Packages

19

Outcomes:

By the end of this chapter you should be able to:

- *identify the role of* ***packages*** *in organizing classes*;
- *create and deploy their own packages in Java*;
- *access classes residing in their own packages*;
- *run Java applications from the command line*;
- *deploy Java applications as* ***JAR*** *files*;
- *access libraries that are not part of the standard Java framework to access data on a remote database using the* ***Java Database Connectivity (JDBC)*** *and* ***Hibernate ORM*** *technologies.*

19.1 Introduction

From as early as Chap. 2 of this book you have been familiar with the idea of a *package*, and have been using packages in order to access classes residing in external folders. In this chapter we will take a more in-depth look at Java's package concept and see how you can deploy your own applications.

19.2 Understanding Packages

A **package**, in Java, is a *named collection* of *related classes*. You have already been using packages to access pre-written classes. For example, to store objects in an ordered collection you made use of the `ArrayList` class, which resides in the `util` package. To read and write to files you used classes in the `io` package. To organise the layout of your JavaFX applications you used classes such as `Scene`

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_19

and Group from the scene package. Giving meaningful names to a set of related classes in this way makes it easy for programmers to locate these classes when required. Packages can themselves contain other packages. For example, as well as containing classes for organising the layout of your JavaFX applications, the scene

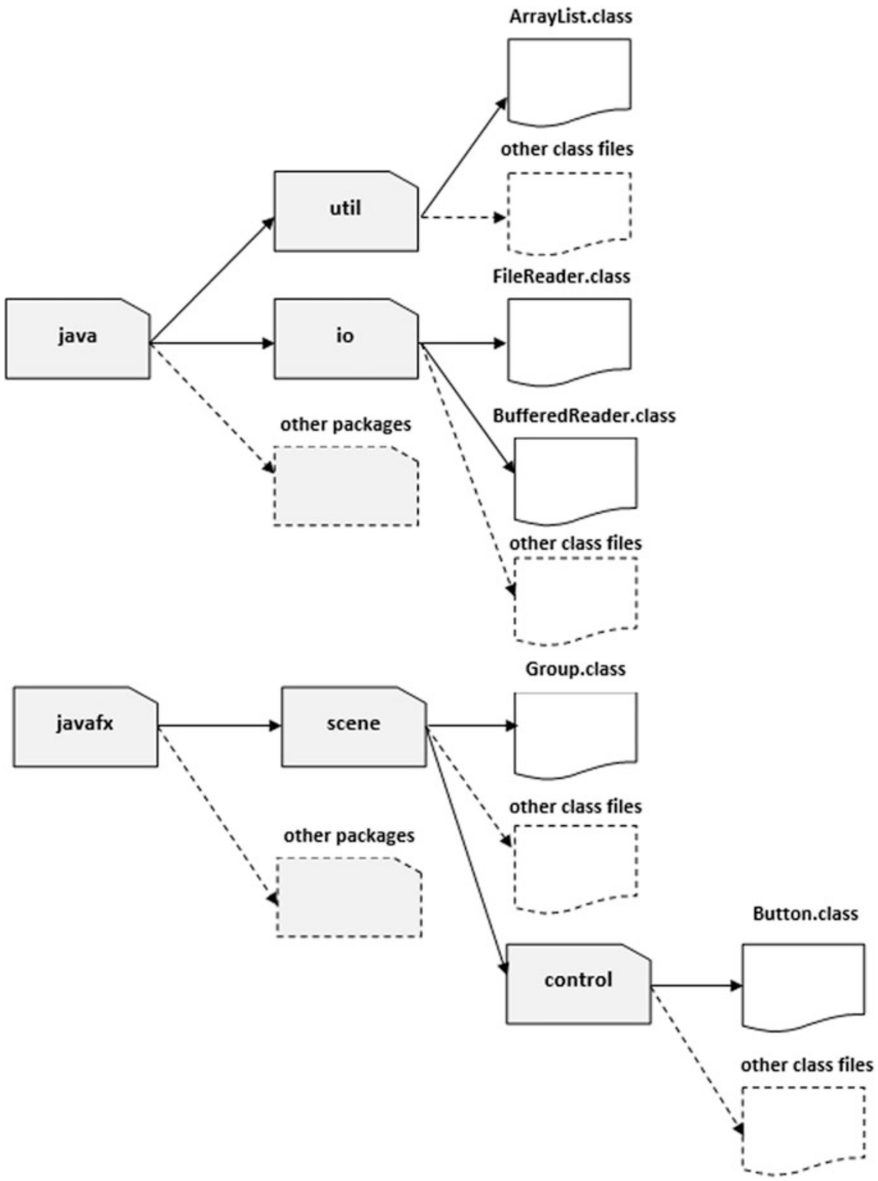


Fig. 19.1 A sample of the java package hierarchy

package also contains the `control` package, which contains JavaFX control classes such as `Button` and `Label`, since this group of classes is still logically related to JavaFX's collection of scene related files.

