CHAPTER 9
Creating Packages and Using Access Specifiers

What is a package in Java? In Java, package is a collection of related classes and interfaces. Just like we organize our files into folders, we normally have a Documents folder to keep all document files, a Music folder to keep all music files, a Videos folder to keep all video files; similarly we organize our classes and interfaces in Java into various packages.

In the earlier chapters and exercises, we have defined a few classes like the `Rectangle` and the `Cuboid` classes in Chapter 6 and then the `Account`, `SavingsAccount` and the `CurrentAccount` classes in Exercise 8.1. Now the classes like `Rectangle` and `Cuboid` are related to geometrical shapes, and the `Account`, `SavingsAccount` and the `CurrentAccount` are related to the banking domain. We may like to group these separately.

This can be done by organizing the classes into different packages. A package is like a namespace, to which all the related classes and interfaces belong. In Java, we do not have a procedure to create a new package, we only define the classes and interfaces within a package. It is only when we have class(es) and/or interface(s) in a package that a package exists. We do not have a package being created, and then class(es) and/or interface(s) being added in a package. If we want a package, we need to define some class or interface with the appropriate `package` statement.

9.1 USES OF PACKAGE AND IMPORT STATEMENTS

When we create a Java source file, we can declare that all the classes and interfaces being defined in a given file will belong to a particular `package` by using the `package` statement. All classes and/or interfaces defined within a single Java source file will belong to the same package as specified in the `package` statement. In a single Java source file the package statement can be used only once, and that too only before any other statements. We can only have comments and white spaces before the `package` statement. In case there is no `package` statement in a Java source file, then all the classes defined in the source file are said to belong to a common unnamed package.

java.lang → These are classes that Java compiler itself uses & therefore they are automatically imported.

When we have a class belonging to a package, then the class is accessed by prefixing it with its package-name. A package-name can contain multiple identifiers separated by a dot '.', as shown in Listing 9.1.

Listing 9.1. Package declaration for `Rectangle` class

```
1 package geometry.shapes;
2
3 class Rectangle {
4     ...     // various members of the Rectangle class.
5 }
```

In the above code listing, the class which is defined will be accessed as geometry.shapes.Rectangle. This is the fully qualified class name for the `Rectangle` class defined in the package geometry.shapes. Note that the package name is made up of two Java identifiers separated by '.'. A package name may be made up of any number of identifiers. This is a sort of a tree structure, where we can have packages and sub-packages. Like we could have a package for `geometry`, which may contain some classes and interfaces and then we could still have a sub-package of `geometry` called geometry.shapes. These are two different packages. Currently, in Java, there is no use of this kind of relation between packages. It may be used in the future. Classes and interfaces belonging to geometry.shapes are not considered to belong to the package `geometry`.

