5. Java Interfaces

Up to this point in the guide, the reader will not have encountered any examples of a *Java interface*. This chapter explains how members, in addition to those accessible to a class by means of inheritance, can be introduced into Java classes by means of a special class known as a Java interface.

5.1 What is a Java Interface?

Examples discussed in previous chapters readily show that *method implementations* embody the application logic of a Java application and *method invocations* run the application. It almost goes without saying, therefore, that the methods implemented in the classes defined in a Java application are fundamental to its operation.

In the light of the content of previous chapters, *what do we know about a method written by a developer or one that is specified by the Java API?* To answer this question, let us re-visit two of the methods in the themed application.

Firstly, the documentation and declaration of the **takeItemOnLoan** method of the **DvdMembershipCard** class is as follows.

```
/**
* This method takes a specific DVD on loan.  It overrides the method in the parent class.
* @param catNo  The catalogue number of the DVD to take on loan.
*/
public void takeItemOnLoan( String catNo ) throws ItemLimitException { }
```

Secondly, let us recall from Chapter Four that the **readMembers** method of the **MediaStore** class calls the **readObject** method of one of the I/O classes in a **try** block. Part of the documentation of the **readObject** method, taken from the Java API, includes the following information:

```
readObject
  public final Object readObject()
                      throws IOException
```

The documentation of both methods includes a number of elements that are known to the user of the method. The known elements of the two methods are listed in the box on the next page.

- the method’s identifier;
- the method’s return type;
- the method’s parameters;
- the exceptions thrown by the method.

The contents of the box above are, in fact, the method’s return type, signature and throws clause. For the sake of brevity, we will refer to the set of (four) components as the method’s *description*.

There is no reason why we cannot extrapolate this notion and state that *any* Java method can be similarly described.

The description of a method is aimed at the *user* of the method, i.e. the code – another method - that invokes it. This means that the description of a method must be known to the developer so that other methods can invoke the method without having to know *how* the method is implemented. All that the calling method needs to know are the four items in the box above.

For example, the calling method of the **`takeItemOnLoan`** method in the themed application is a method that calls **`takeItemsOnLoan`** when one of the buttons of the application’s GUI is pressed. Recalling the simplified code for the calling method from Chapter Four, we can see that the code shown below only needs to know that **`takeItemsOnLoan`** throws an exception and that it takes an argument of the **`String`** type, as follows:

```
// if the button pressed is the 'Borrow DVD' button.
{
        // Get the film from the list of DVDs available for loan.
        // Get the catalogue number of the DVD; this is a String.
        // Use the catalogue number to borrow the film using the member's DVD card.
        // takeItemOnLoan throws an exception.
        try
        {
                membersCard.takeItemOnLoan( catNo );
        }
        catch( ItemLimitException ile )
        {
                messagesArea.setText( ile.getMessage( ) );
        }

} // end if
```

The code shows that the calling method does not need to know how **takeItemOnLoan** is implemented in terms of its statements; the calling method calls **takeItemOnLoan** in the knowledge of its description.

Similarly, the code that calls the **readObject** method of the I/O stream needs to know that **readObject** returns an **Object**, takes no arguments and throws an exception. The calling method in the themed application is shown below.

```
/**
* This method reads the file of members.
*/
public void readMembers( ) {

        try
        { // Start of try block.
                // The String is the path to the file.
                FileInputStream fis = new FileInputStream("C:\\Temp\members.dat");
                ObjectInputStream ois = new ObjectInputStream( fis );
                // Note the cast in the next statement.
                members = ( Member [ ] ) ois.readObject( );
        } // End of try block.
        catch ( IOException e ) { // Start of catch block.
                System.out.println( "Error: " + e.getMessage( ) );
        } // End of catch block.
```

```
                finally
                {
                        ois.close( );
                        fis.close( );
                }

        } // End of readMembers.
```

The code shows that the calling method does not need to know how **`readObject`** is implemented; the calling method calls **`readObject`** in the knowledge of its description specified in the API.

The two examples above show that a calling method does not need to know *how* the method called is implemented; rather, it only needs to know *what* the method does in terms of its description. The outcome of decoupling the *how* and *what* of a method, illustrated by the examples above, means that the implementation of a method can change – to make it more efficient for example – *without* changing its description.

The outcome of the examples and their explanation above serves to emphasise the fundamental importance of a method's description in that it provides the developer with sufficient information to invoke it. Therefore, we can regard the notion of a method's description as a kind of *contract* offered by the method in that it indicates how it is invoked. Extrapolating from the examples above implies that *any* method that invokes another method can do so in the knowledge of the latter's description; the calling method does not need to know how the called method is implemented. In practice, if the body of a method is changed without changing its description, method invocations do not have to be re-written.

The outcome of decoupling method implementation from method description has profound implications in Java. Methods can be published, as in the Java API, so that developers can programme to their description without regard to their implementation.

Publishing the descriptions of methods of Java classes implies that they can be standardised so that a *client application* – i.e. one that invokes published methods – can be written in the knowledge of the description of the methods provided by a *server application* – i.e. one that implements the published method.

(There is a category of applications – that are outside the scope of this guide – in which the client application calls server methods across the Internet. The provider of the server part of the application publishes – to Java developers – the description of methods implemented by the server application so that the developer can write Java clients independently of the implementation of the server. This category of Internet-based application is known as a *Web service*. Web services rely on published method descriptions known as *interfaces*. In the case of a Web service, the client and server applications can be written in any language; in fact, the client and server do not have to be written in the same language given that the client is programmed to the server application's interface.)

The outcome of this section is to emphasise the fundamental importance of a method's description and leads to the concept of a *Java interface*.

A Java interface is a category of Java class that includes the description of one or more methods in a collective contract that other classes can use.

Classes that use a Java interface are said to *implement* that interface. Defining and implementing interfaces is explained in the next section.

5.2 Defining and Implementing a Java Interface

An interface is similar to a class in that it declares members. An interface can declare constants and declares methods by their description; there are no method bodies. Interfaces cannot be instantiated; they can only be implemented by classes or extended by other interfaces.

