Introduction:

- One of the main features of the OOP is its ability to reuse the code. We can do so by using inheritance or extending classes and implementing the interfaces.

- The name of each class has taken from same name space. This means that we have to define unique name for each class in Java. So you have to think some way that class name you choose would be unique and not collide with class name chosen by other programmers.

    1. Java provides mechanism for partitioning the class name space into more manageable chunks. This mechanism is known as **package**. So package is a collection of classes.

    2. By organizing class into package we have the following benefits:

        1. You can define classes' insides a package that are not accessible outside that package.

        2. You can also define class members that are only exposed to other members of the same package.

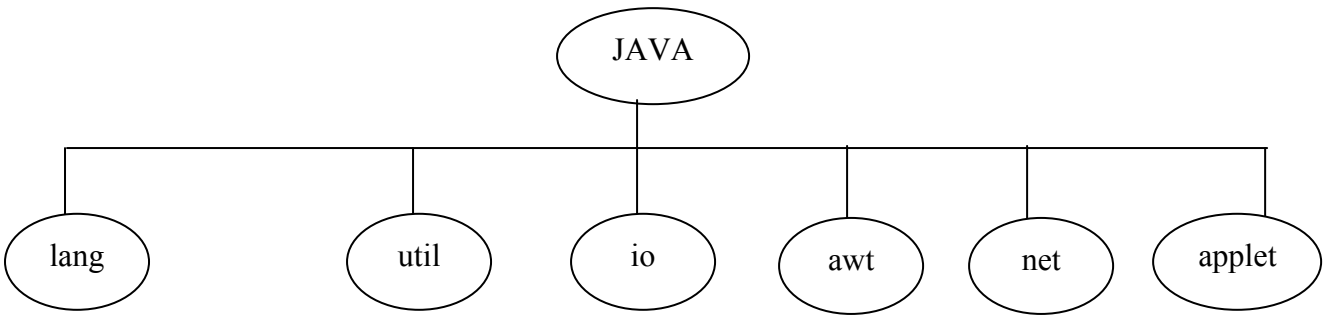
A java source file can contain any (or all) of the following four internal parts:

1. A single package statement (optional)
2. Any number of import statements (optional)
3. A single public class declaration (required)
4. Any number of classes private to the package (optional)

Java packages are classified into 2 categories.

1. Java API Packages
2. User defined Packages.

❖ **Java API Packages:**



- **How to define package or create a package?**

To create our own package involves the following steps:

1. Declare the package at the beginning of the file using the following form;

   package packagename;

   For example;

   package mypackage;

2. Define the class that is to be put in the package and declare it public.

3. Create a subdirectory under root directory where the main source files are stored.

   For example: z:\121212\java\mypackage

4. Store the java source file as the class name.java file in the created subdirectory.

   For example: z:\121212\java\mypackage\myclass.java

5. Compile the file. This creates .class file inside the subdirectory.

6. The subdirectory name must be matching the package name exactly.

7. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

   Package pkg1[.pkeg2][.pkg3]

- Java creates **.class** file for every source file which is declared under the subdirectory or package.

Note: you can't rename a package without renaming the directory in which the classes are stored.

==========

==========

Consider the following example to compile and execute or run packages.

Step 1: Assume that there is following path where we create .java source file.
z:\121212\java

Step 2: now type the following command on the prompt and type step 3 in file.
z:\121212\java\edit myanimal.java
or
Create java source file using **notepad** and save this file inside above path with filename and extension such as **myanimal.java** and type step 3 in file.

Step 3:
```
package animal;
public class myanimal
{
    public String aname[]={"Elephant","Tiger","Dog"};
}
```

❖ **Save this file as myanimal.java**

Step 4: Now again type the following command on the prompt with given code.
z:\121212\java\edit animaldemo.java
or
Create java source file using notepad with given code and save this file in above path with filename and extension such as **animaldemo.java**

```
package animal;
class animaldemo
{
    public static void main(String args[])
    {
        int i, len;
        myanimal m1=new myanimal();     //object of myanimal class

        len=m1.aname.length;        //access aname[] of myanimal class
        for(i=0;i<len;i++)
            System.out.println("Animal[" +(i+1) +"]: " +m1.aname[i]);
    }
}
```

Save this file as animaldemo.java

Step 5: now create the folder **animal** on the prompt and cut both **.java** file into **<animal>** folder **For example; z:\121212\java\animal**

Step 6: now compile the both **.java file** in the following way.

z:\121212\java\javac animal/myanimal.java
z:\121212\java\javac animal/animaldemo.java

Step 7: After successfully compiled animaldemo.java file, run this file.
z:\121212\java\java animal/animaldemo

Note: You can run only those file under which you have defined **main()** method.