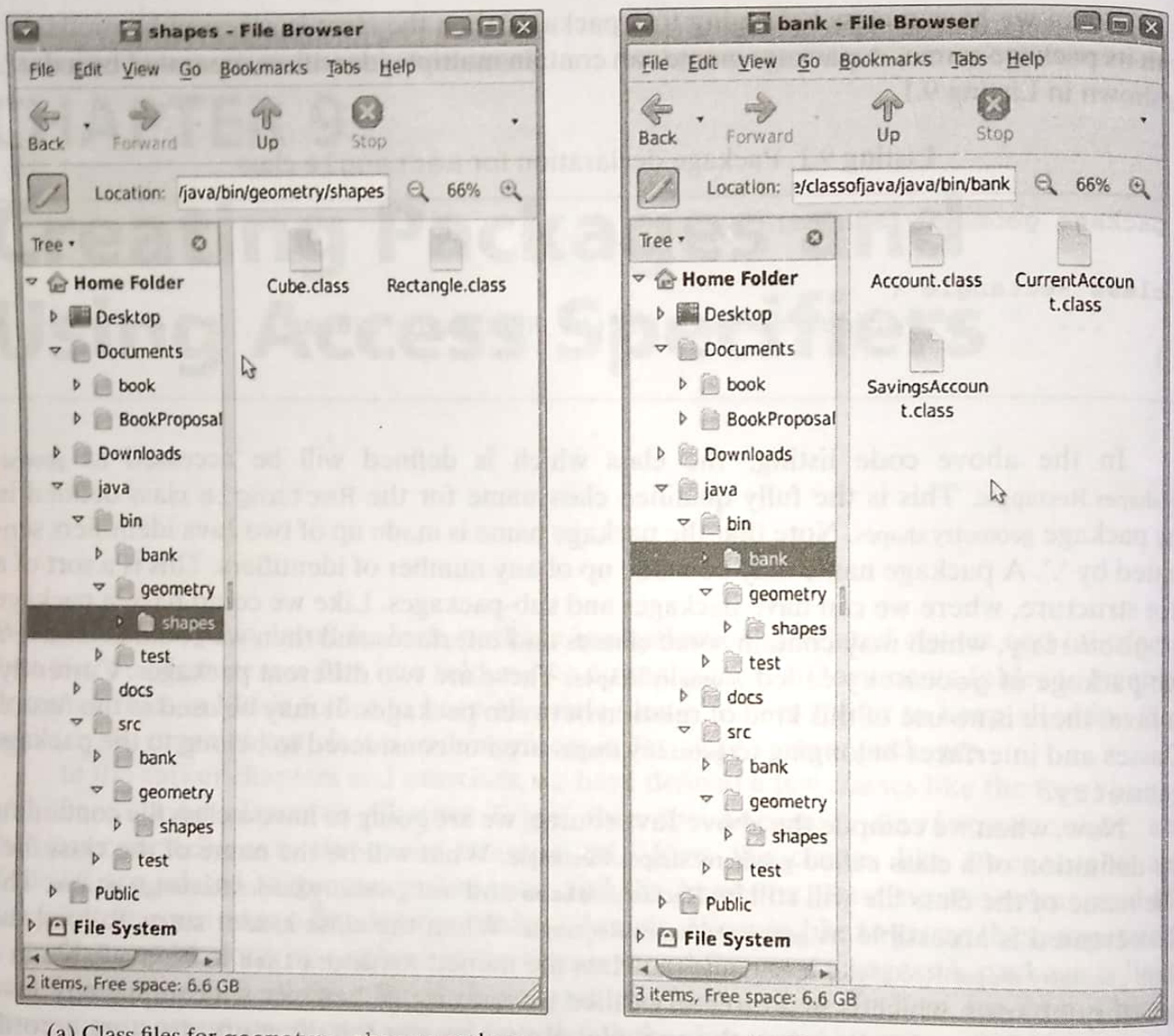
Now, when we compile the above Java source, we are going to have a class file containing the definition of a class called geometry.shapes.Rectangle. What will be the name of the class file? The name of the class file will still be Rectangle.**class** and not geometry.shapes.Rectangle.**class**. The class created is accessible as geometry.shapes.Rectangle. When the class loader starts to load the class geometry.shapes.Rectangle, it expects the class file named Rectangle.**class** to be available in a directory `shapes` which is in a directory called `geometry`. Whenever we compile any Java source file with the -d option then the compiler always creates the directory structure according to the package of the class in the destination directory specified with the -d option. For example, we have defined the class geometry.shapes.Rectangle in a file named Rectangle.java within the `src` directory and we compile the source file from `src` directory by using the command as given below:

```
javac -d ../bin Rectangle.java
```

Then, the compiler would create the directories `geometry` and geometry/shapes in the `bin` directory and the Rectangle.**class** would be stored in the geometry/shapes directory under the `bin` directory. The class geometry.shapes.Rectangle would now be loadable from the `bin` directory.

Similar to the compiler organizing the class files into directories according to a package structure, whenever we compile using the -d option. It is a good practice to organize even our source Java files also according to the package structure. So for the various Java files required for creating the classes in the geometry.shapes package, we would create the directory structure as shown in Figure 9.1.

Now, when we want to use this class in the class `TestRectangle`, it would have to be used with its fully qualified class name, as shown in Listing 9.2.



(a) Class files for `geometry.shapes` package

(b) Class files for bank package

Figure 9.1 Directory structure for classes in various packages

Listing 9.2. `TestRectangle` using full names for classes

```
1 package test;   // declaring TestRectangle as part of package test
2
3 class TestRectangle {
4     public static void main(String[] args) {
5         geometry.shapes.Rectangle r1, r2;  // using the fully
              qualified name
6         r1 = new geometry.shapes.Rectangle(7, 5);
7         ...
8     }
9 }
```

When a class belongs to a package, then it can only be referenced as <package-name>.<class-name>. So, if we define the `TestRectangle` class above as part of the package `test`, then the application can be invoked using the command:

```
java test.TestRectangle
```

Now, looking at how the class geometry.shapes.Rectangle is being used from the `TestRectangle` class, we find that we have to use the fully qualified class name each time we want to use the class. In a Java source file we can declare that we would like to use `Rectangle` instead of the fully qualified name geometry.shapes.Rectangle by using the `import` statement, as follows:

```
1 package test;
2
3 import geometry.shapes.Rectangle; // declaring a substitution for
    the full name.
4
5 class TestRectangle {
6     public static void main(String[] args) {
7         Rectangle r1, r2;  // The compiler will replace Rectangle
              with
8         r1 = new Rectangle(7, 5);  // geometry.shapes.Rectangle.
9     }
10 }
```

In a Java source file, when we declare a `package` statement, we are declaring that all the classes and interfaces defined in this file would belong to the given package. The fully qualified name of the classes and interfaces defined in the file would change to <**package**-name>.<**class**-or-**interface**-name>.