Defining an interface is similar to declaring a class definition. For example, a version of the themed application includes the interface shown below.

```
/** The Interface CardStatus introduces new behaviour to its implementing classes.
* @author  David M. Etheridge.
* @version 1.0, dated 29 November 2008.
*/
```

```
public interface CardStatus {

        /**
        * setStatus sets the status of the card to either "Standard" or "Premier" when it is called.
        * @param  status  A String to set the variable status in the MembershipCard class.
        */
        void setStatus( String status );

        /**
        * getStatus returns the value of the attribute status.
        * @return  The value of the variable status in the MembershipCard class.
        */
        String getStatus( );

        /**
        * setDiscout sets the level of discount of the card.
        * @param  discount  An integer used to set the discount for the card.
        */
        void setDiscount( int discount );

        /**
        * getDiscount returns the value of the discount.
        * @return  The value of the discount.
        */
        int getDiscount( );

} // End of interface definition.
```

It should be noted that method declarations in an interface are terminated with a semi-colon and do not require the modifier public.

In this version of the themed application, the **MembershipCard** class implements the **CardStatus** interface and provides a body for each method declared in the interface. If the developer omits any of these methods, the compiler issues a warning to this effect. A simplified version of the **MembershipCard** class follows, the purpose of which is to show the reader how the interface is implemented by the class.

```
/**
* The MembershipCard class implements the methods required for transactions carried out
* by a member's virtual membership card.
* @author D. M. Etheridge.
* @version 1.0, dated 29 November 2008.
*/
public class MembershipCard implements Serializable, CardStatus {
```

```
// Declare instance variables.
protected int noOnLoan.
protected int maxOnLoan;
protected int discount;
protected String status;

/**
* Constructor for objects of class MembershipCard.
* @param max  The maximum number of items allowed on loan.
*/
public MembershipCard( int max ) {

        // The argument passed to this constructor is used to initialise the
        // maxOnLoan field.
        maxOnLoan = max;

} // End of constructor.

// Methods to return the maximum number of items permitted to be on loan, to return
// the number of items currently on loan, take items on loan and return items on loan
// would follow but are omitted for the purposes of the present discussion.

// Methods associated with the interface CardStatus are on the next page.
```

```
        /**
        * This method sets the variable called status.
        * @param status  The value of the card's status: "Standard" or * "Premier", is passed to
        * the method when it is called.
        */
        public void setStatus( String status ){

                this.status = status;

        } // End of implementation of setStatus.

        /**
        * This method returns the value of the variable called status.
        * @return status  The value of status.
        */
        public String getStatus( ){

                return status;

        } // End of implementation of getStatus.

        /**
        * This method has an empty body in this version of the application.
        */
        public void setDiscount( int discount ) { }

        /**
        * This method also has an empty body in this version of the application.
        * @return discount  The value of discount.
        */
        public int getDiscount( ) { }

    } // End of class definition of MembershipCard.
```

As the code shows, the setter and getter methods for the variable **discount** are empty in this version of the themed application. (An empty method is often referred to as a *stub*.) However **setDiscount** and **getDiscont** must be implemented, either with a body or as a stub, because the contract of the **CardStatus** interface demands as much. If either **setDiscount** or **getDiscount** are omitted by the developer, the compiler would issue a warning to this effect.

The class declaration also shows that an interface with the identifier **Serilaizable** is also implemented by **MembershipCard**. It is evident, therefore, that a class may implement more than one interface.

The definition of a Java interface can be summarized by the following general syntax:

public interface < NameOfInterface > extends < SuperInterfaces >

An interface, unlike a class, can extend any number of interfaces in a comma-separated list.

5.3 The Role of Interfaces as a Means to Introduce Behaviour to a Class

A similar version of the themed application modifies the class definition for **MembershipCard** as follows:

public class MembershipCard extends Card implements Serializable, CardStatus {

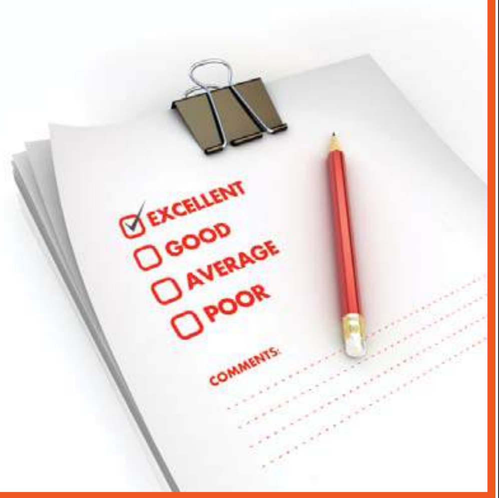
and shows that a class can extend from only one class – as we discovered in Chapter Three – but can implement more than one interface. This means that the **MembershipCard** class not only contains its own methods and those inherited from the **Card** class, it also contains methods declared in the interface **CardStatus**. (The interface **Serializable** is one of a number of *tagged* interfaces. Tagged interfaces do not declare methods; they are used to indicate to the run-time system that an activity is required; in this case the activity required involves reading and writing objects to an input/output stream. [We will encounter input/output streams in Chapter One in *An Introduction to Java Programming 3: Graphical User Interfaces*.] This activity is known as *object serialization* – or *serialization* for short - and is used in the themed application to store an array of **Member** objects to a file. Serialization is achieved by declaring that a class implements the **Serializable** interface.)

Both versions of the class declarations of **MembershipCard** referred to above implement the **CardStatus** interface and, in effect, introduce behaviour to the class in addition to inherited behaviour. Given that Java does not permit multiple inheritance, interfaces are used as an alternative in order to add methods to a class. Adding methods to a class by means of interfaces gives them a very important role in Java.

The role of Java interfaces arises due to the fact that interfaces are not part of the general class hierarchy. An extract from the API for the **Serializable** interface of the **java.io package** shown on the next page illustrates this.

java.io
Interface Serializable
All Known Subinterfaces:
AdapterActivator, Attribute, Attribute, Attributes, BindingIterator, ClientRequestInfo, ClientRequestInterceptor, Codec, CodecFactory, Control … etc.

Selecting the subinterface **`Attribute`** displays to the following page of the API.



Interface Attribute

All Superinterfaces:

Cloneable, Serializable

All Known Implementing Classes:

BasicAttribute

The point of these illustrations is not to find out what the various interface do, it is to note that interfaces have their own internal hierarchy that is not part of the overall class hierarchy. This means that a class can implement more than one interface and an interface can be implemented by more than one unrelated class.

