20

How to use JDBC to work with databases

This chapter shows how to use Java to work with a database. In particular, it shows the JDBC skills needed to create the classes for the database tier of the Product Manager application. Then, the next two chapters show how to create the classes for the presentation tier.

How to work with JDBC

To write Java code that works with a database, you can use *JDBC*, which is sometimes referred to as *Java Database Connectivity*. The core JDBC API is stored in the java.sql package, which comes as part of Java SE.

An introduction to database drivers

Before you can connect to a database, you must make a *database driver* available to your application. Figure 20-1 lists the four types of JDBC database drivers that you can use. Then, it shows how to download a database driver and make it available to your application. For most applications, you'll want to use a type-4 driver. The other three driver types are generally considered less preferable, with the type-1 driver being the least preferable. As a result, you'll only want to use these drivers if a type-4 driver isn't available for the database that you're using.

For a MySQL database, you can use the type-4 driver named Connector/J that's available for free from the MySQL website. For other types of databases, you can usually download a type-4 driver from the website for that database. The documentation for these drivers typically shows how to install and configure the driver.

To use the database driver in an application, you can add the JAR file that contains the database driver to the application's classpath. The easiest way to do that is to use your IDE to add the JAR file for the database driver to your application. To add a MySQL driver to a NetBeans project, you can right-click on the project's Library folder, select the Add Library command, and use the resulting dialog box to select the MySQL JDBC Driver library.

The four types of JDBC database drivers

Type 1	A *JDBC-ODBC bridge driver* converts JDBC calls into ODBC calls that access the DBMS protocol. For this data access method, an ODBC driver must be installed on the client machine. A JDBC-ODBC bridge driver was included as part of the JDK prior to Java 8 but is not included or supported with Java 8 and later.
Type 2	A *native protocol partly Java driver* converts JDBC calls into the native DBMS protocol. Since this conversion takes place on the client, the database client library must be installed on the client machine.
Type 3	A *net protocol all Java driver* converts JDBC calls into a net protocol that's independent of any native DBMS protocol. Then, middleware software running on a server converts the net protocol to the native DBMS protocol. Since this conversion takes place on the server side, the database client library isn't required on the client machine.
Type 4	A *native protocol all Java driver* converts JDBC calls into the native DBMS protocol. Since this conversion takes place on the server, the database client library isn't required on the client machine.

How to download a database driver

- For MySQL databases, you can download a JDBC driver named Connector/J from the MySQL website. This driver is an open-source, type-4 driver that's available for free.
- For other databases, you can usually download a type-4 JDBC driver from the database's website.
- The Connector/J driver for MySQL databases is included with NetBeans. As a result, if you're using NetBeans, you don't need to download this driver.

How to make a database driver available to an application

- Before you can use a database driver, you must make it available to your application. The easiest way to do this is to use your IDE to add the JAR file for the driver to your application.
- To add the MySQL JDBC driver to a NetBeans project, right-click on the Libraries folder, select the Add Library command, and use the resulting dialog box to select the MySQL JDBC Driver library.
- To add any JDBC driver to a NetBeans project, right-click on the Libraries folder, select the Add JAR/Folder command, and use the resulting dialog box to select the JAR file for the driver.

Figure 20-1 An introduction to database drivers

How to connect to a database

Before you can access or modify the data in a database, you must connect to the database as shown in figure 20-2. To start, this figure shows the syntax for a database URL. You can use this syntax within the code that gets the connection.

The first example shows how to use a type-4 MySQL driver to get a connection to a database. To start, you use the getConnection method of the DriverManager class to return a Connection object. This method requires three arguments: the URL for the database, a username, and a password. In the first example, the URL consists of the API (jdbc), the subprotocol for MySQL drivers (mysql), the host machine (localhost), the port for the database service (3306), and the name of the database (mma).

For security reasons, it's considered a best practice to connect to the database with a username that only has the privileges that the application needs. That's why the script that creates the mma database creates a user named mma_user that only has limited privileges to work with the data of the mma database, not to modify its structure. As a result, the user named mma_user is appropriate for the sample applications presented in this book.

The second example shows how to connect to an Oracle database. Here again, you provide the URL for the database, a username, and a password. This is true no matter what type of database you're using with JDBC.

In practice, connecting to the database is often frustrating because it's hard to figure out what the URL, username, and password need to be. So if your colleagues have already made a connection to the database that you need to use, you can start by asking them for this information.

Since the getConnection method of the DriverManager class throws a SQLException, you need to handle this exception whenever you connect to a database. With JDBC 4.0 (Java 6) and later, you can use an enhanced for statement to loop through any exceptions that are nested within the SQLException object. In this figure, the catch block in the first example loops through all the exceptions that are nested in the SQLException object.

To do that, this loop retrieves a Throwable object named t for each nested exception. Then, it prints the exception to the console. This works because the Throwable class is the superclass for all exceptions, and a Throwable object is returned by the iterator for the SQLException class.

The first two examples present a new feature of JDBC 4.0 called *automatic driver loading*. This feature loads the database driver automatically based on the URL for the database.

If you're working with an older version of Java, though, you need to use the forName method of the Class class to explicitly load the driver before you call the getConnection method as shown by the third example. Since this method throws a ClassNotFoundException, you also have to handle this exception.

Even with JDBC 4.0 and later, you might get a message that says, "No suitable driver found." In that case, you can use the forName method of the Class class to explicitly load the driver. However, if automatic driver loading works, it usually makes sense to remove this method call from your code. That way, you can connect to the database with less code, and you don't have to hard code the name of the database driver.

Database URL syntax

```
jdbc:subprotocolName:databaseURL
```

How to connect to a MySQL database with automatic driver loading

```
try {
    // set the db url, username, and password
    String dbURL = "jdbc:mysql://localhost:3306/mma";
    String username = "mma_user";
    String password = "sesame";

    // get connection
    Connection connection =
            DriverManager.getConnection(dbURL, username, password);
} catch(SQLException e) {
    for (Throwable t : e) {
        System.out.println(t);
    }
}
```

How to connect to an Oracle database with automatic driver loading

```
    Connection connection = DriverManager.getConnection(
        "jdbc:oracle:thin@localhost/mma", "mma_user", "sesame");
```

How to load a MySQL database driver prior to JDBC 4.0

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch(ClassNotFoundException e) {
    System.out.println(e);
}
```

Description

- Before you can get or modify the data in a database, you need to connect to it. To do that, you use the getConnection method of the DriverManager class to return a Connection object.
- When you use the getConnection method of the DriverManager class, you must supply a URL for the database. In addition, you usually supply a username and a password, though you might not for an embedded database. This method throws a SQLException.
- With JDBC 4.0 and later, the SQLException class implements the Iterable interface. As a result, you can use an enhanced for statement to loop through any nested exceptions.
- With JDBC 4.0 and later, the database driver is loaded automatically. This feature is known as *automatic driver loading*.
- Prior to JDBC 4.0, you needed to use the forName method of the Class class to load the driver. This method throws a ClassNotFoundException.
- Although the connection string for each driver is different, the documentation for the driver should explain how to write a connection string for that driver.
- Typically, you only need to connect to one database for an application. However, it's possible to load multiple database drivers and establish connections to multiple types of databases.

Figure 20-2 How to connect to a database

How to return a result set and move the cursor through it

Once you connect to a database, you're ready to retrieve data from it as shown in figure 20-3. Here, the first two examples show how to use Statement objects to create a *result set*, or *result table*. Then, the next two examples show how to move the *row pointer*, or *cursor*, through the result set.

Both of the result sets in this figure are read-only, forward-only result sets. This means that you can only move the cursor forward through the result set, and that you can read but not write rows in the result set. Although JDBC supports other types of scrollable, updateable result sets, these features require some additional overhead, and they aren't necessary for most applications.

The first example calls the createStatement method from a Connection object to return a Statement object. Then, it calls the executeQuery method from the Statement object to execute a SELECT statement that's coded as a string. This returns a ResultSet object that contains the result set for the SELECT statement. In this case, the SELECT statement only retrieves a single column from a single row (the product code for a specific product ID) so that's what the ResultSet object contains. You can use this object to check whether a row exists.

The second example works like the first example. However, it returns all of the rows and columns for the Product table and puts this result set in a ResultSet object named products. You can use this object to display all products.

The third example shows how to use the next method of the ResultSet object to move the cursor to the first row of the result set that's created by the first example. When you create a result set, the cursor is positioned before the first row in the result set. As a result, the first use of the next method attempts to move the cursor to the first row in the result set. If the row exists, the cursor moves to that row and the next method returns a true value. Otherwise, the next method returns a false value. In the next figure, you'll learn how to retrieve values from the row that the cursor is on.

The fourth example shows how to use the next method to loop through all of the rows in the result set that's created in the second example. Here, the while loop calls the next method. Then, if the next row is a valid row, the next method moves the cursor to the row and returns a true value. As a result, the code within the while loop is executed. Otherwise, the next method returns a false value and the code within the while loop isn't executed.

Since all of the methods described in this figure throw a SQLException, you either need to throw or catch this exception when you're working with these methods. The ProductDB class presented later in this chapter shows how this works.

Although there are other ResultSet methods, the one you'll use the most with a forward-only, read-only result set is the next method. However, this figure summarizes two other methods (last and close) that you may occasionally want to use for this type of result set.

How to create a result set that contains 1 row and 1 column

```
Statement statement = connection.createStatement();
ResultSet product = statement.executeQuery(
    "SELECT ProductCode FROM Product " +
    "WHERE ProductID = '1'");
```

How to create a result set that contains multiple columns and rows

```
Statement statement = connection.createStatement();
ResultSet products = statement.executeQuery(
    "SELECT * FROM Product ");
```

How to move the cursor to the first row in the result set

```
boolean productExists = product.next();
```

How to loop through a result set

```
while (products.next()) {
    // statements that process each row
}
```

ResultSet methods for forward-only, read-only result sets

Method	Description
`next()`	Moves the cursor to the next row in the result set.
`last()`	Moves the cursor to the last row in the result set.
`close()`	Releases the result set's resources.

Description

- To return a *result set*, you use the createStatement method of a Connection object to create a Statement object. Then, you use the executeQuery method of the Statement object to execute a SELECT statement that returns a ResultSet object.
- By default, the createStatement method creates a forward-only, read-only result set. This means that you can only move the *cursor* through it from the first row to the last and that you can't update it. Although you can pass arguments to the createStatement method that create other types of result sets, the default is appropriate for most applications.
- When a result set is created, the cursor is positioned before the first row. Then, you can use the methods of the ResultSet object to move the cursor. To move the cursor to the next row, for example, you call the next method. If the row is valid, this method moves the cursor to the next row and returns a true value. Otherwise, it returns a false value.
- The createStatement, executeQuery, and next methods throw a SQLException. As a result, any code that uses these methods needs to catch or throw this exception.

Figure 20-3 How to return a result set and move the cursor through it

How to get data from a result set

When the cursor is positioned on the row that you want to get data from, you can use the methods in figure 20-4 to get that data. The examples show how to use the getInt, getString, and getDouble methods of the ResultSet object to return int, String, and double values. However, you can use similar get methods to return other types of data.

The methods in this figure show the two types of arguments accepted by the get methods. The first method accepts an int value that specifies the index number of the column in the result set, where 1 is the first column, 2 is the second column, and so on. The second method accepts a String object that specifies the name of the column in the result set. Although the get methods with column indexes require less typing, the get methods with column names lead to code that's easier to read and understand.

The first example shows how to use column indexes to return data from a result set named products. Here, the first statement uses the getInt method to return the ID for the product, the next two statements use the getString method to return the code and description, and the third statement uses the getDouble method to return the list price. Since these methods use the column index, the first column in the result set must contain the product ID, the second column must contain the product code, and so on.

The second example shows how to use column names to return data from the products result set. Since this code uses the column names, the order of the columns in the result set doesn't matter. However, the column names must exist in the result set or a SQLException is thrown that indicates that a column wasn't found.

The third example shows how you can use the get methods to create a Product object. Here, the first statement creates a new Product object. Then, the next four statements store the data that was retrieved from the ResultSet object in the Product object. Since objects are often created from data that's stored in a database, code like this is commonly used when you use the JDBC API. However, this code is handled automatically by some other database APIs such as JPA (Java Persistence API).

If you look up the ResultSet interface in the documentation for the JDBC API, you'll see that get methods exist for all of the primitive types as well as for other types of data. For example, get methods exist for the Date, Time, and Timestamp classes that are a part of the java.sql package. In addition, they exist for *BLOB objects* (*Binary Large Objects*) and *CLOB objects* (*Character Large Objects*). You can use these types of objects to store large objects such as images, audio, and video in databases.

Methods of a ResultSet object that return data from a result set

Method	Description
`getXXX(columnIndex)`	Returns data from the specified column number.
`getXXX(columnName)`	Returns data from the specified column name.

Code that uses indexes to return columns from the products result set

```
int productID = products.getInt(1);
String code = products.getString(2);
String description = products.getString(3);
double price = products.getDouble(4);
```

Code that uses names to return the same columns

```
int productID = products.getInt("ProductID");
String code = products.getString("Code");
String description = products.getString("Description");
double price = products.getDouble("ListPrice");
```

Code that creates a Product object from the products result set

```
Product p = new Product();
p.setId(productID);
p.setCode(code);
p.setDescription(description);
p.setPrice(price);
```

Description

- The getXXX methods can be used to return all eight primitive types. For example, the getInt method returns the int type and the getLong method returns the long type.
- The getXXX methods can also be used to return strings, dates, and times. For example, the getString method returns an object of the String class, and the getDate, getTime, and getTimestamp methods return objects of the Date, Time, and Timestamp classes of the java.sql package.

Figure 20-4 How to retrieve data from a result set

How to insert, update, and delete data

Figure 20-5 shows how to use JDBC to modify the data in a database. To do that, you use the executeUpdate method of a Statement object to execute SQL statements that add, update, and delete data.

When you work with the executeUpdate method, you just pass a SQL statement to the database. In these examples, the code adds, updates, and deletes a product in the Product table. To do that, the code combines data from a Product object with the appropriate SQL statement. For the UPDATE and DELETE statements, the SQL statement uses the product's code in the WHERE clause to select a single product.

Unfortunately, if you build a SQL statement from user input and use a method of the Statement object to execute that SQL statement, you are susceptible to a security vulnerability known as a SQL injection attack. A *SQL injection attack* allows a hacker to execute SQL statements against your database to read sensitive data or to delete or modify data. For the Product Manager application, for instance, the user might be able to execute a DROP TABLE statement by entering the following code:

```
test'); DROP TABLE Product; --
```

Here, the first semicolon ends the first SQL statement. Then, the database might execute the second SQL statement. To prevent most types of SQL injection attacks, you can use a prepared statement as described in the next figure.

How to use the executeUpdate method to modify data

How to add a row

```
String query =
   "INSERT INTO Product (ProductCode, ProductDescription, ProductPrice) " +
   "VALUES ('" + product.getCode() + "', " +
           "'" + product.getDescription() + "', " +
           "'" + product.getPrice() + "')";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

How to update a row

```
String query = "UPDATE Product SET " +
   "ProductCode = '" + product.getCode() + "', " +
   "ProductDescription = '" + product.getDescription() + "', " +
   "ProductPrice = '" + product.getPrice() + "' " +
   "WHERE ProductCode = '" + product.getCode() + "'";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

How to delete a row

```
String query = "DELETE FROM Product " +
               "WHERE ProductCode = '" + productCode + "'";
Statement statement = connection.createStatement();
int rowCount = statement.executeUpdate(query);
```

Description

- The executeUpdate method is an older method that works with most JDBC drivers. Although there are some newer methods that require less SQL code, they may not work properly with all JDBC drivers.
- The executeUpdate method returns an int value that identifies the number of rows that were affected by the SQL statement.

Warning

- If you build a SQL statement from user input and use a method of the Statement object to execute that SQL statement, you may be susceptible to a security vulnerability known as a SQL injection attack.
- A *SQL injection attack* allows a hacker to bypass authentication or to execute SQL statements against your database that can read sensitive data, modify data, or delete data.
- To prevent most types of SQL injection attacks, you can use prepared statements as described in the next figure.

Figure 20-5 How to insert, update, and delete data

How to work with prepared statements

Each time a Java application sends a new SQL statement to the database server, the server checks the statement for syntax errors, prepares a plan for executing the statement, and executes the statement. If the same statement is sent again, though, the database server checks to see whether it has already received one exactly like it. If so, the server doesn't have to check its syntax and prepare an execution plan for it so the server just executes it. This improves the performance of the database operations.

To take advantage of this database feature, Java provides for the use of *prepared statements* as shown in figure 20-6. This feature lets you send statements to the database server that get executed repeatedly by accepting the parameter values that are sent to it. That improves the database performance because the database server only has to check the syntax and prepare the execution plan once for each statement.

In addition, prepared statements automatically check their parameter values to prevent most types of SQL injection attacks. As a result, it's generally considered a best practice to use prepared statements whenever possible.

The first example uses a prepared statement to create a result set that contains a single product. Here, the first statement uses a question mark (?) to identify the parameter for the SELECT statement, which is the product code for the book, and the second statement uses the prepareStatement method of the Connection object to return a PreparedStatement object. Then, the third statement uses a set method (the setString method) of the PreparedStatement object to set a value for the parameter, and the fourth statement uses the executeQuery method of the PreparedStatement object to return a ResultSet object.

The second example shows how to use a prepared statement to execute an UPDATE query that requires four parameters. Here, the first statement uses four question marks to identify the four parameters of the UPDATE statement, and the second statement creates the PreparedStatement object. Then, the next four statements use set methods to set the four parameters in the order that they appear in the UPDATE statement. The last statement uses the executeUpdate method of the PreparedStatement object to execute the UPDATE statement.

The third and fourth examples show how to insert and delete rows with prepared statements. This works similarly to the second example.

In this figure, the type of SQL statement that you're using determines whether you use the executeQuery method or the executeUpdate method. If you're using a SELECT statement to return a result set, you use the executeQuery method. However, if you're using an INSERT, UPDATE, or DELETE statement, you use the executeUpdate method. In other words, this works the same for a PreparedStatement object as it does for a Statement object.

How to use a prepared statement

To return a result set

```
String sql = "SELECT ProductCode, ProductDescription, ProductPrice "
           + "FROM Product WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, productCode);
ResultSet product = ps.executeQuery();
```

To modify a row

```
String sql = "UPDATE Product SET "
           + "   ProductCode = ?, "
           + "   ProductDescription = ?, "
           + "   ProductPrice = ?"
           + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, product.getCode());
ps.setString(2, product.getDescription());
ps.setDouble(3, product.getPrice());
ps.setString(4, product.getCode());
ps.executeUpdate();
```

To insert a row

```
String sql =
      "INSERT INTO Product (ProductCode, ProductDescription, ProductPrice) "
    + "VALUES (?, ?, ?)";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, product.getCode());
ps.setString(2, product.getDescription());
ps.setDouble(3, product.getPrice());
ps.executeUpdate();
```

To delete a row

