CHAPTER 20

PACKAGES

It is necessary in software development to create several classes and interfaces. After creating these classes and interfaces, it is better if they are divided into some groups depending on their relationship. Thus, the classes and interfaces which handle similar or same task are put into the same directory. This directory or folder is also called a `package`.

Package

A package represents a directory that contains related group of classes and interfaces. For example, when we write statements like:

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

We are importing classes of `java.io` package. Here, java is a directory name and `io` is another sub directory within it. And the '*' represents all the classes and interfaces of that `io` sub directory. Please look at the Figure 20.1.

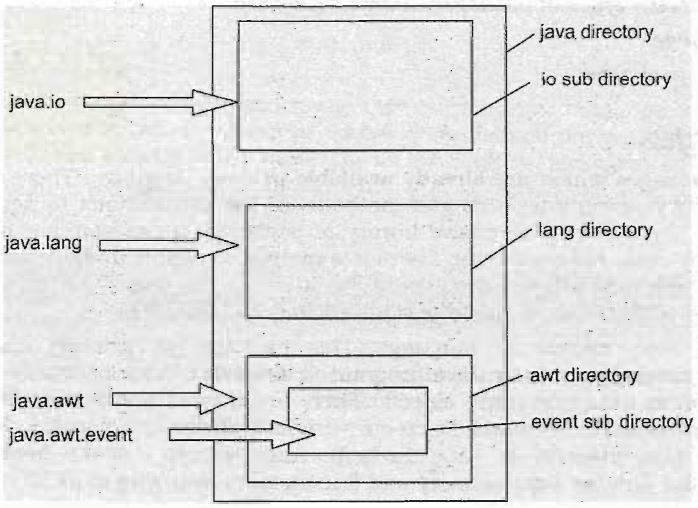


Figure 20.1 Package is a directory.

Important Interview Question

A programmer is writing the following statements in a program:

1. `import java.awt.*;`
2. `import java.awt.event.*;`

Should he write both the statements in his program or the first one is enough?

`event` *is a sub package of* `java.awt` *package. But, when a package is imported, its sub packages are not automatically imported into a program. So, for every package or sub package, a separate import statement should be written. Hence if the programmer wants the classes and interfaces of both the* `java.awt` *and* `java.awt.event` *packages, then he should write both the preceding statements in his program.*

There are several following advantages of package concept:

- Packages are useful to arrange related classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example, in Java, all the classes and interfaces which perform input and output operations are stored in `java.io` package.
- Packages hide the classes and interfaces in a separate sub directory, so that accidental deletion of classes and interfaces will not take place.
- The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different classes. For example, there is a Date class in `java.util` package and also there is another Date class available in `java.sql` package.
- A group of packages is called a library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy. Just think, the packages in Java are created by JavaSoft people only once, and millions of programmers all over the world are daily by using them in various programs.

Different Types of Packages

There are two different types of packages in Java. They are:

- Built-in packages
- User-defined packages

Built-in Packages

These are the packages which are already available in Java language. These packages provide all most all necessary classes, interfaces and methods for the programmer to perform any task in his programs. Since, Java has an extensive library of packages, a programmer need not think about logic for doing any task. For everything, there is a method available in Java and that method can be used by the programmer without developing the logic on his own. This makes the programming easy. Here, we introduce some of the important packages of Java SE:

- `java.lang: lang` stands for language. This package got primary classes and interfaces essential for developing a basic Java program. It consists of wrapper classes which are useful to convert primitive data types into objects. There are classes like String, StringBuffer to handle strings. There is a Thread class to create various individual processes. Runtime and System classes are also present in `java.lang` package which contain methods to execute an application and find the total memory and free memory available in JVM.

Important Interview Question

How can you call the garbage collector?

