Interfaces:

Using interface, you can specify what a class must do, but not how it does it. Interfaces are syntatically similar to classes, but they lack instance variables, and their methods are declared without any body. once it is defined, any no of classes can implement an interface and also one class can implement any no of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. By using interfaces, Java allows "one interface, multiple methods" aspect of polymorphism.

They are designed to support dynamic method resolution at runtime.

Syntax:

```
access specifier interface name
{
    return-type methodname1 ( parameters-list );
    return-type methodname2 ( parameters-list );
    :
    return-type methodnamen ( parameters-list )
    type variable1 = value;
    type variable2 = value;
    :
    type variablen = value;
}
```

→ An interface is similar to a class which can contain variables & methods. It should have methods without body. They are by default public.

→ The access specifier for an interface can be either default or public. If a class implements two or more interfaces, the interfaces are separated with a comma.

→ The variables in an interface are by default public, static & final. so these variables have to be initialized in the interface only.

→ An interface cannot have constructors and main().

→ An interface can be implemented in a child class.

Syntax:

```
class subclassname implements interfacename
{
    // Body of the class
}
```

child(R) subclass.

It is the responsibility of subclass (child) to define all the methods of the super interface.

partial implementation of an interface:

If the child class doesnot define all the methods then it also should be declared as abstract and it is the responsibility of its child to define them.

Example (Interfaces):

Stack: A class called stack can be implemented in an array (or) a linked list (or) a binary tree, and so on. No matter how the stack is implemented, the interface to the 'stack' remains the same with the methods push() and pop().

```
interface Stack
{
    void push(int item); // store an item
    int pop(); // retrieve an item.
}
```

→ we can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.

→ If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything.

→ one interface can inherit another by use of keyword 'extends'

// Interfaces

```
interface Calculator
{
    void add();
    void sub();
    void mul();
    void div();
    void moddiv();
}
```

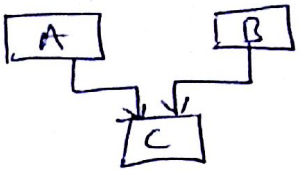
Interface

```
class NormalCalci implements Calculator
{   int a,b;
    public void add()
    {
        s.o.pln("addition:" + (a+b));
    }
    public void sub()
    {
        s.o.pln("substraction =" + (a-b));
    }
    public void mul()
    {
        s.o.p("multiplication:" + (a*b));
    }
    public void div()
    {
        s.o.p("division =" + (a/b));
    }
    public void moddiv()
    {
        s.o.p("modulo division =" + (a%b));
    }
    NormalCalci(int x, int y)
    {
        a=x; b=y;
    }
}
class InterfaceDemo
{
    p.s.v.main(String args[])
    {
        NormalCalci ob= new NormalCalci(10,5);
        ob.add(); ob.sub();
        ob.mul(); ob.div(); ob.moddiv();
    }
}
```

// Calculator a= new calculator(); // Error

Multiple inheritance using interfaces

→ Java does not support multiple inheritance directly with the classes but by using interfaces it does this.



If both the parents are classes and contains same ~~methods~~ members then the child gets two copies of each member and creates ambiguity. Multiple inheritance in java is possible only when one of the parent is an interface and the other is a class

(or) both the parents are interfaces.

Example: (1 one interface and one class.

```
interface percentage
{
    void cal();
}
class Student
{
    String name;
    int age, m1, m2;
    Student(String s, int a, int x, int y)
    {
        name = s; age = a; m1 = x; m2 = y;
    }
    void display()
    {
        s.o.pln("name = "+name + "age = "+ age);
        s.o.pln("m1 = "+ m1 + "m2 = "+ m2);
    }
}
class Results extends Student implements percentage
{
    Results(String s, int l, int m, int n)
    {
        super(s, l, m, n);
    }
    public void cal()
    {
        double P = (m1+m2)/200 x 100;
        s.o.pln("percentage = "+ P);
    }
}
```

```
class MultipleInheritanceDemo
{
    p.s.v.main(String args[])
    {
        Results ob1 = new Results("CSE1", 30, 65, 75);
        Results ob2 = new Results("CSE2", 30, 55, 65);

        ob1.display();
        ob1.cal();
        ob2.display();
        ob2.cal();
    }
}
```