Interfaces combine with classes in that the API specifies which classes implement which interface, as is shown above for the interface **Attribute**.

5.4 Interfaces as Types

This chapter has shown that a Java class can inherit from only one class but that it can implement more than one interface. This means that an object can have multiple types: its own type, superclass types – as we saw in Chapter Three - and the types of all interfaces that the class implements.

Consider, for example, the version of the themed application shown in the screen shot:

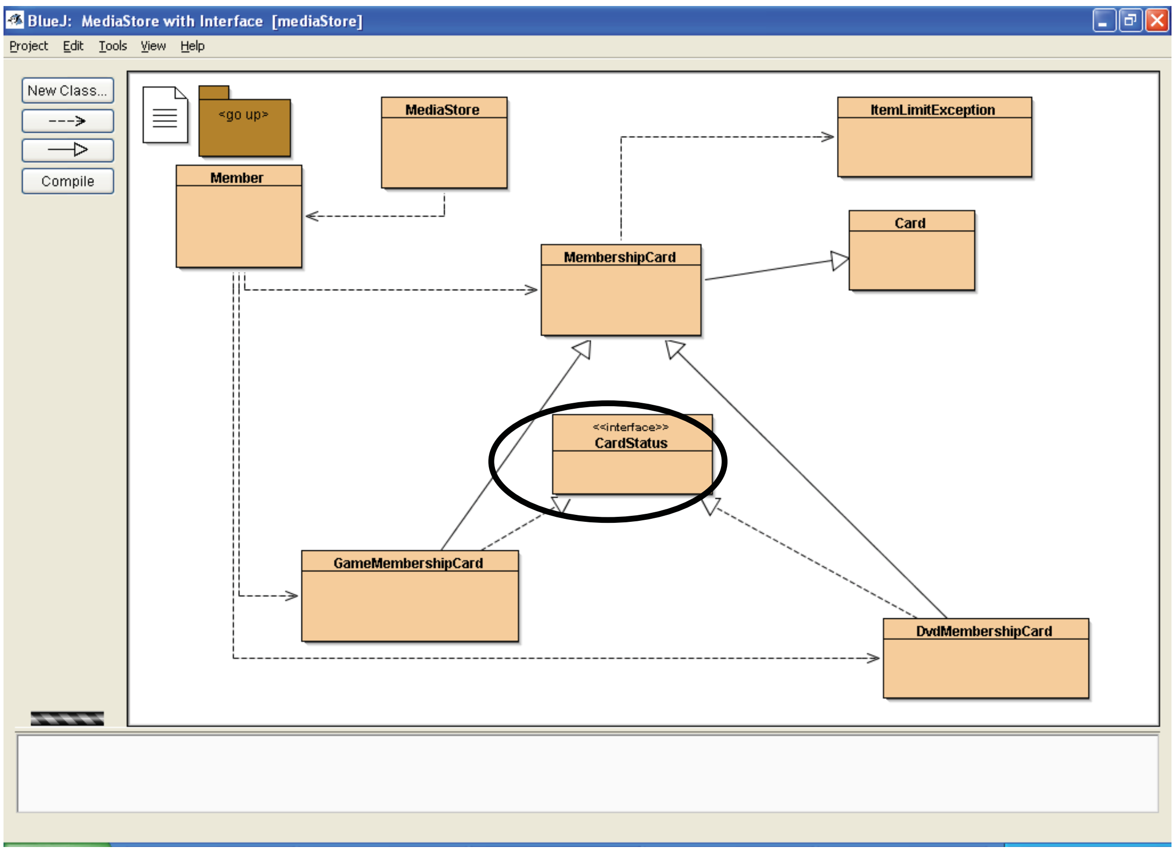


Figure 5.1 The class diagram for (a version of) the themed application

For the purposes of the present discussion, it is not important to know what each class in the application does, rather, the reader should note that the figure shows that `MembershipCard` inherits from `Card` and has two subclasses which both implement the `CardStatus` interface.

In the context of the application shown in Figure 5.1, the following statements compile:

```
// Note: the constructors for MembershipCard, DvdMembershipCard and
// GameMembershipCard take an integer argument.
// Instantiate an object whose type is the same as the run-time type.
Card cardOne = new Card( );
// Instantiate an object whose type is the superclass of the run-time type.
Card cardTwo = new MembershipCard( 10 );
// Instantiate an object whose type is the interface implemented by one of its run-time types.
CardStatus cardThree = new DvdMembershipCard( 10 );
// Instantiate an object whose type is the interface implemented the other run-time type.
CardStatus cardFour = new GameMembershipCard( 10 );
```

The third and fourth statements show that when a class type variable is declared to be an interface type, the variable can be used as an object reference to an object that implements that interface. In other words, the instance can be accessed by a reference of the interface type.

Now that it has been have shown - by example - that interfaces can be declared as class types, this raises a question: *how can we use an interface as a type?*

5.4.1 The Use of Interfaces as Class Types

Referring to the example in Section 5.4, let us assume that a substantial amount of code has to be written for a class that process instances of `DvdMembershipCard` objects. The code would, firstly, instantiate, an instance of `DvdMembershipCard` as follows:

```
// Instantiate an object whose type is the interface implemented by DvdMembershipCard.
CardStatus membersCard = new DvdMembershipCard( 10 );
```

The code that follows this declaration might include a number of statements that include the variable `membersCard`.

Let us also assume that code also has to be written for a similar class that process instances of `GameMembershipCard` objects. This code would also instantiate an instance of a `GameMembershipCard` object as shown next:

```
// Instantiate an object whose type is the interface implemented by GameMembershipCard.
CardStatus membersCard = new GameMembershipCard( 10 );
```

Let us assume that the code that follows this declaration is the same as the code that processes a member's DVD card. (This assumption is based on the notion that card processing is the same irrespective of the type of card.)

This means that the code that processes a member's DVD card can be copied to form the code that processes a member's games card. The only statement that would need to be re-written is the one that calls the constructor of the card; all other statements that include the variable **membersCard** would not have to be re-written.

A similar use of interfaces as types arises when a number of similar classes implement the same interface. For example let us assume that there are a number of data structures that can be used to store a specific primitive data type or object type, all of which implement the same interface.