```
String sql = "DELETE FROM Product "
           + "WHERE ProductCode = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, product.getCode());
ps.executeUpdate();
```

Description

- When you use *prepared statements* in your Java applications, the database server only has to check the syntax and prepare an execution plan once for each SQL statement. This improves the efficiency of the database operations. In addition, it prevents most types of SQL injection attacks.
- To specify a parameter for a prepared statement, type a question mark (?) in the SQL statement.
- To supply values for the parameters in a prepared statement, use the set methods of the PreparedStatement interface. For a complete list of set methods, look up the PreparedStatement interface of the java.sql package in the documentation for the Java API.
- To execute a SELECT statement, use the executeQuery method. To execute an INSERT , UPDATE, or DELETE statement, use the executeUpdate method.

Figure 20-6 How to work with prepared statements

Two classes for working with databases

Now that you know how to use JDBC to work with a database, you're ready to learn how to code a class that allows you to map an object to a table in a database. In particular, you're ready to learn how to map the Product object to the Product table in the mma database. But first, you'll be introduced to a utility class that you can use to get a connection to a database.

The DBUtil class

Figure 20-7 shows a class named DBUtil. This class begins by declaring a static variable for the Connection object. Then, it provides a private constructor that prevents other classes from creating an object from the DBUtil class. That's a good practice since the DBUtil class only provides two static methods.

The first static method, the getConnection method, returns a Connection object that provides a connection to the database. This method begins by checking whether the connection has already been opened. If so, it returns that connection. As a result, this method only opens one connection to the database. This helps the application run efficiently after it opens the first connection to the database.

However, if the connection hasn't been opened, this code opens a connection by creating a Connection object. To do that, this code specifies the URL, username, and password for the database. Here, the code connects to the mma database as the user named mma_user. This works as described earlier in this chapter.

After specifying the URL, username, and password for the database, the getConnection method uses the static getConnection method of the DriverManager class to automatically load the appropriate database driver and connect to the database. If this method is successful, this method returns the Connection object to the calling method. Otherwise, this method throws a SQLException. Then, the catch block catches this exception, wraps it in the custom DBException class presented in chapter 16, and throws that exception.

The second static method, the closeConnection method, closes the one connection that's opened by this class. Typically, you call this method when the application exits. In other words, you close the connection to the database when you close the application. This prevents resource leaks.

Both of the static methods include the synchronized keyword. As a result, each method must finish before you can call another method, which is what you want. For example, if you call the getConnection method, you want it to finish executing before the close method can begin executing.

The DBUtil class

```
package murach.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBUtil {

    private static Connection connection;

    private DBUtil() {}

    public static synchronized Connection getConnection() throws DBException {
        if (connection != null) {
            return connection;
        }
        else {
            try {
                // set the db url, username, and password
                String url = "jdbc:mysql://localhost:3306/mma";
                String username = "mma_user";
                String password = "sesame";

                // get and return connection
                connection = DriverManager.getConnection(
                        url, username, password);
                return connection;
            } catch (SQLException e) {
                throw new DBException(e);
            }
        }
    }

    public static synchronized void closeConnection() throws DBException {
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                throw new DBException(e);
            } finally {
                connection = null;
            }
        }
    }
}
```

Description

- To make it easier to get and close a connection to a database, you can use a utility class like the one shown in this figure.

Figure 20-7 The DBUtil class

The ProductDB class

Figure 20-8 presents the complete code for the ProductDB class. This class maps the Product object to the Product table of the mma database.

The getAll method returns a List object that contains all of the Product objects that are stored in the Product table of the mma database. Here, the first statement creates a string that contains a SQL statement that selects all columns from the Product table and sorts them in ascending order by the ProductID column. The second statement creates an ArrayList object that can store Product objects. And the third statement uses the static getConnection method of the DBUtil class to get a connection to the database.

After getting the connection, this code uses a try-with-resources statement to create the PreparedStatement and ResultSet objects that are needed by this method. That way, these objects are automatically closed when the try block ends.

Once the ResultSet object has been created, the getAll method uses a loop to read each row in the result set. Within the loop, the first statement uses the getInt method of the ResultSet object to get the int value for the ProductID column of the result set. The next two statements use the getString method to return strings for the Code and Description columns. And the fourth statement uses the getDouble method to return a double value for the ListPrice column. Then, this loop creates the Product object, stores the data in it, and adds it to the products array list.

If this code executes successfully, the getAll method returns the products array list. However, if a SQLException is thrown anywhere in the try block, the catch block creates a new DBException that stores the SQLException and throws this exception to the calling method. That way, the calling method can throw or handle this exception. Ultimately, this should lead to the exception being handled at a level that's appropriate for the application, often by having the user interface display an appropriate user-friendly message to the user.

The ProductDB class — Page 1

```
package murach.db;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import murach.business.Product;

public class ProductDB {

    public static List<Product> getAll() throws DBException {
        String sql = "SELECT * FROM Product ORDER BY ProductID";
        List<Product> products = new ArrayList<>();
        Connection connection = DBUtil.getConnection();
        try (PreparedStatement ps = connection.prepareStatement(sql);
             ResultSet rs = ps.executeQuery()) {
            while (rs.next()) {
                int productID = rs.getInt("ProductID");
                String code = rs.getString("Code");
                String name  = rs.getString("Description");
                double price = rs.getDouble("ListPrice");

                Product p = new Product();
                p.setId(productID);
                p.setCode(code);
                p.setDescription(name);
                p.setPrice(price);
                products.add(p);
            }
            return products;
        } catch (SQLException e) {
            throw new DBException(e);
        }
    }
```

Figure 20-8 The ProductDB class (part 1 of 3)

The get method returns a Product object for a product that matches the specified product code. To do that, it uses a prepared SQL statement to return a result set. Then, it calls the next method of the result set to attempt to move the cursor to the first row in the result set. If successful, this method continues by reading the columns from the row, closing the result set, creating a Product object from this data, and returning the Product object.

Unlike the getAll method, the result set used by the get method can't be created in the try-with-resources statement. That's because before the result set can be opened, the value of the parameter in the prepared statement must be set. However, the prepared statement can still be created in the try-with-resources statement, which means that it doesn't have to be closed explicitly.

If no product row contains a product code that matches the specified code, this method closes the result set and returns a null to indicate that the product couldn't be found. In addition, if a SQLException is thrown anywhere in this method, this method creates a new exception from the DBException class and throws this exception to the calling method. That way, the method that calls the get method can throw or handle this exception.

As you review the get method, note that if the try block throws a SQLException, the result set isn't explicitly closed. In most cases, that's not a problem because the result set is automatically closed when the prepared statement that was used to create the result set is closed. If you wanted to close the result set explicitly, though, you could do that by adding a finally clause to the try statement.

The add method begins by creating a prepared statement that can be used to insert values into three columns of the Product table. Then, it sets the values of the three parameters in the prepared statement to the values stored in the Product object that's passed to it. Finally, it calls the executeUpdate method of the prepared statement. If a SQLException is thrown anywhere in this method, this code stores that exception in a DBException and throws it to the calling method.

The ProductDB class — Page 2

```
public static Product get(String productCode) throws DBException {
    String sql = "SELECT * FROM Product WHERE Code = ?";
    Connection connection = DBUtil.getConnection();
    try (PreparedStatement ps = connection.prepareStatement(sql)) {
        ps.setString(1, productCode);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            long productId = rs.getLong("ProductID");
            String name = rs.getString("Description");
            double price = rs.getDouble("ListPrice");
            rs.close();

            Product p = new Product();
            p.setId(productId);
            p.setCode(productCode);
            p.setDescription(name);
            p.setPrice(price);

            return p;
        } else {
            rs.close();
            return null;
        }
    } catch (SQLException e) {
        throw new DBException(e);
    }
}

public static void add(Product product) throws DBException {
    String sql
            = "INSERT INTO Product (Code, Description, ListPrice) "
            + "VALUES (?, ?, ?)";
    Connection connection = DBUtil.getConnection();
    try (PreparedStatement ps = connection.prepareStatement(sql)) {
        ps.setString(1, product.getCode());
        ps.setString(2, product.getDescription());
        ps.setDouble(3, product.getPrice());
        ps.executeUpdate();
    } catch (SQLException e) {
        throw new DBException(e);
    }
}
```

Figure 20-8 The database class for the Email List application (part 2 of 3)

The update method uses a prepared SQL statement to update an existing product in the Product table with the data that's stored in the Product object that's passed to it. Note that this method only works if the Product object has a product ID that exists in the Product table. Like the add method, this method throws a DBException if it isn't able to execute successfully.

The deleteProduct method uses a prepared SQL statement to delete the product row that has the same product code as the Product object that's passed to it. Like the add and update methods, the delete method throws a DBException if it isn't able to execute successfully.

The code for this class only uses methods from JDBC 1.0. That's because this is still the most common way to use JDBC to work with databases. The disadvantage of this technique is that you must understand SQL. However, SQL is easy to learn, and most programmers who work with databases already know how to use it. In fact, some programmers prefer using SQL so they have direct control over the SQL statement that's sent to the database.

Each method in this class that needs a connection uses the DBUtil class to get a connection. As a result, the first method call opens a connection to the database. Then, subsequent method calls use this connection until it is closed. Typically, the application closes this connection when the user exits. Since opening a database connection can be a relatively time-consuming process, this works much more efficiently than opening and closing a connection for each method call.

The ProductDB class Page 3

```
    public static void update(Product product) throws DBException {
        String sql = "UPDATE Product SET "
                + "Code = ?, "
                + "Description = ?, "
                + "ListPrice = ? "
                + "WHERE ProductID = ?";
        Connection connection = DBUtil.getConnection();
        try (PreparedStatement ps = connection.prepareStatement(sql)) {
            ps.setString(1, product.getCode());
            ps.setString(2, product.getDescription());
            ps.setDouble(3, product.getPrice());
            ps.setLong(4, product.getId());
            ps.executeUpdate();
        } catch (SQLException e) {
            throw new DBException(e);
        }
    }

    public static void delete(Product product) throws DBException {
        String sql = "DELETE FROM Product "
                   + "WHERE ProductID = ?";
        Connection connection = DBUtil.getConnection();
        try (PreparedStatement ps = connection.prepareStatement(sql)) {
            ps.setLong(1, product.getId());
            ps.executeUpdate();
        } catch (SQLException e) {
            throw new DBException(e);
        }
    }
}
```

Figure 20-8 The database class for the Email List application (part 3 of 3)

Code that uses the ProductDB class

Figure 20-9 shows some code that uses the ProductDB class. You can use code like this in the user interface for a console application.

The ProductDB, DBException, and DBUtil classes are stored in the mma.db package. If you're calling this code from a different package, you need to include import statements for these classes as shown in the first example. In addition, you need to include an import statement for the Product class that's stored in the mma.business package.

The second example shows how to get all products. To do that, you begin by declaring a List<Product> object to store the products. Then, you call the static getAll method from the ProductDB class. Since this code throws a DBException, you need to handle this exception. In this case, the code handles the exception by printing some information to the console. To start, it prints a user-friendly message to the console. This should give the user a good idea of what went wrong. Then, it prints the exception to the console. This should provide more detailed technical information that may help a programmer or administrator resolve the problem.

The third example shows how to get a single product. This works much like the first example. However, it uses the static get method of the ProductDB class to get a Product object for the specified code. This code assumes that the variable named code stores a product code.

The fourth example shows how to add a product. This works much like the previous two examples. However, it uses the static add method of the ProductDB class to add a Product object to the database. If successful, this code prints a message to the console to indicate that it worked. This code assumes that the variable named product stores a Product object.

The fifth example shows how to delete a product. This works much like the fourth example. However, it uses the ProductDB class to delete rows in the database. Like the fourth example, this example assumes that the product variable stores a Product object.

The sixth example shows how to close a connection. To do that, this code calls the closeConnection method of the DBUtil class. Then, it handles the DBException that's thrown by this method by printing some information to the console. Typically, you execute code like this when your application exits.

How to import the business and database classes

```
import mma.business.Product;
import mma.db.ProductDB;
import mma.db.DBException;
import mma.db.DBUtil;
```

How to get all products

```
List<Product> products;
try {
    products = ProductDB.getAll();
} catch (DBException e) {
    System.out.println("Error! Unable to get products");
    System.out.println(e + "\n");
}
```

How to get a product with the specified product code

```
Product product;
try {
    product = ProductDB.get(code);
} catch (DBException e) {
    System.out.println("Error! Unable to get product for code: " + code);
    System.out.println(e + "\n");
}
```

How to add a product

```
try {
    ProductDB.add(product);
    System.out.println(product.getDescription() + " was added.\n");
} catch (DBException e) {
    System.out.println("Error! Unable to add product.");
    System.out.println(e + "\n");
}
```

How to delete a product

```
try {
    ProductDB.delete(product);
    System.out.println(product.getDescription() + " was deleted.\n");
} catch (DBException e) {
    System.out.println("Error! Unable to delete product.");
    System.out.println(e + "\n");
}
```

How to close a connection

```
try {
    DBUtil.closeConnection();
} catch (DBException e) {
    Console.display("Error! Unable to close connection.");
    Console.display(e + "\n");
}
```

Description

- You can use the static methods of the ProductDB class to map Product objects to the rows of the Product table.
- You can use the DBUtil class to close the database connection when the application exits.

Figure 20-9 Code that uses the ProductDB class

Perspective

Now that you've finished this chapter, you should understand how to use JDBC to store data in a database and to retrieve data from a database. Although there's much more to learn about working with databases, those are the essential skills. To enhance your database skills, you can learn more about database management systems like MySQL or Oracle and the SQL that works with them.

You should also understand that the JDBC API requires the developer to write a significant amount of low-level code. This provides a good conceptual background if you are new to Java and databases. In addition, many legacy applications use JDBC. As a result, you may need to use it when working on old applications.

However, for new development, you might prefer to use another data access API such as JPA (Java Persistence API). This API handles much of the object to database mapping automatically. As a result, it doesn't require the developer to write as much code, and the code that the developer does need to write is often easier to maintain.

Summary

- With JDBC 4.0 (Java 6) and later, the *database driver* that's used to connect the application to a database is loaded automatically. This is known as *automatic driver loading*.
- With JDBC 4.0 and later, you can loop through any exceptions that are nested within the SQLException object.
- A Java application can use one of four driver types to access a database. It's considered a best practice to use a type-4 driver if one is available for that database.
- You can use JDBC to execute SQL statements that select, add, update, or delete one or more rows in a database. You can also control the location of the *cursor* in the result set.
- You can use *prepared statements* to supply parameters to SQL statements. Since prepared statements provide better performance and security than regular statements, you should use them whenever possible.

Exercise 20-1 Work with JDBC

In this exercise, you'll write JDBC code that works with the MySQL database named mma that was described in the previous chapter.

Review the code and test the application

1. Open the project named ch20_ex1_DBTester that's in the ex_starts folder.
2. Expand the Libraries folder for this project and note that it includes a JAR file for the MySQL database driver.
3. Make sure the MySQL server is running.
4. Review the code in the source files and run the project. It should print all of the rows in the Product table to the console three times with some additional messages.

Write and test code that uses JDBC

5. Write the code for the printFirstProduct method. This method should print the first product in the list of products to the console. Use column names to retrieve the column values.
6. Run this application to make sure it's working correctly.
7. Write the code for the printProductByCode method. This method should print the product with the specified code to the console. Use a prepared statement to create the result set, and use indexes to retrieve the column values.
8. Run this application to make sure it's working correctly.
9. Write the code for the insertProduct method. This method should add the product to the database and print that product to the console.
10. Run this application to make sure it's working correctly. If you run this application multiple times, it should display an error message that indicates that the product can't be added because of a duplicate key.
11. Write the code for the deleteProduct method. This method should delete the product that was added by the insertProduct method.
12. Run this application to make sure it's working correctly. You should be able to run this application multiple times without displaying any error messages.

Exercise 20-2 Modify the Product Manager application

In this exercise, you'll modify a Product Manager application that works with the MySQL database named mma that was described the previous chapter.

Review the code and test the application

1. Open the project named ch20_ex2_ProductManager that's in the ex_starts folder.
2. Expand the Libraries folder for this project and note that it includes a JAR file for the MySQL database driver.
3. Open the ProductDB class and review its code. Note that it provides all of the methods presented in this chapter, including an update method.
4. Open the Main class and review its code. Then, run this application. It should let you view and store product data in a database.

Modify the JDBC code

5. In the ProductDB class, modify the getAll method so it uses column numbers instead of column names to get the data for the row.
6. Run this application to make sure this code works correctly.
7. In the ProductDB class, add a private method that can create a Product object from the current row in the result set like this:

```
private static Product getProductFromRow(ResultSet rs)
throws SQLException {
```

8. In the ProductDB class, modify the getAll and get methods so they use the getProductFromRow method to get Product object from the current row. Note how this reduces code duplication and makes your code easier to maintain.
9. Run this application to make sure this code works correctly.

Add an update command

10. In the Main class, modify the code so it includes an update command. This command should prompt the user for the product code to update. Then, it should prompt the user for a new description and price like this:

```
Enter product code to update: java
Enter product description: Murach's Beginning Java
Enter price: 54.50
```

11. In the Main class, add code that gets the specified product from the database, sets the new data in that product, and updates the database with the new data. If successful, this should display a message like this:

