UNIT-II

Packages- Defining a Package, CLASSPATH, Access protection, importing packages.

Interfaces- defining an interface, implementing interfaces, Nested interfaces, applying interfaces, variables in interfaces and extending interfaces.

Stream based I/O(java.io) – The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and writing Files, Random access file operations, The Console class, Serialization, Enumerations, auto boxing, generics.

PACKAGES

DEFINING PACKAGE:

Packages are containers for classes that are used to keep the class name space compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

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

import java.io.*;

Here we are importing classes of **java.io.**package. Here, **java** is a directory name and **io**is another sub directory within it. The **'*'**represents all the classes and interfaces of that **io** sub directory. We can create our own packages called user-defined packages or extend the available packages. User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

USE: Packages provide re-usability.

ADVANTAGES OF PACKAGES:

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 in java.sql package.

A group of package called a library. The classes and interfaces of a package are likes books in a library and can be reused several times. This reusability nature of packages makes programming easy.

Just think, the package 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 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 extensie 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(Integer, Character, Float etc), which are useful to convert primitive data types into objects. There are classes like String, SttringBuffer, StringBuilder classes 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.

Java.util:util stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector, Arrays, etc. thses classes are 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 helps to develop GUI(Graphical user Interfaces) 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.

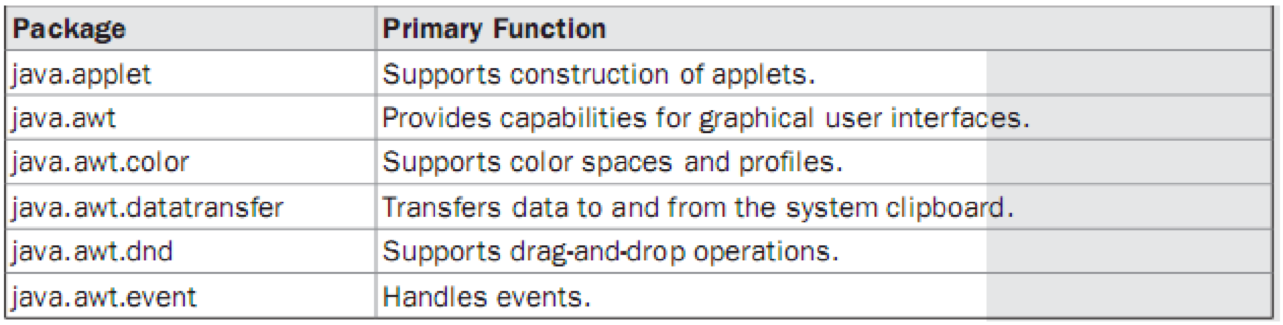
Javax.swing:this package helps to develop GUI like java.awt. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.

Java.net:net stands for network. Client-Server programming can be done by using this package. Classes related to obtaining authentication for 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 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.



Package	Primary Function
java.applet	Supports construction of applets.
java.awt	Provides capabilities for graphical user interfaces.
java.awt.color	Supports color spaces and profiles.
java.awt.datatransfer	Transfers data to and from the system clipboard.
java.awt.dnd	Supports drag-and-drop operations.
java.awt.event	Handles events.

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.

CREATING AND IMPORTING PACKAGES:

```
package packagename; //to create a package

package packagename.subpackagename;//to create a sub package within a
package. e.g.: package pack;
```

The first statement in the program must be package statement while creating a package.

While creating a package except instance variables, declare all the members and the class itself as public then only the public members are available outside the package to other programs.

Program 1: Write a program to create a package pack with Addition class.

```
 package pack; public class Addition
 {
     private double d1,d2;

    public Addition(double a, double b)

   { d1 = a;

    d2 = b; }

 public void sum()

{ System. out. println ("Sum of two given numbers is : " + (d1+d2) );

} }
```

Compiling the above program:

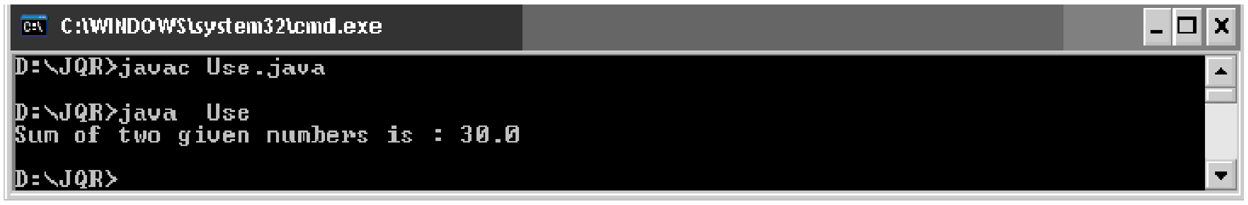
javac -d . Addition.java

The –d option tells the Java compiler to create a separate directory and place the .class file in that directory (package). The (.) dot after –d indicates that the package should be created in the current directory. So, out package pack with Addition class is ready.

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

```
//Using the package pack
 import pack.Addition; class Use
{ public static void main(String args[])
{ Addition ob1 = new Addition(10,20); ob1.sum();
}

}
```

Output:

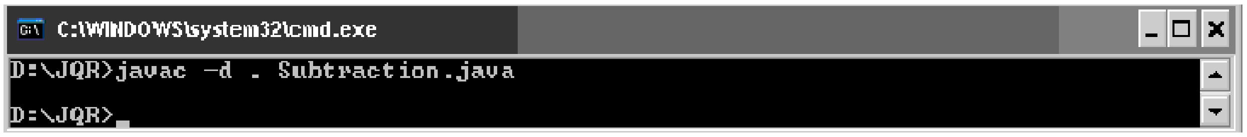


```
D:\JQR>javac Use.java

D:\JQR>java  Use
Sum of two given numbers is : 30.0

D:\JQR>
```

Program 3: Write a program to add one more class Subtraction to the same package pack.

```
//Adding one more class to package pack:
 package pack;
public class Subtraction

{

private double d1,d2;

public Subtraction(double a, double b)
{

d1 = a; d2 = b;
}

public void difference()

{ System.out.println ("Sum of two given numbers is : " + (d1 - d2) );

}

}
```

Compiling the above program:

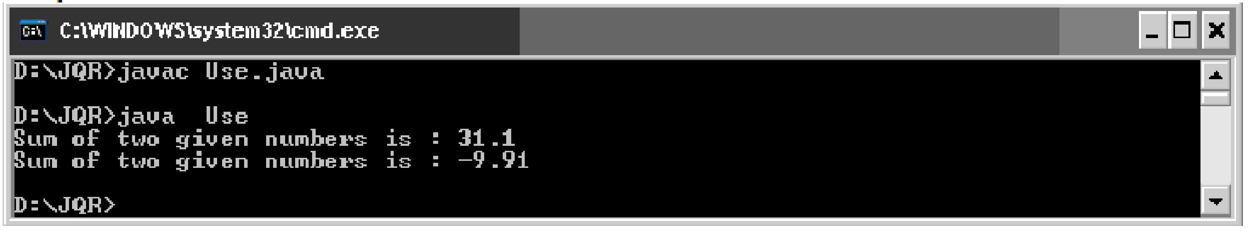

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Subtraction.java

D:\JQR>_
```

Program 4: Write a program to access all the classes in the package pack.

```
//To import all the classes and interfaces in a class using import pack.*;
 import pack.*;-
class Use

{ public static void main(String args[])

{ Addition ob1 = new Addition(10.5,20.6); ob1.sum();
Subtraction ob2 = new Subtraction(30.2,40.11); ob2.difference();
}

}
```

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 it gets no byte code and hence it flags some errors.

Output:

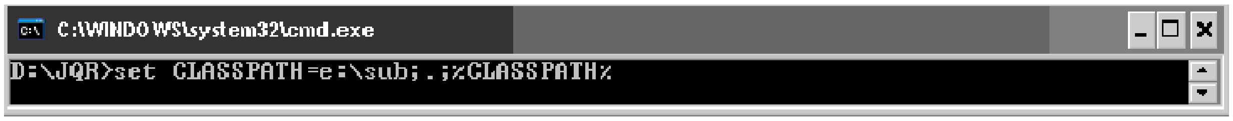

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Use.java

D:\JQR>java  Use
Sum of two given numbers is : 31.1
Sum of two given numbers is : -9.91

D:\JQR>
```

UNDERSTANDING CLASSPATH:

If the package pack is available in different directory, in that case the compiler should be given information regarding the package location by mentioning the directory name of the package in the classpath.