Q. **Explain Access protection of packages**

The class is java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

1. Subclasses in the same package.
2. Non-subclasses in the same package.
3. Subclasses in different packages.
4. Classes that are neither in the same package nor subclasses.

The four access specifiers, private, public, protected and default access specifier.

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

2. **Private:** Anything declared **private** can be accessed within class only.

3. **Protected:** Anything can be declared as **protected** can be accessed in same package and different package in subclasses only which is directly extended by super class.

4. **Default access:** Anything can be declared as default can be accessed within same package in subclasses and non-subclass.

A **class** has only two possible access level: **default and public.** When a class is declared as **public,** it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

The following table shows you member accessibility of classes.

Sr.No	Access location	Private	No modifier or Default	Protected	Public
1	Same class	Yes	Yes	Yes	Yes
2	Subclass in Same Package	No	Yes	Yes	Yes
3	Non-subclass in same package	No	Yes	Yes	Yes
4	Subclass in other packages	No	No	Yes	Yes
5	Non-subclasses in other packages	No	No	No	Yes

The following example shows all combinations of the access control modifiers

```
package p1;

public class protection
 {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;
```

```
        public protection()
        {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

This file save as **protection.java (file 1)**

```
class derived extends protection
{
        derived()
        {
                System.out.println("derived constructor");
                System.out.println("n = " + n);

                // class only
                // System.out.println("n_pri = " + n_pri);      //error
                   System.out.println("n_pro = " + n_pro);
                   System.out.println("n_pub = " + n_pub);
        }
}
```

This file save as derived.java

```
class samepackage
{
                samepackage()
                {
                        protection p = new protection();
                        System.out.println("same package constructor");
                        System.out.println("n = " + p.n);

                    // class only
                    // System.out.println("n_pri = " + p.n_pri);        //error

                        System.out.println("n_pro = " + p.n_pro);
                        System.out.println("n_pub = " + p.n_pub);
                }
}
```

This file is saved as samepackage.java

```
package p2;
class protection2 extends p1.protection
 {
                    protection2()
                    {
                        System.out.println("derived other package constructor");

                      // class or package only
                      // System.out.println("n = " + n);  //error

                      // class only
                      // System.out.println("n_pri = " + n_pri);     //error

                        System.out.println("n_pro = " + n_pro);
                        System.out.println("n_pub = " + n_pub);
                     }
}
```

This file save as protection2.java

```
class otherpackage
{
                    otherpackage()
                    {
                        p1.protection p = new p1.protection();
                        System.out.println("other package constructor");
                      // class or package only
                      // System.out.println("n = " + p.n);

                      // class only
                      // System.out.println("n_pri = " + p.n_pri);

                      // class, subclass or package only
                      // System.out.println("n_pro = " + p.n_pro);

                       System.out.println("n_pub = " + p.n_pub);
                     }
}
```

This file save as otherpackage.java

```
package p1;
public class demo
{
                    public static void main(String args[])
                    {
                        protection ob1 = new protection();
                        derived ob2 = new derived();
                        samepackage ob3 = new samepackage();
                     }
}
```

This file is save as demo.java in package p1