We can call garbage collector of JVM to delete any unused variables and unreferenced objects from memory using `gc()` *method. This* `gc()` *method appears in both Runtime and System classes of* `java.lang` *package. For example, we can call it as:*

System.gc();

Runtime.getRuntime().gc();

- `java.util`: `util` stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector, Arrays, etc. These classes are called collections. There are also classes for handling date and time operations.
- `java.io`: `io` stands for input and output. This package contains streams. A stream represents flow of data from one place to another place. Streams are useful to store data in the form of files and also to perform input-output related tasks.
- `java.awt`: `awt` stands for abstract window toolkit. This package helps to develop GUI (Graphics User Interface) where programs with colorful screens, paintings and images etc., can be developed. It consists of an important sub package, `java.awt.event`, which is useful to provide action for components like push buttons, radio buttons, menus etc.
- `javax.swing`: This package helps to develop GUI like `java.awt`. The 'x' in `javax` represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, `javax.swing` is an extended package of `java.awt`.
- `java.net`: `net` stands for network. Client-Server programming can be done by using this package. Classes related to obtaining authentication for a network, creating sockets at client and server to establish communication between them are also available in `java.net` package.
- `java.applet`: Applets are programs which come from a server into a client and get executed on the client machine on a network. Applet class of this package is useful to create and use applets.
- `java.text`: This package has two important classes, `DateFormat` to format dates and times, and `NumberFormat` which is useful to format numeric values.
- `java.sql`: `sql` stands for structured query language. This package helps to connect to databases like Oracle or Sybase, retrieve the data from them and use it in a Java program.

User-defined Packages

Just like the Built-in packages shown earlier, the users of the Java language can also create their own packages. They are called user-defined packages. User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

Let us see how to create a package of our own and use it in any other class. To create a package the keyword `package` used as:

```
package packagename;  //to create a package
package packagename.subpackagename;  //to create a sub package within a
                                //package
```

The preceding statements in fact create a directory with the given package name. We should of course add our classes and interfaces to this directory. In program 1, we are creating a package with the name `pack` and adding a class `Addition` to it. This `Addition` class has a method `void sum()` which performs addition of two given numbers. One point we should understand while creating the class in a package is to declare all the members and the class itself as `public`, except the instance variables. The reason is only that the public members are available outside the package to other programs.

Program 1: Write a program to create a package with the name pack and store Addition class in it.

```
//STEP 1: creating a package pack with Addition class
package pack;  //pack is the package name
public class Addition
{
        //instance vars
        private double d1,d2;

        public Addition(double a, double b)
        {
                d1=a;
                d2=b;
        }

        //method to find sum of two numbers
        public void sum()
        {
                System.out.println("Sum= "+(d1+d2));
        }
}
```

Output:

```
C:\> javac -d . Addition.java
C:\>
```

See the output to understand how to compile a Java program that contains a package. The -d option (switch) tells the Java compiler to create a separate sub directory and place the .class file there. The dot (.) after -d indicates that the package should be created in the current directory i.e., C:\. We have written as:

```
javac -d . Addition.java
```

The preceding command means create a package (-d) in the current directory (.) and store Addition.class file there in the package. The package name is specified in the program as pack. So the Java compiler creates a directory in C:\ with the name as pack and stores Addition.class there. Please observe it by going to pack sub directory which is created in C:\.

So, our package with Addition class is ready. The next step is to use the Addition class and its sum() method in a program. For this purpose, we write another class Use as shown in Program 2. In this program, we can refer to the Addition class of package pack using membership operator (.) as,

```
pack.Addition
```

Now, to create an object to Addition class, we can write as:

```
pack.Addition obj = new pack.Addition(10, 15.5);
```

Program 2: Write a program which depicts how to use the Addition class of package pack.

```
//STEP 2: Using the package pack
class Use
{
        public static void main(String args[ ])
        {
                //create Addition class object
                pack.Addition obj = new pack.Addition(10, 15.5);

                //call the sum() method
                obj.sum();
        }
}
```

Output:

```
C:\> javac Use.java
C:\> java Use
Sum= 25.5
```

Every time we refer to a class of a package, we should write the package name before the class name as `pack.Addition` in the preceding program. This is inconvenient for the programmer. To overcome this, we can use import statement only once in the beginning of the program, as:

```
import pack.Addition;
```

Once the import statement is written as shown earlier, we need not to use the package name before the class name in the rest of the program and we can create the object to Addition class, in a normal way as:

```
Addition obj = new Addition(10, 15.5);
```

This is shown in Program 3, given here.