The CLASSPATH is an environment variable that tells the Java compiler where to look for class files to import.

If our package exists in e:\sub then we need to set class path as follows:



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

We are setting the classpath to e:\sub directory and current directory (.) an

%CLASSPATH% means retain the already available classpath as it is.

Creating Sub package in a package: We can create sub package in a package in the

format: package packagename.subpackagename;

e.g.: package pack1.pack2;

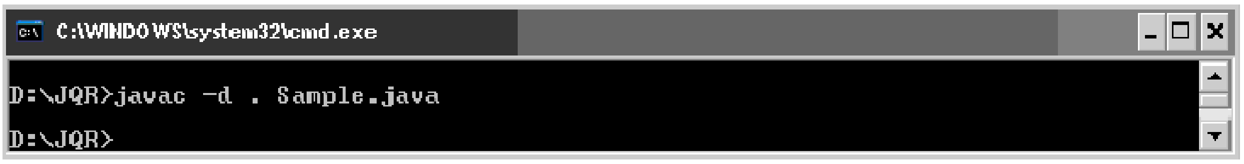
Here, we are creating pack2 subpackage which is created inside pack1 package.
To use the classes and interfaces of pack2, we can write import statement as:

import pack1.pack2;

Program 5: Program to show how to create a subpackage in a package.

```
//Creating a subpackage in a package
package pack1.pack2;
public class Sample

{ public void show ()

{

System.out.println ("Hello Java Learners");

}

}
```

Compiling the above program:



```
D:\JQR>javac -d . Sample.java

D:\JQR>
```

ACCESSING A PACKAGES:

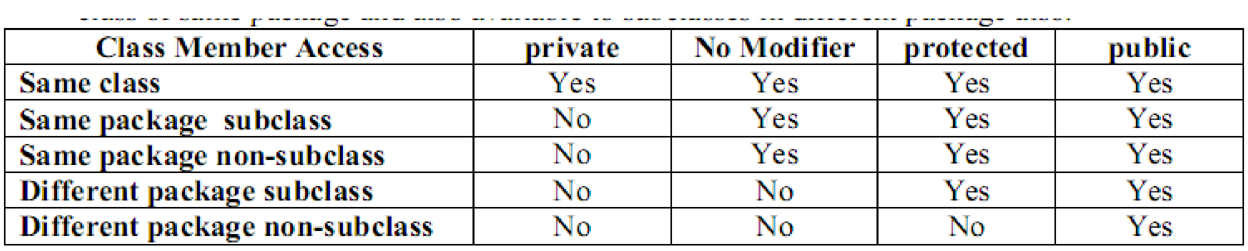
Access Specifier: Specifies the scope of the data members, class and methods.

private members of the class are available with in the class only. The scope of private members of the class is “CLASS SCOPE”.

public members of the class are available anywhere . The scope of public members of the class is "GLOBAL SCOPE".

default members of the class are available with in the class, outside the class and in its sub class of same package. It is not available outside the package. So the scope of default members of the class is "PACKAGE SCOPE".

protected members of the class are available with in the class, outside the class and in its sub class of same package and also available to subclasses in different package also.

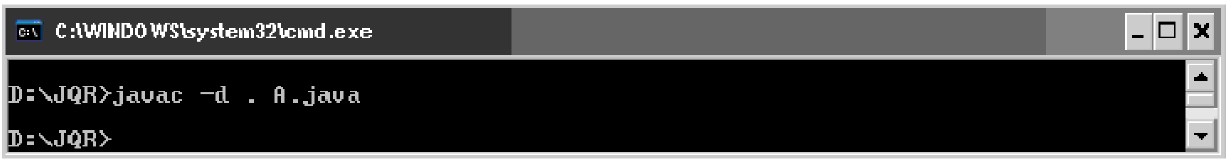


Class Member Access	private	No Modifier	protected	public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Program 6: Write a program to create class A with different access specifiers.

```
//create a package same

package same;

public class A


{ private int a=1;

  public int b = 2;

  protected int c = 3;

  int d = 4;

}
```

Compiling the above program:

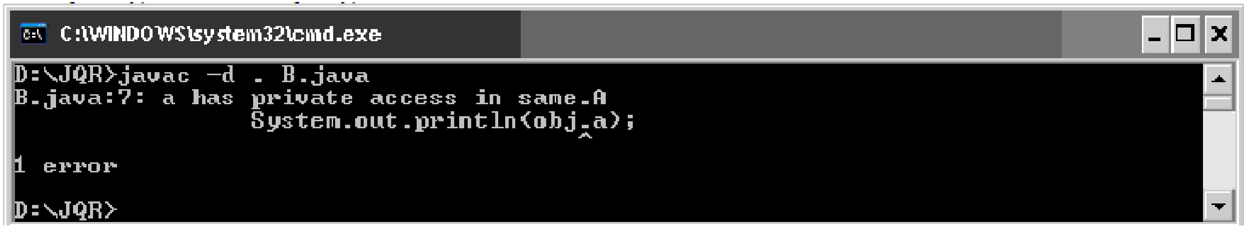


```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . A.java
D:\JQR>
```

Program 7: 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[])

{


        A obj = new A();

          System.out.println(obj.a);

          System.out.println(obj.b);

          System.out.println(obj.c);

          System.out.println(obj.d);


         } }
```

Compiling the above program:

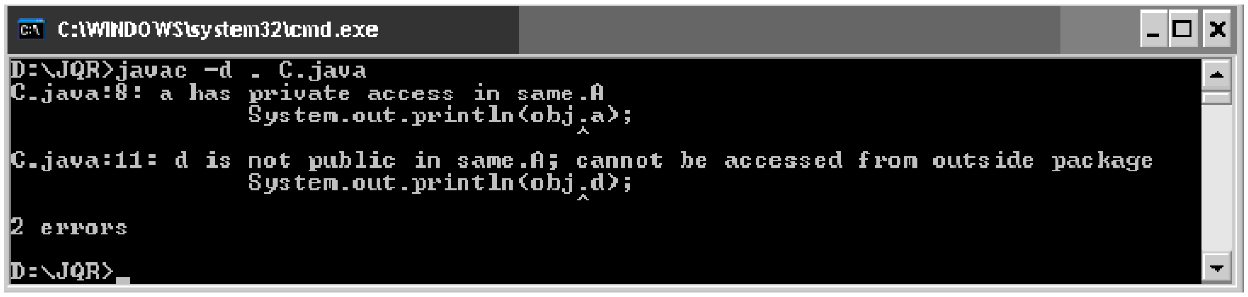


```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . B.java
B.java:7: a has private access in same.A
                System.out.println(obj.a);
                                      ^
1 error

D:\JQR>
```

Program 8: Write a program for creating class C of another package.

```
package another;

import same.A;

public class C extends A

{

public static void main(String args[])

  {

  C obj = new C();

    System.out.println(obj.a);

    System.out.println(obj.b);

    System.out.println(obj.c);

    System.out.println(obj.d);

  }

}
```

Compiling the above program:



```
D:\JQR>javac -d . C.java
C.java:8: a has private access in same.A
               System.out.println(obj.a);
                                     ^
C.java:11: d is not public in same.A; cannot be accessed from outside package
               System.out.println(obj.d);
                                     ^
2 errors

D:\JQR>_
```

• DEFINING AN INTERFACE AND IMPLEMENTING AN INTERFACE

A programmer uses an abstract class when there are some common features shared by all the objects. A programmer writes an interface when all the features have different implementations for different objects. Interfaces are written when the programmer wants to leave the implementation to third party vendors. An interface is a specification of method prototypes. All the methods in an interface are abstract methods.

An interface is a specification of method

prototypes. An interface contains zero or more

abstract methods.

All the methods of interface are public, abstract by default.

An interface may contain variables which are by default public static

final. Once an interface is written any third party vendor can

implement it.

All the methods of the interface should be implemented in its implementation classes.

If any one of the method is not implemented, then that implementation class should be declared as abstract.

We cannot create an object to an interface.

We can create a reference variable to an

interface. An interface cannot implement

another interface. An interface can extend

another interface.

A class can implement multiple interfaces.

An interface is defined much like a class. This is the general form of an interface:

```
access modifier interface name {

return-type method-name1(parameter-list);
return-type method-name2(parameter-list);

type final-varname1 = value;

type final-varname2 = value;

// ...


return-type method-nameN(parameter-list);

type final-varnameN = value;

}
```

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {


// class-body


}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract. For example:

```
abstract class Incomplete implements Callback

{ int a, b;

void show()

{ System.out.println(a + " " +

b);

}


// ...