Instantiating an object of one of these data structures is generalised as follows:

```
// SomeDataStructure implements AnInterface.
AnInterface dataStructure = new SomeDataStructure( );
```

followed by code that includes the variable **dataStructure**.

If we wish to change the data structure, in the light of testing the application, we need to re-write only one line of code, i.e. the call to the constructor of the new data structure, as follows:

```
// SomeOtherDataStructure implements AnInterface.
AnInterface dataStructure = new SomeOtherDataStructure( );
```

Subsequent statements that include the variable **dataStructure** do no have to be re-written.

The examples in this section show how the compatibility rules for using interfaces as types can be exploited to minimise re-writing code and gives rise to significant re-useability of code.

5.5 Summary of Java Interfaces

The principal concepts associated with Java interfaces are summarised next.

- Java allows a class to inherit from only one superclass but allows the class to implement one or more interfaces.
- Interfaces are not part of the class hierarchy; unrelated classes can implement the same interface.
- Interfaces can be declared as class types at compile time.

The ways in which Java interfaces can be used is summarised as follows.

- Declaring methods that one or more classes can implement.
- Determining an object's programming interface without the need to reveal the body of the methods implemented in the class.
- Exposing similarities amongst unrelated classes without forcing a "is a" class relationship.
- Simulating multiple inheritance by declaring that a class implements more than one interface.

A close inspection of the title of the window displayed in Figure 5.1 reveals that it includes the term **[mediaStore]** (in square brackets). The component of the application, whose classes are shown in Figure 5.1, referred to as **[mediaStore]** is an example of what is known as a *Java package*. The next chapter explains the purpose of packages in a Java application.

6. Grouping Classes Together in a Java Application

Chapter Twelve explains how classes and interfaces can be grouped together, using a concept not unlike that of a generic namespace arrangement. Collecting a number of related classes together in a single structure known as a *package* makes them easier to manage and avoids potential naming conflicts.

6.1 An Introduction to Java Packages

As an illustration of namespace management, assume that a banking application has two classes named **Account**: one of the classes is the definition of a class for a current account and the other class is that for a savings account. The name of the two classes can be the same (**Account**) as long as they are placed in separate packages. This is because each package creates a new namespace such that the types in one package do not conflict with the same types in another package.

A simple analogy might help the reader to appreciate the need to manage class names and their namespaces. The use of packages in a Java application is rather like the obvious distinction between a business address in Main Street in Freetown and a similar address in Main Street in Freeville as

Top Ten Records, Main Street, Freetown   and   Top Ten Records, Main Street, Freeville

The notation for the address of each Top Ten Records store is for the obvious benefit to everyone concerned.

The outcome of placing the classes named **Account** in separate packages means that their unique, *fully-qualified* names are written as follows:

```
currentAccount . Account      // refers to the Account class in the package named
                              // currentAccount
```

and

```
savingsAccount . Account      // refers to the Account class in the package named
                              // savingsAccount
```

The notation used for the fully-qualified name of a class means that it can be referred to unambiguously and accessed by means of the usual ‘ **.** ‘ (dot) selector.

6.2 Creating Packages

A version of the themed application shown in Figure 6.1 below indicates that classes of the application are bundled together in one of two packages depending upon their function in the application.

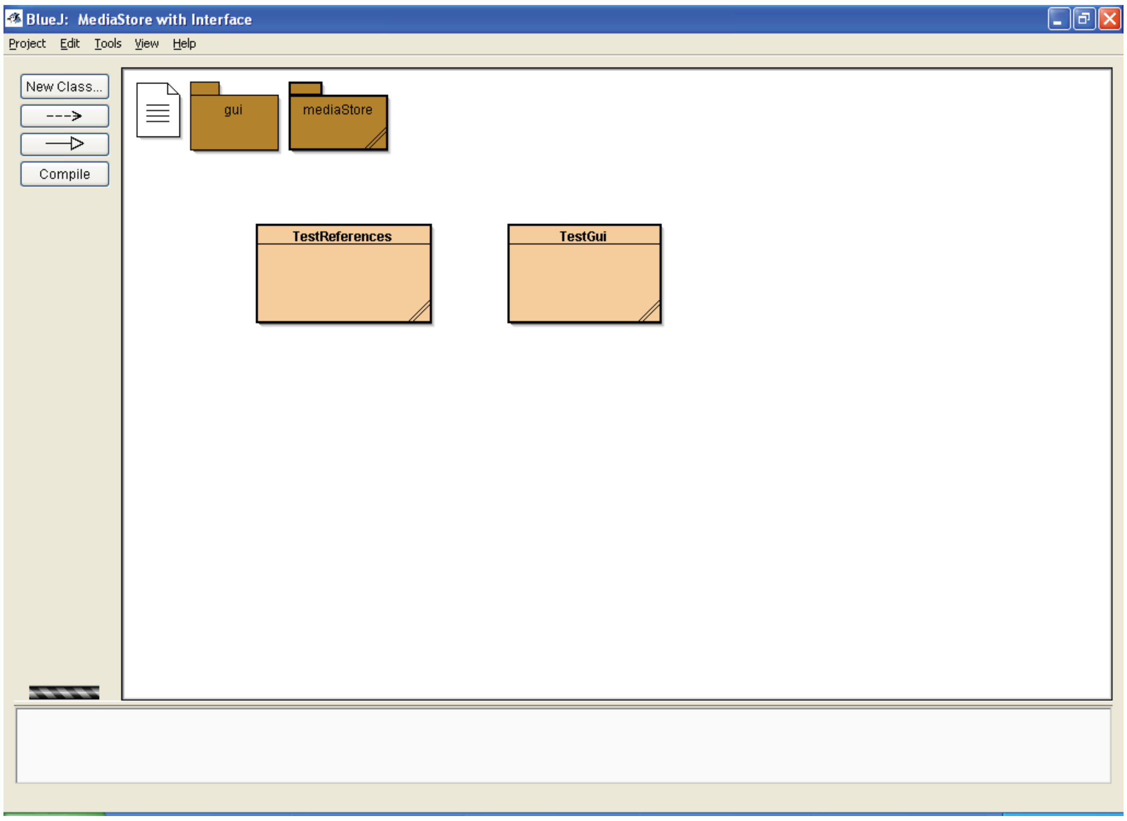

BlueJ: MediaStore with Interface
Project Edit Tools View Help
New Class...
Compile
gui
mediaStore
TestReferences
TestGui

Figure 6.1 The package structure of the themed application

The two test classes – i.e. the ones that include a **main** method – named **TestGui** and **TestReferences** are not in a named package and are said to be in the *default package* of the application.