```
Murach's Beginning Java was updated in the database.
```

12. Run this application to make sure this code works correctly.

21

How to develop a GUI with Swing (part 1)

Up until now, all of the code you've been working with in this book has used a console interface. However, most modern applications—such as word processors, spreadsheets, web browsers, and so on—use a graphical user interface, often abbreviated as GUI. In this chapter and the next, you'll learn the basics of developing GUI applications with Swing, which is the most popular library for developing GUI applications with Java.

An introduction to GUI programming

Up until the late 1980s (and even into the 90s), most computer applications used text-based user interfaces. The introduction of the Apple Macintosh, followed shortly after by Microsoft Windows, changed all that. By the mid 1990s, most major applications used graphical user interfaces, also called *GUIs*.

In the early days, programming GUIs was very complicated because the APIs for doing so were extremely low level. Thankfully, that situation has changed, and modern APIs make writing good looking GUIs much easier.

A summary of GUI toolkits

If you want to use Java to develop a GUI, you can choose from several APIs, which are often called *toolkits*. Figure 21-1 shows the most common ones, though there are others available as well.

AWT (*Abstract Window Toolkit*) was the first GUI toolkit for Java. AWT allows you to create native operating system *components* (commonly called *widgets*). However, this approach has several limitations. Since it's operating system dependent, widgets take up different amounts of space on different operating systems. As a result, a GUI that looks perfect on one operating system might look terrible on another. In addition, because AWT can only use the native components of an operating system, it lacks advanced widgets.

To address the shortcomings of AWT, Java 2 included a new toolkit named Swing. Swing is built on top of AWT, but creates its own widgets instead of using the ones provided by the operating system. This has the advantage of ensuring that GUIs look the same on all platforms that Java runs on. In addition, it allows the toolkit to have more advanced widgets.

Unfortunately, in the early days, Swing was slow and inefficient. In addition, the early Swing widgets were relatively ugly and didn't look like native widgets. Fortunately, these problems with Swing no longer exist today. Today, Swing performs much better, and, modern themes provided with Java can make Swing applications look like native applications.

To compete with early versions of Swing, IBM created *SWT* (*Standard Widget Toolkit)*. Unlike Swing, SWT uses native widgets. As a result, SWT widgets look like native widgets. In addition, in the early days, SWT had a speed advantage over Swing.

However, SWT also has two drawbacks. First, because it relies on native widgets, you are required to ship a native library for each operating system you want to support. Second, the core SWT widget set is primitive. As a result, it's difficult to create advanced user interfaces with it.

JavaFX is a newer framework for developing *rich Internet applications* (*RIAs*). However, since JavaFX completes with Adobe's already popular Flash plugin, it was late to the game. In addition, the introduction of HTML 5 is making proprietary browser plugins such as Adobe Flash, Oracle Java, and Microsoft Silverlight, obsolete. As a result, JavaFX has been virtually ignored by developers.

Common Java GUI libraries

Library	Description
AWT	Abstract Window Toolkit. This is the oldest Java GUI library. It uses the native widgets from the operating system. This library is not recommended for modern application development due to problems with cross-platform portability.
Swing	Created to replace AWT and fix the cross-platform portability problems. Swing widgets are not native, but can use themes to look identical to native widgets. Most modern Java GUIs are written using Swing.
SWT	Standard Widget Toolkit. This library was created by IBM to fix performance problems and the non-native look and feel of earlier versions of Swing. This library uses native widgets. However, it's not as powerful as Swing without adding additional libraries.
JavaFX	Designed to replace the aging Swing library and make it easier to develop rich Internet applications (RIAs). However, it has been slow to catch on.
Pivot	An open-source Apache project that allows you to create modern looking GUIs that work with Internet resources. Like JavaFX, this library has been slow to catch on.

Description

- When creating GUIs with Java, you have several possible libraries that you can use. Of these libraries, Swing is the most widely used.

Figure 21-1 A summary of Java GUI toolkits

Pivot is an open-source Apache project that was intended primarily to compete with JavaFX. Although it was initially designed for producing rich Internet applications that run in a browser, it works just as well for developing desktop applications.

Although JavaFX and Pivot are newer than Swing, they are not nearly as popular as Swing. Instead, Swing dominates the world of GUI programming with Java and is what you are most likely to encounter in the real world. As a result, this book covers Swing instead of JavaFX or Pivot. However, if you need to write an application that works extensively with Internet resources, you might want to investigate JavaFX or Pivot.

The inheritance hierarchy for Swing components

Like all Java APIs, Swing is object oriented, and the Swing classes exist in a hierarchy. Figure 21-2 shows a partial diagram of this hierarchy, along with some of the most commonly used classes.

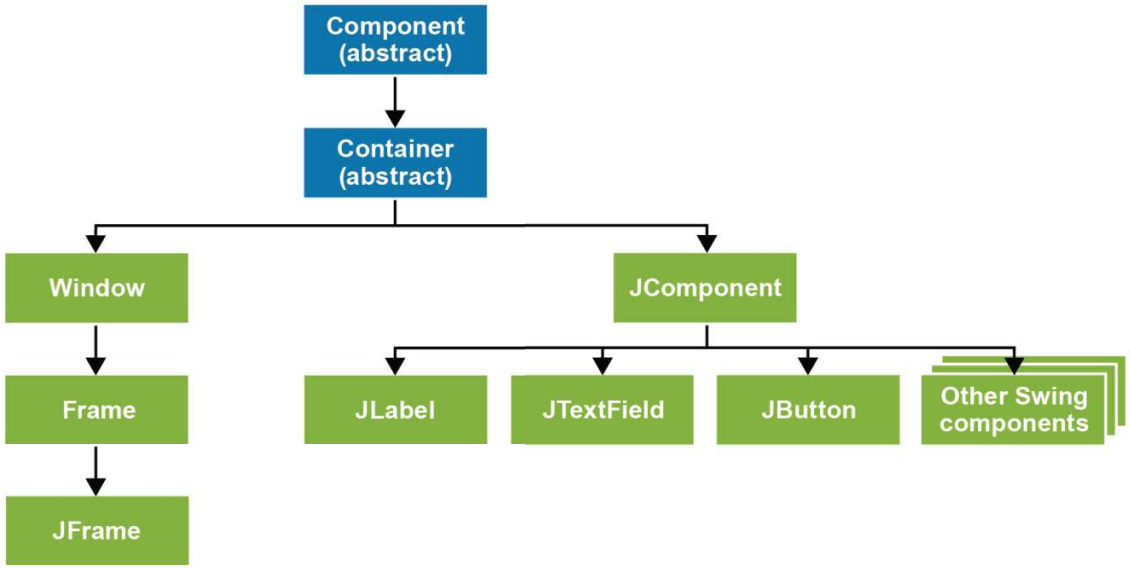
As mentioned in the previous figure, Swing builds on the older AWT framework. As a result, all Swing components ultimately inherit the abstract AWT Container class, which itself inherits the abstract AWT Component class. In this diagram, all Swing classes begin with the letter J (for Java). This distinguishes them from AWT components with the same name. For example, Frame is an AWT component, and JFrame is a Swing component.

As this diagram shows, most Swing components inherit from the JComponent class. As a result, any method that accepts a JComponent object as a parameter, accepts most Swing widgets. In this diagram, the exception is a JFrame widget. This widget is commonly called a *top-level widget* because it inherits from the Window class and holds other widgets. Typically, a JFrame widget is used as the main window for the application.

A *container* is a type of component that can hold other components. The diagram in this figure shows that all Swing widgets inherit from the Container class. As a result, all Swing widgets can act as a container to hold other widgets. That means, it's possible for a JButton widget to hold another JButton widget. However, it's more common to use a container widget such as a JFrame widget to hold other widgets. Later in this chapter and the next, you'll learn about some other container widgets including the JPanel, JScrollPane, and JDialog widgets.

As a general rule, you should not mix AWT and Swing components. This typically results in bugs and display problems. For example, you should not put a JButton widget inside a Frame object. Instead, you should put a JButton widget inside a JFrame.

The Swing inheritance hierarchy



A summary of these classes

Class	Description
`Component`	An abstract AWT class that defines any object that can be displayed. For instance, frames, panels, buttons, labels, and text fields are derived from this class.
`Container`	An abstract AWT class that defines any component that can contain other components.
`Window`	The AWT class that defines a window without a title bar or border.
`Frame`	The AWT class that defines a window with a title bar and border.
`JFrame`	The Swing class that defines a window with a title bar and border.
`JComponent`	A base class for Swing components such as JPanel, JButton, JLabel, and JTextField.
`JPanel`	The Swing class that defines a panel, which is used to hold other components.
`JButton`	The Swing class that defines a button.
`JLabel`	The Swing class that defines a label.
`JTextField`	The Swing class that defines a text field.

Description

- The Swing library is built upon the AWT library. In other words, Swing classes often inherit AWT classes.
- Most Swing classes begin with the letter J (for Java).

Figure 21-2 The inheritance hierarchy for Swing components

How to create a GUI that handles events

When you develop a GUI with Swing, you typically begin by displaying a frame that acts as the main window for the application. Then, you add components such as panels and buttons to the frame. Finally, you handle the events that occur when the user interacts with these components.

How to display a frame

A JFrame widget has a border and contains all of the normal window controls for your operating system such as minimize, maximize, and close buttons. Figure 21-3 shows an example of an empty JFrame widget along with the code that displays it.

To set the title of the frame, you can either pass it as a parameter to the JFrame constructor, or you can create a JFrame object and then use the setTitle method.

To set the size of the frame, you can use the setSize method. This method takes two integers which represent the width and height in pixels. In this figure, for example, the frame is set to a width of 600 pixels and a height of 400 pixels. Another way to size a frame is to use the pack method. This method takes no parameters and sizes the frame just large enough to hold all of the components it contains. If you don't set the size, the frame has a width and height of zero, and you can't see any of the components it contains. As a result, you typically want to set the size of the frame.

To set the location of the frame, you can pass a true value to the setLocationByPlatform method. Then, the operating system determines the location of the frame, which is usually what you want.

When the user clicks on the close button of a frame, you typically want to close the frame and exit the application. The easiest way to do this is to call the setDefaultCloseOperation method and pass it the EXIT_ON_CLOSE value as show in the figure. Although there are other possible values that you can use with this method, the EXIT_ON_CLOSE value is a quick and easy way to close a frame. If you don't include this code and the user clicks the close button, the frame closes, but the application continues running. That's because, the close button only closes the frame by default, but doesn't end the application.

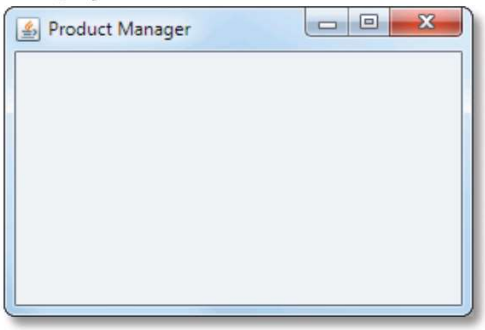
Before you display a frame, you can set the look and feel to any look and feel that's available from Swing. To do that, you can use the UIManager class that's available from the javax.swing package as shown in this figure. In this figure, the code sets the look and feel for the Swing components to the current operating system. As result, this application looks like a Windows application on Windows, a Mac application on Mac OS X, and a Linux application on Linux. In practice, you typically have to do some more work to get the application to act exactly like a native Mac application. However, that's beyond the scope of this book.

Finally, to display the frame, you can pass a true value to the setVisible method.

The package that contains the Swing classes

```
javax.swing
```

An empty frame



A main method that creates and displays a frame

```
public static void main(String[] args) {
    JFrame frame = new JFrame("Product Manager");
    frame.setSize(600, 400);
    frame.setLocationByPlatform(true);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    try {
        UIManager.setLookAndFeel(
                UIManager.getSystemLookAndFeelClassName());
    } catch (ClassNotFoundException | IllegalAccessException |
             InstantiationException | UnsupportedLookAndFeelException e) {
        System.err.println("Unsupported look and feel.");
    }

    frame.setVisible(true);
}
```

Description

- A *frame* typically defines the main window of an application.
- To set the title of a frame, you can use the constructor of the JFrame class. Or, you can call the setTitle method of the JFrame object.
- To set the size of a frame, you call the setSize method and send it the height and width in pixels. Or, you can call the pack method to automatically size the window so it is just large enough to hold all of its components.
- To let the operating system set the location of a frame, you can call the setLocation-ByPlatfrom method and send it a true value.
- To exit the application when the user selects the close button, you can set the default close operation. If you don't, the application may keep running after the window closes.
- To set the look and feel to use the default platform look and feel, you call the setLookAndFeel method of the UIManager class inside a try-catch block.
- To make the frame visible, you call the setVisible method and pass it a true value.

Figure 21-3 How to work with frames

How to add a panel to a frame

Usually, the easiest way to build a GUI application is to use an invisible container known as a *panel* to group components. To do that with Swing, you can use the JPanel class. Then, you can add one or more panels to the frame.

Figure 21-4 shows an example of how to add a panel to a frame. To start, it creates a JPanel widget named panel. Then, it calls the add method of the frame to add the panel to the frame. Since a panel is invisible, this doesn't display anything on the frame. To do that, you need to add one or more widgets to the panel.

How to add buttons to a panel

The second example adds two buttons to the panel. To do that, it creates two buttons from the JButton class. Then, it uses the add method of the panel to add the buttons to the panel.

To set the text of the button, you can use the constructor of the JButton class. Or, you can create the JButton widget and call its setText method. In this figure, for example, the text for both buttons is set by the constructor for the JButton widget.

The JButton widget provides some other methods that are commonly used. For example, you can use the setEnabled method to disable a button and gray it out. If you want to disable an OK button until the user has filled in the required data, for instance, you can use this method. Or, you may want to display a *tooltip*, which is a common graphical user interface element that's displayed when the user hovers the pointer over an item for about one second. To do that, you can use the setToolTipText method.

A common method of the JFrame and JPanel objects

Method	Description
`add(Component)`	Adds the specified component to the frame or panel.

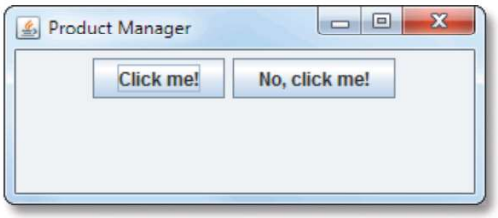
Code that adds a panel to a frame

```
JPanel panel = new JPanel();
frame.add(panel);
```

Code that adds two buttons to a panel

```
JButton button1 = new JButton("Click me!");
JButton button2 = new JButton("No, click me!");
panel.add(button1);
panel.add(button2);
```

Two buttons in a panel



Common constructors and methods of the JButton object

Constructor	Description
`JButton(String)`	Sets the text displayed by the button.
Method	**Description**
`setText(String)`	Sets the text displayed by the button.
`setEnabled(boolean)`	Sets whether the button is enabled or disabled and grayed out.
`setToolTipText(String)`	Sets the text for the tooltip. If the user hovers the mouse over the button for about one second, the tooltip is displayed.

Description

- A *panel* is an invisible container that's used to group other components.
- To create a panel, you can create a JPanel object. Then, you can add it to a frame.
- To create a button, you can create a JButton object. Then, you can add it to a frame or a panel.
- You should add all of the panels and components to the frame before calling its setVisible method. Otherwise, the user interface may flicker when it's first displayed.

Figure 21-4 How to work with buttons

How to handle a button event

The buttons displayed by the code in the previous figure don't do anything when you click on them. That's because there is nothing to handle the *event* that's generated by the button when it's clicked.

Each time the user interacts with your GUI, an event is fired. For example, when the user clicks on a button, an event is fired notifying your code that the button was clicked. An event is also fired each time the user presses a key on the keyboard, or moves the mouse.

If you want to respond to the event that is fired when the button is clicked, you need to register something that listens for the event. This is referred to as an *action listener* or *event listener*.

Figure 21-5 shows how you can use the addActionListener method of the button to register a method that handles the event when the button is clicked. This method is known as an *event handler*. When the button is clicked, an ActionEvent object is created and sent to the event handler. This ActionEvent object contains a lot of information about the event. However, in simple cases like this, you don't need any of that information. Instead, you just need to know that the button fired an ActionEvent.

Prior to Java 8, the most common way to handle an ActionEvent was to use an anonymous inner class as shown in the first example. Here, an anonymous class is coded within the parentheses of the addActionListener method. This code begins by creating a new ActionListener object that contains a method named actionPerformed that accepts an ActionEvent argument. Then, the code within the actionPerformed method is executed when the user clicks the button. In this case, this method just prints a message of "Button 1 clicked!" to the console. However, this method could contain more complex code such as code that updates a database.

With Java 8 and later, you can use a lambda instead of an anonymous class as shown in the second example. In this case, the lambda is cleaner, more concise, and easier to understand than an anonymous inner class would be. As a result, if you're using Java 8 or later, you'll probably want to use lambdas. However, if you plan to distribute your application to users who might be using Java 7 or earlier, you'll probably want to use anonymous methods.

Although this figure shows how to handle the event that occurs when a user clicks a button, other Swing components also generate events that you can handle. For example, when a user changes the contents of a text box, it generates an event that you can handle. Handling those types of events is outside the scope of this book. However, you can use the concepts presented in this figure to get started with them, and you can use the API documentation to get more information about them.

The package that contains the events

```
java.awt.event
```

A common method of most components

Method	Description
`addActionListener(ActionEvent)`	Adds an event listener to a component such as a button so it can listen for and respond to events such as click events.

Code that adds an event handler to a button

With an anonymous class (prior to Java 8)

```
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button 1 clicked! ");
    }
});
```

With a lambda expression (Java 8 and later)

```
button1.addActionListener((ActionEvent e) -> {
    System.out.println("Button 1 clicked!");
});
```

Description

- To make a button do something when clicked, you add an action listener by calling its addActionListener method.
- The addActionListener method accepts an ActionListener object that includes an actionPerformed method that accepts an ActionEvent object. Within the actionPerformed method, you write the code that's executed when the button is clicked.
- Prior to Java 8, it was common to write anonymous inner classes to handle actions. However, with Java 8 or later, you can use a lambda expression, which is cleaner and more concise.

Figure 21-5 How to handle the event that occurs when a button is clicked

How to work with layout managers

In the old days of GUI design, it was common to place non-resizable components wherever you wanted them. This type of layout is known as *fixed width layout*. Although fixed width layout is quick and easy, it causes several problems. For example, if the window is resized, the components inside don't resize with it, which means it's possible for components to disappear outside the boundaries of the window. Furthermore, if the user's operating system uses a different font size, it's possible for components to end up on top of each other or hidden underneath other components. To address this problem, most modern GUI toolkits, including Swing, use a concept known as a *layout manager*.

A summary of layout managers

Java provides several built-in layout managers such as the ones summarized in figure 21-6. Some are easy to use, but are not flexible. Others are harder to use, but are flexible and can produce complex layouts. Still others are intended primarily for use by GUI tools that generate GUI code for programmers rather than for use by programmers who are coding by hand. In addition, there are several third-party layout managers that are available for free that might be easier to use for some tasks.

This chapter shows how to use two of the most common layout managers included with Java: FlowLayout and BorderLayout. The next chapter shows how to use the GridBagLayout manager. The Product Manager application presented in this chapter and the next uses all three of these layout managers. Once you are familiar with the basics of using these layout managers, you can search the Internet for information about the other layout managers, which might be easier to use for certain types of tasks.

By default, a JFrame widget uses the BorderLayout manager, and a JPanel widget uses the FlowLayout manager. If the default layout manager provided by a container doesn't suit your needs, you can change it to a different one by calling the container's setLayout method, as shown in this figure. This code changes the layout manager of the frame from its default of BorderLayout to FlowLayout instead.

The package that contains the layout managers

```
java.awt
```

A summary of layout managers

Manager	Description
`BorderLayout`	Lays out components by providing five areas that can each hold one component. The five areas are north, south, east, west, and center.
`FlowLayout`	Lays out components by flowing them from left to right, or right to left, similar to words on a page.
`GridBagLayout`	Lays out components in a rectangular grid as shown in the next chapter.
`GridLayout`	Lays out components in a rectangular grid of cells where each cell is the same size.
`CardLayout`	Lays out components on a card where only one card is visible at a time. This layout is rarely used, but can be useful for creating wizard dialogs that guide the user through a step-by-step procedure.
`BoxLayout`	Lays out components in a horizontal or vertical row of cells. This layout is rarely used, but it can be useful for producing rows of buttons.

A common method of most container components

Method	Description
`setLayout(layout)`	Sets the layout to the specified layout manager.

To change a container's layout manager

```
JFrame frame = new JFrame();
frame.setLayout(new FlowLayout());
```

Description

- A *layout manager* determines how your components are placed in the container and how they behave if the container is resized or if the font size changes.
- By default, a JFrame uses the BorderLayout manager.
- By default, a JPanel uses the FlowLayout manager.

Figure 21-6 A summary of layout managers

How to use the FlowLayout manager

Of the three layout managers presented in this book, the FlowLayout manager is the easiest to use. It adds components in a row and aligns the components according to the alignment parameter. By default, the FlowLayout manager centers components.

The first example in figure 21-7 begins by creating a new JFrame widget with a title of "Product Manager". Then, this code sets the size to a width of 600 pixels and a height of 400 pixels. Next, this code calls the frame's setLayout method and passes it a new instance of the FlowLayout manager. Since this statement doesn't specify an alignment for the FlowLayout manager, it uses the default alignment of center.

After setting the layout manager, this code adds four buttons to the frame labeled Button 1, Button 2, Button 3, and Button 4. As a result, the layout manager arranges these buttons from left to right in the same order that the code added them to the frame.

If there isn't enough space in the container to hold all of the components, FlowLayout wraps the components to the next row. This is similar to how a word processor wraps text when it reaches the right margin. Once again, the layout manager arranges the components according to the alignment we specified, which in this case, is center.

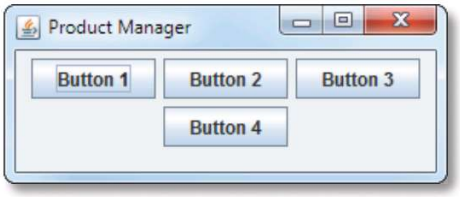
To change the default alignment, you can supply an alignment parameter to the constructor of the FlowLayout class. In the second example, for instance, the alignment parameter tells FlowLayout that it should align the components with the left edge of the frame, similar to left aligning text in a word processor. It's also possible to right align the components.

Code that creates a FlowLayout and adds four buttons to it