}
```

Here, the class Incomplete does not implement callback( ) and must be declared as abstract. Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface. Anested interface can be declared as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

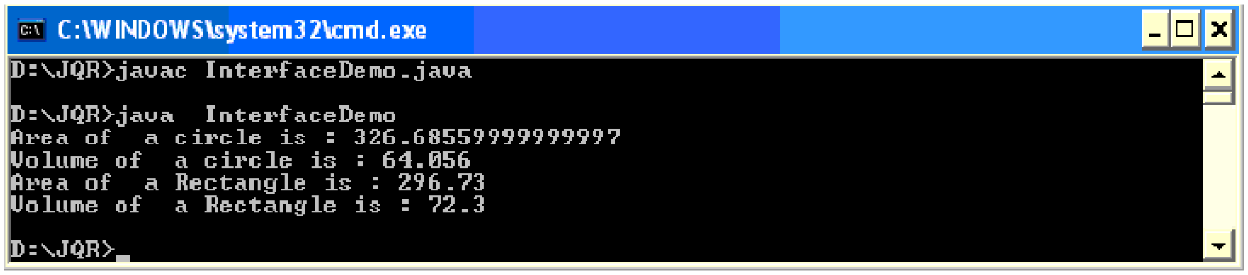
Here is an example that demonstrates a nested interface:

```
// A nested interface example.
// This class contains a member interface.

class A {

// this is a nested interface

public interface NestedIF

{ booleanisNotNegative(int

x);

}


}


// B implements the nested interface.

class B implements A.NestedIF

{ publicbooleanisNotNegative(int x)


{ return x < 0 ? false : true;


}}


class NestedIFDemo {

public static void main(String args[]) {

// use a nested interface reference

A.NestedIFnif = new B();

if(nif.isNotNegative(10))

System.out.println("10 is not negative");
if(nif.isNotNegative(-12))

System.out.println("this won't be

displayed");

}}
```

Program 1: Write an example program for interface

```
interface Shape

{ void area (); void volume (); double pi = 3.14;
}


class Circle implements Shape

{ double r;

Circle (double radius)

{ r = radius;

}

public void area ()


{ System.out.println ("Area of a circle is : " + pi*r*r );

}

public void volume ()

{ System.out.println ("Volume of a circle is : " + 2*pi*r);

} }

class Rectangle implements Shape

{ doublel,b;

Rectangle (double length, double breadth)

{  l = length; b = breadth;
}

public void area ()
{ System.out.println ("Area of a Rectangle is : " + l*b ); } public void volume ()
{ System.out.println ("Volume of a Rectangle is : " + 2*(l+b));

} }

class InterfaceDemo

{ public static void main (String args[])

{ Circle ob1 = new Circle (10.2); ob1.area ();
ob1.volume ();

Rectangle ob2 = new Rectangle (12.6, 23.55); ob2.area ();

ob2.volume ();

} }
```

Output:



C:\WINDOWS\system32\cmd.exe

```
D:\JQR>javac InterfaceDemo.java

D:\JQR>java  InterfaceDemo
Area of  a circle is : 326.68559999999997
Volume of  a circle is : 64.056
Area of  a Rectangle is : 296.73
Volume of  a Rectangle is : 72.3

D:\JQR>_
```

APPLYING INTERFACE:

To understand the power of interfaces, let’s look at a more practical example. In earlier chapters, you developed a class called Stack that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be “growable.” The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods push( ) and pop( ) define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let’s look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called IntStack.java. This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {

void push(int item); // store an
item int pop(); // retrieve an item

}
```

Applications are:

Abstractions,Multiple Inheritance

VARIABLES IN INTERFACES:

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations.) If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything.