Abstract class

1. It consists of both abstract and concrete methods
2. It does not allow multiple Inheritance
3. It has constructors but cannot be initialised.
4. The variables in abstract classes are not by default public, static & final.
5. Methods are not by default public
6. Defined with the keyword "abstract"
7. When defining abstract method of a superclass, it need not be public in the child class

Syntax:

```
abstract classname
{
    data members;
    abstract methods();
}
```

Interface

1. It consists of only abstract methods.
2. It allows multiple Inheritance.
3. It cannot have constructors.
4. The variables in an interface are by default public, static & final.
5. methods are by default public.
6. Defined with the keyword "interface"
7. When defining interface methods in child class the modifier public is mandatory.

Syntax:

```
interface interfacename
{
    type final variablename1 = values;
    :
    returntype methodname1(parameters);
    :
}
```

Example: // Two interfaces

```
Interface A
{
   void display1();
}
Interface B
{
   void display2();
}
class C implements A,B
{
   public void display1()
   {
      S.o.pln("Hello");
   }
   public void display2()
   {
      S.o.pln("world");
   }
}
class MulInhDemo1
{
   P.S.V.main(String args[])
   {
      C ob = new C();
      ob.display1(); ob.display2();
   }
}
```

Interface Reference variable refering to subclass Object

```
interface A
{
   void fun();
}
class B implements A
{
   public void fun()
   {
      =
   }
}
```

```
class InterfaceDemo
{
   P.S.V.m(String args[])
   {
      B ob1 = new B();
      A ob;   → reference.
      ob = ob1;
      ob.fun();
   }
}
```

packages

A package is a group of related classes and interfaces. By using packages we can achieve reusability in java. packages act as "containers" for classes.

Advantages

① The classes contained in the packages of other programs can be easily reused.

② With a single import stmt we can get all the classes into the usage of the program.

③ packages solve name space problems. [two classes in two different packages can have the same name]

→ we can import either all the classes of the package (or) only one class of the package as

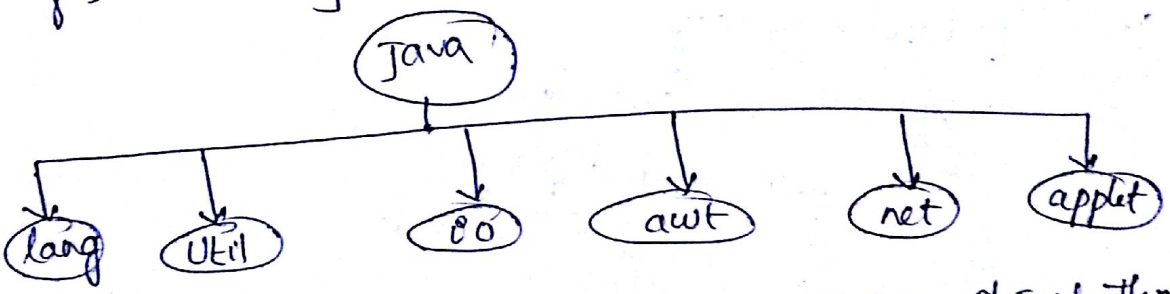
```
import java.util.*;   → all classes
import java.util.Date; → for Date class
```

→ classification of packages

① Java API packages [pre defined packages comes along with JDK]

② User defined packages.

→ ① Java API provides a large no of classes grouped into different packages according to functionality.



java.lang: It contains the classes system, string, object, thread, Exception etc. These classes are required for every java program. For this reason even if this package not imported, it is implicitly imported. [This facility exist for lang package only].

java.util : It contains Date, Random, Calender, Data structures like stack, vectors, hashtables, etc classes. These are called utility classes.

java.io : It contains FileInputStream, FileOutputstream, etc classes these classes are used for input/output operations.

java.awt : It contains Button, Frame, TextField etc classes. These classes gives java a front-end capability and also graphics environment.

java.net : It contains Socket, ServerSocket, etc classes, These classes are useful for writing LAN communication programs

java.applet : It contains Applet, AppletContext ... etc classes. These classes are useful to write Applets that made java an Internet language.