```
JFrame frame = new JFrame("Product Manager");
frame.setSize(600, 400);
frame.setLayout(new FlowLayout());
frame.add(new JButton("Button 1"));
frame.add(new JButton("Button 2"));
frame.add(new JButton("Button 3"));
frame.add(new JButton("Button 4"));
```

The FlowLayout when all components fit on one line

The FlowLayout when components wrap to the next line



A common constructor of the FlowLayout class

Constructor	Description
`FlowLayout(alignment)`	Sets the horizontal alignment of this manager. To set this alignment, you can specify the LEFT, RIGHT, or CENTER constants of the FlowLayout class.

How to change the default layout

```
frame.setLayout(new FlowLayout(FlowLayout.LEFT));
```

Description

- If the components don't fit on one line, the FlowLayout manager wraps the components to a new line.
- The FlowLayout manager can align components to the left, right, or center. The default is center.

Figure 21-7 How to work with the FlowLayout manager

How to use the BorderLayout manager

The BorderLayout manager has five areas: north, south, east, west, and center. Each of these areas can hold a single component.

Figure 21-8 shows a frame that uses the BorderLayout manager to display one button in each of the five areas. This code begins by creating a frame like the one in the previous figure. However, since BorderLayout is the default layout manager for a frame, this code doesn't set the layout manager.

After creating the frame, this code adds a new button to the north area of the frame by calling the add method of the frame. Then, it passes the component as the first argument and the area as the second argument. Here, the component is a new JButton widget that says "North", and the BorderLayout.NORTH constant specifies the north area. This causes the layout manager to size this component horizontally to fill the entire width of the frame and vertically to its preferred height, which is the height of the button.

In a similar fashion, this code adds a new button to the south area of the frame. Like the north area, the layout manager sizes the component horizontally to fill the entire width of the frame and vertically to its preferred height.

After adding a button to the north and sound areas, this code adds a button to the east and west areas. In this case, the layout manager sizes these components to fill all of the vertical space between the north and south areas. In addition, it sizes these components to their preferred width, which is the width of the button.

Finally, the code adds a button to the center area of the frame. In this case, the layout manager sizes this component both vertically and horizontally to fill any remaining space not used by the components in the other areas.

If you resize this window, the layout manager adjusts the size of the components in each area. The north and south components resize horizontally to fill the new space, but their height remains the same. The east and west components resize vertically to fill the new space but their width remains the same. And the center component resizes to fill any remaining space.

If you don't provide a component for one or more of the areas in the BorderLayout, that area has a size of zero. For example, if you remove the button from the east area, the east area has a size of 0. As a result, the component in the center expands to the right edge of the frame.

The BorderLayout manager is often used to lay out the main screen of an application. For example, suppose you were writing an email client. You might use the north area as a toolbar, the west area to hold a list of folders, the center area to hold the list of emails, and the south area to hold a status bar. In this case, you might not use the east area.

When working with the BorderLayout manager, it's important to remember that each area can only hold one component. As a result, if you want to add multiple components, you need to group those buttons into a panel before you add them. For example, if you want to add a toolbar with multiple buttons, you need to group those buttons into a panel (probably using FlowLayout for that panel) and then add the panel to the north area of the BorderLayout.

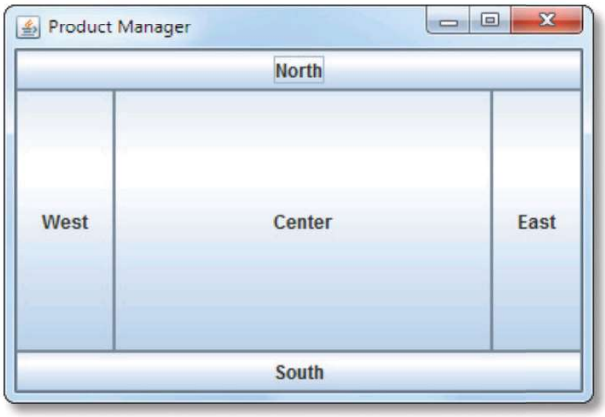
A common method of the BorderLayout class

Method	Description
`add(component, area)`	Adds the specified component to the specified area. To set the area, you can specify the NORTH, SOUTH, EAST, WEST, or CENTER constants of the BorderLayout class.

Code that adds five buttons to each area of a BorderLayout

```
JFrame frame = new JFrame("Product Manager");
frame.setSize(600, 400);
frame.add(new JButton("North"), BorderLayout.NORTH);
frame.add(new JButton("South"), BorderLayout.SOUTH);
frame.add(new JButton("East"), BorderLayout.EAST);
frame.add(new JButton("West"), BorderLayout.WEST);
frame.add(new JButton("Center"), BorderLayout.CENTER);
```

The JFrame produced by the above code



Description

- A BorderLayout has five areas: NORTH, SOUTH, EAST, WEST, and CENTER.
- The default area is CENTER. As a result, if you don't specify the area, the component is added to the center.
- The NORTH and SOUTH areas get their preferred height, which usually means they are tall enough to hold whatever is placed in them. The preferred width is ignored.
- The WEST and EAST areas get their preferred width, which usually means they are wide enough to hold whatever is placed in them. The preferred height is ignored.
- The CENTER area gets whatever space is left over and expands or contracts as the window is resized.
- Each area of a BorderLayout can only hold one component. If you need to add multiple components to an area, you can add them to a panel and then add the panel to the area.

Figure 21-8 How to work with the BorderLayout manger

How to work with tables

Many applications, especially applications that work with a database, need to display rows of data in a table. To do that, you can use a JTable widget. But first, you typically begin by creating a model for the table. A table model holds the data for the table and determines how the data is displayed in the table.

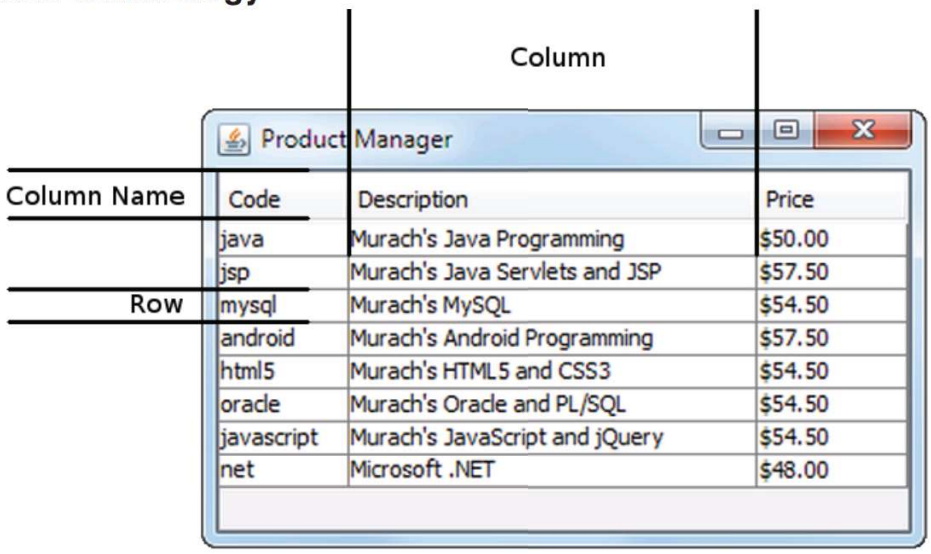
How to create a model for a table

Figure 21-9 begins by showing a JTable widget along with some terminology that's necessary to work with the table and its model. The first row of the table, commonly referred to as the table header, contains a column name for each column of the table. In this figure, the column names are "Code", "Description", and "Price". A column usually stores a certain type of data. In this figure, the first column stores product codes, the second column stores descriptions, and the third column stores prices. And a row typically stores a complete record. In this figure, each row stores all of the information for a product.

To create a model for a table, you can extend the AbstractTableModel class that's part of Swing. Then, you can override the methods in the first table of this figure. Of these methods, you must override the first three, and you should override the fourth if you want to include column names.

In addition, you can add other methods to the class for the model. For example, you may want to add a method that makes it easy to get a Product object from the model. Or, you may need to add a method that calls the fireTableDataChanged method to notify the table that the data in the model has been updated. That way, the table can refresh itself so it displays the current data. You'll see how this works in the next figure.

JTable terminology



AbstractTableModel methods that you typically override

Method	Description
`getRowCount()`	Returns an integer for the number of rows in the table.
`getColumnCount()`	Returns an integer for the number of columns in the table.
`getValueAt(row, column)`	Returns an object for the value at the given row and column index of the table.
`getColumnName(column)`	Returns a string representing the name of the column at the specified column index. This value is used as the column label in the table header.

An AbstractTableModel method that you can call

Method	Description
`fireTableDataChanged()`	Notifies all listeners that all cell values in the table's rows may have changed.

Description

- The easiest way to create a custom table model is to extend the AbstractTableModel class.
- At a minimum, you need to override the getRowCount, getColumnCount, and getValueAt methods. In addition, it's common to override the getColumnName method.
- When you update the database, you can call the fireTableDataChanged method to refresh all data that's displayed in the table.
- The AbstractTableModel class contains other methods, such as fireTableCellUpdated and fireTableRowsInserted, that you can call to update part of a table. These methods can be much more efficient, especially for a large table. See the API documentation for more information.

Figure 21-9 How to create a table model

The ProductTableModel class

Figure 21-10 shows the code for the class that defines the model that's used by the table in the previous figure. This class is named ProductTableModel, and it begins by extending AbstractTableModel.

Within the class, the first statement declares a variable named products that stores a List of Product objects. Then, the second statement declares and initializes an array of strings for the names of the three columns. These column names are Code, Description, and Price.

The constructor for the table model calls the getAll method of the ProductDB class described in the previous chapter to retrieve a List of Product objects from the database. Then, it assigns this list of products to the variable named products.

The getRowCount method is the first of three methods that you are required to override when extending the AbstractTableModel class. This method returns an integer value for the number of rows in the table. Since this value is equal to the number of products in the products list, this code returns the value of the size method of the products list.

The getColumnCount method is the second method you are required to override. This method returns an integer value for the number of columns. To do that, this returns the length of the COLUMN_NAMES array, which is the same as the number of columns.

The getColumnName method isn't required, but it's necessary if you want to display column names. This method takes a column index value and returns the name of the column for the specified index. To do that, this method returns the string at that index of the COLUMN_NAMES array.

The getValueAt method is the third method you are required to override. This method defines a row index and a column index as parameters. Then, it returns the value at the specified row and column. To do that, this method uses a switch statement to determine the column to display. For a column index of 0, this code calls the get method of the products list to return a Product object at the specified row index. Then, it uses method chaining to call the getCode method of that Product object. For a column index of 1, this code calls the getDescription method of the Product object at the specified row index. And for a column index of 2, this code calls the object's getPriceFormatted method.

One thing that can cause some confusion for new programmers is that you define these four methods, but you never call them anywhere. That's because the JTable widget calls these methods whenever it needs information about how to display the table. For example, when the JTable widget needs to know what value to display for the description column in the first row of the table, it calls the getValueAt method with a row index of 0, and a column index of 1. Then, the getValueAt method gets the Product object from the first row of the product list (remember that list indexes start at 0) and calls its getDescription method to return the description.

The ProductTableModel class

```
package mma.ui;

import java.util.List;
import javax.swing.table.AbstractTableModel;

import mma.business.Product;
import mma.db.ProductDB;
import mma.db.DBException;

public class ProductTableModel extends AbstractTableModel {
    private List<Product> products;
    private final String[] COLUMN_NAMES = { "Code", "Description", "Price" };

    public ProductTableModel() {
        try {
            products = ProductDB.getAll();
        } catch (DBException e) {
            System.out.println(e);
        }
    }

    @Override
    public int getRowCount() {
        return products.size();
    }

    @Override
    public int getColumnCount() {
        return COLUMN_NAMES.length;
    }

    @Override
    public String getColumnName(int columnIndex) {
        return COLUMN_NAMES[columnIndex];
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        switch (columnIndex) {
            case 0:
                return products.get(rowIndex).getCode();
            case 1:
                return products.get(rowIndex).getDescription();
            case 2:
                return products.get(rowIndex).getPriceFormatted();
            default:
                return null;
        }
    }
```

Figure 21-10 The ProductTableModel class (part 1 of 2)

Part 2 of figure 21-10 shows two more methods of the table model. When you write the code for an application, you can call these methods to work with the data in the table. You'll see how this works later in this chapter.

The getProduct method returns a Product object for the row at the specified index. This lets you get the Product object for the row that's selected by the user, which is useful if you want to update or delete this product in the database.

The databaseUpdated method updates the old data in the table model with new data from the database. Then, it calls the fireTableDataChanged method of the table model. This notifies the JTable widget that the data in the model has changed. This allows the JTable widget to display the new data. If you modify the underlying data in the database, you can call this method to synchronize the data that's displayed in the table with the data that's in the database.

The ProductTableModel class (continued)

```
    Product getProduct(int rowIndex) {
        return products.get(rowIndex);
    }

    void databaseUpdated() {
        try {
            products = ProductDB.getAll();
            fireTableDataChanged();
        } catch (DBException e) {
            System.out.println(e);
        }
    }
}
```

Description

- When a row represents a business object such as a Product object, you can include a method that returns an object for the specified row index.
- To synchronize the data in the table with the data in the database, you can include a method that updates the model with the data from the database and then calls the fireTableDataChanged method. This causes the table to display the new data that's stored in the model.

Figure 21-10 The ProductTableModel class (part 2 of 2)

How to create a table

Once you create a model for a table, you can create a JTable widget to display the data that's stored in that model. The first example of figure 21-11 shows how.

This code begins by creating a ProductTableModel object from the class shown in the previous figure. Then, it creates a new JTable widget by passing the model to the constructor of the JTable class. Alternately, this code could call the empty constructor of the JTable class and then use the setModel method to set the model.

After setting the model for the table, this code sets the selection mode of the table to SINGLE_SELECTION mode. As a result, users can only select one row at a time. However, you can use two other values to set the selection mode.

First, you can use the SINGLE_INTERVAL_SELECTION mode to allow the user to select multiple rows by holding the Control or Shift keys. However, this only works if those rows are contiguous. In other words, it only works if the selected rows are next to each other.

Second, if you want to allow more possibilities, you can use the MULTIPLE_INTERVAL_SELECTION mode to allow the user to select multiple rows by holding the Control or Shift keys. However, in this mode, those rows don't have to be contiguous.

After setting the selection mode, this code calls the setBorder method of the table and passes a null value to that method. As a result, this table doesn't include a border. If you plan to add scrollbars to a table as shown in the next figure, this makes sure that the GUI doesn't display two borders: one for the table and one for the scrollbar. This is a best practice whenever the component is going to take up the entire space of the container.

How to get the selected row or rows

The second example shows how to get the selected row or rows from a table after it's displayed. This code starts by calling the getSelectedRow method of the table. This gets the index for the first row that the user selected.

After getting the row index, this code checks to make sure that the user selected a row. If so, it passes the row index to the getProduct method of the model for the table. This gets the Product object that corresponds with the selected row.

It makes sense for this example to use the getSelectedRow method because the first example only allows the user to select a single row. However, if the first example had allowed the user to select multiple rows, it would make sense for this example to use the getSelectedRows method to get the indexes for all selected rows.

Some common constructors and methods of the JTable class

Constructor	Description
`JTable(model)`	Constructs a new JTable and sets the model for the table to the specified TableModel object.
Method	**Description**
`setModel(model)`	Sets the model for the table to the specified TableModel object.
`setSelectionMode(mode)`	Sets the selection mode to one of the ListSelectionModel values. This determines how the user can select items from the table.
`setBorder(border)`	If you enclose the table in a scroll pane, setting the border to null may improve the appearance.
`getSelectedRow()`	Returns an int value for the index of the first selected row or -1 if no rows are selected.
`getSelectedRows()`	Returns an array of int values for the indexes of the selected rows.

ListSelectionModel values

Value	Description
`SINGLE_SELECTION`	Allows selecting only one row at a time.
`SINGLE_INTERVAL_SELECTION`	Allows selecting multiple rows, but only if they are contiguous.
`MULTIPLE_INTERVAL_SELECTION`	Allows for selection of multiple rows at the same time, even if they are not contiguous

Code that creates a table

```
ProductTableModel productTableModel = new ProductTableModel();
JTable productTable = new JTable(productTableModel);
productTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
productTable.setBorder(null);
```

Code that gets data from the table

```
int rowIndex = productTable.getSelectedRow();
if (rowIndex != -1) {
    Product product = productTableModel.getProduct(rowIndex);
}
```

Description

- Once you have created a model for a table, you can create the table and set the model in the table. Then, you can use the methods of the table to work with it.

Figure 21-11 How to create a table

How to add scrollbars to a table

It's common for a table to contain so many rows or columns that it doesn't fit on one screen. Fortunately, Swing provides a container called JScrollPane that you can use to add scrollbars to a table. Of course, JScrollPane also works with any other component, such as text areas, where the content of the component is too large to fit on one screen.

The first example in figure 21-12 shows how to create a scroll pane and add a table to it. To do that, this code passes the table to the constructor of the JScrollPane class. Then, it adds the JScrollPane object to the frame.

The second code example shows an alternate and common method of adding a component to a scroll pane and adding the scroll pane to the frame. Since this example doesn't maintain a reference to the JScrollPane object, you can't call any of its methods later. However, this is usually fine as long as you don't need to change the default scroll pane policy.

If you need to change the scroll pane policy, you can use the setHorizontalScrollBarPolicy and setVerticalScrollBarPolicy methods of the scroll pane as shown in the third example. These methods determine when the scrollbars are visible to the user. The default for both scrollbars is AS_NEEDED. As a result, if all the content fits in the container, the scrollbars are hidden. However, if the content doesn't fit, the scrollbars are displayed. This could happen, for example, if you resize the window to make the container smaller.

If you want to make it so the scrollbars are always visible, even if they are not needed, you can set the policy to ALWAYS. In this case, the scrollbars are displayed even when they are not needed, but they don't do anything if you try to use them.

If you want to make it so the scrollbars are never visible, you can set the policy to NEVER. This hides the scrollbars. However, the user can still scroll the component by using the arrow keys and the Page Up and Page Down keys.

To add a component to a scroll pane

```
JScrollPane scrollPane = new JScrollPane(productTable);
frame.add(scrollPane);
```

An alternate way of adding a scroll pane

```
frame.add(new JScrollPane(productTable));
```

A constructor and two methods of the JScrollPane class

Constructor	Description
`JScrollPane(component)`	Adds the component to the scroll pane.
Method	**Description**
`setHorizontalScrollBarPolicy(policy)`	Controls when the horizontal scrollbar is visible using one of the constants from the next table.
`setVerticalScrollBarPolicy(policy)`	Controls when the vertical scrollbar is visible using one of the constants from the next table.

ScrollPaneConstants

Value	Description
`HORIZONTAL_SCROLLBAR_AS_NEEDED`	Display the horizontal scrollbar only if the content is wider than the view port and can't all be displayed at the same time. This is the default.
`HORIZONTAL_SCROLLBAR_NEVER`	Never display the horizontal scrollbar. It's still possible to scroll left and right using the left and right arrow keys on the keyboard.
`HORIZONTAL_SCROLLBAR_ALWAYS`	Always display the horizontal scrollbar, whether it is needed or not.
`VERTICAL_SCROLLBAR_AS_NEEDED`	Display the vertical scrollbar only if the content is longer than the viewport and can't all be displayed at the same time. This is the default.
`VERTICAL_SCROLLBAR_NEVER`	Never display the vertical scrollbar. It's still possible to scroll the content up and down using the up and down arrow keys and the page up / page down keys on the keyboard.
`VERTICAL_SCROLLBAR_ALWAYS`	Always display the vertical scrollbar, whether it is needed or not.

To make the scrollbars always visible