It is as if that class were importing the constant fields into the class name space as final variables. The next example uses this technique to implement an automated "decision maker":

```
import java.util.Random;

interface

SharedConstants { int

NO = 0;

int YES = 1;
```

```
        int MAYBE = 2;

        int LATER = 3;

        int SOON = 4;

        int NEVER = 5;

        }

class Question implements SharedConstants{ Random rand = new Random();
int ask() {

int prob = (int) (100 * rand.nextDouble()); if (prob < 30)
return NO;       // 30% else if (prob < 60) return YES;      // 30%

        else if (prob <
        75)
        return LATER;     //
                          15%
        else if (prob <
        98)
        return SOON;      //
                          13%
        else

        return NEVER;     // 2%

        }

        }

class AskMe implements SharedConstants{ static void answer(int result)
{ switch(result) { case NO:
System.out.println("No"); break;
case YES: System.out.println("Yes"); break;
case MAYBE: System.out.println("Maybe"); break;
case LATER: System.out.println("Later"); break;


case SOON:

System.out.println("Soon"); break;
case NEVER:

System.out.println("Never"); break;
} }

public static void main(String args[]) { Question q = new Question(); answer(q.ask());
answer(q.ask());

answer(q.ask());

answer(q.ask());}}
```

Notice that this program makes use of one of Java's standard classes: Random. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method nextDouble( ) is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Late

r

Soo

n No

Yes
```

EXTENDING INTERFACES:

One interface can inherit another by use of the keyword extends. The syntax is the same as

for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

// One interface can extend one or more

interfaces..

```
interface A {
  void meth1();
  void meth2();}

  // B now includes meth1() and meth2() -- it adds meth3().
  interface B extends A {

  void meth3();}


  // This class must implement all of A and B
  class MyClass implements B {

  public void meth1()


  { System.out.println("Implement
  meth1().");

  }
  public void meth2()

  { System.out.println("Implement meth2().");

  } public void meth3()

  { System.out.println("Implement meth3().");

  }}
```

```
class IFExtend {

public static void main(String arg[]) {

MyClassob = new MyClass(); ob.meth1();
ob.meth2();

ob.meth3();

}}
```

As an experiment, you might want to try removing the implementation for meth1( ) in MyClass. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment.

Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

DIFFERENCES BETWEEN CLASSES AND INTERFACES:

Classes	Interfaces

Classes have instances as variables and methods with body	Interfaces have instances as abstract methods and final constants variables.
Inheritance goes with extends keyword	Inheritance goes with implements keywords.
The variables can have any acess specifier.	The Variables should be public, static, final
Multiple inheritance is not possible	It is possible
Classes are created by putting the keyword class prior to classname.	Interfaces are created by putting the keyword interface prior to interfacename(super class).
Classes contain any type of methods. Classes may or may not provide the abstractions.	Interfaces contain mustly abstract methods. Interfaces are exhibit the fully abstractions

Stream based I/O(java.io)

The java.io package contains nearly every class you might ever need to perform input and output (I/

O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are ,**FileInputStream**and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

```
import   java.io.*;   public
class CopyFile {

  public static void main(String args[]) throws IOException

  {

    FileInputStream in = null;
    FileOutputStream out =
    null;


    try {

      in = new FileInputStream("input.txt");

      out = new FileOutputStream("output.txt");
```

```
        int c;

        while ((c = in.read()) != -1)

          { out.write(c);

        }

      }finally {

        if (in != null) {

          in.close();

        }

        if (out != null) {


          out.close();

        }

      }

    }

  }
```

Now let's have a file **input.txt** with the following content:

```
This is test for copy file.
```

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java

$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, where as Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are ,**FileReader**and **FileWriter.**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*; public
class CopyFile {

  public static void main(String args[]) throws IOException

  {

    FileReader in =
    null; FileWriter out
    = null;



    try {

      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");



      int c;

      while ((c = in.read()) != -1)

        { out.write(c);

      }

    }finally {



     if (in != null) {
       in.close();
      }
      if (out != null)
        { out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware if C or C+

+ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.

- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

Following is a simple program which creates **InputStreamReader**to read standard input stream until the user types a "q":

```
import java.io.*;



public class ReadConsole {

  public static void main(String args[]) throws IOException

  {

    InputStreamReadercin = null;



    try {

      cin = new InputStreamReader(System.in);
      System.out.println("Enter characters, 'q' to quit.");
      char c;

      do {

        c = (char) cin.read();
        System.out.print(c);

      } while(c != 'q');

    }finally {

     if (cin != null)

      { cin.close();