An `import` statement is written as **import** <fully-qualified-name>;.

In a Java source file, when we declare an `import` statement, we declare that in this file, wherever we use `<class-name>` or `<interface-name>` it should be substituted with the specified `<fully-qualified-name>`. A Java source file may contain any number of `import` statements. All import statements have to precede any class, interface or enum declaration in the Java source file. The `import` statement should follow the `package` statement if the `package` statement is used in the file.

An `import` statement in Java does not add any class or interface into the existing code. It is simply a declaration that in this file we would be using only the class or interface names instead of the fully qualified class or interface names. This declaration is used only by the compiler. This declaration of the fully qualified name can also be done for all the classes and interfaces of a given package by using a wildcard as follows:

```
1 package test;
2
3 import geometry.shapes.*; // declaring a substitution for the full
    name,
```

```
4                                   // for all classes and interfaces within
                                        a single package
5  class TestRectangle {
6      public static void main(String[] args) {
7          Rectangle r1, r2;
8          r1 = new Rectangle(7, 5);
9          r2 = new Cuboid(7, 5, 4);
10     }
11 }
```

Let us look at how a compiler uses the `import` statement to resolve class and interface names in a Java source file.

In the above class definition of `TestRectangle` class, which belongs to the package `test`, when the compiler comes across the token `Rectangle` in Line 8, then the first thing that the compiler tries is that whether there is some class called test.Rectangle available (since the current package of the file is `test`). If such a class is found (such a class can be loaded by the compiler), then the compiler would substitute the token `Rectangle` in Line 8 with test.Rectangle. If it does not find such a class or interface, then the compiler would look for `import` statement, which ends with the `Rectangle`. If the compiler finds an `import` statement which ends with `Rectangle`, then the compiler would substitute the token `Rectangle` in Line 8 with the fully qualified name mentioned in the `import` statement. If there is no `import` statement which ends with `Rectangle`, then the compiler would use the `import` statements which end with wildcard '*'. Try to substitute the '*' with the token `Rectangle` and check if such a class is found. If it finds such a class then it would substitute the token `Rectangle` in Line 8 with the fully qualified name for the token. In case there are more than one such `import` statements with wild cards where the substitution of `Rectangle` is a valid class, then the compiler would report an error.

Note we cannot have two `import` statements ending with the same non-wild-card token.

So, every class or interface, which is part of a package, has a fully qualified name by which it is identified.

A Java source file can have a `package` statement, indicating the package for which classes and interfaces are being defined in the file. This becomes the default package for use within the file.

When the compiler has to resolve a type-name, which is not a qualified name, (qualified names are the ones which have a prefix), then either it is a primitive type or it would be qualified using the current package. Only if it is not a valid type for the current package will the compiler use the `import` statements to try and resolve to a qualified-type-name. The `import` statement has no effect at runtime. It is only used at the compile time by the compiler to resolve unqualified type names used in the Java source file.

In the `HelloWorld` and subsequent applications, we have been using the classes like `String` and `System`. There is no such class called `String` or `System`. The actual class names are java.lang.String and java.lang.System. We never mentioned any `import` for these classes, yet these are resolved to their fully qualified names. This is because the compiler by default assumes an `import` statement for the classes in the java.lang package, i.e. even if we don't write, the following `import` statement is always assumed by the compiler:

```
import java.lang.*;
```

Now let us assume that we have a class defined in the `test` package by the name of `String`, i.e. we have a class called test.String. If we now try compiling the code for the test.TestRectangle class as given in Listing 9.1, the compilation goes through without any errors, but when we try executing the application by using the command

```
java test.TestRectangle
```

it gives an error that the class does not contain the main method. This happens because the compiler has resolved the `String[]` in Line 6 as test.String[], whereas when we try executing the Java runtime using the command then it looks for the method main with the parameter as java.lang.String[]. This can only be corrected by using the full name for the `String` class.

9.2 USE OF STATIC IMPORTS

Just like we have the import statement which helps the compiler in resolving data types, we have another form of import statement, which can help the compiler to resolve static member names in the Java source file. With static imports, we can specify the fully qualified name of a static member of a class or an interface. The static imports are also specified right at the beginning, i.e. before any class or interface definition is specified, but following a package declaration, if present. The mechanism of resolving the static member names is similar to the way type names are resolved by the compiler, i.e. first it would try to use the static imports which are without wild cards. If the static member name being resolved does not match the last element of any of the static import statements, then it would try to use the static imports with wild cards. The syntax for writing s static import is as shown in Listing 9.3.