The package name actually corresponds to the *name of the directory* (or folder as some operating systems call it) in which all the given classes reside. All the core Java packages themselves reside in a global Java directory, named simply `java`. This directory is not itself a package but a store for other packages. Since Java was launched a few additional global directories have been developed. In particular, the `javafx` directory contains all the packages and classes required for JavaFX development. Figure 19.1 illustrates part of this hierarchy of packages.

As you can see from Fig. 19.1, packages contain class files (that is the compiled Java byte code), not source files (the original Java instructions). This means the location of the original Java source files is unimportant here. They may be in the same directory as the class files, in another directory or, as in the case of the predefined Java packages, they may even no longer be available.

19.3 Accessing Classes in Packages

Suppose you are writing the code for a new class. You will recall how you can give it access to a class contained within a package. Just referencing the class won't work. For example, let's assume a class you are writing needs a `DecimalFormat` object. The following will not compile:

```
public class SomeClass
{
   private DecimalFormat someFormatObject; // a problem here
}
```

This won't compile because the compiler won't be able to find a class called `DecimalFormat`. One way to tell the compiler where this class file resides is, as you already know, to add an **`import`** statement above the class. This class is in the text package so the following would be appropriate:

```
import java.text.DecimalFormat; // allows compiler to find the DecimalFormat.class file

public class SomeClass
{
   private DecimalFormat someFormatObject; // now this will compile
}
```

Can you see how the **`import`** statement matches the directory structure we illustrated in Fig. 19.1? Effectively the compiler is being told to look for the `DecimalFormat` class in the text directory (package), which in turn is in the java directory (whose location is already known to the Java run-time system). The location of a file is often referred to as the **path** to that file. In the Windows operating systems this path would be expressed as follows:

```
java\text\
```

In other operating systems forward slashes may be used instead of backward slashes. The Java **import** statement simply expresses this path but uses dots instead of backward or forward slashes.

The asterisk notation, that we also met in Chap. 2, allows you to have access to *all* class files in the given package. Note that there can only ever be one '.*' in an **import** statement and the '.*' must follow a package name. However, as you have already seen in previous chapters, you can have as many **import** statements as you require. Here are some examples of valid and invalid **import** statements:

```
import java.*.*; // illegal as contains more than one '.*'
import java.*;   // illegal as 'java' is not a package
import java.text.*;  // fine, allows access to all classes in text package
import javafx.scene.control.Button; // fine, allows access to the Button class in control package
```

Although there is no overhead in allowing access to all files in a package via the asterisk notation, we have used the convention of explicitly listing every class imported individually in this book for the sake of clarity.

It is actually possible to access classes from within packages *without* the need for an **import** statement. To do this, references to any such classes must be appended onto the package name itself. Returning to the DecimalFormat example, we could have removed the **import** statement and referred to the package directly in the class as follows:

```
public class SomeClass
{
   /* appending the class name onto the package name avoids the need to import
      the given package */

   private java.text.DecimalFormat someFormatObject;
}
```

The package plus class name is in fact the proper name for this class. An **import** statement just provides us with a convenient shorthand so that we do not always have to include the package name with the class name. As you can imagine, having to append the class name onto the package name every time we use a class from a package would be very cumbersome, so the **import** statement is preferable. There are times, however, when the long name is necessary.

The long class name can be useful when the class name on its own clashes with the name of another class. For example, let us assume we have developed our own class called Scanner—perhaps as part of a warehouse application. Giving this name to the class is not a great idea as there already is a Scanner class in the util package, but it is possible to choose this name if we wish. Now, let us assume that the constructor for this class takes a **double** value that represents the price of a product. The ScannerApp1 program below requires both this newly developed Scanner class and Java's Scanner class in the util package.

This *ScannerApp1* program will not compile

```
import java.util.Scanner;// Java's Scanner class

// this program will not compile!
public class ScannerApp1
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in); // Java's Scanner class
        System.out.println("Enter price: ");
        double price = keyboard.nextDouble();
        // the next line will cause a compiler error
        Scanner product = new Scanner (price); // our own Scanner class
        // more code here
    }
}
```

As you can see, we are referencing two Scanner classes here: Java's Scanner class and our newly developed Scanner class. Not surprisingly, this will result in a compiler error! In the case of the ScannerApp program, the compiler will assume the correct Scanner file is the Scanner that has been imported and used to create a keyboard object:

```
Scanner keyboard = new Scanner(System.in); // this will compile ok
```

The second use of Scanner, to create an object called product from our own class, will then be the one that will cause the compiler error, as the compiler is expecting this Scanner class to be the Scanner class from the util package, which has no such constructor for receiving a double:

```
Scanner product = new Scanner (price);// this will cause a compiler error
```

To resolve this name clash, we can use the extended package name in the code to differentiate between the two classes. Let's see how to do this in the ScannerApp2 program below:

This *ScannerApp2* program will compile

```
// note we do not import Scanner from util

public class ScannerApp2
{
    public static void main(String[] args)
    {
       // use full path name to Scanner file in util
       java.util.Scanner keyboard = new java.util.Scanner(System.in);
       System.out.println("Enter price: ");
       double price = keyboard.nextDouble();
       // the next line will not now cause a compiler error
       Scanner product = new Scanner (price); // our own Scanner class
       // more code here
    }
}
```

You can see the full path name has been used for Java's `Scanner` class in the code itself

```
java.util.Scanner keyboard = new java.util.Scanner(System.in);
```

Now we no longer have a name clash with our class and the complier can process references to both classes.