```
// Demo package p2.
package p2;

public class demo
{
                public static void main(String args[])
                {
                        protection2 ob1 = new protection2();
                        otherpackage ob2 = new otherpackage();
                 }
}
```

This file is demo.java in package p2

Q. Explain Importing packages:

There is no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.

For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java Program.
If you are going to refer to a few dozen classes in your application, however the **import** statement will save a lot of typing.

The general form of the **import** statement:

import pkg1[.pkg2].(classname | *);

Here, **pkg1** is the name of a top-level package, and **pkg2** is the name of the package inside **pkge1** separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*****), which indicates that the java compiler should import the entire package.

- **import java.util.date;**
- **import java.io.*;**

All of the standard java classes included with java is stored in a package called **java.** The basic language functions are stored in a package inside of the java package called **java.lang.** Normally, you have to import every package inside of the **java** package called **java.lang.**

Normally, you have to import every package or class that you want to use, but since java is useless without much of the functionality in **java.lang,** it is implicitly imported by the compiler for all programs.

import java.lang.*;

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will a compiler-time error and have to explicitly name the class specifying its package.

Note that the statement must end with a **semicolon (;).** The **import** statement should appear before any class definition in a source file. Multiple import statements are allowed. The following is an example of importing a particular class.

import firstpkg.secondpkg.myclass

After defining this statement, all the members of the class myclass can be directly accessed using the class name or its objects (as the case may be) directly without using the package name.

We can also use another approach as follows:

import packagename .*;

Here **packagename** may denote a single package or a hierarchy of packages as mentioned earlier.

The star **(*)** indicates that the compiler should search this entire package hierarchy when it encounters a class name. This implies that we can access all classes contained in the above package directly.

For example,

```
package mypack;
/*
        now the balance class, its constructor, and its show() method are public. This means
        that they can be used non-subclass code outside their package.
*/
 public class balance
{
                String name;
                double bal;
                public balance(String n, double b)
                {
                    name=n;
                    bal=b;
                }
                public void show()
                {
                    if(bal<0)
                            System.out.println("-->");
                    else
                            System.out.println(name +": Rs." +bal);
                }
}
```

This file is save as balance.java in package mypack

Now create another .java file

```
import mypack.*;

class testbalance
{
                public static void main(String args[])
                {
//because of balance is public you may use balance class and its constructor
                    balance test=new balance("Rudra Singh",150.50);
                    test.show();
                }
}
```

This file is save as testbalance.java

Q: What do you mean by * when we import a package?

When we import a package using (*), all **public classes** are imported. However, we may prefer to **"not import"** certain classes. That is, we may like to hide these classes from accessing outside of the package. Such classes should be declared **"not public" For example,**

```
package p1;
public class x
{
                // body of x                    //public class, available outside
}
class y
{
                //body of y          // not public, hidden from outside the package
}
```

Here, the class y which is not declared public is hidden from outside of the package p1. This class can be seen and used only by other classes in the same package.

Now consider the following code, which imports the package p1 that contains class x and class y;

```
import p1.*;

x object = new x();          //ok class x is available here
y object = new y();          // not ok, y is not available here.
```

Note: Java source file can have only one class declared as **public**, we cannot put two or more **public classes** together in a **.java file**. This is because of the file name should be same as the name of the public class with .java extension.

Q: Explain interfaces:

In C++ the implementation of multiple inheritance proves difficult and adds complexity to the language. Java provides an alternate approach known as **interface** to support the concept of multiple inheritance. Although a java class cannot inherit more than one superclass. It can implement more than one interface. For example, a definition likes the following not valid in java.

```
class A extends B extends C
{
        ----------
        ----------
}
```

Interfaces are syntactically similar to classes, but they lack instance variables and their methods are declared without any body. It means you can define interfaces which don't make any assumptions about how they are implemented. Any number of classes can implement an **interface.** One class can implement any number of **interfaces.**

By providing the interface keyword, java allows you to fully utilize the **"one interface, multiple methods"** aspects of polymorphism.

Using the keyword **interface**, you can fully abstract a class interface from its implementation. That is by using **interface**, you can specify what a class must do, but not how it does it.

- **Defining an interface**

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

```
access interface name
{
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1=value;
        type final-varname1=value;
        ----------------------------
        ----------------------------
        return-type method-nameN(parameter-list);
        type final-varnameN=value;
}
```

Here access is either **public** or not used. When no access specifier is included, then the default access results and the interface is only available to other members of the package in which it is declared.

When it is declared as **public,** the interface can be used by any other code. Name is the name of the interface, and can be any valid identifier.

Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final and static,** meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All the methods and variables are implicitly **public** if the interface, itself, is declared as **public.**

For example,

Example 1:
```
interface callback
{
        void callback (int param);
}
```
Example 2:
```
interface item
{
        final int code=1001;
        final String name="DVD"
        void display();
}
```
Example 3:
```
interface area
{
        final float pi=3.14f;
        float compute (float x, float y)
        void show();
}
```

- **implementing interfaces**

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

```
access class classname [extends superclass]
                [implements interface [,interface…]]
{               //class-body

}
```

Here, access is either **public** or not used. 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.

Here is a small example class that implements the **callback** interface shown earlier.