Program 3: Write a program which is using the import statement to import a package and its classes into a program.

```
//STEP 2: Using the package pack
import pack.Addition;
class Use
{
        public static void main(String args[ ])
        {
                //create Addition class object
                Addition obj = new Addition(10, 15.5);

                //call the sum() method
                obj.sum();
        }
}
```

Output:

```
C:\> javac Use.java
C:\> java Use
Sum= 25.5
```

Of course, we have seen how to create a package pack and add a class Addition to it. Similarly, we can add another class Subtraction with `sub()` method which performs subtraction of two numbers. To add another class to the package, the same procedure should be repeated. First of all, write the Subtraction class with the package statement, as shown in Program 4. And then write `Subtraction` class and its members as public.

Program 4: Write a program to add another class Subtraction to the same package `pack`.

```
//Adding another class to the package :pack
package pack;
public class Subtraction
{
        //a static method to return result of subtraction
        public static double sub(double a, double b)
        {
                return (a - b);
        }
}
```

Output:

```
C:\> javac -d . Subtraction.java
C:\>
```

See the preceding output. Here, the Java compiler checks whether the package with the name pack already exists or not. If it is existing, then it adds `Subtraction.class` to it. If the package pack does not exist then Java compiler creates a sub directory with the name pack in the current directory and adds `Subtraction.class` to it. Since package pack is already existing, the `Subtraction.class` file is also added to the existing package where already `Addition.class` file is available.

Let us see how to use the `Subtraction` class in our `Use.java` program. The Program 3 can be rewritten as shown here.

Program 5: Let us make a program using both the Addition and Subtraction classes of the package pack.

```
//Using the package pack
import pack.Addition;
import pack.Subtraction;
class Use
{
        public static void main(String args[ ])
        {
                //create Addition class object
                Addition obj = new Addition(10, 15.5);

                //call the sum() method
                obj.sum();

                //call the sub() method and pass values
                double res = Subtraction.sub(10, 15.5);

                System.out.println("Result= "+ res);
        }
}
```

Output:

```
C:\> javac Use.java
C:\> java Use
Sum= 25.5
Result= -5.5
```

We should understand that generally a package is created by a programmer and it is used another programmer in some other program. For example, the package pack with Addition and Subtraction class is created by a programmer and another user is using that package

Use.java program. Now the question is how the other user knows that the package pack has got Addition class and Subtraction class, and there are sum() and sub() methods available in those classes? For this purpose, the user takes the help of API (application programming interface) document which contains the description of all the packages, classes, methods etc. The API document is discussed later in this chapter.

Observe in Program 4, we are using multiple import statements as:

```
        import pack.Addition;
        import pack.Subtraction;
```

When we want to use classes of the same package, we need not write separate import statements as shown earlier. We can write a single import statement as:

```
        import pack.*;
```

Here, '*' represents all the classes and interfaces of a particular package, in this case, it is the package pack.

Important Interview Question

What is the difference between the following two statements:

1) import pack.Addition;

2) import pack.;*

In statement 1, only the Addition class of the package pack is imported into the program and in statement 2, all the classes and interfaces of the package pack are available to the program.

If a programmer wants to import only one class of a package say BufferedReader *of* java.io *package, he can write:*

```
        import java.io.BufferedReader;
```

This is straight and the Java compiler links up the BufferedReader of java.io package with the program. But, if he writes import statement as:

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

In this case, the Java compiler conducts a search for BufferedReader class in java.io package, every time it is used in the rest of the program. This increases load on the compiler and hence compilation time increases. However, there will not be any change in the runtime.

Let us rewrite Program 4, as shown here. Here, we are using import statement to represent all the classes of the package pack, as:

```
        import pack.*;
```

In this case, please be sure that any of the Addition.java and Subtraction.java programs will not exist in the current directory. Delete them from the current directory as they cause confusion for the Java compiler. The compiler looks for byte code in Addition.java and Subtraction.java files and there is no byte code available there and hence it flags some errors.