The package named **gui** contains a single class named **MediaStoreGui**. The statements

```
package gui;              // the class MediaStoreGui is in this package
import mediaStore.*;  // the class has access to the classes contained in the other package of
                          // the application
```

are written *before* the class declaration and state that the class named **MediaStoreGui** is in the package named gui and that it requires access to all of the classes in the package **mediaStore** by virtue of the wildcard ‘ * ‘ written to the right-hand side of the (dot) selector in the **import** statement.

This version of the themed application has only one class that displays the graphical user interface (GUI) for the application. Nevertheless, it makes sense to place this class in its own package: other versions of the application might contain additional classes that are concerned with the display of the GUI that and would be placed in the package named **gui**.

The **import** statement is required because components of the GUI require access to classes in the **mediaStore** package to invoke their methods and run the application.

The package named **mediaStore** contains eight classes, each of which begins with the **package** statement

```
package mediaStore;
```

The eight classes are bundled together in their own package because they represent the business logic of the application. None of these classes has any role in the display of the GUI; therefore, there is no requirement to import the contents of the **gui** package into the package named **mediaStore**.

Whilst the two test classes *could* have been placed in their own package, this version of the themed application is used to illustrate the contents of the default package of the application. Both test classes require access to classes in both packages, which means that the following **import** statements appear at the beginning of each source code file:

```
import gui.*;
import mediaStore.*;
```

There is no package statement at the head of the source code file of the two test classes; this means that these two classes are automatically placed in the default package of the application.

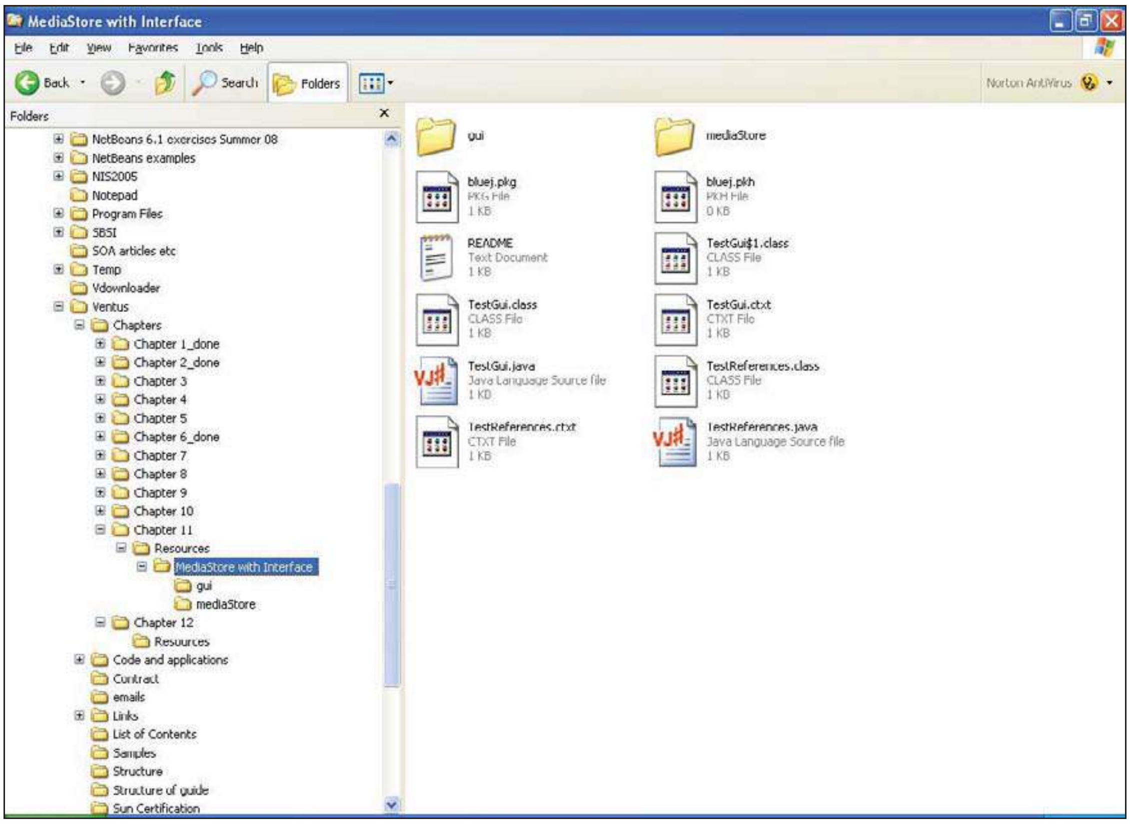
The version of the themed application discussed above illustrates the principal reasons why classes and interfaces are bundled in a package; i.e. the types in a package are functionally related in the context of an application. A number of other reasons derive from this; they are summarised in the box shown on the next page.

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

Reasons why packages are employed in a Java application:

- developers can easily determine from the structure of an application how application logic is partitioned amongst its packages;
- developers can easily find related classes in an application;
- packages in the Java API imply that their contents are a collection of related classes (see later in this chapter);
- the names of types in one package won't conflict with the same names in another package;
- types in a package can have unrestricted access to one another but restricted access to types outside the package (see later in this chapter).

The structure of an application, in terms of its packages, is reflected in its directory structure as used by the operating system that hosts the application. For example, the package structure of the version of the themed application shown in Figure 6.1 above is reflected in its directory and folder structure in Windows™ as shown in Figure 6.2 below.