Listing 9.3. Example showing use of `static import`

```
1 package test;
2
3 import geometry.shapes.*; // declaring a substitution for the full
      name,
4                            // for all classes and interfaces within
                                a single package
5 import static java.lang.System.out;
6 import static java.lang.Math.*;
7
8 class TestRectangle {
9     public static void main(String[] args) {
10          Rectangle r1, r2;
11          r1 = new Rectangle(7, 5);
12          r2 = new Cuboid(7, 5, 4);
13          out.println("area of r1 is "+r1.area());
14          out.println("square root of r1\'s area "+sqrt(r1.area()));
15      }
16 }
```

In Listing 9.3, when the compiler comes across the `out` token, in Lines 13 and 14, it is expecting a variable, and it finds a `static import` specified in Line 5, whose last element matches the name being resolved (`out`). So the compiler would substitute the `out` in Lines 13

and 14 with java.lang.System.out. In Line 14, when the compiler comes across the token sqrt, it expects a method name, there is no static import whose last element matches sqrt, so it would then try to use the static imports with the wild card. Now it tries to substitute the sqrt for the wild card and it would find that the static import with wild card in Line 6 results in a valid fully qualified static member name, i.e. java.lang.Math.sqrt is a valid static member. So the sqrt in Line 14 is substituted by java.lang.Math.sqrt. Use static imports with caution. From a readability point of view, if static imports are not used then the usage of static members within the code is easier to identify since they would be preceded by their class names. The static import statement is a feature introduced in the Java programming Language from Java 5 onwards.

9.3 USE OF CLASSPATH FOR CLASS LOADING

The classes from the class files created by a compiler have to be loaded, at the runtime as well as by the compiler at the compile time. The class file Rectangle.class for the class geometry.shapes.Rectangle can be loaded from a directory which contains the sub-directory geometry, having a sub-directory shapes containing the Rectangle.class class file. In order to be able to load class files from any location, the Java Compiler and the Java runtime class loader use an environment variable called CLASSPATH. The CLASSPATH environment variable, like the PATH environment variable, is a list of directory names from where classes may be loaded. The directory names are separated by : in case of UNIX and LINUX platforms and by ; in case of Windows platform. The directory list for CLASSPATH may even contain names of archive files which contain the class files, e.g. zip files and the jar files.

When there is no CLASSPATH environment variable set in the system, then the compiler and the Java runtime will be able to load classes from the current directory only. Both the commands java and javac have switch to specify the directories from where the classes may be loaded, instead of using the CLASSPATH environment variable. Normally, we organize our source code under the src folder with subdirectories matching the package structure, and classes under the bin folder with class files organized according to the package structure. So for the purpose of compiling the source file, one can use the javac command as follows:

```
javac -d ../bin -classpath ../bin geometry/shapes/Rectangle.java
```

In the above command the switch -d ../bin is used to specify the destination directory of the compiled class files. In this case the sub-directories for the package geometry.shapes would be created by the compiler itself under the bin directory. In the above command the switch -classpath ../bin is specified to let the compiler know that if any classes other than the standard API classes are required to be loaded, then they can be loaded only from the bin directory. This -classpath option would ignore any CLASSPATH setting if present.

In case of the java command, the switch for specifying the directories containing the non-API classes is -cp as shown in the example below:

```
java -cp ../bin test.TestRectangle
```

In order to execute the Java application in test.TestRectangle class, we need not change the directory to the bin directory; instead, even while we are in the src directory, we could use the command as given above.

9.4 ACCESS SPECIFIERS

Now if we look at the whole picture, we find that the classes and interfaces which are data types are grouped into various packages, and these classes and interfaces in turn have other types of members which are the various variables, methods, constructors, etc. So, we find that packages can have members which are classes and interfaces belonging to a package. Even if we do not define a class or an interface as part of a package, then they are members of a common unnamed package. The classes and interfaces in turn have other kinds of members, which are the instance variables, class variables, methods, static methods and constructors. Java programming language provides access specifiers to allow or restrict the use of the various members from other class definitions and interface definitions.

9.4.1 Access Specifiers for Members of a Package

For the classes and interfaces that are a part of a package, there are only two access specifiers available. A particular class or an interface can have either a public access specifier or a default access specifier.