19.4 Developing Your Own Packages

You might be surprised to know that all the classes that you have developed so far already reside in a *single* package. This may seem strange as you didn't instruct the compiler to add your classes to any package. In fact, what actually happens is that if you don't specifically ask your classes to be put in a package, then they all get added to some large unnamed package.

In order to locate and deploy your class files easily, and avoid any name clashes in the future, it would be a good idea to use named packages to organize your classes.

As an example, let's go back to our *Hostel* application from Chaps. 11 and 12 and create a unique package in which to put our class files—we will call this package `hostelApp`.[1] To instruct the compiler that you wish to add the classes that make up this application into a package called `hostelApp`, simply add the following **`package`** command at the top of each of the original source files:

```
package hostelApp;
```

This line instructs the compiler that the class file created from this source file must be put in a package called `hostelApp`. Here, for example, is the `Payment` class with this **`package`** line added:

```
package hostelApp; // add this line to the top of the source file

public class Payment
{
   // as before
}
```

[1] We will stick to the standard Java convention of beginning package names with a lower-case letter.

When you compile this class you will find a directory called `hostelApp` will have been created and the resulting `Payment.class` file will be placed into this directory.[2]

All the classes that make up this *Hostel* application (such as `Tenant`, `Hostel` and so on) will need to be amended in a similar way:

1. Add the following line to the top of each source file

```
package hostelApp;
```

2. Ensure that the compiled class files are placed in the `hostelApp` directory.

TestPackage

```
import hostelApp.Payment; // import a class from the hostelApp

public class TestPackage
{
    public static void main(String[] args)
    {
        Payment p = new Payment ("January", 725); // access the Payment class
        System.out.println(p);// calls Payment's toString method
    }
}
```

Now, if we were developing applications in the future that wish to make use of any of the classes in our `hostelApp` package we could import them like classes from any other package. For example, the `TestPackage` program below imports the `Payment` class from the `hostelApp` package and then creates a `Payment` object before printing it on the screen:

19.5 Package Scope

Up until now we have declared all our classes to be **`public`**. This has meant they have been visible to all other classes. When we come to adding our classes into our own packages, this becomes particularly important. This is because *classes can be made visible outside of their package only if they are declared as* **`public`**. Unless they are declared as **`public`**, classes by default have what is known as **package** scope. This means that they are visible *only to other classes within the same package*.

[2]If you are developing a class from scratch that you wish to add into a package, your Java IDE can be used so that the `package` line is inserted into your code for you and the required directory structure created. If you are using your Java IDE to *revisit* classes previously written outside of a package (as in this example), you may need to ensure that the resulting directory structure is reflected in the project you are working in. Refer to your IDE's documentation for details about how to do this.

Not all classes in the package need be declared as **`public`**. Some classes may be part of the implementation only and the developer may not wish them to be made available to the client. These classes can have **package** scope instead of **`public`** scope. In this way, packages provide an extra layer of security for your classes.

In the case of our *Hostel* application, we might choose to make the `Hostel` class **`public`**, and keep all the other files required in this application hidden within the package by giving them package scope. To give a class package scope, just remove the **`public`** modifier from in front of the class declaration. For example, returning to the `Payment` class, we can give this package scope as follows:

```
package hostelApp; // this class is added into the package

class Payment // this class has package scope
{
   // as before
}
```

Now, when the `hostelApp` package is imported into another class, this other class has access only to the `Hostel` class; not to classes like `Payment` which are hidden in the package with package scope. This is demonstrated in the code fragment below:

```
import hostelApp.*; // this imports only the public classes in the package

public class SomeOtherClass
{
   // the next line will not compile as Payment is hidden in the hostelApp package
   Payment p = new Payment("January", 725);
}
```

19.6 Running Applications from the Command Line

Way back in Chap. 1 we discussed the process of compiling and running Java programs. If you remember, we said that if you are working within a Java IDE you will have simple icons to click in order to carry out these procedures. If, however, you are working from a command line, like a DOS prompt for example, you would use the **`javac`** command (followed by the name of the source file) to compile a source file and **`java`** (followed by the name of a class) to run an application.[3] As a simple example, here is the very first program we showed you back in Chap. 1:

[3]When installing Java an **environment variable** called PATH should automatically have been set so the operating system can locate the necessary Java tools to compile, run and deploy Java programs. If this has not been done refer to your operating system's instructions for setting this variable The PATH variable should point to the *bin* folder in your JDK folder, for example *C:\ProgramFiles\Java\jdk1.8.0_121\bin.*

HelloWorld

```
public class HelloWorld
{
    public static void main(String[] args)
    {
         System.out.println ("Hello world");
    }
}
```

Notice this program was not placed in a package. Assuming you are in the directory containing this source code file, you can compile this class using the `javac.exe` tool as follows:

javac HelloWorld.java

This will produce a Java class file (`HelloWorld.class`). Assuming you are now in the directory that contains this class file you can then run this program with the `java.exe` tool as follows:

java HelloWorld

As noted in Chap. 1, when running a class file you do not include the `.class` extension.