```
scrollPane.setHoizontalScrollBarPolicy(
        ScrollPaneConstants.HORIZONATAL_SCROLLBAR_ALWAYS);

scrollPane.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
```

Description

- You can use a *scroll pane* to add scrollbars to any component that might not fit in a window.
- By default, scroll panes only display the scrollbars if they are needed.

Figure 21-12 How to work with scroll panes

How to work with built-in dialog boxes

If you work with GUIs, you may sometimes need to use a dialog box to display a message to the user. For example, you may need to display a notification or a warning. Other times, you may need to use a dialog box to confirm an operation. For example, you may need to ask a simple question to determine whether it's ok to delete a file.

Manually writing GUI code for common dialog boxes like these would be tedious. Fortunately, Swing provides built-in dialog boxes, and you can use the JOptionPane class to work with them.

A dialog box created from the JOptionPane class is always displayed on top of the parent container. It is always *modal*, which means that the user cannot interact with any other part of the application until they have responded to the dialog. And it is always *blocking*, which means that the dialog halts the currently running thread and prevents it from resuming until the user has responded to the dialog.

How to display a message

Figure 21-13 shows how to use the JOptionPane class to display a message to the user. To do that, you can call the static showMessageDialog method. This method takes four parameters: (1) the parent container, (2) the message, (3) the title, and (4) the type of message.

Typically, the parent for a dialog box is one of the frames of the application. However, it can also be a custom dialog box like the one described in the next chapter.

The JOptionPane can display the five message types listed in this figure. In most cases, the main difference between these types is the icon that's displayed by the message box. This icon may vary between operating systems and different look and feel settings. In addition, some operating systems may play a sound for certain types of messages.

The first example displays an information message. As a result, it uses the information icon. Since the purpose of this dialog box is to display a message, it only contains one button labeled "OK". After reading the message, the user can click this button to close the dialog and continue with the application.

The second example displays an error message. This example works like the first example. However, since it's an error message, it uses the error icon.

Again note that these icons may look different depending on the operating system and look and feel. These examples both use the default Swing look and feel.

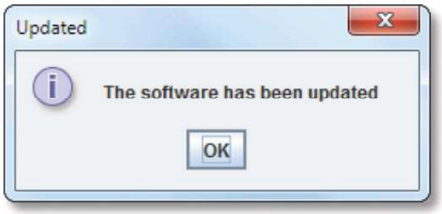
JOptionPane message types

Type	Description
ERROR_MESSAGE	An error icon.
WARNING_MESSAGE	A warning icon.
INFORMATION_MESSAGE	An information icon.
QUESTION_MESSAGE	A question icon.
PLAIN_MESSAGE	No icon.

To display an information message

```
JOptionPane.showMessageDialog(frame, "The software has been updated.",
    "Updated", JOptionPane.INFORMATION_MESSAGE);
```

The dialog box that's displayed



To display an error message

```
JOptionPane.showMessageDialog(frame,
    "The internet could not be accessed because it doesn't exist.",
    "Resource doesn't exist", JOptionPane.ERROR_MESSAGE);
```

The dialog box that's displayed

Description

- To show a dialog box that displays a message, you use the static showMessage-Dialog method of the JOptionPane class. This method has four parameters: (1) the parent container, (2) the message, (3) the title, and (4) the type of dialog.
- Java uses the parent container parameter to determine where to display the dialog. On Windows, Java typically centers the dialog box on the parent container.
- By default, the dialog box has a single OK button.
- The dialog box is *modal*. This means that it is always on top of the rest of the application. As a result, users can't continue until they have responded to the dialog.
- The appearance of the icons displayed by the dialogs varies depending on the look and feel that the application is using. In addition, these icons may vary depending on the locale.

Figure 21-13 How to display a message

How to confirm an operation

Figure 21-14 shows how to use the JOptionPane class to display a message that confirms an operation by asking the user a question. To do that, you can call the showConfirmDialog method. This method works similarly to the showMessageDialog method described in the previous figure. However, the fourth parameter of the showConfirmDialog method determines which buttons the dialog should display. In addition, this method returns a value that indicates which button the user selected.

The showConfirmDialog can display a dialog box with the button options listed in this figure. Of these, the YES_NO_OPTION type and the OK_CANCEL_OPTION type are the most commonly used.

The first example displays a dialog box that asks the user whether he or she wants to create a new file. Here, the first three parameters of the method work the same as the first three parameters of the method shown in the previous figure. However, the fourth parameter uses the YES_NO_OPTION value to specify that the dialog should display Yes and No buttons. In addition, this code assigns the return value of the showConfirmDialog method to an int variable named option. In part 2 of this figure, you'll learn how to use this variable to determine which button the user selected.

By default, the showConfirmDialog uses the QUESTION_MESSAGE type. This message type displays a question mark method as shown in the first example. However, you may want to use an icon that gives a stronger warning if the action the user is performing is going to delete a file, change data, or perform an operation that can't be undone. To do that, you can supply an optional fifth parameter that uses one of the message type values shown in the previous figure. In this figure, for instance, the second code example supplies a fifth parameter that specifies the WARNING_MESSAGE type.

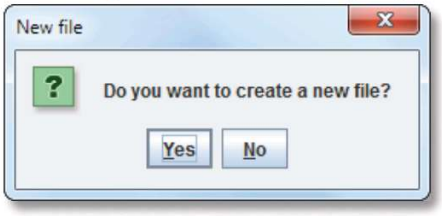
JOptionPane option types

Type	Buttons
DEFAULT_OPTION	OK
YES_NO_OPTION	Yes and No
YES_NO_CANCEL_OPTION	Yes, No, and Cancel
OK_CANCEL_OPTION	OK and Cancel

To display a question dialog with Yes and No buttons

```
int option = JOptionPane.showConfirmDialog(frame,
        "Do you want to create a new file?", "New file",
         JOptionPane.YES_NO_OPTION);
```

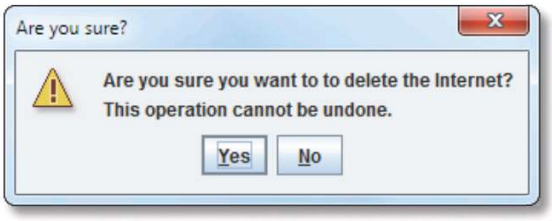
The dialog box that's displayed



To display a question dialog with a warning icon

```
int option = JOptionPane.showConfirmDialog(frame,
        "Are you sure you want to to delete the Internet?\n"
                + "This operation cannot be undone.", "Are you sure?",
                JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE);
```

The dialog box that's displayed



Description

- To display a dialog box that lets the user confirm or cancel an operation, you can use the static showConfirmDialog method of the JOptionPane class.
- The first three parameters of the showConfirmDialog method work the same as the showMessageDialog method shown in the previous figure.
- The fourth parameter determines the buttons that are displayed.
- By default, the showConfirmDialog method displays a question icon. However, you can specify another type of icon by supplying the fifth parameter.
- To wrap a long message to a second line, you can use the line break character (\n).

Figure 21-14 How to confirm an operation (part 1 of 2)

Part 2 of figure 21-14 shows how to use the return value of the showConfirmDialog method to determine which button the user selected. This method can return any of the values listed in this figure. Of course, the possible return values depend on the type of dialog. For example, a dialog box of the YES_NO_OPTION returns the YES_OPTION or the NO_OPTION. In other words, it never returns the CANCEL_OPTION or the OK_OPTION. However, all dialog types can return the CLOSED_OPTION that indicates that the user closed the dialog box without selecting any of the buttons.

The code example uses a switch statement to determine which button the user selected. To do that, this switch statement provides one case for each possible return value: the YES_OPTION, the NO_OPTION, and the CLOSE_OPTION. Then, it prints a different message to the console depending on which option the user selected. Alternately, it's possible to use an if/else statement instead of a switch statement. However, as a general rule, switch statements are more efficient. In addition, for code like this, they're easier to read.

Remember that the dialogs produced by JOptionPane are blocking. This means that they halt the thread and prevent it from continuing until the user has responded to the dialog. As result, the switch statement doesn't run until the user selects one of the dialog buttons or closes the dialog.

JOptionPane return values

Value	Description
YES_OPTION	The Yes button was selected.
NO_OPTION	The No button was selected.
CANCEL_OPTION	The Cancel button was selected.
OK_OPTION	The OK button was selected.
CLOSED_OPTION	The message box was closed without making any selection.

A switch statement that determines which button was selected

```
int option = JOptionPane.showConfirmDialog(frame,
        "Are you sure you want to to delete the Internet?\n"
                + "This operation cannot be undone.", "Are you sure?",
                JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE);

switch (option) {
    case JOptionPane.YES_OPTION:
        System.out.println("You clicked the Yes button.");
        break;
    case JOptionPane.NO_OPTION:
        System.out.println("You clicked the No button.");
        break;
    case JOptionPane.CLOSED_OPTION:
        System.out.println("You closed the dialog.");
        break;
}
```

Description

- When the user selects an option from a dialog box, the showConfirmDialog method closes the dialog box and returns an int value for the option.
- To determine the option selected by the user, it's common to use a switch statement. However, it's also possible to use an if/else statement.

Figure 21-14 How to confirm an operation (part 2 of 2)

The Product Manager frame

This chapter finishes by presenting the complete code for the main window of the Product Manager application. This code uses most of the concepts and components you learned about in this chapter.

The user interface

Figure 21-15 shows the user interface for the Product Manager application. The main window of the application is a JFrame widget that uses the BorderLayout manager. This frame displays two components.

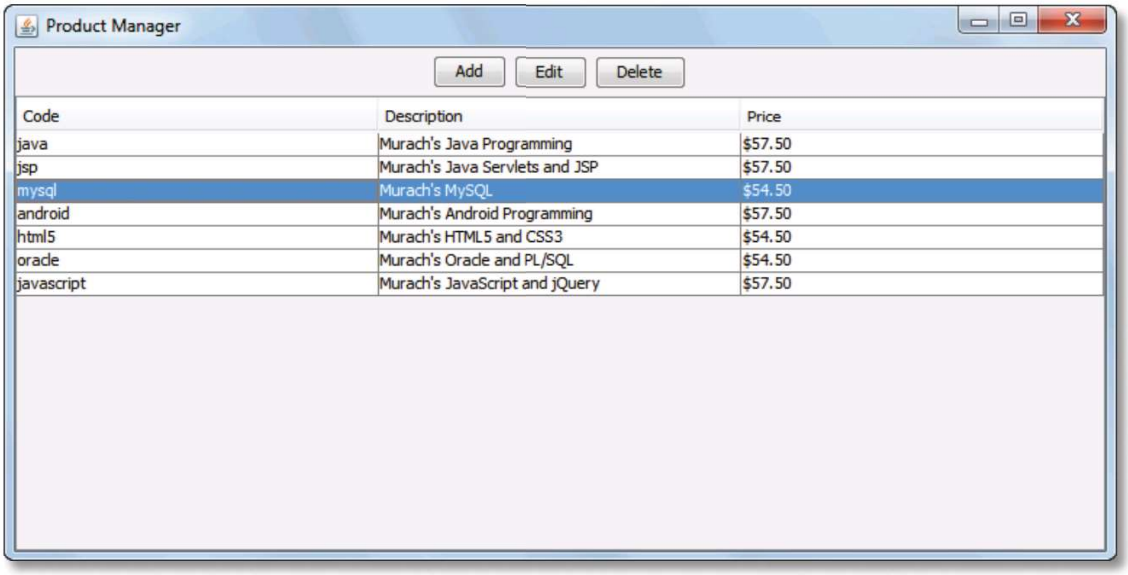
The first is a JPanel widget that uses the FlowLayout manager to display three JButton widgets. The frame displays this panel in the NORTH area of its layout.

The second is a JScrollPane widget that contains a JTable widget. This table displays the data for the products that are stored in the database. The frame displays this scroll pane in the CENTER area of its layout.

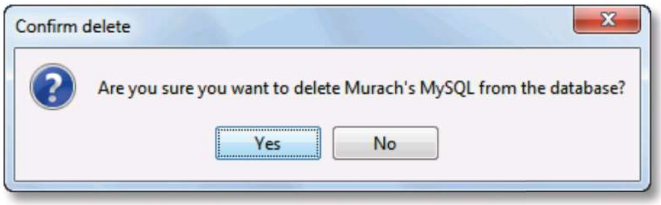
To delete a product, the user can select the row for the product and click the Delete button. This displays the Confirm Delete dialog box shown in this figure. If the user selects the Yes button to confirm the deletion, this deletes the product from the database and updates the data that's displayed in the JTable widget.

At this point, the Add and Edit buttons haven't been implemented. As a result, if the user clicks on either of these buttons, the application displays a dialog box that indicates that the feature hasn't been implemented yet. In the next chapter, you'll learn how to implement these buttons so they can be used to add a new product or update an existing product.

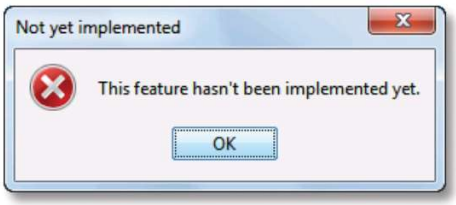
The Product Manager frame



The dialog box for the Delete button



The dialog box for the Add and Edit buttons



Description

- The main window for the Product Manager application is a frame that displays a panel that contains three buttons, as well as a table that contains the data for several products.
- If the user selects the row for a product and clicks the Delete button, the application displays a dialog box that allows the user to confirm the deletion.
- If the user selects the Add or Edit buttons, the application displays a dialog box that indicates that these features haven't been implemented yet. You'll learn how to implement these features in the next chapter.

Figure 21-15 The user interface for the Product Manager application

The ProductManagerFrame class

Figure 21-16 shows the code that creates the main window for the Product Manager application. To start, the class declaration declares a class named ProductManagerFrame that extends the JFrame class. This is a common practice since it allows you to store all code that defines this frame within a single class.

This class begins by defining two instance variables. First, it defines a JTable widget named productTable. This widget displays the table of products to the user. Second, it defines an instance of the ProductTableModel class described earlier in this chapter. This model stores the data for the table widget.

The constructor of this frame starts by using the UIManager to attempt to set the look and feel to the native look and feel for the operating system that the application is running on. This step is optional, but it usually improves the appearance of the application since it matches the look and feel of other applications on the same operating system.

After setting the look and feel, the constructor sets the title, size, and location of the frame. Then, it sets the default close operation so the application exits if the user closes the window. If you don't include this line of code, the window closes, but the application continues to run in the background.

After setting the default close operation, the constructor calls the buildButtonPanel method that's shown later in this class to get a panel that contains the three buttons for this application. Then, it adds this panel to the NORTH area of the BorderLayout. This is possible because a JFrame widget uses the BorderLayout manager by default.

After adding the button panel to the NORTH area, the constructor calls the buildProductTable method that's shown later in this class to get the JTable widget for the product table. In addition, it assigns this JTable widget to the productTable variable.

After creating the JTable widget, the constructor adds a JScrollPane widget that contains the JTable widget to the CENTER area of the BorderLayout. Since CENTER is the default area, this code could have omitted this argument. However, including this argument clearly identifies the area of the BorderLayout.

Finally, the constructor calls the setVisible method to display the user interface to the user. This statement is coded last because coding it earlier in the constructor can sometimes cause the user interface to flicker when it's first displayed.

The buildButtonPanel method begins by creating a JPanel object named panel. By default, a JPanel object uses the FlowLayout manager. Since that's what we want in this case, this code doesn't set a layout.

This method continues by creating the Add, Edit, and Delete buttons. Then, it sets a tooltip for some of these buttons. For example, it sets a tooltip of "Edit selected product" for the Edit button. Next, it uses lambda expressions to add action listeners to each button. Each of these listeners call an appropriately named method. For example, the action listener for the Add method calls the doAddButton method that's coded later in this class.

The ProductManagerFrame class

```
package mma.ui;

import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.ListSelectionModel;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

import mma.business.Product;
import mma.db.ProductDB;
import mma.db.DBException;

public class ProductManagerFrame extends JFrame {
    private JTable productTable;
    private ProductTableModel productTableModel;

    public ProductManagerFrame() {
        try {
            UIManager.setLookAndFeel(
                    UIManager.getSystemLookAndFeelClassName());
        } catch (ClassNotFoundException | InstantiationException |
                 IllegalAccessException | UnsupportedLookAndFeelException e) {
            System.out.println(e);
        }
        setTitle("Product Manager");
        setSize(768, 384);
        setLocationByPlatform(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        add(buildButtonPanel(), BorderLayout.NORTH);
        productTable = buildProductTable();
        add(new JScrollPane(productTable), BorderLayout.CENTER);
        setVisible(true);
    }

    private JPanel buildButtonPanel() {
        JPanel panel = new JPanel();

        JButton addButton = new JButton("Add");
        addButton.addActionListener((ActionEvent) -> {
            doAddButton();
        });
        panel.add(addButton);

        JButton editButton = new JButton("Edit");
        editButton.setToolTipText("Edit selected product");
        editButton.addActionListener((ActionEvent) -> {
            doEditButton();
        });
        panel.add(editButton);
```

Figure 21-16 The code for ProductManagerFrame (part 1 of 2)

After creating these three buttons, this code adds these three buttons to the panel. Finally, the last statement in the buildButtonPanel method returns the panel.

The next three methods are the event handler methods for the action listeners. For now, the methods for the Add and Edit buttons just display a dialog box with a message indicating the feature hasn't been implemented yet. In the next chapter, you'll learn how to add code to these methods so the Add and Edit buttons can be used to add new products or to edit existing products.

However, the method for the Delete button contains code that deletes the selected product from the database. To start, this code uses the getSelectedRow method of the JTable widget to get the index for the selected row. Then, this code checks whether this index is equal to -1. If so, it displays a message dialog that indicates that the user must select a product before clicking the Delete button. Otherwise, it displays a dialog that contains Yes and No buttons that allow the user to confirm the operation. If the user clicks the Yes button, this code uses the delete method of the ProductDB class described in the previous chapter to delete the product. Then, it calls the fireDatabaseUpdatedEvent method that's coded later in this class. In turn, this method calls the databaseUpdated method of the table model to update the data that's displayed by the JTable widget.

The code that calls the methods of the JOptionPane class specifies the keyword named *this* as the first argument of the method. This tells the JOptionPane class that the current object (the ProductManagerFrame object) is the parent for the dialog box.

The buildProductTable method creates a ProductTableModel object and assigns it to the productTableModel instance variable. Remember that the constructor for ProductTableModel class makes a call to the ProductDB class to get all of the records. As a result, this creates a model that contains a list of products that corresponds with the products in the database.

After creating the model, this code creates a new JTable widget and sets its model. To do that, it passes the productTableModel variable to the constructor of the JTable class.

After creating the JTable widget, the next line of code sets the selection mode of the table to a value of SINGLE_SELECTION. As a result, the user can only select one row at a time.

Because this table is displayed in a JScrollPane widget, this code also sets the border of the table to null. This makes sure that the Product Manager application doesn't display two borders: one for the table and one for the scroll pane.

Finally, this code returns the table. Remember that the constructor of this class calls the buildProductTable method to get this table. Then, it adds this table to a scroll pane, and it adds the scroll pane to the center area of the frame.

The ProductManagerFrame class

