UNIT-3
Interfaces and Packages

Syllabus:
Interfaces: Interfaces - Interfaces vs. Abstract classes, defining an interface, implementing interfaces, extending interfaces.
Packages - Defining, Creating and Accessing a Package, Understanding CLASSPATH, Access protection. Importing packages.

abstract class:

- A class that is declared with abstract keyword, is known as abstract class in java.
- It can have abstract methods, non-abstract methods (method with body), variables and constructor.
- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- It shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- **Ways to achieve Abstraction**
  There are two ways to achieve abstraction in java
  1. Abstract class
  2. Interface (fully abstraction)
- A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

```
abstract class A
{
      // may contain abstract and normal methods
}
```

- A method that is declared as abstract and does not have implementation is known as abstract method.

```
abstract void print();//no implementation
```

- Example :

```
abstract class Base
{
  abstract void print1();
  void print2()
   {
       System.out.println("Normal method");
   }
}
```

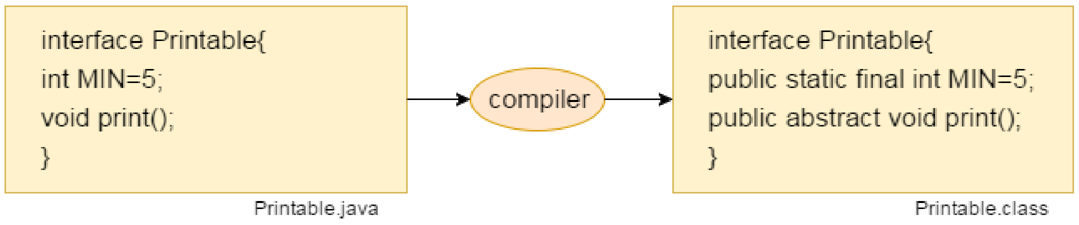
```
class Derived extends Base
{
     void print1()
     {
         System.out.println("abstract method");
     }
     public static void main(String args[])
     {
         Derived obj = new Derived();
         obj.print1();
         obj.print2();
     }
}
```

Output: `Normal method`
`abstract method`

****

Interface:

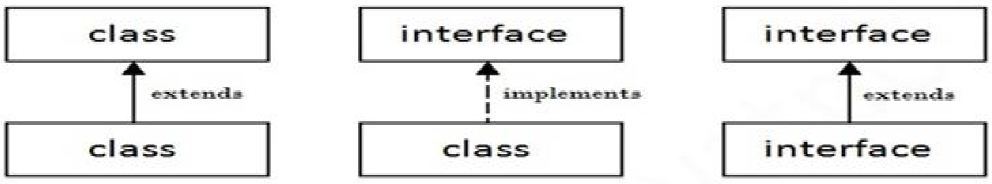
- An **interface in java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, Interface fields are public, static and final by default, and methods are public and abstract.



```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

- **Relationship between classes and interfaces**



- a class extends another class, an interface extends another interface but a **class implements an interface**.

Implementing interface:

- interface implementation provided inside a class using ‘implements’ keyword

Example (Implementing interfaces):

```
interface ISample
{
   int i=10;
   void display();
}
class Sample implements ISample
{
   public void display()
   {
      System.out.println("display method i="+ i);
   }
}
class InterfaceDemo
{
   public static void main(String args[])
   {
      Sample s=new Sample();
       s.display();
   }
}
```

Output: `display method i=10`

Extending interfaces:

- interfaces can extended by another interface with ‘extends’ keyword, but the extending interface implementation provided by a class by using ‘implements’ keyword.
- **Example (extending interfaces):**

```
interface ISample1
{
   int i=10;
   void display1();
}

interface ISample2 extends ISample1
{
   int j=20;
   void display2();
}
```

```
class Sample1 implements ISample2
{
   public void display1()
   {
      System.out.println("display1 method i="+ i);
   }
   public void display2()
   {
      System.out.println("display2 method j="+ j);
   }
}
class InterfaceDemo1
{
   public static void main(String args[])
   {
       Sample s=new Sample();
       s.display1();
       s.display2();
   }
}
```

Output:

```
display1 method i=10
display2 method j=20
```

Interface vs abstract class:

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can have static methods, main method and constructor.	Interface can't have static methods, main method or constructor.
5) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
6) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

7) Example:
abstract class Sample
{
abstract void show();
}
	Example:
	interface ISample
	{
	void show();
	}

****

Q: Is it possible an abstract class can implement interface?

Answer:

- **Yes, it is possible**, An abstract class can implement any interface.
- But we can't create an object for an abstract class. So it is compulsory to extend abstract class by another class.
- Now by creating object for extending class, we can invoke members of abstract class.
- **Example:**