When any member of a package has a default access specifier (no access modifier is mentioned), then such a member is known and usable only within its package. The default access specifier is also known as the package level access. So in the example above, since the class geometry.shapes.Rectangle has a default access specifier, it would not be known in the class test.TestRectangle, which is outside the geometry.shapes package.

When any member of a package has a public access specifier, then such a member is known everywhere, i.e. it is usable even outside the package. Most of the time the classes and interfaces which we create have to be allowed to be used from outside the package, and so they are normally declared to be public. Note that in case of a public class or interface, which are members of a package, the class or interface has to be defined in the file by the same name as the class or the interface name. So the classes that we have defined for geometry.shapes.Rectangle and geometry.shapes.Cuboid must be written in separate files named Rectangle.java and Cuboid.java, respectively, and declared to be `public`. Then these classes can be used from the test.TestRectangle class (Tables 9.1 and 9.2).

9.4.2 Access Specifiers for Members of a Class

For the members of a class there are four access specifiers available. A particular member of a class can be declared to be `private`, default (no access specifier mentioned), `protected` or `public`. All members of an interface are implicitly `public`.

When any member of a class (which is the member of a package) is declared to be private, then that private member is accessible only within the class. The accessibility of private members of a nested type is discussed in Chapter 12. `private` members are not inherited by a sub-class.

TABLE 9.1 Access specifiers for members of a package

Access specifier	Within package	Outside package
(default)	Accessible	Not accessible
`public`	Accessible	Accessible

TABLE 9.2 Access specifiers for members of a class

Access specifier	Within class	Within package	Sub-classes outside package	Non-sub-classes outside package
private	Accessible	Not accessible	Not accessible	Not accessible
(default)	Accessible	Accessible	Not accessible	Not accessible
protected	Accessible	Accessible	Not accessible but inherited	Not accessible
public	Accessible	Accessible	Accessible	Accessible

When any member of a class is declared with a default access specifier (no explicit access specifier is mentioned), then that member is accessible from anywhere within the package to which the class belongs. All classes declared without the package are considered to belong to a single unnamed package. Default members are not inherited by sub-classes outside the package but are inherited by sub-classes within the same package only.

When any member of a public class is declared to be protected, then that member is accessible from anywhere within the package to which the class belongs, and such a member is also inherited by the sub-classes which are outside the package. This inherited member is accessible and usable in the sub-class, which may be outside the package. The protected member of a class cannot be accessed from the sub-class outside the package, when used on an instance of this class itself. The protected member is inherited by a sub-class and can thus be used only as a member of the sub-class instance, but not as a member of the super-class instance.

When any member of a public class is declared to be public, then that member has no access restrictions.

`public` and `protected` members of a non-public member of a package are considered to be default and have only package level accessibility, i.e. if a class or an interface, which is a member of a package, is not `public`, then its `public` and `protected` members will be considered to be default.

The four access specifiers for the members of a class have a hierarchy based on the scope of accesibility. The hierarchy has `private` as the most narrow access, followed by default, which is wider compared to `private`, but narrower compared to `protected` and `public`. The `protected` is broader compared to `private` and default, but narrower compared to `public`.

Most of the time the following are the commonly used access specifiers for the different kinds of members:

Classes and interfaces which are members of a package are normally declared to be public.

The instance variables are most of the time declared to be private, and some of the instance variables may be declared to be protected, to allow use from the sub-classes. However, most of the time instance variables are declared to be private; it would be required in order to have data encapsulation.

The static variables, which are also final, are most of the time declared to be public. Other static variables (non-final) are commonly declared to be private, and there are public static methods which may set or get the values of these variables.

The constructors are most of the time public. In case of abstract classes, the constructors are normally protected. When following a singleton pattern, the constructors are private.

These above access specifiers, for various members, are not rules, these are only indications about how they are commonly used.

9.4.3 Access Specifiers for Overriding Methods

When a sub-class overrides a method from the super-class, then what access specifier is allowed for the overriding method depends on the access specifier of the method being overridden. The rule here is that the overriding method is not allowed to have a weaker (narrower) access specifier compared to the method being overridden.

EXERCISE 9.1 Update the `Account`, `SavingsAccount` and the `CurrentAccount` classes of Exercise 8.1 to belong to a package called `bank`. Now use appropriate access modifiers for the members of all the three classes above. Also update the `TestAccount` class to belong to a package called `test`, and now test which members of the three classes are accessible from `test.TestAccount`.

Hint: In the above exercise, all the instance variables in `Account` may be kept private except for the `balance`, which is required to be used from the sub-classes.

The updated listing for `Account`, `SavingsAccount` and `CurrentAccount` are given in Listings 9.4–9.6.