User defined packages:

To create a package of your own simply include a package command as the first stmt in a Java source file. Any classes declared within that file will belong to the specified package. The package stmt defines a name space in which classes are stored.

General form : `package pkgname;`

Ex.
```
package mypackage;
public class firstclass
{
    public void display()
    {
        S.O.println("Hello");
    }
}
```

→ The class which you want to keep in the package should declare as public.

CLASSPATH :

Save the program as Firstclass.java in your directory.

① If we want to create a package in our current directory then
> javac -d . Firstclass.java → then a package with the name mypackage is created in your current directory

If we want to create in some other directory then

c:\mgit> javac -d F:\CSE Firstclass.java then the package mypackage is created in F:\CSE directory.

(or)

① manually create folder with the same name as 'packagename' used in the program ie mypackage. Then

```
> cd mypackage
> mypackage | javac Firstclass.java
> mypackage | cd ..
> java mypackage.Firstclass   ← To run a program
```

Example: Importing java packages

```
package P1;
class Balance
{
    String name;
    double bal;
    Balance(String s, double b)
    {
        name = s;
        bal = b;
    }
    void showBal()
    {
        s.o.pln("customer:" + name + "has" + bal + "balance");
    }
}
class packageDemo
{
    p.s.v.main(String args[])
    {
        Balance customer[] = new Balance[3];
        customer[0] = new Balance("CSE1", 4000);
        customer[1] = new Balance("CSE2", 5000);
        customer[2] = new Balance("CSE3", 6000);
        for(int i=0; i<3; i++)
            customer[i].showBal();
    }
}
```

Access Control @) visibility control

Java provides four types of access modifiers as
① public ② private ③ default ④ protected.

① public : Anything declared public can be accessed from anywhere.

② private, cannot be accessed @ seen out of its own class.

③ default : accessed in same, subclasses ~~etc~~, it is a package level access modifier.

④ protected: accessed in same class, subclasses and non-subclasses on the same package and differed package subclass.

	public	private	default	protected
same class	yes	yes	yes	yes
Same package subclass	yes	No	yes	yes
Same package non-subclass	yes	No	yes	yes
Deffrent package subclass	yes	No	No	yes
Different package Non-subclass	yes	No	No	No

Example:

→ projection.java (file name)

```
package P1;
public class protection
{ int a=1; // default
  private int b=2;
  protected int c=3;
  public int d=4;
  public protection()
  { S.O.P("same package & same class");
    S.O.Pln("a=" + a);
    S.O.Pln("b=" + b);
    S.O.Pln("c=" + c);
    S.O.Pln("d=" + d);
  }
}
```

filename (Derived.java)

```
package P1;
class Derived extends protection
{
    Derived()
    {
        S.o.p(" same package derived class");
        S.o.p("a=" +a);
        // S.o.p("b=" +b);   error (private access)
        S.o.p("c=" +c);
        S.o.p("d=" +d);
    }
}
```

filename: samepackage.java

```
package P1;
class Samepackage
{
    Samepackage()
    {
        protection p = new protection();
        S.o.p(" same package non subclass");
        S.o.p("a=" +p.a);
        // S.o.p("b=" +p.b);   error
        S.o.p("c=" + p.c);
        S.o.p("d=" + p.d);
    }
}
```

→ Samepackagedemo.java

```
package P1;
class Samepackagedemo
{
    public static void main(String args[])
    {
        protection ob1 = new protection();
        Derived ob2 = new Derived();
        Samepackage ob3 = new Samepackage();
    }
}
```

protection2.java

```
package P2;
class protection2 extends P1.protection
{
    protection2()
    {
        S.o.pln("Different package subclass");
        // S.o.pln("a=" + a);   error
        // S.o.pln("b=" + b);   error
        s.o.pln("c=" + c);
        s.o.pln("d=" + d);
    }
}
```

otherpackage.java

```
package P2;
class otherpackage
{
    otherpackage()
    {
        P1.protection p = new P1.protection();
        s.o.pln("other package non-subclass");
        s.o.pln("a=" + p.a); // error
        s.o.pln("b=" + p.b); // error
        s.o.pln("c=" + p.c); // error
        s.o.pln("d=" + p.d); valid (public)
    }
}
```

otherpackageDemo.java

```
package P2;
class otherpackageDemo
{
    P.s.v.main(String args[])
    {
        protection2 ob1 = new protection2();
        otherpackage ob2 = new otherpackage();
    }
}
```