```
interface ISample
{
   int i=10;
   void display();
}
abstract class Sample implements ISample
{
   public void display()
   {
      System.out.println("display method i= "+ i);
   }
}

class SampleDemo extends Sample
{
   public static void main(String args[])
   {
      SampleDemo s=new SampleDemo();
       s.display();
   }
}
```

Output:

```
    display method i= 10
```

****

Packages:

- A **package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- User-defined packages that are created by programmer by using ‘package’ keyword.

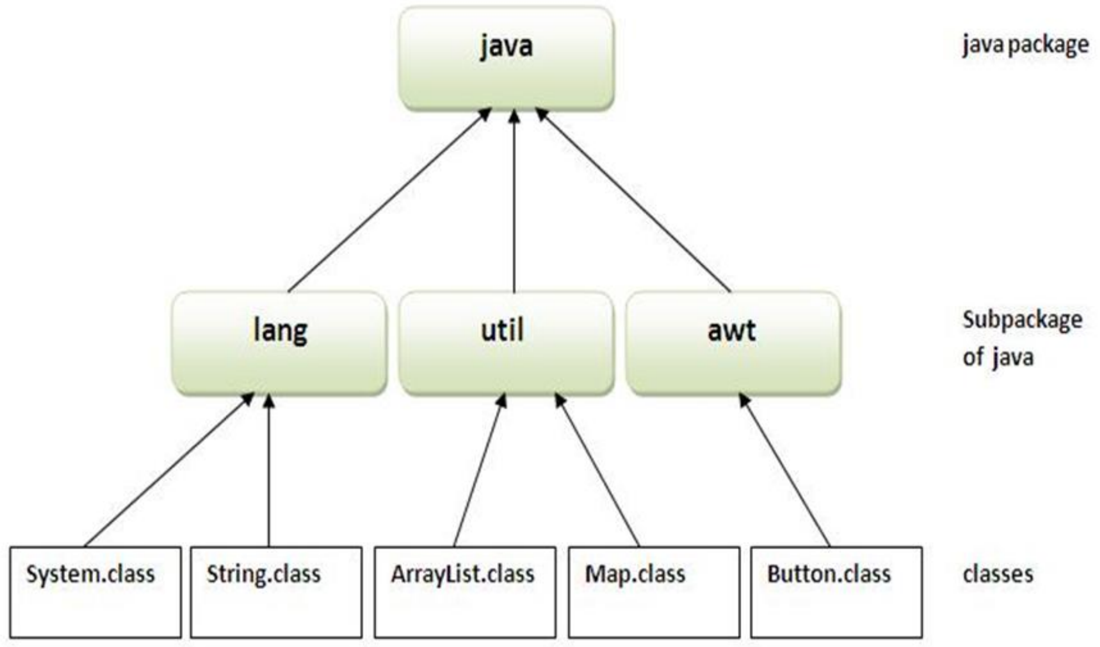
```
package packname;
```

- **Advantages of Package**

  1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

  2) Java package provides access protection.

  3) Java package removes naming collision.



Creating a package:

- The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple
{
   public static void main(String args[])
   {
      System.out.println("Welcome to package");
   }
}
```

Compilation process of package program:

- Following is the procedure for compiling the program

```
javac -d directory javafilename
```

- For **example**

```
javac -d . Simple.java
```

- The '-d ' specifies the destination where to put the generated class file.
- We can use any directory name d:/abc (in case of windows) etc.
- If you want to keep the package within the same directory, you can use . (dot).

To run java package program

- we need to use fully qualified name like

```
packagename.javafile
```

Example: `mypack.Simple` to run the class.

To compile and execute above simple package program:

Compile: `javac -d . Simple.java`

Run: `java mypack.Simple`

Output:`Welcome to package`

- The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The **. (dot)** represents the current folder.

****

Importing or accessing packages:

- Generally if we want to access any class belongs to specific package, then it is compulsory to import package that contain class.
- Example: suppose if we want to access **Scanner** class**,** we have to import java.util package.

```
import java.util.Scanner;
```

- Similar to built-in package we have to import user defined packages.
- There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

Using packagename.*:

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.
- **Example**

```
//save by A.java
package pack;
public class A
{
    public void msg()
     {
           System.out.println("Hello");
     }
}

//save by B.java
package mypack;
import pack.*;
class B
{
    public static void main(String args[])
      {
            A obj = new A();
            obj.msg();
      }
}
Output: Hello
```

Using packagename.classname:

- If you import package.classname then only declared class of this package will be accessible.
- **Example**

```
//save by A.java
package pack;
public class A
{
  public void msg()
    {
        System.out.println("Hello");
    }
}
```

```
//save by B.java
package mypack;
import pack.A;
class B
{
   public static void main(String args[])
   {
      A obj = new A();
      obj.msg();
   }
}
```
Output: Hello