Listing 9.4. `Account.java`

```
1 package bank;
2
3 public abstract class Account {
4
5     private int accountNumber;
6     private String name;
7     protected double balance;
8
9     public Account(int acno, String n, double openBal) {
10        this.accountNumber = acno;
11        this.name = n;
12        this.balance = openBal;
13    }
14
15    public int getAccountNumber() {
16        return this.accountNumber;
17    }
18
19    public String getName() {
20        return this.name;
21    }
22
23    public double getBalance() {
24        return this.balance;
25    }
26
27    public void deposit(double amt) {
28        this.balance += amt;
29    }
30
31    public boolean withdraw(double amt) {
```

```
32          if (this.balance < amt) {
33              return false;
34          }
35          this.balance -= amt;
36          return true;
37      }
38
39      public void display() {
40  //          System.out.println("Account:"+this.accountNumber+","+this
        .name+","+this.balance);
41          System.out.println(this);
42      }
43
44      private static int lastAccountNumber = 1000;
45
46      public Account(String n, double openBal) {
47          this(++lastAccountNumber, n, openBal);
48      }
49
50      public String toString() {
51          return this.getClass().getName()+":"+this.accountNumber+","
                +this.name+","+this.balance;
52      }
53
54      public boolean equals(Object obj) {
55          if (this.getClass() != obj.getClass()) {
56              return false;
57          }
58          return this.accountNumber == ((Account)obj).accountNumber;
59      }
60
61      public int hashCode() {
62          return this.accountNumber;
63      }
64  }
```

Listing 9.5. SavingsAccount.java

```
1  package bank;
2
3  public class SavingsAccount extends Account {
4      public SavingsAccount(int acno, String n, double openBal) {
5          super(acno, n, openBal);
6      }
7
8      public SavingsAccount(String n, double openBal) {
9          super(n, openBal);
10     }
11 }
```

Listing 9.6. CurrentAccount.java

```
1  package bank;
2
3  public class CurrentAccount extends Account {
4
5      private static final double minimumBalance = 5000;
6      private static final double penalty = 100;
7
8      public CurrentAccount(int acno, String n, double openBal) {
9          super(acno, n, openBal);
10     }
11
12     public CurrentAccount(String n, double openBal) {
13         super(n, openBal);
14     }
15
16     public boolean withdraw(double amt) {
17         if (!super.withdraw(amt)) {
18             return false;
19         }
20         if (this.balance < minimumBalance) {
21             this.balance -= penalty;
22         }
23         return true;
24     }
25 }
```

9.5 REVISITING javadoc

Earlier in Chapter 2, we had used javadoc command to generate documentation for the classes which we define in a Java file. At that time we did not have any public classes and had used the additional switch -package to generate the documentation. The javadoc command has the following switches which could be used to control the documentation generated by javadoc:

- -public
- -protected
- -package
- -private

Using these switches we can control which members are included in the documentation generated by javadoc as shown in Table 9.3.

9.5.1 Generating javadoc According to the Target User

In the javadoc command we could use one of the several switches to control the members included in the documentation. Table 9.3 gives the members included in the documentation

TABLE 9.3 Members included by javadoc depending on the switch used

Switch	Private	Default	Protected	Public
-private	Included	Included	Included	Included
-package	Not included	Included	Included	Included
-protected	Not included	Not included	Included	Included
-public	Not included	Not included	Not included	Included

generated by javadoc command. The default switch is -protected. So if we do not use any of the switches specified in Table 9.3, then by default the documentation generated will include only the protected and public members. This is the documentation which would be useful for normal other users of our classes. When our classes are to be used in various applications by other users, then this documentation is sufficient. When one wants to generate documentation to be used by other members of a development team, then the -package switch would be useful, since the other members are also working on classes which are part of the same package. They would be able to use even the default members. When we want to generate documentation for someone who is going to maintain the given class, then the -private switch is more useful since it will include even the private members in the classes.

EXERCISE 9.2 Generate HTML documentation for all the classes of Exercise 9.1. The documentation should be created in the docs folder only and the folder for source code. Now view the index.html file in the docs folder using a browser. Add appropriate documentation comments for the various elements of the classes.

9.5.2 Using the Java APIs

When we install JDK, we get src.zip file along with the installation. This file contains the source code for the API classes. There is a full Javadoc documentation which is generated from this source and is available as part of documentation for the appropriate version of Java from the Sun Microsystem's download page. When we extract from the documentation downloaded zip file, there is a docs folder which is created. We also find an API folder as a sub-folder in the docs folder. We have an index.html file in the API folder, which is the javadoc documentation of the entire Java APIs. The documentation shows three frames as shown in Figure 9.2.