```
class client implements callback        //implement callback's interface
{
                public void callback(int p)
                {
                    System.out.println("Callback called with" +p);
                }
                void nonifacemeth()
                {
System.out.println("Classes that implement interfaces" +may also define other members, too");
                }
}
```

Notice that **callback()** is declared using the **public** access specifier. It means when you implement an interface method, it must be declared as **public.**

Note: when you implement an interface method, it must be declared as **public.**

- **Accessing implementation through interface reference**

or

- **How to access interface method through interface reference?**

We create an instance of each class using the **new** operator. After creating an instance of each class, we declare an object of interface class without using **new** operator. Now we assign the reference to the class object to the interface object. When we call the method of interface, the method of class that is being referred by interface object is invoked. So above callback interface will be implemented as following.

```
class interfacedemo
{
                public static void main(String arg[])
                {
                    client c1=new client();          //object of client class
                    callback i1;                 //object of callback interface
                    i1=c1;           // i1 refers to the client object
                    i1.callback(10);                 //i1 refers to the client object

                    callback i2=new client();       //object of callback interface
                                                     //refers to the client object
                    i2.callback(20);                         //i2 refers to the client object
                }
}
```

- **Variables in interfaces**

When you include the interface in a class, 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. So you must initialize variable as constant value when you define interface. When you implement it in a class you can't modify that value in a class. Because all the variable declared inside interface will be **final**.

For example

```
interface variable
{
        int x=10;
        int y=20;
        int z=30;
        int m=40;
        int n=50;
}
```

- **Extending interfaces**

Like classes, interfaces can also be extended. That is, an interface can be sub-interfaced from other interfaces. The new sub-interface will inherit all the members of the super-interface in the manner similar to subclasses. This is done by using keyword **extends** as shown bellow:

```
interface name2 extends name1
{
        body of name2
}
```

For example, we can put all the constants in one interface and the methods in other. This will enable us to use the constants in classes where the methods are not required. **For example**

```
interface     itemconstants
{
    int code=1001;
    String name="DVD"
}
interface item extends itemconstants
{
    void display();
}
```

The interface item would inherit both the constants **code** and **name** into it. Note that the variables **name** and **code** are declared like simple variables. It is allowed because all the variables in an interface are treated as constants through the keywords **final** not present.

We can also combine several interfaces together into a single interface. Following declarations are valid:

```
interface item extends itemconstants, itemmethods
{
        ---------
        ---------
}
```

Remember the following points:

1. While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in superinterfaces. After all, subinterfaces are still interfaces, not classes.

2. Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

For example,

```
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 can not be removed anyone.

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().");
        }
}
```

3. When an interface extends two or more interfaces, they are separated by commas.
4. Interfaces cannot extend classes. Because of this would violate the rule that an interface can have only abstract methods and constants.

//A complete example of class and interface

```
interface callback
{
        void callback (int param);
}
class client implements callback        //implement callback's interface
{
        public void callback(int p)
        {
            System.out.println("Callback called with:" +p);
        }
        void nonifacemeth()
        {
System.out.println("Classes that implement interfaces " +"may also define other members, too");
        }
}
class interfacedemo
{
        public static void main(String arg[])
        {
            client c1=new client();          //object of client class
            callback i1;              //reference of callback interface
            i1=c1;          // i1 refers to the client object
            i1.callback(10);                 //i1 call callback() method

            callback i2=new client();  //object of client to reference variable
                                                    //refers to the client object
            i2.callback(20);                        //i2 call callback() method
            c1.nonifacemeth();    //call non interface method by class object
        }
}
```

Save above file as: interfacedemo.java

Output:

Callback called with: 10

Callback called with: 20

Classes that implement interfaces may also define other members, too