Stream based I/O (java.io)

(Input | output)

The 'io' package supports Java's basic I/O (input | output) system, including file I/O. The applet package supports applets.

Java does provide strong, flexible support for I/O as it relates to files and networks.

Streams:

Java programs perform I/O through Streams. A stream is an abstraction that either produces (or) consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. i.e an input stream can abstract many different kinds of input: from a disk file, a keyboard, (or) a network socket. And an output stream may refer to the console, a disk file, (or) a n/w connection.

Byte streams & Character streams:

→ Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used (also) reading (or) writing binary data.

→ character streams provide a convenient means for handling input and output of characters. They use Unicode

The Byte stream classes

{① InputStream ② OutputStream} abstract classes.

The abstract classes InputStream and OutputStream define several methods like read() and write() by which we can read and write bytes of data resp. Both methods are declared as abstract inside InputStream and OutputStream.

The character stream classes

{① Reader ② Writer } abstract classes

The abstract classes Reader and Writer define several methods like read() and write() by which we can read and write characters of data res/.

The Byte Stream classes

Stream class	meaning
BufferedInputStream	Buffered Input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	output stream that writes to a byte array
DataInputStream	It contains methods for reading the Java standard data types.
DataOutputStream	It contains methods for writing " " " "
FileInputStream	It reads from a file
FileOutputStream	It writes to a file.
FilterInputStream	describes stream input
FilterOutputStream	describes stream output
⋮	⋮
RandomAccessFile	Supports random access file I/O
SequenceInputStream	It is combination of two or more input streams that will be read sequentially, one after the other.

The Character Stream I/O classes

Stream class	meaning
BufferedReader	Buffered input character stream.
BufferedWriter	Buffered output character stream.
CharArrayReader	reads from a character array.
CharArrayWriter	writes to a character array.
FileReader	reads from a file.
FileWriter	writes to a file.
InputStreamReader	translates byte to character.
LineNumberReader	counts lines.
OutputStreamWriter	translates characters to bytes.
⋮	⋮
Reader	describes character stream input
StringReader	reads from a string
StringWriter	writes to a string
Writer	describes character stream output.

The predefined streams

All java programs automatically import java.lang package. It defines a class called System, which encapsulates several aspects of the run-time environment. System class also contains three predefined stream variables in, out, and error. These fields are declared as public and static within System class.

System.out refers to the standard output stream, by default it is console. System.in refers to standard input, which is the keyboard by default. System.err refers to standard error stream, which is also is the console by default.

→ System.in is an object of type InputStream.
→ System.out . . . . OutputStream.
→ System.err . . . . PrintStream

These three are byte streams, even though they are used to read and write characters from and to the console.

Reading Console Input:

Console input is accomplished by reading from System.in. To obtain a character-based stream that is attached to the console, wrap System.in in a BufferedReader object, to create a character stream. BufferedReader supports a buffered input stream.