When you run a class that resides in a package you must amend this slightly. As an example, let's once again consider the *Hostel* application. When you run an application you must run the class that contains the main method. The appropriate class in our application is the JavaFX `Hostel` class. Remember, we have now added this class to a `hostelApp` package:

```
package hostelApp; // Hostel class part of the hostelApp package
// import statements here
public class Hostel extends Application
{
   // as before
}
```

If we were in the *hostelApp* folder that contained this class we could try running the `Hostel` class from the command line as follows:

java Hostel

Unfortunately, this won't work as the system won't be able to find a class of the given name. In order to run a class that is contained within a package you must append the class name onto the name of the package (with a '.' symbol). You will now need to be in the directory *above* the package directory. So in this case you will need to be in the directory above the *hostelApp* directory. You can then run the `Hostel` class by using the following command:

java hostelApp.Hostel`

If you do not wish to run the Java commands from the directories containing the relevant files you can set an environment variable called the **CLASSPATH** so that it points to the relevant directory (or directories) and then run these commands from any directory. See your operating system's documentation for how to do this.

Before we move on, let's just stop and have a look at the parameter that we always give to `main` methods:

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

As you know, this means that `main` is given an array of `String` objects as a parameter. How are these `String` objects passed on to `main`? Up until now we have not discussed them at all. Well, values for these strings can be passed to `main` when you run the given class from the command line. Often, as in our previous program, there is no need to pass any such strings and this array of strings is effectively empty. Sometimes, however, it is useful to send in such parameters. They are sent to `main` from the command line by listing the strings, one after the other after the name of the class as follows:

java ClassName firstString secondString otherStrings

As you can see, the strings are separated by spaces. Any number of strings can be sent in this way. For example, if a program were called `ProcessNames`, two names could be sent to it as follows:

java ProcessNames Aaron Quentin

Notice that, were the strings to contain spaces, they must be enclosed in quotes:

java ProcessNames "Aaron Kans" "Quentin Charatan"

These strings will be placed into `main`'s array parameter (`args`), with the first string being at `args[0]`, the second at `args[1]` and so on. The number of strings sent to `main` is variable. The `main` method can always determine the number of strings sent by checking the `length` of the array (`args.length`). The `ProcessNames` class takes the array of strings and displays them on the screen.

ProcessNames

```
public class ProcessNames
{
    public static void main(String[] args)
    {
       if (args.length != 0)// check some arguments have been sent
       {
            // loop through all elements in the 'args' array
            for (int i=0; i<args.length; i++)
            {
                // access individual strings in array
                System.out.println("hello " + args[i]);
            }
       }
    }
}
```

Notice, to keep things simple we have not added this class to a package. We can run this program from the command line as follows:

java ProcessNames “Batman and Robin” Superman

Notice “`Batman and Robin`” needed to be surrounded by quotes as it has spaces in it, whereas `Superman` does not. Running this program would produce the obvious result:

```
hello Batman and Robin
hello Superman
```

19.7 Deploying Your Packages

A very common way of making your packages available to clients is to convert them to JAR files. A JAR file (short for Java Archive) has the extension `.jar` and is simply a compressed file. Most IDEs provide a means of creating JAR files but JAR files can also be created from the command prompt with the **`jar.exe`** tool. This tool is provided with the JDK and also with most standard IDEs. Assuming that we are in the directory above the `hostelApp` package then the correct statement to create a suitable JAR file is:

jar cvf hostel.jar hostelApp

As you can see there are various switches that are used with the `jar` program. The ones used above have the following effect:

c: create a new JAR file;
v: provide full (verbose) output to report on progress;
f: provide a name for the JAR file.

After these switches comes the name of the output file—`hostel.jar` in our case. Finally,we must list the files we wish to be included. In the above example we require only the package directory, `hostelApp`, but you may have additional files here such as sound and image files that are not part of the package.

If you are working in a graphics environment, and there is a JVM installed on your computer, then it is possible to create a JAR file that will run the program by double-clicking on its icon. We call such a JAR file an *executable* JAR file.

While it is possible to use Java tools from the command line to create such executable JAR files, as you could see when we showed you how to create a standard JAR file from the command line, this can be quite verbose and prone to error. So, these days, IDEs will provide very simple tools to both create a JAR file and to make this JAR file executable if you choose. We provide instructions on the accompanying web-site on how to do this for a popular Java IDE.

19.8 Adding External Libraries

Sometimes it is necessary to use libraries that are not part of the standard Java framework. One of the most common examples of this is in the development of applications that require the use of data held on a database which is stored either locally or on another machine on the network. In this section we will briefly explore two technologies—**Java Database Connectivity (JDBC)** and **Hibernate ORM** (Object/Relational Mapping), often referred to simply as *Hibernate*.