Program 6: Write a program to use import statement in a different way. Remove Addition.java and Subtraction.java from current directory and then test this program.

```
        //Using the package pack
        import pack.*;
        class Use
```

```
{
        public static void main(String args[ ])
        {
                //create Addition class object
                Addition obj = new Addition(10, 15.5);

                //call the sum() method
                obj.sum();

                //call the sub() method and pass values
                double res = Subtraction.sub(10, 15.5);

                System.out.println("Result= "+ res);
        }
}
```

Output:

```
C:\> javac Use.java
C:\> java Use
Sum= 25.5
Result= -5.5
```

Of course, the package pack is available in the current directory. If the package is not availabl the current directory, then what happens? Suppose our program is running in C:\ and the pack pack is available in the directory D:\sub. In this case, the compiler should be given informat regarding the package location by mentioning the directory name of the package in class path.

Class path represents an operating system's environment variable which stores active direc path such that all the files in those directories are available to any programs in the sys Generally, it is written in all capital letters as CLASSPATH.

Important Interview Question

What is CLASSPATH?

The CLASSPATH *is an environment variable that tells the Java compiler where to look for class f to import.* CLASSPATH *is generally set to a directory or a JAR (Java Archive) file.*

To see what is there currently in the CLASSPATH variable in your system, you can type in Win 98/2000/Me/NT/XP/Vista:

```
C:\> echo %CLASSPATH%
```

Suppose, preceding command has displayed class path as:

```
c:\rnr;.
```

This means the current class path is set to rnr directory in C:\ and also to the current di represented by dot (.). Our package pack does not exist in either rnr or current directo package exists in D:\sub. This information should be provided to the Java compiler by sett class path to d:\sub, as shown here:

```
C:\>set CLASSPATH=D:\sub;.;%CLASSPATH%
```

In the preceding command, we are setting the class path to sub directory and current direc And then we typed %CLASSPATH% which means retain the already available class path as it is. is necessary especially when the class path in your system is already set to an imp application that should not be disturbed.

Please create a new directory in `D:\` with the name `sub` and copy our package `pack` directory into that directory. Now, set the class path to that `sub` directory as:

```
C:\>set CLASSPATH=D:\sub;.;%CLASSPATH%
```

Then execute our program `Use.java` which is in `C:\`, by typing:

```
C:\> javac Use.java
C:\> java Use
```

Alternately, you can mention the class path at the time of executing the program at command line using –cp (classpath) option, as:

```
C:\> javac -cp D:\sub;.  Use.java
C:\> java -cp D:\sub;.  Use
```

Remember, here our `Use.java` program is in the current directory and the package pack is available in `D:\sub` directory. So, it is possible to use the package by setting the class path to `D:\sub;.` and execute the program.

The JAR Files

A JAR (Java Archive) file is a file that contains compressed version of `.class` files, audio files, image files or directories. We can imagine a `.jar` file as a zipped file (`.zip`) that is created by using WinZip software. Even, WinZip software can be used to extract the contents of a `.jar` file. The difference is that a `.jar` file can be used as it is but whereas the `.zip` file can not be used directly. The files should be extracted first from a `.zip` file, and then used.

Important Interview Question

What is a JAR file?

A Java Archive file (JAR) is a file that contains compressed version of several `.class` files, audio files, image files or directories. JAR file is useful to bundle up several files related to a project and use them easily.

Let us see how to create a `.jar` file and related commands which help us to work with `.jar` files:

- To create `.jar` file, JavaSoft people have provided jar command, which can be used in the following way:

```
jar cf jarfilename inputfiles
```

Here, cf represents `create file`. For example, assuming our package `pack` is available in `C:\` directory, to convert it into a jar file with the name `pack.jar`, we can give the command as:

```
C:\> jar cf pack.jar pack
```

Now, `pack.jar` file is created.

- To view the contents of a .jar file, we can use the jar command as:

```
jar tf jarfilename
```

After setting the CLASSPATH permanently as shown in the preceding steps to the pack.jar file, it is available any where in that computer system. Our program (Use.java) which uses the package may present in any directory, it can be compiled and run without any problem.

Interfaces in a Package

It is also possible to write interfaces in a package. But whenever, we create an interface the implementation classes are also should be created. We cannot create an object to the interface but we can create objects for implementation classes and use them. Let us see how to do this. We write an interface to display system date and time in the package mypack as shown in the following Program 6.