As shown in the figure, we can access documentation of any class or interface of the API by first selecting the package in the left top frame. In this example, we have selected the package java.lang. Next, we select the class or interface, whose documentation we are interested in. We select the class name from the second frame, which is below the frame for the packages. In this example, we have selected the class called Object. The right-hand frame now shows the documentation of the Object class. There are lots of links for the various kinds of members within the class. We have links to directly jump to a list of fields, or constructors or methods, etc. This can be explored further. Keep exploring the APIs, they contain a lot of documentation and information about the various classes.

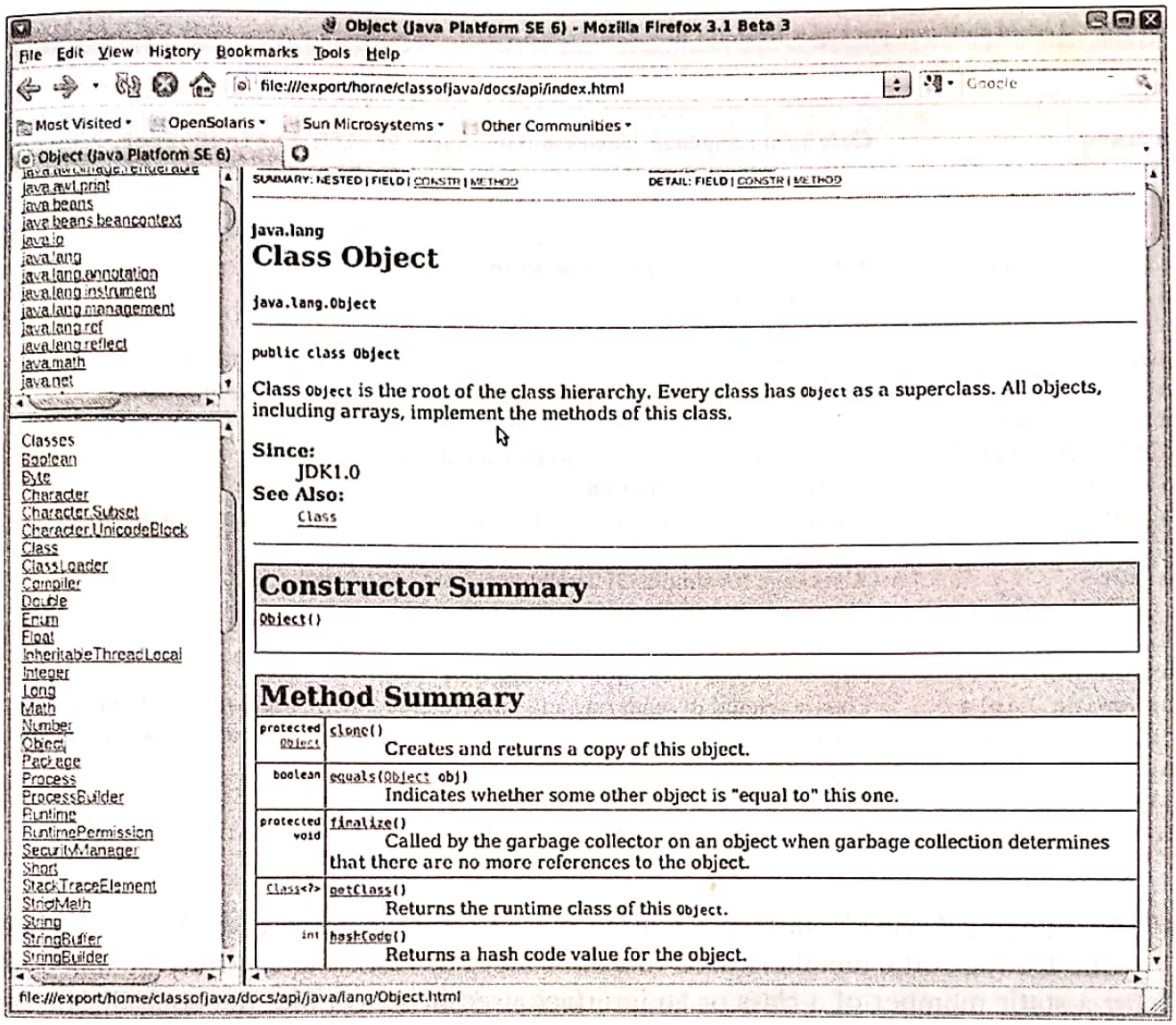


Figure 9.2 javadoc for the APIs

9.5.3 Commonly Used Packages from the Java APIs

Table 9.4 shows the list of most commonly used packages from the Java APIs.