It is not our intention here to give a detailed explanation of how these technologies are used, but rather to give a concrete example of how it is possible to use libraries that are not part of the Java framework. This should, however, also serve to whet the appetites of those of you who wish to learn how to access a database via a Java application.

19.8.1 Accessing Databases Using JDBC

Database manufacturers provide JDBC *drivers* by means of which Java programs can access their databases. A driver is a piece of software that enables communication between two programs, or between a software program and a piece of hardware, by translating the output of one program into a form understood by the other one.

In order to use a particular driver we would download it from the manufacturer's website in the form of a `.jar` file. We would then need to ensure that the CLASSPATH points to the location of this file; this is most easily done by adding the `.jar` file to the run-time or compile-time configurations within an IDE such as NetBeans™ or Eclipse™. Instructions for doing this will be found within the documentation for the particular IDE.

In this chapter we are going to be referring to a MySQL™ database. The driver, which is called `Driver.class`, is contained within the `.jar` file. At the time of writing the current version is:

```
mysql-connector-java-5.1.45-bin.jar
```

In today's version of Java, all we have to do to make the driver accessible is to add it to the CLASSPATH as described above. In previous versions we would have needed to include the following code (which, as you see, refers to the driver by its full path name):

```
try
{
   Class.forName("com.mysql.jdbc.Driver");
}

catch(ClassNotFoundException e)
{
}
```

Table 19.1 The *products* table

Field	Data type	Length
serialNumber (*key field*)	char	(25)
make	char	(10)
description	char	(25)
price	decimal	(10, 2)

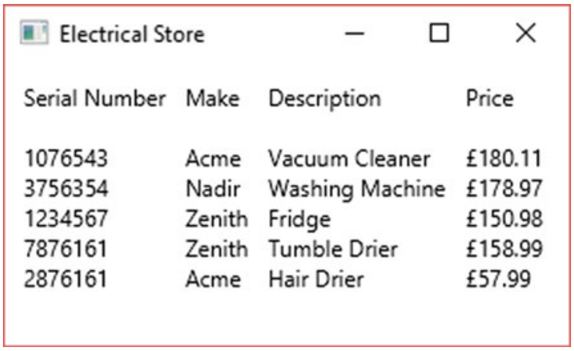
Java provides a package known as `java.sql`. This package provides the means by which our Java programs can contain commands written in standard SQL (Structured Query Language), which is the well-established means of writing database instructions. In this chapter we are not going to teach you SQL, but will assume you are familiar with some basic commands.

For our example we have set up a little database called *ElectricalStore* that contains a table called *products*. This table is described below (Table 19.1); it is assumed that you are familiar with relational databases and the data types available.

We have populated this database, and, in order to query it, we have developed a class called `ProductQuery`. This class, once an instance of it is created, executes just one SQL statement:

```
select * from products;
```

Those of you who are familiar with SQL will know that this query retrieves all the fields from all the records in the *products* table. The information obtained is then displayed in a text area as you can see from Fig. 19.2.



```
Serial Number  Make    Description      Price

1076543        Acme    Vacuum Cleaner   £180.11
3756354        Nadir   Washing Machine  £178.97
1234567        Zenith  Fridge           £150.98
7876161        Zenith  Tumble Drier     £158.99
2876161        Acme    Hair Drier       £57.99
```

Fig. 19.2 Displaying the information from the *ElectricalStore* database

Take a look at the `ProductQuery` application below, and then we will go through it with you.

ProductQuery

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class ProductQuery extends Application

{
    public static final String URL = "jdbc:mysql://localHost/ElectricalStore?useSSL=true";
    public static final String USERNAME = "Kub";
    public static final String PASSWORD = "SydneyPaper";

    @Override
    public void start(Stage stage)
    {
        // create VBoxes to act as display columns
        VBox data1 = new VBox();
        VBox data2 = new VBox();
        VBox data3 = new VBox();
        VBox data4 = new VBox();

        data1.getChildren().add(new Text("Serial Number\n"));
        data2.getChildren().add(new Text("Make\n"));
        data3.getChildren().add(new Text("Description\n"));
        data4.getChildren().add(new Text("Price\n"));

        // configure the visual components
        HBox root = new HBox(10);
        root.setPadding(new Insets(10));
        root.getChildren().addAll(data1, data2, data3, data4);
        Scene scene = new Scene(root, 300, 150);
        stage.setTitle("Electrical Store");
        stage.setScene(scene);
        stage.show();

        Connection con;
        Statement st;
        ResultSet result;

        try
        {
            // connect to the database
            con = DriverManager.getConnection(URL, USERNAME, PASSWORD);

            // create an SQL statement
            st = con.createStatement();

            // execute an SQL query
            result = st.executeQuery("select * from products");

            while(result.next()) // move to next record
            {
                // retrieve and display first field
                data1.getChildren().add(new Text (result.getString(1)));
                // retrieve and display second field
                data2.getChildren().add(new Text(result.getString(2)));
                // retrieve and display third field
                data3.getChildren().add(new Text(result.getString(3)));
                // retrieve and display fourth field
                data4.getChildren().add(new Text("£" + result.getString(4)));
            }
        }

        catch(SQLException e) // handle the SQLException
        {

        }
    }

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

You can see that we have declared some `String` constants as attributes—these will be explained shortly, when they are used.