Program 7: Write a program to create an interface with a single method to display system date and time.

```
//Create MyDate interface in the package mypack
package mypack;
public interface MyDate
{
        void showDate(); //public abstract
}
```

Output:

```
C:\> javac -d . MyDate.java
C:\>
```

Compile the preceding code and observe that the Java compiler creates a sub directory with the name mypack and stores MyDate.class file there. This MyDate.class file denotes the byte code of the interface. The next step is to create implementation class for MyDate interface where showDate() method body should be written. This is done in Program 7. DateImpl is an implementation class where the body for showDate() method is written. Here, we create an object to Date class (of java.util package) which by default stores the system date and time.

Program 8: Write a program to create an implementation class for the MyDate interface with the name DateImpl and storing it in the same package mypack.

```
//This is implementation class of MyDate interface
package mypack; //store DateImpl class also in mypack
import mypack.MyDate;
import java.util.*;

public class DateImpl implements MyDate
{
        public void showDate()
        {
                //Date class object by default stores system date and time
                Date d = new Date();
                System.out.println(d);
        }
}
```

Output:

```
C:\> javac -d . DateImpl.java
C:\>
```

When the preceding code is compiled, `DateImpl.class` file is created in the same package mypac
`DateImpl` class contained `showDate()` method which can be called and used in any other progra
This is shown in Program 8. In this program, `DateDisplay` is a class where we want to us
`DateImpl` class. So, an object to `DateImpl` is created and the `showDate()` method is called. Th
displays system date and time.

Program 9: Write a program which shows how to use the DateImpl which is an implementati
class of MyDate interface.

```
//Using the DateImpl of mypack
import mypack.DateImpl;
class DateDisplay
{
	public static void main(String args[ ])
	{
		//create DateImpl object
		DateImpl obj = new DateImpl();

		//call showDate()
		obj.showDate();
	}
}
```

Output:

```
C:\> javac DateDisplay.java
C:\> java DateDisplay
Tue Aug 07 21:14:43 IST 2007
```

An alternative approach where the `MyDate` interface reference can be used to refer to the `DateIm`
class object to access all the methods of the `DateImpl` class is shown in Program 9.

Program 10: Write a program where `MyDate` interface reference is used to object of `DateImpl` class

```
//This is another version of Program 8.
import mypack.MyDate;
import mypack.DateImpl;
class DateDisplay
{
	public static void main(String args[ ])
	{
		//MyDate interface reference is used to refer to DateImpl object
		MyDate obj = new DateImpl();

		//call showDate()
		obj.showDate();
	}
}
```

Output:

```
C:\> javac DateDisplay.java
C:\> java DateDisplay
Tue Aug 07 21:14:43 IST 2007
```

Creating Sub Package in a Package

We can create sub package in a package in the format:

```
package pack1.pack2;
```

Here, we are creating pack2 which is created inside pack1. To use the classes and interfaces of pack2, we can write import statement as:

```
import pack1.pack2;
```

This concept can be extended to create several sub packages. In the following program, we are creating `tech` package inside `dream` package by writing the statement:

```
package dream.tech;
```

And in this sub package, we are storing Sample class with a method `show()`.

Program 11: Let us make a program to learn how to create a sub package in a package.

```
//Creating a sub package tech in the package dream
package dream.tech;
public class Sample
{
        public void show()
        {
                System.out.println("Welcome to Dream tech");
        }
}
```

Output:

```
C:\> javac -d . Sample.java
C:\>
```

When the preceding program is compiled, the Java compiler creates a sub directory with the name dream. Inside this, there would be another sub directory with the name `tech` is created. In this tech directory, Sample class is stored. Suppose the user wants to use the Sample class of `dream.tech` package, he can write a statement as:

```
import dream.tech.Sample;
```

Program 12: Let us make a program using Sample class of `dream.tech` package.

```
//Using the package dream.tech
import dream.tech.Sample;
class Use
{
  public static void main(String args[])
  {
    //create an object to Sample class
    Sample s = new Sample();

    //call the show() method
    s.show();
  }
}
```

Output:

```
C:\> javac Use.java
C:\> java Use
Welcome to Dreamtech
```

Suppose, in the preceding program, we want to use the import statement by using a '*' to refer to the classes of the package, we can write:

```
import dreamtech.*;
```

In this case, any source code related to Sample class (e.g. Sample.java) should be deleted from t current directory (i.e., from C:\). Then compiling Program 11 and executing it will yield the corr result.