MediaStore with Interface

Folders
- NetBeans 6.1 exercises Summer 08
- NetBeans examples
- NIS2005
- Notepad
- Program Files
- SBSI
- SOA articles etc
- Temp
- Vdownloader
- Ventus
  - Chapters
    - Chapter 1_done
    - Chapter 2_done
    - Chapter 3
    - Chapter 4
    - Chapter 5
    - Chapter 6_done
    - Chapter 7
    - Chapter 8
    - Chapter 9
    - Chapter 10
    - Chapter 11
      - Resources
        - MediaStore with Interface
          - gui
          - mediaStore
    - Chapter 12
      - Resources
  - Code and applications
  - Contract
  - emails
  - Links
  - List of Contents
  - Samples
  - Structure
  - Structure of guide
  - Sun Certification

gui	mediaStore
bluej.pkg PKG File 1 KB	bluej.pkh PKH File 0 KB
README Text Document 1 KB	TestGui$1.class CLASS File 1 KB
TestGui.class CLASS File 1 KB	TestGui.ctxt CTXT File 1 KB
TestGui.java Java Language Source file 1 KB	TestReferences.class CLASS File 1 KB
TestReferences.ctxt CTXT File 1 KB	TestReferences.java Java Language Source file 1 KB

Figure 6.2 The directory of the application shown in Figure 6.1

Figure 6.2 shows that the two test classes are in the default (unnamed) package and remaining classes have been placed in one of two packages. The IDE used to develop the application shown in figures 12.1 and 12.2 automatically creates a Windows™ folder for each package created by the developer. In practice, all IDEs will create a Windows™ folder for each package created by the IDE.

6.3 Naming Convention

Figures 12.1 and 12.2 show that the author (of this guide) has named the packages of the themed application with a single word that starts with a lower case letter. This simple convention works well unless independent developers use the same package and class names for applications that are in the public domain or that are accessible by members of the Java development community.

To overcome the potential problem of more than one application containing a class named **mediaStore.Member** – to take, as an example, one of the classes of the themed application – there is a convention in the Java developers' community whereby an organisation uses its Internet domain name in reverse to start package names. For example, the package named **mediaStore** shown in figures 12.1 and 12.2 should – by convention – be named along the following lines:

u**k.ac.bcu.tic.etheridge.mediaStore**

so that the class named **Member** is given the fully-qualified name of

u**k.ac.bcu.tic.etheridge.mediaStore.Member**

Up to this point in the chapter, we have largely considered packages provided by the developer of an application. The Java language itself is partitioned into a number of packages as is suggested by the statements at the head of the class definition for the **MediaStore** class of the **mediaStore** package of the themed application, as follows:

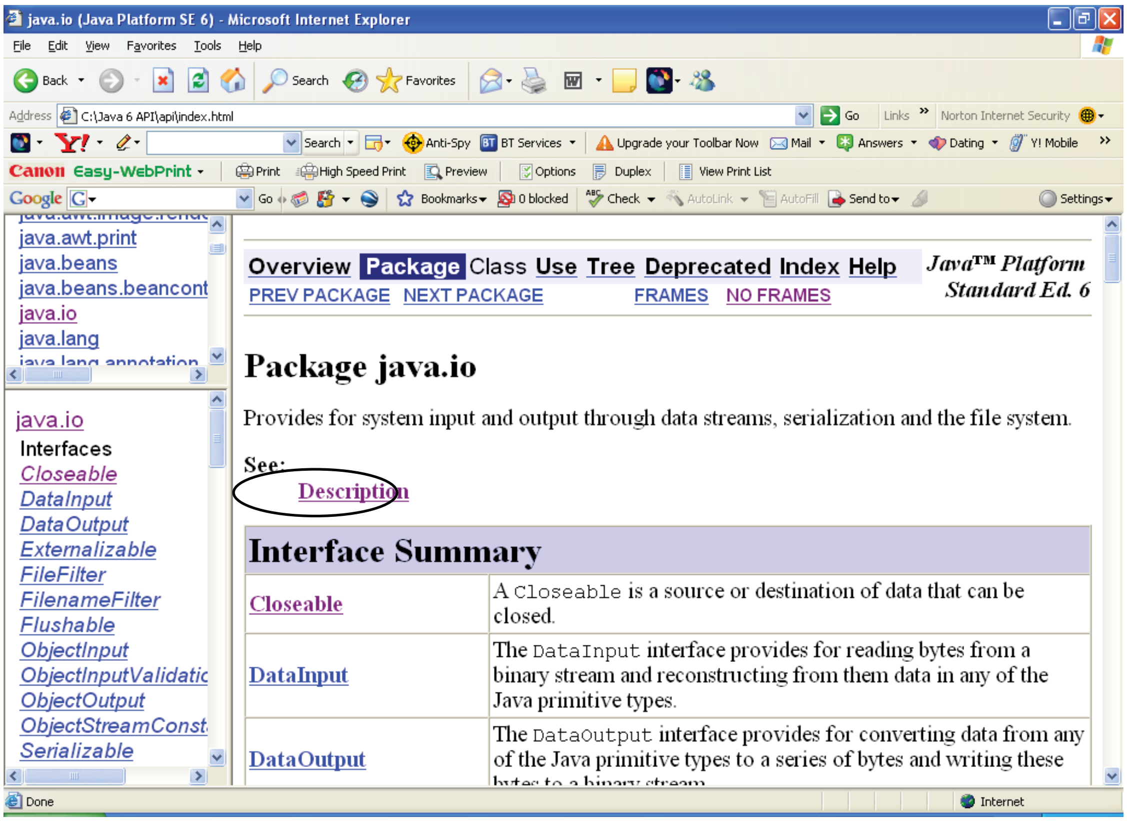
```
package mediaStore;
import java.io.*;
```

The first statement denotes that the **MediaStore** class is a member of the **mediaStore** package. The second statement implies that the class requires access to *all* of the classes of the **java.io** package.

The next section looks briefly at the use of package in the Java language.

6.4 Packages in the Java Language

The mane of the majority of the packages in the Java language begins with **java** or **javax**. For example, the relevant page of the Java API for the **java.io** package is shown in Figure 6.3 below.


java.io (Java Platform SE 6) - Microsoft Internet Explorer

Address C:\Java 6 API\api\index.html

java.awt.print
java.beans
java.beans.beancont
java.io
java.lang

java.io
Interfaces
Closeable
DataInput
DataOutput
Externalizable
FileFilter
FilenameFilter
Flushable
ObjectInput
ObjectInputValidatic
ObjectOutput
ObjectStreamConst
Serializable

Overview **Package** Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Java™ Platform Standard Ed. 6

Package java.io

Provides for system input and output through data streams, serialization and the file system.