In the `start` method we have created four `VBoxes`, one for each field of the data records—these will be lined up horizontally in an `HBox`. We have then declared a `Connection` object, a `Statement` object and a `ResultSet` object, none of which you have previously encountered. These are part of the `java.sql` package; their use will become clear in a moment.

We now declare a **`try`** … **`catch`** block because all of the methods we are going to use to communicate with the database throw `SQLExceptions`. The first thing we need to do within this block is to establish a connection with the database:

```
con = DriverManager.getConnection(URL, USERNAME, PASSWORD);
```

The `getConnection` method of `DriverManager` establishes a connection with the database referred to by the parameter `URL`, which was defined in the attribute declarations as:

```
public static final String URL = "jdbc:mysql://localHost/ElectricalStore?useSSL=true";
```

This is the correct format for the MySQL database called *ElectricalStore* which resides on the local machine and which uses the SSL security protocol. Other databases will require a slightly different format, the details of which can be found in the documentation for that product. Note that *localhost* is the way in which operating systems refer to the local machine—it is in fact an alias for IP (Internet Protocol) address 127.0.0.1, the normal loopback IP. If the database were located on another machine on the network, then this would be replaced by its name or IP address.[4] If the port number is required, this is placed after the name of the database separated by a colon—for example `Electrical Store:3306`. As you can see, the `getConnection` method receives, in addition to the URL (uniform resource locator), the user name and password; if this is not required, there is a version of `getConnection` that accepts the URL only (some databases allow the username and password to be embedded in the URL).

The method returns a `Connection` object, which we have assigned to the attribute `con`. A `Connection` object created in this way has a number of methods that allow communication with the database. One of these methods is called `createStatement`, and it is the next one we use:

```
st = con.createStatement();
```

[4]The system administrator will, of course, have had to set up the correct permissions for the database.

As you saw, we previously declared a `Statement` object, `st`, and this is now assigned the return value of the `createStatement` method. A `Statement` object is used for executing SQL statements and returning their results; in the next line we use its `executeQuery` method:

```
result = st.executeQuery("select * from products");
```

The data returned by executing the query is assigned to a `ResultSet` object, `result`. A `ResultSet` object holds a tabular representation of the data, and a pointer is maintained to allow us to navigate through the records. The `next` method moves the pointer to the next record, returning **`false`** if there are no more records. The individual fields are returned with methods such as `getString`, `getDouble` and `getInt`. You can see how we have used these methods in the `ProductQuery` class:

```
while(result.next()) // move to next record
{
   data1.getChildren().add(new Text(result.getString(1)));
   data2.getChildren().add(new Text(result.getString(2)));
   data3.getChildren().add(new Text(result.getString(3)));
   data4.getChildren().add(new Text("£" + result.getString(4)));
}
```

You can see that, as we are using `VBox`es to display our data, we have created a `Text` object for each string and added this to the correct `VBox`.

The version of `getString` that we have used here takes an integer representing the position of the field—in our example 1 is the *serialNumber*, 2 is *make* and so on. There is also a version of `getString` that accepts the name of the field. So, for example, we could have used, for the second field:

```
data2.getChildren().add(new Text(result.getString(make)));
```

Since all we are doing is displaying the data we used `getString` for the last field, even though it holds numeric data—this saved us the trouble of doing any formatting on it. If we had wished to do any processing with this data, we could have used the `getDouble` method to retrieve it as a **`double`** rather than a `String`.

19.8.2 Accessing Databases Using Hibernate

Hibernate is a more recent technology than JDBC; it allows us to store and retrieve whole objects from a database.

If you create your Hibernate project within Netbeans™ you will find that the necessary libraries will be added to the project. Otherwise you will need to download the files from the Hibernate website. Normally this will be in the form of a `.zip` file which contains a folder called *required*. In this folder you will find the `.jar` files that you will need to add to the CLASSPATH. As with the JDBC example in the previous section, you will need to add the relevant driver to the run-time configuration. We will use the same MySQL™ database as before, so you will need the same driver as in the previous section.

The first thing you will need to do is create a Java class for the objects that we will be dealing with—in our case we will need a `Product` class:

Product

```
public class Product
{
    private String stockNumber;
    private String manufacturer;
    private String item;
    private double unitPrice;

    public Product(String stockNumberIn, String manufacturerIn, String itemIn, double unitPriceIn)
    {
        stockNumber = stockNumberIn;
        manufacturer = manufacturerIn;
        item = itemIn;
        unitPrice = unitPriceIn;
    }

    public Product()
    {

    }

    public String getStockNumber()
    {
        return stockNumber;
    }

     public void setStockNumber(String stockNumberIn)
    {
        stockNumber = stockNumberIn;
    }

    public String getManufacturer()
    {
        return manufacturer;
    }

    public void setManufacturer(String manufacturerIn)
    {
        manufacturer = manufacturerIn;
    }


    public String getItem()
    {
        return item;
    }

    public void setItem(String itemIn)
    {
        item = itemIn;
    }

    public double getUnitPrice()
    {
        return unitPrice;
    }

    public void setUnitPrice(double unitPriceIn)
    {
        unitPrice = unitPriceIn;
    }
}
```

As you can see, the attributes of this class correspond to the fields in the database that you saw in the last section. Hibernate expects there to be `set-` and `get-` methods for these attributes, and also expects there to be an empty constructor.