If the dreamtech package is not in the current directory, then we should set the class path to t directory which holds the dreamtech package. For example, copy dreamtech package into so other directory, say e:\temp. To link up this package with Use.java program, we should set class path in the current directory as:

```
C:\> set CLASSPATH=E:\temp;.;%CLASSPATH%
```

Now compile and run the Use.java program to yield correct output.

Access Specifiers in Java

We have seen that any class and its members within a package should be declared as public. Th only the class and its members will be available for use outside the package. Let us discuss w exact difference is there among various access specifiers in Java.

An access specifier is a key word that is used to specify how to access a member of a class or class itself. It means, we can use access specifiers with the members of a class or with the c also. There are four access specifiers in Java: private, public, protected and default. We do not default key word for specifying the default access specifier. If no other access specifier is u then Java compiler will take it as default access specifier.

In the Figure 20.2, we are taking three classes, class A, class B and class C. Whereas class A class B exists in the same package (same directory), class C is away from them in another pack (another directory). Let us assume that there are four members with private, public, protected a default specifiers in class A. Now let us discuss which of these members will be available to cla and class C.

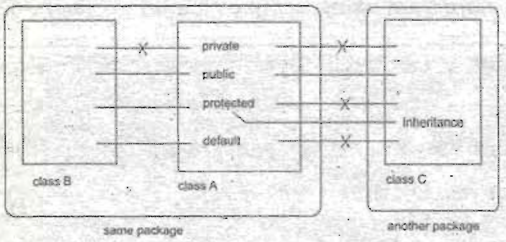


Figure 20.2 Availability of access specifiers in packages

- ❑ private members of class A are not available to class B or class C. This means, any pr member's scope is limited only to that class where it is defined. So the scope of private a specifier is class scope.

- public members of class A are available to class B and also class C. This means, public members are available every where and their scope is `global scope`.
- protected members of class A are available to class B, but not in class C. But if, class C is a sub class of class A, then the protected members of class A are available to class C. So, protected access specifier acts as public with respect to sub classes.
- When no access specifier is used, it is taken as default specifier. Default members of class A are accessible to class B which is within the same package. They are not available to class C. This means, the scope of default members is `package scope`.

Important Interview Question

What is the scope of default access specifier?

Default members are available within the same package, but not outside of the package. So their scope is package scope.

Let us create class A with private, public, protected and default instance variables, as shown here.

Program 13: Write a program to create class A with different access specifiers.

```
//class A of same package
package same;
public class A
{
        private int a=1;
        public int b=2;
        protected int c=3;
        int d=4; //default
}
```

Output:

```
C:\> javac -d . A.java
C:\
```

Now, create another class B in the same package as class A, as shown here. In class B, we are trying to access class A's members. When class B is compiled there would be 1 error saying that the private member of class A is not accessible in class B. It means, the public, protected and default members are accessible in class B. This is verified in Program 13.

Program 14: Write a program for creating class B in the same package.

```
//class B of same package
package same;
import same.A;
public class B
{
        public static void main(String args[ ])
        {
                //access the members of class A
                A obj = new A();
                System.out.println(obj.a);
                System.out.println(obj.b);
                System.out.println(obj.c);
                System.out.println(obj.d);
        }
}
```

Output:

```
C:\> javac -d . B.java
B.java:10: a has private access in same.A
     System.out.println(obj.a);
                           ^
1 error
```

Let us create class C in another package. We want to access all the members of class A from C. I this case, only the public member of class A will be available to class C. Other members such a private, protected and default are not available to C.

Program 15: Write a program for creating class C of another package

```
//class C of another package
package another;
import same.A;
public class C
{
     public static void main(String args[ ])
     {
          //access the members of class A
          A obj = new A();
          System.out.println(obj.a);
          System.out.println(obj.b);
          System.out.println(obj.c);
          System.out.println(obj.d);
     }
}
```

Output:

```
C:\> javac -d . C.java
C.java:10: a has private access in same.A
     System.out.println(obj.a);
                           ^
C.java:10: c has protected access in same.A
     System.out.println(obj.c);
                           ^
C.java:10: d is not public in same.A; cannot be accessed from outside package
     System.out.println(obj.d);
                           ^

3 errors
```

If class C is a sub class of class A, then the protected members of class A will be available to clas in addition to public members. This is shown in Program 15.