See:
Description

Interface Summary

Closeable	A `Closeable` is a source or destination of data that can be closed.
DataInput	The `DataInput` interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.
DataOutput	The `DataOutput` interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream

Done Internet

Figure 6.3 The opening page of the java.io package in the API

The lower left-hand pane shows the first part of the list of interfaces, classes and exceptions associated with the package. There are about 60 classes in the **`java.io`** package that are available to the Java developer.

Although the statement

import java.io.*;

referred to above implies that the author wishes to import *all 60* classes of the **`java.io`** package into the **`MediaStore`** class of the themed application, he used only four of these classes in the code. In this case, four import statements would have sufficed, as shown on the next page.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

The set of import statements imports the four classes of the `java.io` package required for the `Member` class. (Chapter Thirteen examines some of the classes of the `java.io` package in the context of what are known as input/output streams.)

The use of the ‘import all’ statement

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

is useful when developing an application because it means that the developer can use *any* of the classes of – in this case - the `java.io` package without having to insert specific import statements.

The large, central pane shown in Figure 6.3 begins with a brief description of the contents of the `java.io` package and includes a hyperlink to a more detailed description. This kind of overall structure of linked HTML pages in the Java API is typical of most packages, of which there are about 225 in the Java Version 6 Standard Edition API. The sheer number of packages – in excess of 200 – confirms the large scale and very wide functionality of the standard edition of the Java language. Clearly, in a guide such as this one, the author can only scratch the surface of what is available to the developer in the API.

At this point, it is worth mentioning that the Java run-time system automatically imports the (classes of) the **java.lang** package into all applications. The **java.lang** package contains about 40 classes, including widely-used ones such as the wrapper classes - **Boolean**, **Character**, **Double**, **Integer** and so on - that are object representations of the primitive data types, as well as classes such as **System**, **String** and **Thread**. The rationale for this automatic inclusion is that commonly-used classes should be readily available without having to insert import statements for them.

6.5 Using and Accessing Package Members

We have seen that the **import** statement provides access to one or more members of a named package in a class definition. In order to use a public member from *outside* its package, one of the following must be done:

1. The member must be referred to by its fully-qualified name.
2. The member must be imported.
3. The member's entire package must be imported.

Referring to a member by its fully-qualified name is required each time the reference is used in Java statements and, therefore, is a satisfactory approach when there are relatively few such uses. For frequent references, it makes more sense to import the member at the head of the class definition. Once a member has been imported by an **import** statement, it can be referred to in the code by its class name rather than by its fully-qualified name.

However, care should be taken when importing members of packages when, for instance, there are identically-named classes in them. For example, consider the following code snippet:

```
package mynewpackage;
import currentAccount.Account;
import savingsAccount.Account;
```

A reference to the type **Account** in the class definition that follows the **package** statement and **import** statements would produce a compiler error because the compiler does not know which of the two **Account** classes is being referred to. In such a case, the compiler will be able to distinguish between the two **Account** types when their fully-qualified name is used in the code even though both **Account** classes have been specifically imported.

When a member's *entire* package is imported, *any* member of that package can be referred to by its class name in the subsequent code.

6.5.1 Controlling Access to Package Members

Let us recall the access levels for the access modifiers listed in Table 3.1; the table is reproduced in Table 6.1 on the next page

Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default: no specifier	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Table 6.1 Access levels to class members

The first row of Table 6.1 reaffirms what we know about the access modifier ‘private’: i.e. the class itself has access to its **`private`** members as we would expect.

The second row can be regarded as a class’s *package contract* whereby if no modifier is specified classes in the same package have access to such members and trust one another to access each other’s members. A class’s package contract can be used to hide things such as implementation details that are not required to be known to classes outside the package. This type of contract means that such details can be changed without changing a class’s *public contract.*

A class’s public contract declares it as a type that is available to developers using its containing package. Thus the fourth row shows that *all* classes have access to **`public`** members, regardless of their package and parentage. We will return to the third row in due course.

As we work our way down the ‘Modifier’ column from ‘**`private`**’, ‘**`default`**’ (i.e. package level), ‘**`protected`**’ and ‘**`public`**’, each of the four access levels embraces a wider scope of the kind of class that can access that member.

Let us return to the third row of the table that was first displayed in Chapter Nine: the *protected contract.* The third row indicates what level of access is provided when a class member is declared to be ‘protected’. The first column indicates that other members of the class itself have access to the **`protected`** member of that class; the second column indicates that classes in the same package, regardless of their parentage, have access to the **`protected`** member of the class; the third column indicates that subclasses of the class have access to the **`protected`** member, regardless of what package they are in. However, the subclass-protected table entry requires further comment.

The subclass-protected entry in the table can be stated more precisely: *a protected member can be accessed from a class via object references that are of the same type as the class or one of its subtypes.* Despite the apparent clarity of this statement, it gives rise to an interesting twist as the following example aims to illustrate.

Suppose that we have a simple class in a package named **packageone** that includes a **protected** member, as shown by the following code.