LESSONS LEARNED

- In Java all the classes can be organized into packages. The package for a class is specified by using the `package` statement in the file which contains the class definition. The `package` statement is the first statement in a Java file.
- There is a default unnamed package for classes defined in a Java file which does not have a `package` statement.
- The `import` statement is used by the compiler to resolve the fully qualified name for the classes and interfaces used in a Java file. For resolving the fully qualified name for a class or an interface used in a Java file, the compiler first tries to check if it exists in the current package. If not, it checks using the `import` specified using full names, and the last preference is given to the `import` statements using wild-cards. There is no effect of the import statement at the runtime.

TABLE 9.4 List of common packages from APIs

Package	Description
java.lang	Core Java Language, classes and interfaces String, StringBuffer, StringBuilder, Math, System, Comparable, etc.
java.lang.reflect	Reflection-related classes and interfaces
java.util	Date, Calendar and related classes. Collection framework and other utility classes
java.util.regex	Pattern and Matcher classes for regex
java.util.logging	Logging API
java.util.concurrent	High-level concurrency API
java.io	Classes for file management and stream classes for input and output
java.nio	Buffer classes for use with buffered based input output using channels
java.nio.channels	Channel classes used for input and output
java.nio.charset	Charset class and their encoders and decoders
java.text	Classes for text formatting
java.net	Classes for using TCP- and UDP-based communication over network and URL-related classes
java.awt	A package for simple GUI-related classes
java.awt.image	Classes related to image buffering and rendering
javax.imageio	Classes for loading and saving images in various formats
javax.swing	Pure Java components for creating GUI
javax.swing.table	Classes related to rendering and modelling of JTable and table column
javax.swing.tree	Classes and interfaces related to rendering and modelling of a JTree in swing
java.applet	Class and interface for creating an Applet
java.sql	JDBC API for database connectivity
java.rmi	Classes for creating remote objects used RMI

- The static import statement is used by the compiler to resolve the fully qualified name for the static members used in a Java file. For resolving the fully qualified name for a static member of a class or an interface used in a Java file, the compiler first tries to check if it exists in the current class or interface. If not, it checks using the static import specified using full names, and the last preference is given to the static import statements using wild-cards. The static import statement is used along with the other import statements. The static import is used as the keyword import followed by static and then followed by the static member specification.
- Access specifiers may be specified for classes and interfaces which belong to a package and also to members of a class. The members of a package can have access specifier of either public or default. The members of a class can have any of the four access specifiers: (a) public, (b) protected, (c) default and (d) private. The access specifiers are rated from weaker to stronger, where private is the weakest, default is stronger than private but weaker than protected, protected is weaker than public, which is the strongest.
- Overriding methods cannot narrow the accessibility, i.e. an overriding method cannot have a weaker access specifier than that of the method being overridden.
- The javadoc command can be used with the access specifier switches like -private, -package, -protected and -public to indicate what kind of members must be included in the generated documentation based on the access specifiers.

EXERCISES

1. State which of the following are true or false:
   - **(a)** The `toString()` method can be overridden to be `protected`.
   - **(b)** The default access specifier for members of a class in Java is `private`.
   - **(c)** The default access specifier for members of a class in Java is `public`.
   - **(d)** Methods in an interface can be declared to be `protected`.
   - **(e)** A class declared in a package can be declared to be `private`.
   - **(f)** We do not need `import` statements if the full package name and class name are specified each time we refer to a class in a Java file.
   - **(g)** A sub-class outside the package of the super-class inherits all the methods of its super-class except for the private methods.
   - **(h)** The constructor of a class cannot be declared to be `private`.
   - **(i)** The `finalize()` method cannot be declared to be `private`.
   - **(j)** The only way to use classes from other packages than current and java.lang package is using `import`.
2. Fill in the blanks in the following:
   - **(a)** The environment variable _____ is used to indicate the locations from where classes can be loaded.
   - **(b)** Members of a class specified as _____ are accessible only to methods of the class.
   - **(c)** The access specifiers for the method implemented from an interface in a class has to be _____ only.
   - **(d)** When overriding methods from an interface, only _____ access specifier is valid.
   - **(e)** A `protected` method of a `public` class may be overridden to be _____ or _____ in any of its sub-classes. (specify the appropriate access specifiers)
   - **(f)** An `import` statement in a Java file may only be preceded by _____ statement only.
3. Explain the uses of the keywords `package` and `import`.
4. What are the access specifiers available in Java? Explain each of them. State which of these can be applied to members of a class, and which can be applied to members of a package.
5. Explain how the compiler resolves the fully qualified name of a class or an interface.
6. What is the rule for the access specifier of overriding methods?
7. Explain with an example the use of `static import` in a Java file.