Program 16: Write a program for creating class C is a sub class of class A.

```
//class C of another package
package another;
import same.A;
public class C extends A
{
     public static void main(String args[])
     {
          //access the members of class A
          C obj = new C();
          System.out.println(obj.a);
          System.out.println(obj.b);
          System.out.println(obj.c);
          System.out.println(obj.d);
     }
```

```
    }
```

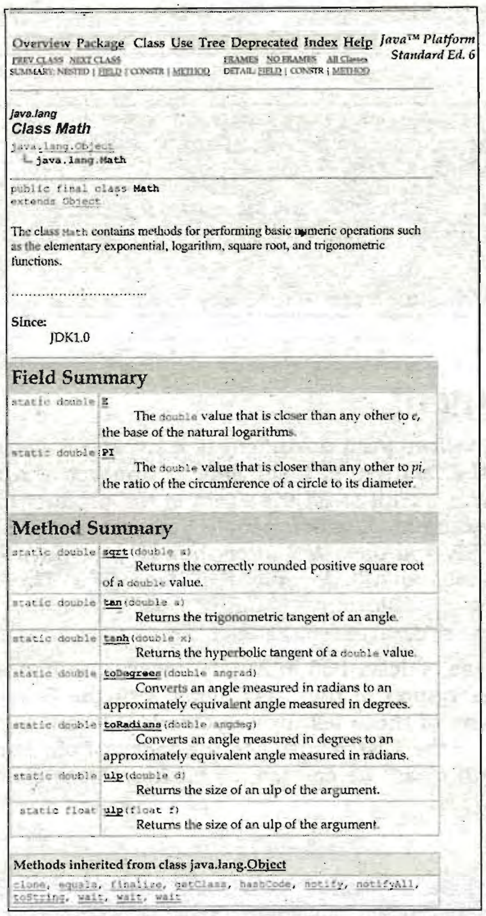
Output:

```
    C:\> javac -d . C.java
    C.java:10: a has private access in same.A
         System.out.println(obj.a);
                                ^
    C.java:10: d is not public in same.A; cannot be accessed from outside package
         System.out.println(obj.d);
                                ^
    2 errors
```

Creating API Document

Application Programming Interface (API) document is a hypertext markup language (html) file that contains description of all the features of a software, a product or a technology. API document is like a reference manual that is useful to all the users of the product to understand all the features and how to use them. For example, JavaSoft people have created API document separately for the three parts, Java SE, Java EE and Java ME after they created Java language. In Figure 20.3, we have shown some part of original API document of Java SE. The Java documentation can be downloaded from the following page of Sun Micro systems site, `http://java.sun.com/javase/downloads/index.jsp`.

Every feature of Java language is described in API document. We can select any package, any class in that package and the description of the class along with the fields, constructors, methods will appear. When we click on any of these features, a detailed description of the feature is displayed. We can also click on Index of the first row to see the index of all items in alphabetical order. For example, to know about Math class, we can simply click on Math in the index to have a page , as shown in Figure 20.3



Overview Package Class Use Tree Deprecated Index Help *Java™ Platform Standard Ed. 6*

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang
Class Math

```
java.lang.Object
  └ java.lang.Math
```

```
public final class Math
extends Object
```

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

....

Since:
JDK1.0

Field Summary

`static double`	**`E`** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	**`PI`** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

`static double`	**`sqrt`**`(double a)` Returns the correctly rounded positive square root of a `double` value.
`static double`	**`tan`**`(double a)` Returns the trigonometric tangent of an angle.
`static double`	**`tanh`**`(double x)` Returns the hyperbolic tangent of a `double` value.
`static double`	**`toDegrees`**`(double angrad)` Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
`static double`	**`toRadians`**`(double angdeg)` Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
`static double`	**`ulp`**`(double d)` Returns the size of an ulp of the argument.
`static float`	**`ulp`**`(float f)` Returns the size of an ulp of the argument.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Figure 20.3 API document of Java SE

There are three important uses of API document:

- ❑ API document is useful for the user to understand all the features of the product. The user understand how to create an object to a class, which fields and methods are available in class and how to use them.
- ❑ API document makes programming easy. By understanding the methods of the API, programmer will be able to use those methods and construct his programs easily.
- ❑ Any third party vendor can provide implementation classes for the interfaces depending on information available on the interfaces in the API.