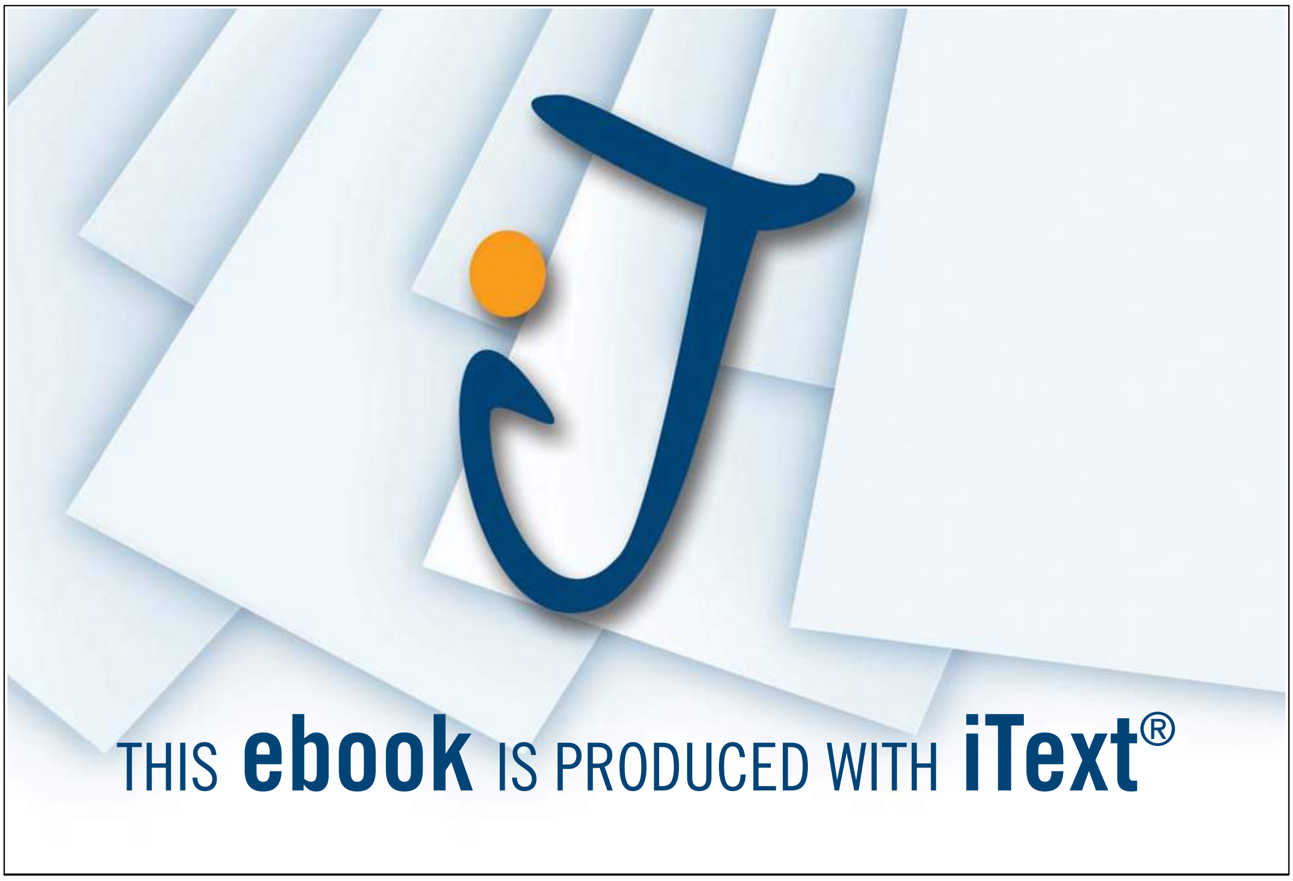
```
package packageone;
public class Cat {
        // The class Mammal is also in packageone but its code is not included here.
        protected Mammal mammal;
        public void setSpeciesName( ) { }
} // end of class definition
```

A subclass of **Cat** is in a separate package.

```
package packagetwo;
public class DomesticCat extends packageone.Cat {
        // Inherits the protected variable mammal of the Mammal type.
        // Other members of DomesticCat would follow next.
} // end of class definition
```

Suppose that **DomesticCat** overrides the method **setSpeciesName** with a simple implementation.

```
public void setSpeciesName( ) {
        // The current object is a subclass of Cat, so access to the protected member inherited
        // from Cat is allowed.
        this.mammal = null;
}
```



Given that the current object (**this**) is a subclass of **Cat**, access to the protected member **mammal** is allowed even though **Cat** and **DomesticCat** are in different packages.

Suppose that one of the methods of **DomesticCat** takes a **DomesticCat** as an argument and accesses its protected member, as follows.

```
public void aMethod( DomesticCat florence ) {
        packageone.Mammal  m = florence.mammal;
}
```

This method compiles because the method accesses the protected member of an object passed as an argument. Access is allowed because the class attempting to access the protected member (**mammal**) is a **DomesticCat** and the type of the reference **florence** is also a **DomesticCat**.

Finally, suppose also that the class **DomesticCat** defines an overridden method, as follows.

```
public void aMethod( packageone.Cat florence ) {
        packageone.Mammal  m = florence.mammal;
}
```

The statement

```
packageone.Mammal  m = florence.mammal;
```

doesn’t compile; the compiler issues the message:

mammal has protected assess in packageone.Cat

The reason why the compiler issues the message is that the class attempting to access the protected member is **DomesticCat** and the type of the reference **florence** is **Cat**. Access to the protected member is not allowed because the **Cat** type is not the same as or a subclass of **DomesticCat**. Although every **DomesticCat** object a subclass of a **Cat** object, not every **Cat** object is a **DomesticCat**.

Although some of the points made in this sub-section may appear to be rather erudite at first, it *is* important that the leaner emerges from this section with a reasonable understanding of access levels and how they are controlled by means of access contracts. Even though some of the implications of controlling access to class members may appear to be technically difficult, they will be more easily understood when the reader encounters compiler messages that issue warnings about access to protected members.

6.6 Compiling and Running Package Members

It is likely that the learner will use a learner-level IDE such as BlueJ to write their first Java programmes to learn the basics of the language. When the learner gains experience, he or she can progress to the use of a professional-level IDE such as NetBeans™. In any event, an IDE typically provides buttons or menu options to use to compile and run a Java application; the IDE will ‘find’ and interoperate with the Java compiler installed on the learner’s computer.

If the learner does not use a learner-level IDE in the first instance, code can be written using a simple text editor and source code files can be compiled from the DOS prompt in the case of a Windows™ operating system.

Referring to the example shown in Section 6.5.1, the package named **`packageone`** includes a test class named **`UserClassOne`** with a **`main`** method. This class can be compiled from the DOS prompt with the following command:

C:\ > javac packageone \ UserClassOne.java

and **`main`** can be executed as follows:

C:\ > java packageone . UserClassOne

The fully-qualified class name and directory path are, as we would expect, in parallel. This means that the developer can go to the directory that *contains* the folder named **`packageone`** and compile and run any of the classes in that package. Similarly, the develop could compile all of the classes in the package named **`packagetwo`** as follows

C:\ > javac packagetwo \ *.java

As long as the programmes **`javac`** and **`java`** are on the computer's PATH environment variable, source code files can be compiled from the DOS prompt. The compiler issues the same messages in the DOS prompt window as it does if an IDE is used to compile and run the application.

The next chapter examines some of the classes of the **`java.io`** package that provide for input to and output from applications.