      }

    }
  }
}
```

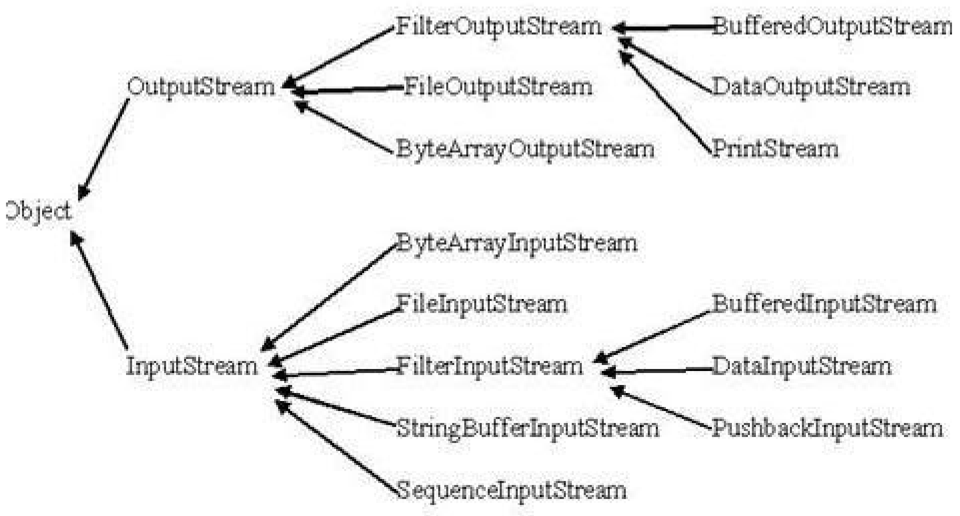
Let's keep above code in ReadConsole.java file and try to compile and execute it as below. This program continues reading and outputting same character until we press 'q':

```
$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to
quit. 1

1

e

e

q

q
```

Reading and Writing Files:

As described earlier, A stream can be defined as a sequence of data. The **InputStream**is used to read data from a source and the **OutputStream**is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream**and **FileOutputStream**, which would be discussed in this tutorial:

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read

the file.: InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new
FileInputStream(f);
```

Once you have *InputStream*object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

SN	*Methods with Description*
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	***public int available() throwsIOException{}*** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links:

- ByteArrayInputStream
- DataInputStream

FileOutputStream:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream

object to write the file: OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new
FileOutputStream(f);
```

Once you have *OutputStream*object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

SN Methods with Description

public void close() throws IOException{} This method closes the file output stream.
1 Releases any system resources associated with the file. Throws an IOException.

protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no

2 more references to this stream. Throws an IOException.

3 ***public void write(int w)throws IOException{}***

This methods writes the specified byte to the output stream.

4 ***public void write(byte[] w)***
Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links:

- ByteArrayOutputStream
- DataOutputStream

Example:

Following is the example to demonstrate InputStream and

OutputStream: import java.io.*;

public class fileStreamTest{

```
  public static void main(String
    args[]){ try{ byte bWrite [] =
    {11,21,3,40,5};
    OutputStreamos = new
    FileOutputStream("test.txt"); for(int x=0;
    x <bWrite.length ; x++){

      os.write( bWrite[x] ); // writes the bytes

    }

    os.close();



    InputStream is = new
    FileInputStream("test.txt"); int size =
    is.available();



    for(int i=0; i< size; i++)

      { System.out.print((char)is.read() + " ");

    }

    is.close();

  }catch(IOException e)

    { System.out.print("Exception");

  }

  }

}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

File Navigation and I/O:
There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java:

A directory is a File which can contains a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories:

There are two useful **File** utility methods, which can be used to create directories:

- The **mkdir( )**method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates

"/tmp/user/java/bin" directory: import

java.io.File;

```
public class CreateDir {

  public static void main(String
    args[]) { String dirname =
    "/tmp/user/java/bin"; File d =
    new File(dirname);
```

```
      // Create
      directory
      now.
      d.mkdirs();
   }
 }
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories:

You can use **list( )**method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;

public class ReadDir {
  public static void main(String[] args) {

File file
  = null;
String[]
  paths;

      try{
```

```
      // create new
      file object file =
      new
      File("/tmp");


      // array of files
      and directory
      paths =
      file.list();


      // for each name in the
      path array for(String
      path:paths)

      {

        // prints filename and directory
        name System.out.println(path);

      }

    }catch(Exception e){

      // if any
      error
      occurs
      e.print
      StackT
      race();

    }

  }

}
```

This would produce following result based on the directories and files available in your **/tmp**