```
        JButton deleteButton = new JButton("Delete");
        deleteButton.setToolTipText("Delete selected product");
        deleteButton.addActionListener((ActionEvent) -> {
            doDeleteButton();
        });
        panel.add(deleteButton);

        return panel;
    }

    private void doAddButton() {
        JOptionPane.showMessageDialog(this,
            "This feature hasn't been implemented yet.",
            "Not yet implemented", JOptionPane.ERROR_MESSAGE);
    }

    private void doEditButton() {
        doAddButton();
    }

    private void doDeleteButton() {
        int selectedRow = productTable.getSelectedRow();
        if (selectedRow == -1) {
            JOptionPane.showMessageDialog(this,
                    "No product is currently selected.",
                    "No product selected", JOptionPane.ERROR_MESSAGE);
        } else {
            Product product = productTableModel.getProduct(selectedRow);
            int ask = JOptionPane.showConfirmDialog(this,
                    "Are you sure you want to delete " +
                        product.getDescription() + " from the database?",
                    "Confirm delete", JOptionPane.YES_NO_OPTION);
            if (ask == JOptionPane.YES_OPTION) {
                try {
                    ProductDB.delete(product);
                    fireDatabaseUpdatedEvent();
                } catch (DBException e) {
                    System.out.println(e);
                }
            }
        }
    }

    public void fireDatabaseUpdatedEvent() {
        productTableModel.databaseUpdated();
    }

    private JTable buildProductTable() {
        productTableModel = new ProductTableModel();
        JTable table = new JTable(productTableModel);
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        table.setBorder(null);
        return table;
    }
}
```

Figure 21-16 The code for ProductManagerFrame (part 2 of 2)

Perspective

In this chapter, you learned some essential skills for developing a GUI with Swing, the most widely used GUI toolkit for Java. In particular, you learned how to create the main window of an application. You learned how to add buttons. You learned how to handle the events that occur when these buttons are clicked. You learned how to work with a table that displays data from a database. And you learned how to work with built-in dialog boxes. In the next chapter, you'll learn how to create data entry forms that allow you to add and update data in a database.

Summary

- *Swing* is the most commonly used GUI toolkit for Java.
- An element of a GUI is called a *component* or a *widget*.
- Swing provides pluggable look and feels that you can change.
- A component that can contain other components is called a *container*. Three commonly used containers are frames, panels, and scroll panes.
- A JFrame widget usually serves as the main window of your application.
- A JPanel widget is an invisible component that you can use to group other components.
- A JButton widget displays a button.
- To handle the *event* that occurs when the user clicks a JButton widget, you can use the addActionListener method to add an *action listener*, or *event listener*.
- An *event handler* is a method that responds to GUI events such as a button click.
- A *layout manager* lays out the components in a container and controls how they respond to events such as resizing the window.
- The BorderLayout and FlowLayout managers are two of the most commonly used layout managers.
- A JTable widget displays tabular data, such as tables from a database.
- A *table model* provides tabular data for a JTable widget. The easiest way to create a table model is to extend the AbstractTableModel class.
- A JScrollPane adds scrollbars to a component such as a JTable widget so that the user can scroll through the component if it doesn't all fit in the container.
- You can use the methods of the JOptionPane class to displays a dialog box that contains an informational message or a dialog box that confirms an operation.

22

How to develop a GUI with Swing (part 2)

In the last chapter, you learned how to create the main window for the Product Manager application, including the table that displays the products that are stored in the database. In this chapter, you'll learn how to create a data entry form that allows you to add new products to the database or to edit existing ones.

How to work with labels and text fields

Most applications use one or more *text fields*, also known as *text boxes*, to allow the user to enter data. In addition, most applications use one or more *labels* to display text that labels the text boxes for the user. In figure 22-1, for example, the Add Product window contains three labels and three text fields.

How to work with labels

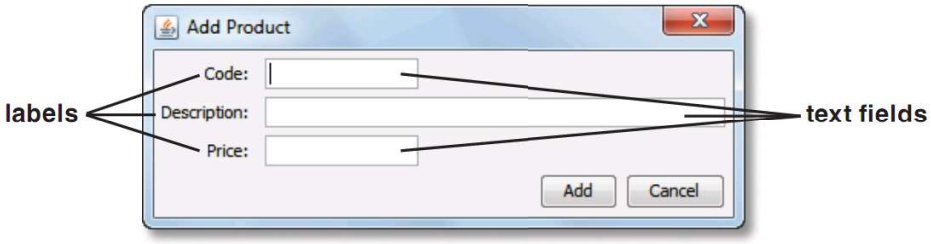
You can use the JLabel class to create a label. For example, the first label in this figure displays "Code:" to label the text field that contains the product code.

The first example creates a label and adds it to a panel. Here, the first statement uses the constructor of the JLabel class to create a label and set its text. However, it's also possible to use the setText method to set the text of a label. Then, the second statement adds the label to the panel.

The second example shows how to create a label and add it to a panel in a single statement. Here, the code doesn't create a variable that refers to the JLabel object. As a result, you can't call any methods of the JLabel class later. However, this is usually fine since you rarely need to call methods from a label.

Although the labels in this figure display text, a label can also display an image. For example, you can use a label to display an icon that identifies a text field, or to display a photograph of a product. However, using images in labels is not covered in this book.

A window that contains labels and text fields



A common constructor and method of the JLabel class

Constructor	Description
`JLabel(String)`	Constructs a new JLabel and sets the text displayed by the label to the specified string.
Method	**Description**
`setText(String)`	Sets the text displayed by the label to the specified string.

How to create a JLabel object and add it to a container

```
JLabel codeLabel = new JLabel("Code:");
panel.add(codeLabel)
```

A common way of adding a JLabel to a container

```
panel.add(new JLabel("Code:"));
```

Description

- A JLabel component defines a *label*, which is a non-editable widget that typically displays text that labels other components such as text fields.
- Since you rarely need to call any methods from a JLabel object, it's a common practice to call the constructor of the JLabel class from within an add method. This does not create a variable that refers to the JLabel object.
- A JLabel object can also display an image. However, that's not covered in this book.

Figure 22-1 How to work with labels

How to work with text fields

The JTextField class defines a text field that can display text to the user and allows the user enter or edit text. Figure 22-2 summarizes some of the most common constructors and methods of the JTextField class. Then, it presents some examples that use these constructors and methods.

The first example creates a new text field named codeField. Then, it uses the setColumns method to set the width of this text field to approximately 20 characters. This means that the width of the text field depends on the font size for the text field. However, some layout managers ignore this setting. As a result, you can't always count on it to work correctly. In that case, you can use the setPreferredSize and setMinimumSize methods as described later in this chapter.

The second example shows how to create the same text field as the first example. However, this example uses the constructor to set the number of characters.

The third and fourth examples show how to get and set the text that's stored in a text field. Here, the third example uses the getText method of the text field to get the string that's stored in the text field and assign it to a String variable. Conversely, the fourth example uses the setText method of the text field to store the specified string in the text field.

By default, a text field is editable. As a result, the user can enter text into it or edit the text that's already in it. However, if you want to create a read-only text field, you can pass a false value to its setEditable method as shown in the fifth example. Then, the user can't enter text into it.

So, why wouldn't you use a label if you want to display text that the user can't edit? One reason is that the user can select and copy text from a read-only text field. This is useful if you want to display some text to the user that they can copy to the clipboard and paste somewhere else, but you don't want them to be able to edit the text. For example, you might want to allow the user to copy and paste an exception stack trace into an email to help the programmer debug a problem.

The setEnabled method works similarly to the setEditable method as shown in the sixth example. However, it also grays out the text box and doesn't allow selecting or copying text from the field. This is useful if you want to prevent users from entering data into a text field until after they have completed other steps.

The setPreferredSize and setMinimumSize methods provide another way besides the setColumns method to set the width of the text field. Like the setColumns method, some layout managers ignore these methods. As a result, you can't always count on them to work correctly. Unlike the setColumns method, these methods set the width in pixels, and they are common to all components. Later in this chapter, you'll see an example of how to use these methods to fix a common problem with Swing layout.

Common constructors and methods of the JTextField class

Constructor	Description
`JTextField(columns)`	Creates a text field with the specified number of columns (characters).
`JTextField(text)`	Creates a text field containing the specified text.
`JTextField(text, columns)`	Creates a text field that starts with the specified text and contains the specified number of columns (characters).
Method	**Description**
`getText(text)`	Gets the text contained by the text field.
`setText(text)`	Sets the text displayed by the text field.
`setColumns(int)`	Sets the width of the text field in columns (characters). Many layout managers ignore this setting.
`setEditable(boolean)`	Sets whether the user can edit the text or not.
`setEnabled(boolean)`	Sets whether the text field is enabled. If set to false, the text field is disabled, which means that it is grayed out and the user can't change the text in it.

How to create a text box for approximately 20 characters

```
JTextField codeField = new JTextField();
codeField.setColumns(20);
```

Another way to create the same text box

```
JTextField codeField = new JTextField(20);
```

How to get text from a text box

```
String productCode = codeField.getText();
```

How to set text in a text box

```
codeField.setText("java");
```

How to create a read-only text box

```
codeField.setEditable(false);
```

How to disable a text box

```
codeField.setEnabled(false);
```

Description

- A JTextField component defines a *text field* that displays text to the user and allows the user to enter text. This type of component is also known as a *text box*.

Figure 22-2 How to work with text fields

How to use the GridBagLayout manager

The GridBagLayout manager is more complicated than the layout managers described in the previous chapter. However, it's also more powerful and more flexible.

An introduction to the GridBagLayout manager

Using the GridBagLayout manager, you can lay out components in a grid like the one shown in in figure 22-3. Of course, the heavy black lines don't appear in the application when it's run. We added them to help you visualize the six cells of this grid. Here, the grid has two columns and three rows. As a result, the grid has six cells, and each cell holds a component.

With the GridBagLayout manager, you can create layouts that resize properly to fit the window size, and automatically look right if the user is using a font size other than the default. However, it can often require a fair amount of tweaking and testing to get a layout to look exactly right.

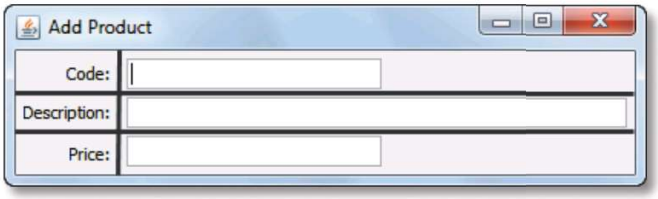
Using the GridBagLayout manager typically involves using the three classes described in this figure. These classes are stored in the java.awt package and are designed to work together.

First, the GridBagLayout class defines the layout manager that you provide to the container's setLayout method. This works the same as it does for the BorderLayout and FlowLayout managers described in the previous chapter.

Second, the GridBagConstraints class contains multiple fields that control how components are laid out within the grid. This includes how much padding they have, how they resize if the window is resized, and so on. You'll learn more about using this class in the next figure.

Third, the Insets class defines the amount of space between components in the grid, as well as the amount of space between components and the edge of the container. This space is often referred to as *padding*.

Six components in a grid of two columns and three rows



The package that contains the layout manager classes

```
java.awt
```

Three classes for working with the GridBagLayout manager

Class	Description
`GridBagLayout`	Creates a layout manager that can lay out components in a grid.
`GridBagConstraints`	Specifies how the GridBagLayout manager lays out its components. This object controls horizontal and vertical space, whether components resize or not, how they resize, and several other aspects of component behavior.
`Insets`	Specifies the amount of padding that the GridBagConstraints object applies between a component and the edge of the container.

Description

- The GridBagLayout manager is more complicated than the layout managers presented in the previous chapter. However, it's also the most powerful and flexible layout manager in Swing.
- To lay out components in a grid, you can use the GridBagConstraints class with the GridBagLayout class as shown in the next figure.
- To specify the padding between components, you can use the Insets class with the GridBagConstraints class as shown in figure 22-5.

Figure 22-3 An introduction to the GridBagLayout manager

How to lay out components in a grid

Figure 22-4 shows how to use a GridBagConstraints object to control the layout of components inside a container that uses the GridBagLayout manager. This figure starts by showing some fields of the GridBagConstraints class.

To start the gridx and gridy fields control the position on the grid where the component is placed. In this figure, for example, both the gridx and gridy fields are set to a value of 0 for the "Code:" label. As a result, the layout manager places this label in the cell at the top left corner of the grid. Here, the coordinates represent cells, not pixels.

The first text field has a gridx value of 1 and a gridy value of 0. As a result, the layout manager places this text field in the cell that's at the intersection of the second column and the first row. The second text field has a gridx value of 1 and a gridy value of 1. As a result, the layout manager places this label in the cell at the intersection of the second row and the second column. And so on.

The anchor field controls where the layout manager places the component within the cell if the cell is larger than the component. By default, the layout manager places the component in the center of the cell. However, you can specify an anchor value of LINE_END to align the component with the right side of the cell. In this figure, for example, the first three labels have an anchor field that's set to this value. Similarly, you can use an anchor value of LINE_START to align the component with the left side of the cell. In this figure, for example, all of the text fields have an anchor field that's set to this value.

Although LINE_START and LINE_END are two of the most common values for the anchor field, many other possible values exist. To view a complete list, see the API documentation for the anchor field.

The LINE_START and LINE_END values are relatively new to Java. Prior to their introduction, developers commonly used the WEST and EAST values instead of the LINE_START and LINE_END values. However, for new development, it's recommended that you use the newer LINE_START and LINE_END values because they are more portable across regions.

The gridwidth and gridheight fields determine how many cells the component takes up. As a result, setting the gridwidth field to 2 causes the component to take up two horizontal cells rather than one. In this figure, for example, the fourth label takes up two horizontal cells rather than one.

The code example begins by creating a JPanel object. By default, a JPanel object uses the FlowLayout manager. That's why the second statement uses the setLayout method to set the layout manager to the GridBagLayout manager.

After setting the layout manager, this code creates a new GridBagConstraints object. In practice, it's common to use the variable named c for this, so that's what this code does.

After creating the GridBagConstraints object, this codes sets the gridx, gridy, and anchor fields of the GridBagConstraints object. Then, it adds the components to the panel, and passes the GridBagConstraints object as the second parameter.

This code resets the gridx, gridy, and anchor fields before adding every label. Reusing a GridBagConstraints object as shown in this figure makes it easy to view and edit the coordinates and alignment of each component. However, if

Some common fields of the GridBagConstraints class

Field	Description
`gridx, gridy`	Sets the x and y coordinates of the component in the grid where 0, 0 is the top left cell of the grid.
`anchor`	Sets where the component is displayed if the component is smaller than the cell that is its display area. The most commonly used are LINE_START and LINE_END. In older versions of Java, these were called WEST and EAST.
`gridwidth, gridheight`	Sets the number of horizontal and vertical cells that the component occupies.

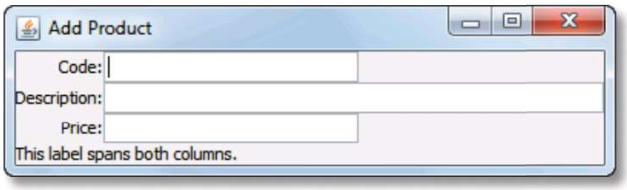
Code that uses a GridBagConstraints object

```
JPanel panel = new JPanel();
panel.setLayout(new GridBagLayout());

GridBagConstraints c = new GridBagConstraints();

c.gridx = 0; c.gridy = 0; c.anchor = GridBagConstraints.LINE_END;
panel.add(new JLabel("Code:"), c);
c.gridx = 1; c.gridy = 0; c.anchor = GridBagConstraints.LINE_START;
panel.add(codeField, c);
c.gridx = 0; c.gridy = 1; c.anchor = GridBagConstraints.LINE_END;
panel.add(new JLabel("Description:"), c);
c.gridx = 1; c.gridy = 1; c.anchor = GridBagConstraints.LINE_START;
panel.add(descriptionField, c);
c.gridx = 0; c.gridy = 2; c.anchor = GridBagConstraints.LINE_END;
panel.add(new JLabel("Price:"), c);
c.gridx = 1; c.gridy = 2; c.anchor = GridBagConstraints.LINE_START;
panel.add(priceField, c);
c.gridx = 0; c.gridy = 3; c.anchor = GridBagConstraints.LINE_START;
c.gridwidth = 2;
panel.add(new JLabel ("This label spans both columns."), c);
```

The resulting layout



Description

- You can reuse the same GridBagConstraints object for multiple widgets. However, you must reset any values you don't want to use in the next widget.

Figure 22-4 How to lay out components in a grid

you don't reset values, you may end up accidentally reusing the values that were set for a previous component, which may cause strange layout behavior. As a result, you need to make sure to reset all values, or you need to create a new GridBagConstraint object for each component.

How to add padding

By default, a GridBagConstraints object doesn't provide any padding between components or between components and the edge of the container. That's why the components displayed in the previous figure don't have any padding between them. And that's why there's no padding between the components and the edge of the container.

To make this layout more visually appealing, you can use the insets field of the GridBagConstraints object to apply some padding as shown in figure 22-5. This field uses an Inserts object to determine the amount of padding around a component. Here, the four parameters of the Insets object are the top, left, bottom, and right padding in pixels. If you think of a clock, these values are in a counterclockwise direction, starting at the top of the clock.

In this figure, the insets field of the GridBagConstraints object is set to an Insets object that specifies 5 pixels of padding for the top, left, and right of the component and no padding for the bottom of the component. Since this code uses the same Insets object for all components in the container, the label at the bottom of the layout touches the bottom edge of the container. However, the rest of the components are separated by 5 pixels of padding. This padding makes this layout more visually appealing than the layout in the previous figure.

When working with an Insets object, there are two important things to remember. First, unlike a GridBagConstraints object, you can't change the values of an Insets object and then reuse it. If you do, your layout will behave strangely. Second, if you set 5 pixels of top spacing and 5 pixels of bottom spacing, the spacing between components is actually 10 pixels. In other words, the layout uses five pixels of space at the bottom of one component and five pixels of space at the top of the component in the next row for a total of 10 pixels.

Of course, the same thing is true with left and right spacing. In this figure, for example, the left side of the longest label has a small amount of padding between its first letter and the left edge of the window. Similarly, the right edge of the longest text field has a small amount of padding between it and the right edge of the window. However, there is twice the padding between the end of the labels and the beginning of the text fields. That's because this area uses the padding from the right side of the label and the padding from the left side of the text box.

Most of the time, you only need to use the insets field of the GridBagConstraints object to specify external padding. However, if you need to specify internal padding, you can use the ipadx and ipady fields shown in this figure. These fields determine how much padding is placed inside the component. For example, if you set the ipadx and ipady values of a JButton component to 10, the layout displays 10 pixels of padding between the text of the JButton component and the edges of the button.

More common fields of the GridBagConstraints class

Field	Description
`insets`	Uses an Insets object to specify how much external padding should be applied to the component.
`ipadx, ipady`	Sets how much internal padding to add to the width and height of the component. The value applies to both sides.

The Insets class

Constructor	Description
`Insets(top, left, bottom, right)`	Specifies the top, left, bottom, and right padding in pixels.

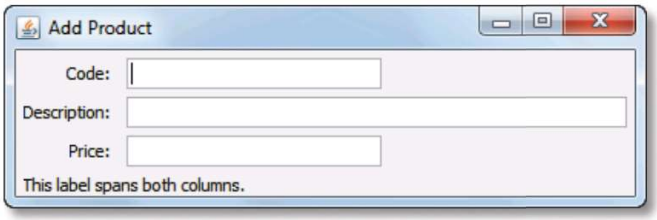
To create an Insets object and add it to a GridBagConstraints object