You might be asking how the application will know which attribute in the Java class corresponds to which field of the database. The answer is that this information needs to be supplied in an XML mapping file, usually named `hibernate.hbm.xml`. Our file looks like this:

hibernate.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
                                        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name = "Product" table = "products">
        <id name = "stockNumber" column = "serialNumber"/>
        <property name = "manufacturer"  column = "make"/>
        <property name = "item" column = "description"/>
        <property name = "unitPrice" type = "double" column = "price"/>
  </class>
</hibernate-mapping>
```

If you are using an IDE wizard to create your Hibernate application, you will find that the first two lines are added into your file automatically—otherwise you can do it manually.

It is assumed here that you know a little about XML—but even if you don't, it is not hard to see how the attributes are mapped onto field names. Notice that the *id* tag specifies which attribute corresponds to the key field, while the *property* tag deals with the other attributes.

There is one more thing we need to do before writing our Hibernate application, which is to write a configuration file (normally called `hibernate.cfg.xml`) that will provide the information needed about the database. Here is ours for the MySQL database we described in the previous section:

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                                          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/ElectricalStore?useSSL=true</property>
    <property name="hibernate.connection.username">Kub</property>
    <property name="hibernate.connection.password">SydneyPaper</property>
  </session-factory>
</hibernate-configuration>
```

As before, using a wizard will cause the first two lines to be inserted for you. The wizard will then allow you to add the rest of the information via a design dialogue screen, or you can do it manually.

The example above is self explanatory. We have needed to add only three properties, the URL that points to the database together with the name and password. Other properties might be necessary to add, depending on the system; for example if the specific driver name were required, that could be added (for a MySQL database) as follows:

```
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
```

Now we come to the application itself. Our little program, which we have called `ProductQuery2`, will do only as much as the one in the previous section, namely to display all the information about the current items held.

Once you have taken a look at it, we will explain what it is all about.

ProductQuery2

```
// accessing a database using Hibernate

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import java.util.ArrayList;
import org.hibernate.query.Query;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

public class ProductQuery2 extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create VBoxes to act as display columns
        VBox data1 = new VBox();
        VBox data2 = new VBox();
        VBox data3 = new VBox();
        VBox data4 = new VBox();

        data1.getChildren().add(new Text("Serial Number\n"));
        data2.getChildren().add(new Text("Make\n"));
        data3.getChildren().add(new Text("Description\n"));
        data4.getChildren().add(new Text("Price\n"));

        HBox root = new HBox(10);
        root.setPadding(new Insets(10));
        root.getChildren().addAll(data1, data2, data3, data4);

        // configure the stage
        Scene scene = new Scene(root, 300, 150);
        stage.setTitle("Electrical Store");
        stage.setScene(scene);
        stage.show();

        // create a Configuration object
        Configuration cfg = new Configuration();

        // link the configuration object to the database properties
        cfg.configure("hibernate.cfg.xml");

        // specify the mapping file
        cfg.addResource("hibernate.hbm.xml");


        // create a Session object to act as an interface between the Java application and Hibernate
        ServiceRegistry serviceRegistry
                = new StandardServiceRegistryBuilder().applySettings(cfg.getProperties()).build();
        SessionFactory  sessionFactory = cfg.buildSessionFactory(serviceRegistry);
        Session session = sessionFactory.openSession();

        // query the database
        Query query = session.createQuery("from Product");

        // create a list of products from the query
        ArrayList<Product> list = (ArrayList) query.list();

        // display the product details for each product in the list
        for(Product pr : list)
        {
            data1.getChildren().add(new Text(pr.getStockNumber()));
            data2.getChildren().add(new Text(pr.getManufacturer()));
            data3.getChildren().add(new Text(pr.getItem()));
            data4.getChildren().add(new Text("£" + pr.getUnitPrice()));
        }

        // close the session
        session.close();
        sessionFactory.close();
        StandardServiceRegistryBuilder.destroy(serviceRegistry);

    }

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

You can see from the code that after creating and configuring the visual components, the first thing we have done is to create a `Configuration` object which read the information from the XML files we created:

```
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
cfg.addResource("hibernate.hbm.xml");
```

We now use this object to create a `Session` object, which is the interface between the Java application and the database. It is the equivalent of a JDBC `Connection` object. To do this you follow the rather verbose steps below:

```
ServiceRegistry serviceRegistry
                 = new StandardServiceRegistryBuilder().applySettings(cfg.getProperties()).build();
SessionFactory sessionFactory = cfg.buildSessionFactory(serviceRegistry);
Session session = sessionFactory.openSession();
```

The `Session` class has a number of methods, one of which is `createQuery`, which returns a Hibernate query. Hibernate has its own query language, **Hibernate Query Language (HQL)** which is similar to SQL; HQL queries are translated into SQL in order to query the database. The only one of these we will show you here is the "From" clause which will retrieve entire objects. You can see how it is used in our application:

```
Query query = session.createQuery("from Product");
```

The Hibernate `Query` class has a method called `list`; this returns a `List` object which we have stored as an `ArrayList` of `Products`:

```
ArrayList<Product> list = (ArrayList) query.list();
```

It is now a simple matter of scrolling through the list of products, displaying all the fields as we do so:

```
for(Product pr : list)
{
     data1.getChildren().add(new Text(pr.getStockNumber()));
     data2.getChildren().add(new Text(pr.getManufacturer()));
     data3.getChildren().add(new Text(pr.getItem()));
     data4.getChildren().add(new Text("£" + pr.getUnitPrice()));
}
```

Before the program terminates, we need to close it down properly. This involves three steps:

```
 session.close();
 sessionFactory.close();
 StandardServiceRegistryBuilder.destroy(serviceRegistry);
```

We mentioned earlier that it is also possible to store whole objects as well as to retrieve them. If you have created a `Product` object, say `pr`, then you could store it with the following lines of code, which should come after the routine for opening a session.

```
Transaction tx = session.beginTransaction();

try
{
    session.save(pr);
    tx.commit();
}

catch(Exception e)
{
    if(tx!=null) tx.rollback();
}

finally
{
    session.close();
    sessionFactory.close();
    StandardServiceRegistryBuilder.destroy(serviceRegistry);
}
```

For a write operation such as this we need to start a transaction, which we do with the `beginTransaction` method of `Session`. We then store our object with the `save` method of `Session` (this has to be in **`try`**. … **`catch`** block). Hibernate does not automatically commit the save to the database, so we need to do this by calling the `commit` method of `Transaction`. Should there be any problem we have rolled back the change within the **`catch`** block. We have conveniently placed our close routines in the **`finally`** clause.

You might be wondering which technology to use, JDBC or Hibernate. There is no rule about this, and opinions differ. However, as a general rule of thumb, if all you want to do is to query an existing database with SQL then JDBC might be simpler. However, if your main aim to write a Java program that stores its data in a database then Hibernate might be the best solution.

19.9 Self-test Questions

1. What role do *packages* have in the development of classes?

2. Identify valid and invalid **`import`** statements amongst the following list:

   ```
   import java.*;
   import javafx.scene.*;
   import java.util.Scanner;
   import javax.scene.control.Button;
   import java.application.Application;
   import java.text.*.*;
   ```

3. Consider the following outline of a class, used in a computer game, that makes reference to JavaFX's `Button` class:

```
public class GameController
{
     private Button myButton;
     // more code here
}
```

   At the moment the line referencing the `Button` class will not compile. Identify three different techniques to allow this class with a `Button` attribute to compile.

4. What is the purpose of the **CLASSPATH** environment variable?

5. You were asked to develop a time table application in programming exercise 8 of Chap. 8. Later, in programming exercise 3 of Chap. 14 you were asked to enhance this application with exceptions. Finally you were asked to develop a JavaFX interface for this application in programming exercise 6 of Chap. 17.

   (a) How would you place the classes that make up the time table application into a package called `timetableApp`?
   (b) What is meant by **package** scope and how would you give the `Booking` and `TimeTable` classes package scope?
   (c) How would you run this application from the command line?
   (d) What is the purpose of a JAR file and how would you create a JAR file for the `timetableApp` package form the command line?

6. Explain the fundamental differences between the JDBC and the Hibernate technologies in terms of their approach to accessing databases.

19.10 Programming Exercises

1. Make the changes discussed in this chapter so that the *Hostel* application is now part of a package called `hostelApp`.

2. Run the *Hostel* application from the command line.

3. Use your IDE to create an executable JAR file for your *Hostel* application then run your *Hostel* application by clicking this executable JAR file.

4. Make the changes to the time table application, discussed in self-test question 5 above, so that the application can be run from the command line and by clicking an executable JAR file.

5. Write a program that accepts a list of names from the command line and then sorts and displays these names on the screen. Run this program from the command line with a variety of names.

6. There are several Java packages that we have not yet explored. Browse your Java documentation to find out about what kind of classes these packages offer. For example, the `lang` package contains a class called `Math`, which has a **`static`** method called random designed to generate random numbers. There is also a random number class, `Random`, in the `util` package. Read your Java documentation to find out more about these random number generation techniques. Then write a program that generates five lottery numbers from 1 to 50 using:

   (a) the **`static`** `random` method of the `Math` class in the `lang` package;

   (b) the random number class, `Random`, in the `util` package.

7. In the previous chapter we developed several programs—for example `TextFileTester`—that stored and accessed data by creating files within the application. See if you can convert one of these programs so that the data is stored in a database to which you have access. In order to do this you will need to either have an account on a database system within your organisation, or to be running a database server (such as a MySQL server) locally.