Let us now see how to create an API document for any software:

- ❑ First of all, we should copy all the source code files (`.java`) into a directory. For example, `Addition.java` and `Subtraction.java` of the package pack into a directory `e:\temp`.
- ❑ We should open each of the source code files and provide Java documentation comments using `/**` and `*/`. When documentation comments are written, the `Addition.java` `Subtraction.java` files look like shown here:

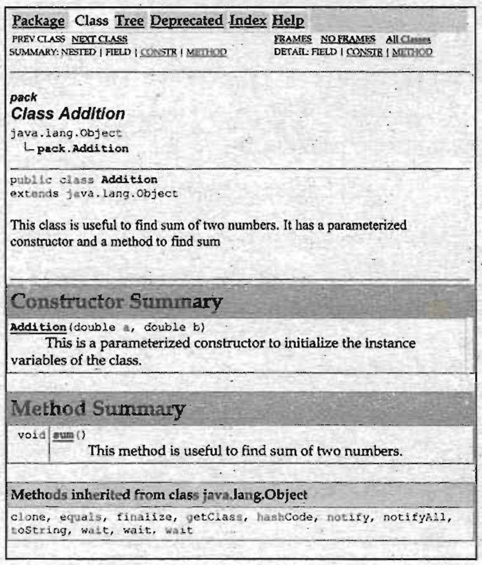
```
//Addition.java

/** This package is useful to perform some arithmetic calculations. It has two
classes by the name Addition and Subtraction.*/

package pack;

/** This class is useful to find sum of two numbers. It has a parameterized
constructor and a method to find sum */

public class Addition
{
 private double d1,d2;

/** This is a parameterized constructor to initialize the instance variables of
the class. */

 public Addition(double a, double b)
 {
  d1=a;
  d2=b;
 }

/** This method is useful to find sum of two numbers. It does not accept any
parameters.

<br>Parameters: nil
<br>Return type: void
<br>Exceptions: nil
*/

 public void sum()
 {
  System.out.println("Sum= "+(d1+d2));
 }
}
```

```
//Subtraction.java
//Adding another class to package: pack

/** This package is useful to perform some arithmetic calculations. It has two
classes by the name Addition and Subtraction.*/

package pack;

/** This class is useful to calculate subtraction on two values */

public class Subtraction
{

/* This method is useful to subtract a number from another and return the result

<br> Parameters: double x, double y
<br> Return type: double
<br> Exceptions: nil
 */
   public static double sub(double x,double y)
   {
     return x-y;
   }
}
```

- After adding Java documentation comments, we should save the `Addition.java` a `Subtraction.java` files. Then, to generate API document, we should give the follo command:

```
    E:\temp> javadoc  *.java
```

Here, we are calling `javadoc` compiler to create the API document. When the preceding comman typed, then several `.html` files are created in `E:\temp` directory. Among them, observe `index.h` file and open it in the browser. It looks as shown here in Figure 20.4.



Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

pack
Class Addition

```
java.lang.Object
  └ pack.Addition
```

```
public class Addition
extends java.lang.Object
```

This class is useful to find sum of two numbers. It has a parameterized constructor and a method to find sum

Constructor Summary

`Addition(double a, double b)`
This is a parameterized constructor to initialize the instance variables of the class.

Method Summary

`void sum()`
This method is useful to find sum of two numbers.

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Figure 20.4 API document for our own software

Conclusion

A package is a directory that stores all the related classes and interfaces. As part of the sof development, several packages are developed which are collectively called a library. These lib are reusable in further projects and make programming very easy. But the question is ho users know about all the packages and their members available in the libraries? To solv problem, API document is provided by the programmers. API document is developed b programmers at the end of software development which helps like a reference manu understand all the features of the software.