```
GridBagConstraints c = new GridBagConstraints();
c.insets = new Insets(5, 5, 0, 5);

c.gridx = 0; c.gridy = 0; c.anchor = GridBagConstraints.LINE_END;
panel.add(new JLabel("Code:"), c);
c.gridx = 1; c.gridy = 0; c.anchor = GridBagConstraints.LINE_START;
panel.add(codeField, c);

// the rest of the code that adds the labels and text boxes
```

The example from the previous figure with insets



Description

- If you forget the order of the parameters for the constructor of the Insets class, it may help to remember that they start at the top and go counterclockwise.

Figure 22-5 How to add padding

How to avoid a common pitfall

Figure 22-6 demonstrates a common pitfall with GUI layout and shows how to avoid it. To start, this figure shows the same form shown in the previous figure after a user resized it by shrinking its horizontal width slightly. This caused all of the text fields to collapse. What's going on here?

Basically, if the container isn't wide enough to display the component at its specified size, the GridBagLayout manager assigns the minimum size to the component instead. By default, a component has a minimum size of 0. As a result, this causes the width of the text fields to completely collapse.

One common way to solve this problem is to use the setMinimumSize method on the text fields to set the preferred and minimum sizes to the same value as shown in this figure. Here, the first statement creates a new Dimension object that's 100 pixels wide and 20 pixels tall. Then, the next four statements set the preferred and minimum sizes of the text fields for the product code and price to that Dimension object.

After that, the sixth statement creates a second new Dimension object with a width of 250 and a height of 20. Then, the next two statements sets the preferred and minimum sizes of text field for the product description to that Dimension object. As a result, the code and price text fields are 100 pixels wide and the description text field is 250 pixels wide.

If you run this code now and shrink the width of the window, the text fields don't collapse. However, the labels begin to collapse as the GridBagLayout manager tries to give the text fields their minimum size and still stay within the window. If you continue to press the issue by shrinking the window even further, the edges of the text fields eventually disappear outside the window.

Because the labels start collapsing, you might say that we solved one problem, but created another. In some sense, that's true. And this shows that the GridBagLayout manager is complex and often requires a lot of tweaking to get a layout to behave exactly as intended.

For the Product Manager application, the user is unlikely to have any need to resize this window. As a result, the solution described in this figure is probably adequate.

Another possible solution to this problem is to make it so the user can't resize the window. To do that, you can call the setResizable method of the window and pass it a value of false. Again, for the Product Manager application, this solution is acceptable. However, you should use this technique with caution as most users expect to be able to resize a window, especially if it helps the window fit better on their display.

There is a third possible solution that's more elegant but also more difficult to code. This solution involves using the weightx, weighty, and fill fields of the GridBagConstraints object to control how the components are resized when the user resizes a window. The weightx and weighty fields control how to distribute any extra horizontal or vertical space among the components. This allows you to specify a percentage for each component. In addition, the fill field determines whether the component resizes if the container is resized. A component can be set to resize either horizontally, vertically, both, or not at all.

A common problem that occurs after horizontal resizing

A constructor of the Dimension class

Constructor	Description
`Dimension(width, height)`	Specifies the width and height of a component in pixels.

Two methods you can use to fix this problem

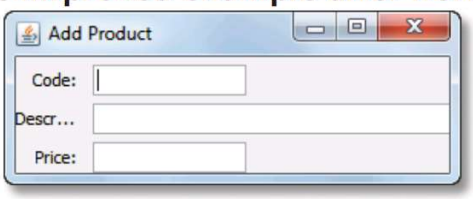
Method	Description
`setMinimumSize(Dimension)`	Specifies the minimum width and height of a component in pixels.
`setPreferredSize(Dimension)`	Specifies the preferred width and height of a component in pixels.

Code that sets the preferred and maximum sizes

```
Dimension dim1 = new Dimension(100, 20);
codeField.setPreferredSize(dim1);
codeField.setMinimumSize(dim1);
priceField.setPreferredSize(dim1);
priceField.setMinimumSize(dim1);

Dimension dim2 = new Dimension(250, 20);
descriptionField.setPreferredSize(dim2);
descriptionField.setMinimumSize(dim2);
```

The improved example after horizontal resizing



Three ways to fix this problem

1. Use the window's setResizable method to prevent the user from resizing the window. However, you should use this technique sparingly since users often expect to be able to resize a window.
2. Use the component's setPreferredSize and setMinimumSize methods. This isn't a perfect solution, but it's acceptable for most applications.
3. Use the weightx, weighty, and fill fields of the GridBagConstraints class to control how the components are resized. This technique can yield the best results, but it also requires the most tweaking.

Description

- When a window is resized horizontally, components (especially text fields) can sometimes collapse to a size of zero. This is a common problem.

Figure 22-6 How to avoid a common pitfall

How to code a data entry form

Most applications need to get input from the user. For example, it's common for an application to need a user to enter or edit the data that's stored in a database. To do this, most applications use a special type of window known as a *form*.

Often, you want to use a special type of window known as a *dialog*, or *dialog box*, to display a form. To do that in Swing, you can use the JDialog component. This allows you to create custom dialogs that work similarly to the built-in dialogs that are available from the JOptionPane class.

How to create a custom dialog

Figure 22-7 begins by showing one of the most common JDialog constructors. This constructor takes three parameters. The first parameter specifies the *owner*, or *parent*, of the dialog. Typically, the parent of a dialog is the JFrame component for the main window of the application.

The second parameter specifies the title of the dialog. This works the same as setting the title for a JFrame component.

The third parameter determines whether the dialog is *modal*. If a dialog is modal, Java prevents the user from interacting with any other part of the application until the user has responded to the dialog. You should use modal dialogs with caution as they can annoy users if they aren't necessary. As a general rule, you should only use a modal dialog when the application can't continue the current operation until the user responds to the dialog.

The code example creates a NameDialog class that defines a dialog. This class begins by extending the JDialog class. Then, it creates a private JTextField component named dialogNameField.

The constructor of the NameDialog class takes one argument: the parent frame. Within this constructor, the first statement uses the super keyword to call the constructor of the JDialog class and it passes the parent frame, a title of "Name Dialog", and a boolean value of true. As a result, the NameDialog component is a modal dialog. The second statement in the constructor continues by setting the default close operation to DISPOSE_ON_CLOSE. As a result, the dialog is closed and its resources are released if the user closes it with the window's close button. Without this code, Java would hide the dialog but not free its resources. And the third statement sets the layout of the dialog to the FlowLayout manager.

The constructor continues by creating the components for the dialog and adding them to the dialog. First, it creates a text field and sets its width to 20 characters. Then, it creates two buttons and adds action listeners to them. Here, the code for the action listener of the Okay button is shown in the next figure, and the code for action listener of the Cancel button just calls the dialog's dispose method. This closes the dialog and releases any resources it was using.

After adding components to the dialog, this code calls the pack method to set the size of the dialog to the minimum size needed to hold the components. Finally, this code calls the setVisible method to make the dialog visible.

A commonly used JDialog constructor

Constructor	Description
`JDialog(parent, title, isModal)`	Specifies the parent frame of the dialog, the title of the dialog, and whether the dialog is modal.

A class for a custom dialog

```
public class NameDialog extends JDialog {

    private JTextField dialogNameField;

    public NameDialog(java.awt.Frame parent) {
        super(parent, "Name Dialog", true);
        this.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        this.setLayout(new FlowLayout());

        dialogNameField = new JTextField();
        dialogNameField.setColumns(20);

        JButton okayButton = new JButton("Okay");
        okayButton.addActionListener((ActionEvent) -> {
            // TODO: Add code here
        });

        JButton cancelButton = new JButton("Cancel");
        cancelButton.addActionListener((ActionEvent) -> {
            dispose();
        });

        this.add(new JLabel("Name:"));
        this.add(dialogNameField);
        this.add(okayButton);
        this.add(cancelButton);
        this.pack();

        this.setVisible(true);
    }
}
```

A statement that displays this dialog

```
new NameDialog(this);
```

The dialog produced by the above code

Description

- A *modal* dialog prevents the user from interacting with other parts of the application until they have responded to the dialog.
- On most operating systems, a JDialog component is displayed on top of its parent frame, has fewer window controls than a JFrame (for example, no maximize or minimize buttons), and does not get its own entry in the taskbar.

Figure 22-7 How to create a custom dialog

At first glance, it might seem there aren't that many differences between a JFrame component and a JDialog component. However, on closer inspection, there are few important differences. First, a dialog is designed to have a parent window. As a result, a dialog is displayed on top of its parent window, even if the dialog is not the active window. Second, on most operating systems, a dialog has fewer window controls. On Windows, for example, a dialog has a close button but no minimize, maximize, or restore buttons. Third, on most operating systems, a dialog doesn't get an entry in the taskbar or window list. As a result, you can't switch directly to the dialog using window switch commands. Instead, you need to switch to the window that owns the dialog. Fourth, as described earlier, a dialog can be modal.

How to pass data between a dialog and its parent

Figure 22-8 shows how to get and set the data contained in a dialog. In this figure, the dialog only works with the data that's stored in a single text field. However, it's possible for a dialog to work with the data that's stored in multiple components. For example, the Product form shown later in this chapter works with the data that's stored in three text fields.

The first two examples show how to set data in a dialog. Here, the first example shows a modified version of the constructor from the previous figure. However, it includes an additional parameter for a String object that specifies the name that's stored in the dialog. This parameter provides a way to pass data to the dialog. Then, the code in the constructor can set the data on the components in the dialog. In this figure, for example, the constructor calls the setText method of the text field to set it to the name parameter.

The second example shows code that calls the constructor of the dialog. This code is stored in the parent frame. It begins by passing the keyword named *this* to refer to the object itself. In this case, this means that this code is passing the frame itself to the NameDialog constructor. Next, it gets the value from a text field in the frame by calling its getText method and passing the String object that's returned as the second parameter. As a result, the text field in the dialog displays the same text as the text field in the frame.

The third and fourth examples show how to get data from a dialog. To do that, the third example adds a public method named setNameText to the frame that the dialog can call. This method takes a String object as a parameter. Within this method, the code sets the text on the text field in the frame by calling its setText method and passing it the name parameter.

The fourth example shows the code for the action listener of the dialog's Okay button. Within this action listener, the first statement gets a reference to the NameFrame object by calling the getOwner method and casting the Window object that's returned to the NameFrame type. This cast is necessary because the Window type doesn't include the setNameText method. Then, the second line of code calls the setNameText method on the frame, and passes it the text that's stored in the text field of the dialog. Finally, the third line of code calls the dispose method of the dialog. This closes the dialog and frees any resources it was using.

How to set data in a dialog

A constructor of the dialog class that allows you to set data

```
public NameDialog(java.awt.Frame parent, String name) {
    super(parent, "Name Dialog", true);
    this.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    this.setLayout(new FlowLayout());

    dialogNameField = new JTextField();
    dialogNameField.setColumns(20);
    dialogNameField.setText(name);
```

Code from the frame that calls this constructor

```
new NameDialog(this, frameNameField.getText());
```

A method of the Window class

Method	Description
`getOwner()`	Returns a Window object for the owner or parent of a dialog. You typically cast this Window object to a more specific type.

How to get data from a dialog

A method in the frame class that the dialog can call

```
public void setNameText(String name) {
    frameNameField.setText(name);
}
```

Code in the dialog that calls this method

```
okayButton.addActionListener((ActionEvent) -> {
    NameFrame frame = (NameFrame) getOwner();
    frame.setNameText(dialogNameField.getText());
    dispose();
});
```

Description

- Within the dialog class, you can code a constructor that allows the main window to set data in the dialog.
- Within the class for the main window, you can code a method that allows the dialog to return set data to the main window.
- To close a dialog, you can call its dispose method. This closes the dialog object and frees any resources that it's using.

Figure 22-8 How to pass data between a dialog and its parent frame

The Product form

This chapter finishes by presenting the code for the Product form. This form is the data entry form used by the main window of the Product Manager application described in the previous chapter.

The user interface

Figure 22-9 shows the user interface for the data entry form. To start, if the user clicks on the Add button in the main window, this form has a title of Add Product, doesn't display any text in its three text fields, and has an Add button. However, if the user clicks on the Edit button in the main window, this form has a title of Edit Product, displays the text for the selected product in its text fields, and has a Save button.

If the user doesn't enter valid data for a product, the application uses the JOptionPane class to display a built-in dialog that asks the user to enter valid data. In this figure, for example, the last dialog displays the message that's displayed if the user leaves one of the text fields blank. However, this application would display a different message if the user had entered an invalid number for the product price.

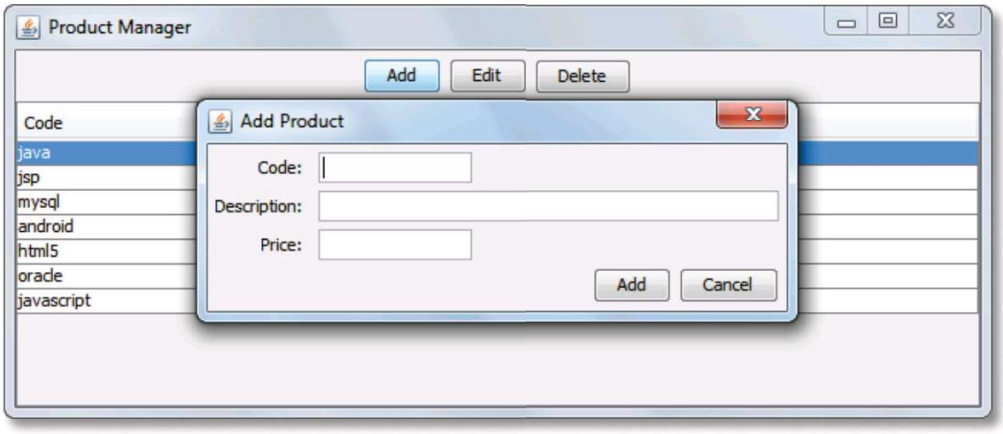
The Product form inherits the JDialog class. As a result, it is a custom dialog. On Windows, a dialog only includes a close button in its upper-right corner. In other words, it doesn't include minimize, maximize, or restore buttons like the main window for the Product Manager application. However, this is specific to the Windows operating system. These buttons may vary depending on the operating system.

The Product form uses the BorderLayout manager. Within this layout, it contains two panels.

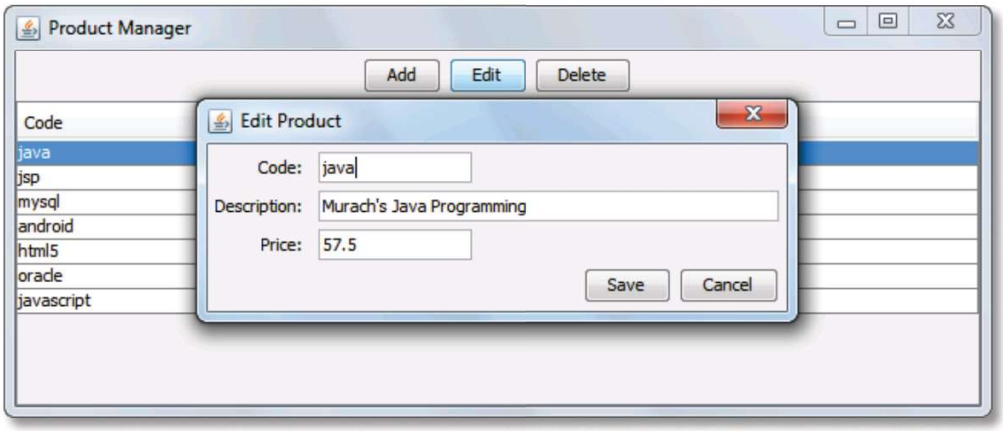
The first panel contains the three labels and their corresponding text fields. This panel uses the GridBagLayout manager to lay out these labels and text fields. Then, this panel is added to the center of the border layout. As a result, it gets all available space that isn't taken up by any other components.

The second panel uses a right-aligned FlowLayout manager. This panel lays out the two buttons for the form. Then, this panel is added to the south area of the border layout. As a result, this panel gets its preferred height, and the width expands to take up the entire horizontal space of the container. In this case, that means that the panel's height is just tall enough to hold the buttons, and the panel's width is the entire width of the window.

The Add Product dialog



The Edit Product dialog



One of the data validation dialogs

Description

- The Product Manager application uses the same dialog to add a new product or to edit an existing product. The only difference in appearance is the title of the dialog and the text that's displayed on the first button.
- When you add or edit a product, the Product form uses built-in dialogs to display validation messages if the fields for the product are missing or not valid.

Figure 22-9 The user interface for the Product form

The ProductForm class

Figure 22-10 shows the code for the ProductForm class. To start, this class extends the JDialog class. This is a common way to create a custom dialog.

Within the class, the first five statements create instance variables that refer to some of the components on the form. The first three are the JTextField components that hold the product code, description, and price. The next two are the JButton components. The first button confirms the operation that adds a new product to the database or updates an existing one. The second button cancels the dialog and closes it.

The sixth statement creates an instance variable of the Product type. This variable can hold the data for the new product that's being added or the existing product that's being edited.

This class defines two constructors. Here, the first constructor displays the Add Product form that's used to add a new product, and the second constructor displays the Edit Product form that's used to edit an existing product.

The first constructor accepts three parameters: (1) the parent frame, (2) the dialog title, and (3) whether the dialog is modal. Within this constructor, the first statement passes these parameters to the constructor of the super class. Then, the second statement calls the initComponents method shown later in this class.

The second constructor adds a fourth parameter: a Product object that contains the data for the selected product. Within this constructor, the first statement calls the previous constructor by using the keyword named *this*, which refers to an instance of the current class. Then, the second statement sets the product instance variable to the product parameter.

After that, this constructor sets the text of the confirm button to "Save". That way, the button on the Edit Product form is labeled "Save" instead of "Add". This is necessary because the initComponents method shown later in this class sets the text of this button to "Add". In addition, this constructor uses the setText methods on the three text fields to set their values to reflect the appropriate values stored in the Product object.

The initComponents method contains the code that performs most of the setup of the dialog. The method begins by creating new instances of JTextField and JButton components for the instance variables declared at the beginning of the class. Next, the code sets the default close operation of the dialog to DISPOSE_ON_CLOSE. As a result, Java disposes the dialog and frees its resources when the user clicks on the close button of the dialog.

The ProductForm class

```
package mma.ui;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.WindowConstants;

import mma.business.Product;
import mma.db.DBException;
import mma.db.ProductDB;

public class ProductForm extends JDialog {
    private JTextField codeField;
    private JTextField descriptionField;
    private JTextField priceField;
    private JButton confirmButton;
    private JButton cancelButton;

    private Product product = new Product();

    public ProductForm(java.awt.Frame parent, String title, boolean modal) {
        super(parent, title, modal);
        initComponents();
    }

    public ProductForm(java.awt.Frame parent, String title,
            boolean modal, Product product) {
        this(parent, title, modal);
        this.product = product;
        confirmButton.setText("Save");
        codeField.setText(product.getCode());
        descriptionField.setText(product.getDescription());
        priceField.setText(Double.toString(product.getPrice()));
    }

    private void initComponents() {
        codeField = new JTextField();
        descriptionField = new JTextField();
        priceField = new JTextField();
        cancelButton = new JButton();
        confirmButton = new JButton();

        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

Figure 22-10 The ProductForm class (part 1 of 4)

The next two lines of code create Dimension objects and assign them a width of 100 pixels and 300 pixels respectively, with both having a height of 20 pixels. Then, the code sets the preferred size and minimum size of the code and price text fields to the shorter Dimension object. It also sets the preferred and minimum size of the description text field to the longer Dimension object. In both cases, this prevents the pitfall described earlier in this chapter where the text fields collapse to a width of zero if the user resizes the dialog.

