="heading">&nbsp;</td>
   <td width="99" class="heading">&nbsp;</td>
</tr>
```

2. Next, add the code necessary to initialize variables and call the delete.jsp include file. Add the following code (shown with the gray screen background) to your file:

```
<%
// Declare and initialize variables with parameters retrieved from the form

   String command = request.getParameter("command");
   String transactionIdString = request.getParameter("transaction_id");

// Declare and initialize variables for database operations

   Connection connection;
   Statement stmt;

// wrap the database code in a try catch block to handle any database related
// errors. the Catch statements are at the bottom.

   try
   {
// Create the connection
// Get the connection from DriverManager, this technique for getting the connection
// is generally not recommended for "real" applications because the password is in
the file.

      Class.forName("com.mysql.jdbc.Driver").newInstance();

      connection = DriverManager.getConnection("jdbc:mysql://localhost/DVDRentals",
                                               "mysqlapp",
                                               "pw1");

// Process the delete command

      if(transactionIdString != null)
      {
         int transactionId = Integer.valueOf(transactionIdString).intValue();

         if("delete".equals(command))
         {
// Include the delete.jsp file
%>
            <%@ include file="delete.jsp" %>
<%
         }
      }
```

3. Now add the actual Delete button by adding the following code (shown with the gray-screen background) to your file:

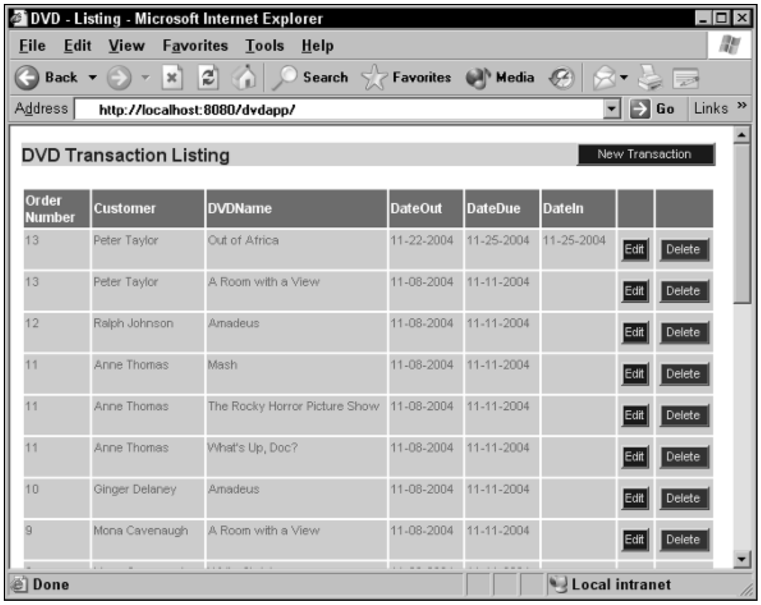
```
         <td class="item">
            <nobr>
            <%=dateInPrint%>
            </nobr>
         </td>
         <td class="item" valign="center" align="center">
            <input type="button" value="Edit" class="edit" onclick="doEdit(this,
<%=transId%>)">
         </td>
         <td class="item" valign="center" align="center">
            <input type="button" value="Delete" class="delete"
onclick="doDelete(this, <%=transId%>)">
         </td>
      </tr>
```

4. Save the index.jsp file.
5. Create a new file named delete.jsp in your text editor, and enter the following code:

```
<%
// Build the DELETE statement with a transactionId parameter reference
String deleteSQL = "DELETE FROM Transactions WHERE TransID = ?";

PreparedStatement preparedStatement = connection.prepareStatement(deleteSQL);

// Set the TransID parameter
preparedStatement.setInt(1, transactionId);

// Execute the DELETE statement
```

```
int rowCount = preparedStatement.executeUpdate();

preparedStatement.close();
%>
```

6. Save the delete.jsp file to the appropriate Web application directory.
7. Open your browser and go to the address `http://localhost:8080/dvdapp`. Your browser should display a page similar to the one shown in the Figure 18-5.

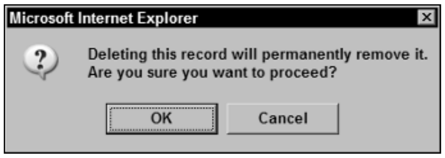

DVD - Listing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/dvdapp/ Go Links
New Transaction
Done
Local intranet

DVD Transaction Listing

Order Number	Customer	DVDName	DateOut	DateDue	DateIn		
13	Peter Taylor	Out of Africa	11-22-2004	11-25-2004	11-25-2004	Edit	Delete
13	Peter Taylor	A Room with a View	11-08-2004	11-11-2004		Edit	Delete
12	Ralph Johnson	Amadeus	11-08-2004	11-11-2004		Edit	Delete
11	Anne Thomas	Mash	11-08-2004	11-11-2004		Edit	Delete
11	Anne Thomas	The Rocky Horror Picture Show	11-08-2004	11-11-2004		Edit	Delete
11	Anne Thomas	What's Up, Doc?	11-08-2004	11-11-2004		Edit	Delete
10	Ginger Delaney	Amadeus	11-08-2004	11-11-2004		Edit	Delete
9	Mona Cavenaugh	A Room with a View	11-08-2004	11-11-2004		Edit	Delete

Figure 18-5

8. Click the Delete button in the row that contains the transaction that you created in a previous Try It Out section (Order number 13, DVD name *Out of Africa*). A message box similar to the one in Figure 18-6 appears, confirming whether you want to delete the record.



Deleting this record will permanently remove it.
Are you sure you want to proceed?

Figure 18-6

9. Click OK to delete the record. The index.jsp file should be redisplayed, with the deleted file no longer showing.

How It Works

In this exercise, you first created an additional column head for a column what will hold the Delete button for each row. You t.hen entered the following code:

```
String command = request.getParameter("command");
String transactionIdString = request.getParameter("transaction_id");
```

Both statements use the `getParameter()` method of the `request` object to retrieve parameter values from the form. The values are then assigned to the appropriate variables, which can then be used in your Java code just like any other variables. Next, you added the code necessary to include the delete.jsp file (which you create in a later step):

```
        if(transactionIdString != null)
        {
           int transactionId = Integer.valueOf(transactionIdString).intValue();

           if("delete".equals(command))
           {
%>
              <%@ include file="delete.jsp" %>
<%
           }
        }
```

The first `if` condition verifies that the `transactionIdString` variable contains a value. If the condition evaluates to true, the `if` block is executed. Next, you uses the `valueOf()` and `intValue()` methods of the `Integer` class to convert the `transactionIdString` value to an integer and assign it to the `transactionId` variable. The next `if` condition specifies that the command value must equal `delete` in order to proceed. If the condition evaluates to true, the delete.jsp file is included in the current file. This means that the Java statements in delete.jsp are executed as though they are actually part of the insert.jsp file.

The last code that you added to the index.jsp file is the HTML cell definition and `<input>` element necessary to add the Delete button to each row displayed on the page:

```
<td class="item" valign="center" align="center">
   <input type="button" value="Delete" class="delete" onclick="doDelete(this,
<%=transId%>)">
</td>
```

As you can see, the input type is button (`type="button"`), the button is named Delete (`value="Delete"`), the style is delete (`class="delete"`), and the `doDelete()` function is executed when the button is clicked. The `doDelete()` function takes two parameters. The `this` parameter merely indicates that it is the current button that is being referenced. The second parameter passes the value in the `transId` variable to the `transactionId` parameter associated with the form. That way, Java knows what record to delete when the `DELETE` statement is executed.

Now your application should be complete, at least this part of the application. You can view, insert, update, and delete transactions. In addition, you can build on this application if you want to extend your application's functionality. The code used to create this application is available online at `www.wrox.com`, and you can use and modify that code as necessary. Keep in mind, however, that at the heart of your application is the MySQL database that manages the data that you need to run your application. The better you understand MySQL and data management, the more effective your applications can be.

Summary

In this chapter, you learned how you can build a Java application that interacts with a MySQL database. The chapter included information about connecting to a MySQL server and database, retrieving data, and modifying data. Because Java is an object-oriented program, much of the discussion and examples in the chapter focused on how Java classes and their objects are used to interact with a MySQL database. You then used this information to create a JSP application that retrieved data from the DVDRentals database and modified that data. Specifically, the chapter covered the following topics:

- ❑ Specifying the Java classes to include in your application
- ❑ Connecting to the MySQL server and selecting a database
- ❑ Using `if` statements to test conditions and take actions
- ❑ Retrieving data, formatting data, and then displaying data
- ❑ Inserting data into your database
- ❑ Updating existing data in your database
- ❑ Deleting data from your database

The principles covered in this chapter can apply to any Java application that interacts with a MySQL database. The important point to remember with Java is that you are always working within the context of objects. These objects provide the structure for everything from establishing database connections to assigning string values to variables. Many of the concepts that you learned in this chapter can be applied to more extensive Java applications, including non-JSP applications. In all cases, the fundamentals of objects and connecting to a MySQL database are the same. And now that you have a foundation in how Java interacts with MySQL, you're ready to move on to the next chapter, which describes how to create a C# ASP.NET application that accesses a MySQL database.

Exercises

In this chapter, you learned how to connect to a MySQL database, retrieve data, and manipulate data from within a Java application. To assist you in better understanding how to perform these tasks, the chapter includes the following exercises. To view solutions to these exercises, see Appendix A.

1. You are creating a JSP file. You plan to use all classes in the java.sql package. What statement should you use to import those classes into your file?
2. You want to create a variable named `strBook` to hold a book title value. You plan to create a `String` object and assign it to the variable. The object will contain the string value *The Open Space of Democracy*, which is the title of the book. What statement should you use to declare and initialize the parameter?
3. You are establishing a connection to a MySQL database from with your JSP application. You plan to connect to the books database using the usr1 user account, which requires the pw1 password. The MySQL server resides on the same computer that the application will run. You have already declared the `conn` variable based on the `Connection` class. What Java statements should you use to establish the connection?

4. You are retrieving data from a MySQL database. You create a `SELECT` statement and assign it to the `selectSql` variable. You then create a `Statement` object and assign it to the `stmt` variable. You now want to execute the statement. You also want to assign the result set to a variable named `rs`. What statement should you use to execute the `SELECT` statement and initialize the `rs` variable?
5. Your application includes the Java code necessary to retrieve data from a MySQL database into a `ResultSet` object. You have assigned that object to a variable named `rs`. The result set includes data from the CDName column and the InStock column of the CDs table. You now want to display each row returned by the query. You plan to use the `System.out.println()` method to print out the values in the column. What java elements and statements should you to retrieve and display the data in the result set?
6. You want to convert a numerical string value to an integer and assign the value to a primitive `int` variable. The string value is 530, and the name of the variable is `intValue`. What Java statement should you use to convert the string value?
7. You are adding code to your Java application to insert data into a MySQL database. You create an `INSERT` statement and assign it to the `insertSql` variable. You also declare a variable named `conn`, which is based on the `Connection` class. You now want to create a `PreparedStatement` object based on the `INSERT` statement and then assign the new object to the `ps` variable. What Java statement should you use?