Syntax:

```
BufferedReader (Reader inputReader);
```

To obtain an InputStreamReader object that is linked to System.in, use the following constructor

```
InputStreamReader ( InputStream inputStream);
```

The following lines of code create BufferedReader that is connected to the keyboard

①
```
BufferedReader br = new BufferedReader (new InputStreamReader
                                        (System.in));
```

②
```
{ InputStreamReader in = new InputStreamReader (System.in);
  BufferedReader br = new BufferedReader (in) }
```

Reading characters

To read a character from BufferedReader, use read() method

```
int read() throws IOException
```

Each time the read() is called, it reads a character from the input stream and returns it as an integer value. It returns -1 when the end of the stream is encountered

Example:

```
import java.io.*;
class BRRead
{
  p.s.v.main(String args[]) throws IOException
  {
    char c;
    BufferReader br = new BufferedReader(new InputStreamReader
                                         (System.in));
```

(or)

```
    InputStream Reader in = new InputstreamReader(System.in);
    BufferedReader br = new BufferedReader(in);
    System.out.println("Enter characters, 'q' to exit.");
    do
    {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

o/p:

```
Enter characters, 'q' to exit.
123abcq
1
2
3
a
b
c
q
```

Reading strings

To read a string from the keyboard, use readLine() which is a member of the BufferedReader class.

String readLine() throws IOException

Example:
```
import java.io.*;
class BRReadLine
{
    p.s.v.main(String args[]) throws IOException
    {
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(in);

        String str;
        str = br.readLine();
        System.out.println(" str);
    }
}
```

Example:
```
import java.io.*;
class Test
{
    p.s.v.main(String args[]) throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String str[] = new String[100];
        S.O.println("Enter lines of text, stop to quit");
        for(int i=0; i<100; i++)
        {
            str[i] = br.readLine();
            if(str[i].equals("stop")) break;
        }
        S.O.println("Entered lines are:");
        for(int i=0; i<100; i++)
        {
            if(str[i].equals("stop")) break;
            S.O.println(str[i]);
        }
    }
}
```

W.A.P to swap two integer numbers.

```
import java.io.*;
class Swap
{
    p.s.v.main( String args[]) throws IOException
    {
        int a, b;
        InputStreamReader isr = new InputStreamReader (System.in);
        BufferedReader br= new BuffenedReader ( isr);
        S.o.pln(" Enter value of a:");
          a = Integer.parseInt( br.readLine());
        S.o.pln (" Enter value of b: ");
          b= Inter.parseInt ( br.readLine());
        S.o.pln (" Before swaping ");
        S.o.pln( "a= " + a + "b= " + b);
          a=a+b; b=a-b; a=a-b;
        S.o.pln(" After swaping ");
        S.o.pln ("a= "+ a + "b= "+ b);
    }
}
```

Programs

① write a java program to find area of a circle (Math.PI)

② WAJP to find factorial of a number

③ WAJP to reverse of a given number

④ WAJP to check whether the given number is palindrome or not

⑤ WAJP to print the series

```
1
1 2
1 2 3
1 2 3 4
```

```
1 2 3 4
1 2 3
1 2
1
```

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

```
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

Writing Console output

It is mostly accomplished with print() and println() methods. These methods are defined by the class PrintStream (which is also implemented by System.out). Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write() to write to the console.

Syntax: `void write(int byteval)`

It writes to the stream the byte specified by byteval.

Reading and writing files

Java provides a number of classes and methods that allow you to read and write files. Here all files are byte-oriented.

The stream classes FileInputStream and FileOutputStream create byte streams linked to files. To open a file, create an object of one of these classes, specifying the name of the file as an argument to the constructor.

Syntax:

```
FileInputStream(String filename) throws FileNotFoundException
FileOutputStream(String filename) throws FileNotFoundException
```

close()

`void close() throws IOException` → for closing a file.

read() → to read from a file

`int read() throws IOException`

It reads a single byte from the file and returns the byte as an integer value. It returns −1 when the end of the file is encountered.

write() : To write to a file.

`void write(int byteval) throws IOException`

Example // Display a text file

```
import java.io.*;
class showFile
{
  p.s.v.main(String args[]) throws IOException
  {
    int i;
    FileInputStream fin;
      try
      {
        fin = new FileInputStream(args[0]);
      }
      catch(FileNotFoundException e)
      {
          S.o.println("File Not found");
          return;
      }
      catch(ArrayIndexOutOfBoundsException e)
      {
          S.o.println("Usage: ShowFile");
          return;
      }
    do
    {
        i = fin.read();
        if(i!=-1)
        S.o.println((char) i);
    } while(i!=-1);
    fin.close();
  }
}
```