The next section of the initComponents method creates the two buttons. To start, it sets the text of the first button to "Cancel" and adds an action listener to it that calls a method named cancelButtonActionPerformed that's shown later in this class. Then, it sets the text of the second button to "Add" and adds an action listener to it that calls a method named confirmButtonActionPerformed. However, if the user is editing an existing product instead of adding one, the setText method call here is overridden by the setText method call in the second constructor as discussed on the previous page.

The next section creates a new JPanel to store the text fields and their corresponding labels. To start, this code sets the layout manager for the panel to the GridBagLayout manager. Then, it adds the text fields and labels for the form to the panel. To make working with the GridBagConstraints object easier, the calls to the panel's add method use a helper method called getConstraints that's shown later in this class. This method takes three parameters: An x coordinate for the grid row, a y coordinate for the grid column, and an anchor value.

The next section creates another JPanel for the buttons. To start, this code sets the layout manager to a FlowLayout manager with right alignment. Then, it adds the confirm and cancel buttons to the layout.

The last section adds the panel that contains the labels and text fields to the center area of the dialog's BorderLayout manager. Then, it adds the panel that contains the buttons to the south area of the dialog's BorderLayout manager. Like a JFrame component, a JDialog component uses the BorderLayout manager by default. As a result, it isn't necessary to explicitly set the layout. Finally, the last statement uses the pack method to size the dialog by packing it so it's just large enough to hold all of the components.

Note that none of this code sets the location of this form or makes this form visible. As a result, the parent frame must do that. For example, the main frame of the Product Manager application could display the Add Product form like this:

```
ProductForm productForm = new ProductForm(this, "Add Product", true);
productForm.setLocationRelativeTo(this);
productForm.setVisible(true);
```

This displays a modal version of this form in the middle of the main window.

The getConstraints method creates and returns a GridBagConstraints object with the specified parameters. This method begins by creating a new GridBagConstraints object. Then, it sets the insets field to a new Insets object that has five pixels of spacing on the top, left, and right, but no spacing on the bottom. Next, it sets the gridx, gridy, and anchor fields of the GridBagConstraints object to the corresponding parameters. Finally, it returns the new GridBagConstraints object.

The ProductForm class (continued)

```
        Dimension shortField = new Dimension(100, 20);
        Dimension longField = new Dimension(300, 20);
        codeField.setPreferredSize(shortField);
        codeField.setMinimumSize(shortField);
        priceField.setPreferredSize(shortField);
        priceField.setMinimumSize(shortField);
        descriptionField.setPreferredSize(longField);
        descriptionField.setMinimumSize(longField);

        cancelButton.setText("Cancel");
        cancelButton.addActionListener((ActionEvent) -> {
            cancelButtonActionPerformed();
        });

        confirmButton.setText("Add");
        confirmButton.addActionListener((ActionEvent) -> {
            confirmButtonActionPerformed();
        });

        // JLabel and JTextField panel
        JPanel productPanel = new JPanel();
        productPanel.setLayout(new GridBagLayout());
        productPanel.add(new JLabel("Code:"),
                getConstraints(0, 0, GridBagConstraints.LINE_END));
        productPanel.add(codeField,
                getConstraints(1, 0, GridBagConstraints.LINE_START));
        productPanel.add(new JLabel("Description:"),
                getConstraints(0, 1, GridBagConstraints.LINE_END));
        productPanel.add(descriptionField,
                getConstraints(1, 1, GridBagConstraints.LINE_START));
        productPanel.add(new JLabel("Price:"),
                getConstraints(0, 2, GridBagConstraints.LINE_END));
        productPanel.add(priceField,
                getConstraints(1, 2, GridBagConstraints.LINE_START));

        // JButton panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
        buttonPanel.add(confirmButton);
        buttonPanel.add(cancelButton);

        // add panels to main panel
        setLayout(new BorderLayout());
        add(productPanel, BorderLayout.CENTER);
        add(buttonPanel, BorderLayout.SOUTH);
        pack();
    }

    private GridBagConstraints getConstraints(int x, int y, int anchor) {
        GridBagConstraints c = new GridBagConstraints();
        c.insets = new Insets(5, 5, 0, 5);
        c.gridx = x; c.gridy = y; c.anchor = anchor;
        return c;
    }
```

Figure 22-10 The ProductForm class (part 2 of 4)

The next two methods in the class are the event handlers for the buttons. The first event handler is called if the user presses the cancel button. The statement within this method calls the dispose method of the dialog to close the dialog and free its resources.

The second event handler is called if the user presses the confirm button. To start, it uses an if statement that calls the validateData method shown later in this class. This method makes sure the user entered all required data and that all of the data is valid. If so, the event handler continues by calling the setData method shown later in this class. This method gets the data from the form and sets that data in the product instance variable.

The second if statement in the event handler checks whether the application is editing an existing product or adding a new one. To do that, it checks the text of the confirm button. If the text of the confirm button is "Add", this code calls the doAdd method shown later in this class. This method adds a new product to the database. However, if the text of the button is not "Add", this code calls the doEdit method shown later in this class. This method updates an existing product in the database.

The validateData method makes sure that the user has entered data for all three text fields and that the user has entered a valid number for the product price. To start, this code gets String objects for the product code, description, and price variables by calling the getText methods on the corresponding text fields. Then, this code uses an if statement to check whether all three fields in the form contain values. If they don't, the variables defined earlier are null or empty. In that case, the code uses the JOptionPane class to display a dialog that contains a message that asks the user to fill in all fields. Then, it returns a value of false.

This code also verifies that the price the user entered is a valid number. To do this, it codes a statement that attempts to convert the price string to a double value within a try statement. If this statement isn't able to convert the string to a double value, it throws a NumberFormatException, which causes the catch block to be executed.

The catch block uses the JOptionPane class to display a dialog that contains a message that informs the user that the data entered in the price field is not a valid price. Then, it attempts to move the focus to the text field for the price. However, this statement might not be able to move the focus. As a result, you shouldn't count on it working. Finally, the catch block returns a value of false.

If the user has entered data for all three fields, and the price is a valid number, the validateData method returns a value of true. This indicates that the data in the form is valid.

The setData method begins by getting the data contained in the form's text fields by calling their getText methods. Once again, to get a double value for the price, this code needs to convert the String object returned by the getText method to a double value. Then, this method sets this data in the instance variable for the Product object by using its set methods.

For this method, the code doesn't catch the NumberFormatException that may be thrown by the parseDouble method because the validateData method shown earlier has already made sure that the user has entered a valid number for the price.

The ProductForm class (continued)

```
private void cancelButtonActionPerformed() {
    dispose();
}

private void confirmButtonActionPerformed() {
    if (validateData()) {
        setData();
        if (confirmButton.getText().equals("Add")) {
            doAdd();
        } else {
            doEdit();
        }
    }
}

private boolean validateData() {
    String code = codeField.getText();
    String name = descriptionField.getText();
    String priceString = priceField.getText();
    double price;
    if (code == null || name == null || priceString == null ||
            code.isEmpty() || name.isEmpty() || priceString.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Please fill in all fields.",
                "Missing Fields", JOptionPane.INFORMATION_MESSAGE);
        return false;
    } else {
        try {
            price = Double.parseDouble(priceString);
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this,
                    "The data entered in the price field is invalid",
                    "Invalid Price",
                    JOptionPane.INFORMATION_MESSAGE);
            priceField.requestFocusInWindow();
            return false;
        }
    }
    return true;
}

private void setData() {
    String code = codeField.getText();
    String name = descriptionField.getText();
    String priceString = priceField.getText();
    double price = Double.parseDouble(priceString);
    product.setCode(code);
    product.setDescription(name);
    product.setPrice(price);
}
```

Figure 22-10 The ProductForm class (part 3 of 4)

The next two methods, doEdit and doAdd, are almost identical, except that the doEdit method calls the update method of the ProductDB class, and the doAdd method calls the add method of the ProductDB class. Then, these methods both call the dispose method to close the dialog and free its resources. Next, they call the fireDataBaseUpdatedEvent method.

The fireDatabaseUpdatedEvent notifies the main window that a new record has been added to the database or that an existing record has been updated. In turn, the main window notifies the table model that the table data needs to be updated to reflect the changes made to the database as shown in the previous chapter. To do this, the code begins by getting a reference to the parent window of the dialog by calling the getOwner method. However, this method returns a Window object. As a result, this code must cast it to the ProductManagerFrame type before it can call its fireDatabaseUpdatedEvent method.

The ProductForm class (continued)

```
    private void doEdit() {
        try {
            ProductDB.update(product);
            dispose();
            fireDatabaseUpdatedEvent();
        } catch (DBException e) {
            System.out.println(e);
        }
    }

    private void doAdd() {
        try {
            ProductDB.add(product);
            dispose();
            fireDatabaseUpdatedEvent();
        } catch(DBException e) {
            System.out.println(e);
        }
    }

    private void fireDatabaseUpdatedEvent() {
        ProductManagerFrame mainWindow = (ProductManagerFrame) getOwner();
        mainWindow.fireDatabaseUpdatedEvent();
    }
}
```

Figure 22-10 The ProductForm class (part 4 of 4)

Two methods that use the ProductForm class

Figure 22-11 shows the two methods of the ProductManagerFrame class that use the ProductForm class. These methods weren't implemented in the previous chapter, but now you can see how they work. In short, they contain the code that's executed when the user clicks on the Add or Edit button in the main window of the Product Manager application.

The code for the Add button begins by creating a modal Add Product dialog. Then, it attempts to set the location of that dialog so it's relative to the current window. In this case, the current window is the Product Manager window. On Windows, this centers the dialog on the current window. However, this is platform specific, and some operating systems may ignore this method completely.

After attempting to set the location of the dialog, this code uses the setVisible method to display the dialog to the user. As a result, this code displays an Add Product dialog like the one shown earlier in this chapter.

The code for the Edit button begins by getting an index for the selected row from the table of products. If the index is -1, this code uses the JOptionPane class to displays a dialog that contains a message that indicates that no product has been selected. Otherwise, this code gets a Product object that corresponds with the selected row in the table.

After getting a Product object, this code creates a modal Edit Product dialog and passes this Product object to the dialog. This allows the dialog to display the data for the selected product. Then, it attempts to set the location of the dialog relative to the Product Manager window. Finally, it displays the dialog to the user. As a result, this code displays an Edit Product dialog like the one shown earlier in this chapter.

Two methods that use the ProductForm class

```
private void doAddButton() {
    ProductForm productForm = new ProductForm(this, "Add Product", true);
    productForm.setLocationRelativeTo(this);
    productForm.setVisible(true);
}

private void doEditButton() {
    int selectedRow = productTable.getSelectedRow();
    if (selectedRow == -1) {
        JOptionPane.showMessageDialog(this,
                "No product is currently selected.",
                "No product selected",
                JOptionPane.ERROR_MESSAGE);
    } else {
        Product product = productTableModel.getProduct(selectedRow);
        ProductForm productForm =
                new ProductForm(this, "Edit Product", true, product);
        productForm.setLocationRelativeTo(this);
        productForm.setVisible(true);
    }
}
```

Figure 22-11 Two methods that use the ProductForm class

How to use threads with Swing

At this point, you should understand how to code an application like the Product Manager application. However, if you need to perform a long running operation in response to a GUI event, you may run into a problem similar to the one shown in figure 12-12.

A common problem

The first code example calls a method that could potentially take a long time to complete. Because this method was called in response to the user pressing a print button in the GUI, it runs on a special thread known as the *event dispatch thread*, or *EDT*. All actions that are a result of an event run on the EDT, as do all Swing operations that update the display, and respond to events.

Unfortunately, this means that the printProductList method also runs on the EDT, which results in it becoming blocked until the method returns. As a result, it can't process other events, or redraw the screen if required. This causes the application to become unresponsive to any other user events. In addition, the display can become corrupt if it needs to be repainted, such as if it is resized, or minimized and then restored.

At first, you might think you can solve this problem by running the printProductList method on a new thread using the techniques shown in the second code example. Unfortunately, it's unsafe to perform any GUI operations outside of the EDT, so this code causes a new problem, which is that this code runs the JOptionPane dialog on a new thread rather than on the EDT, which can result in display corruption.

How to solve the problem

The solution to this problem is to run the printProductList method on another thread, but perform the GUI updates on the EDT. Fortunately, Swing provides the SwingWorker class for doing just that.

The last code sample in the figure uses the SwingWorker class to solve this problem. To start, this code begins by creating a SwingWorker object using an inner class. Within the inner class, the code overrides the doInBackground method, and calls the printProductList method from it. This method runs on a worker thread, which allows the EDT to continue to process events. Although this method can return an object, this example doesn't need to do that, so it returns a null instead.

The inner class also includes a done method that's called when the doInBackground method finishes. The done method runs on the EDT. As a result, you can safely update the GUI from this method.

The SwingWorker class has other methods as well, including ones that allow you to safely update the GUI before the long running operation has finished. However, these methods aren't covered in this book. For more information, you can view the API documentation for this class or search the Internet.

Code that may cause the GUI to become unresponsive

```
private void doPrintButton() {
    printProductList();    // This method may take a long time to run.
    JOptionPane.showMessageDialog(ProductManagerFrame.this,
        "Product list has been printed.");  // This code updates the GUI.
}
```

Code that keeps the GUI responsive but shouldn't update the GUI

```
private void doPrintButton() {
    Thread printThread = new Thread() {
       // This method runs on a background thread (not the EDT).
       // As a result, the GUI remains responsive.
       // However, it isn't safe to update the GUI from this thread.
       @Override
       public void run() {
            printProductList();  // This method may take a long time to run.
            JOptionPane.showMessageDialog(ProductManagerFrame.this,
                "Product list has been printed.");    // NOT safe!
       }
    };
    printThread.start();
}
```

Code that keeps the GUI responsive and can update the GUI

```
private void doPrintButton() {
    SwingWorker worker = new SwingWorker() {
       // This method runs on a background thread (not the EDT).
       // As a result, the GUI remains responsive.
       @Override
       protected Object doInBackground() throws Exception {
           printProductList();  // This method may take a long time to run.
           return null;
       }

        @Override
        // This method runs on the EDT after the background thread finishes.
        // As a result, it's safe to update the GUI from this method.
        protected void done() {
            JOptionPane.showMessageDialog(ProductManagerFrame.this,
                "Product list has been printed.");    // Safe!
        }
    };
    worker.execute();
}
```

Description

- Any code that runs in response to a Swing event runs on a special thread called the *event dispatch thread* or *EDT*. All code that updates the GUI should run on this thread.
- If you run a task that takes a long time on the EDT, the GUI becomes unresponsive.
- If you use the techniques shown in chapter 18 to start a new thread, the GUI remains responsive. However, you can't safely update the GUI from this new thread.
- To allow the GUI to remain responsive while a long-running task runs and to be able to update the GUI when that task finishes, you can use the SwingWorker class.

Figure 22-12 How to use threads with Swing

Perspective

In this chapter, you learned all of the remaining skills you need to know to complete the Product Manager application. You learned how to use the JLabel and JTextField classes to create labels and text fields for data entry forms. You learned how to work with the GridBagLayout manager to create professional looking forms. You learned how to use the JDialog class to create a modal data entry form. And you learned how to pass data back and forth between two windows in an application.

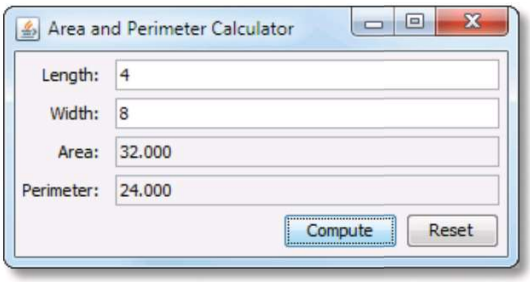
There's still plenty more to learn about working with Swing. For example, you may want to learn about other types of controls such as check boxes, radio buttons, combo boxes, and lists. You may want to learn how to display photographs or other graphics in Swing. You may want to learn about other layout managers. You may want to learn more about using the SwingWorker class to create threads in Swing. Or, you may want to learn how to use a GUI builder tool that's available from your IDE. Whatever you decide to learn next about Swing, the skills presented in the last two chapters are a solid foundation that you can build on.

Summary

- You can use the JLabel class to create *labels* that label other components on a form.
- You can use the JTextField class to create *text fields*, also known as *text boxes* that allow a user to enter data.
- You can use the GridBagLayout manager to create professional looking form layouts that lay out the components in a grid. This class has two helper classes: GridBagConstraints and Insets.
- You can use the JDialog class to create a *dialog* that stays on top of its parent window.
- A dialog can be *modal*, which means it prevents users from interacting with any other part of the application until they have responded to the dialog. However, you should only use modal dialogs when the application cannot complete an operation until the user responds to the dialog.
- Any code that runs in response to a Swing event runs on a special thread called the *event dispatch thread* or *EDT*.
- To prevent the GUI from becoming unresponsive during long running tasks, as well as make sure GUI updates happen safely on the EDT, you can use the SwingWorker class.

Exercise 22-1 Create a new GUI

In this exercise, you'll create a GUI application that calculates the area and perimeter of a rectangle based on its length and width. When you're done, the application should look like this:



Review the existing code for the application

1. Open the project named ch22_ex1_AreaAndPerimeter_start that's in the ex_starts folder.
2. Open the Rectangle class in the murach.business package and review its fields and methods.
3. Open the AreaAndPerimeterFrame class and examine the existing code. Note that:
   - This class extends the JFrame class.
   - This class has a constructor that sets up the frame and calls the initComponents method.
   - This class contains several methods such as the initComponents method that haven't been implemented.
   - This class contains a getConstraints method that has been implemented. This method works like the getConstraints method shown in this chapter.
4. Open the Main class and note that it creates a new instance of AreaAndPerimeterFrame class.
5. Run the application. This should display a frame that doesn't contain any components. Then, resize the frame to make it larger so you can see its title.

Add the components to the frame

6. Add instance variables for the four text fields and two buttons.

7. Add code to the initComponents method that initializes the frame and its components. This method should:
   - Create the four text fields.
   - Modify the text fields for the area and perimeter so the user can't edit them.
   - Set the minimum and preferred dimension for all four fields.
   - Create the two buttons.
   - Create a panel that uses the GridBagLayout manager. Then, add the four labels and text fields to this panel. To do that, you can use the getConstraints method. Finally, add this panel to the center of the frame.
   - Create a panel that uses the FlowLayout manager with right alignment. Then, add the two buttons to this panel. Finally, add this panel to the bottom of the frame.
   - Pack the frame to set its size.
8. Run the application. This should display a frame that looks like the frame shown above. However, if you click on one of the buttons, it should not perform an action.

Handle the events that occur when the user clicks the buttons

9. Add action listeners to both of the buttons. These action listeners should call the computeButtonClicked and resetButtonClicked methods.
10. Implement the computeButtonClicked method. This method should get the text that the user entered into the text fields for the length and width and attempt to convert this text to double values.
    - If the user enters an invalid length or width, this method should use a dialog to display a user-friendly error message. Then, the user can try again.
    - If the user enters a valid length and width, this method should calculate the area and perimeter and set the corresponding text fields. To do that, you can use the Rectangle class.
11. Implement the resetButtonClicked method. This method should set all four text fields to empty strings.