Using fully qualified name:

- If we use fully qualified name then only declared class of this package will be accessible. Now there is no need to import.
- But you need to use fully qualified name every time when you are accessing the class or interface.
- **Example**

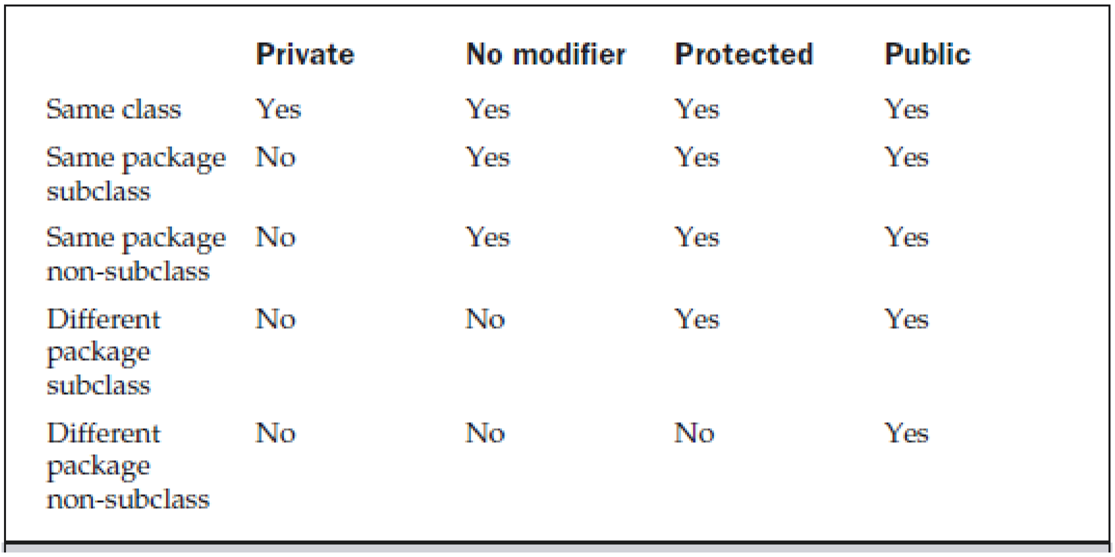
```
//save by A.java
package pack;
public class A
{
      public void msg()
      {
          System.out.println("Hello");
      }
}

//save by B.java
package mypack;
class B
{
     public static void main(String args[])
     {
           // using fully classified name
           pack.A obj = new pack.A();
           obj.msg();
     }
}
```

Output: Hello

Access protection:

- There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
    1. private
    2. default
    3. protected
    4. public
- There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.
- For class we have two access modifiers i.e. public and default.
- The following diagram shows 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

Example:

```
package mypack1;
public class Sample
{
        int a=10; // accessible within same package
       public int b=20; // accessible any where
       private int c=30; // we can’t access outside of class
       protected int d=40; //only accessible by inherited class
       void print1() // accessible within same package
       {
          System.out.println("this is default method");
       }
       public void print2()// accessible any where
       {
          System.out.println("this is public method");
       }
       private void print3() // we can’t access outside of class
       {
          System.out.println("this is private method");
       }
       protected void print4() // only accessible by inherited class
       {
          System.out.println("this is protected method");
       }
}
```

Note: Save as “Sample.java”

```
package mypack2;
import mypack1.*;
class AccessModiefier extends Sample
{
   public static void main(String args[])
   {
        AccessModiefier ac=new AccessModiefier();

        /*we can't access default and
        private members outside of package*/

        System.out.println("default a="+ac.a); //Compile time error
        System.out.println("private c="+ac.c); //Compile time error
        ac.print1(); //Compile time error
        ac.print3(); //Compile time error

        /*can access public members
        and protected members(only by using inheritance)
        outside of package*/

        System.out.println("public a="+ac.b); //Executes
        System.out.println("protected d="+ac.d); //Executes
        ac.print2(); //Executes
        ac.print4(); //Executes
   }
}
```

Note: save as “AccessModiefier.java”

Compilation:
```
javac -d . Sample.java
javac -d . AccessModiefiers.java
```

Execution:
```
java mypack2.java
```

Output:
```
Compile time errors
```

****

Understanding CLASSPATH:

- Consider the following program.

```
//save as Simple.java
package mypack;
public class Simple
{
     public static void main(String args[])
     {
            System.out.println("Welcome to package");
     }
}
```

- If we want to compile the above program in current working directory (Assume `D:\packages`)

  To Compile: `D:\packages>javac –d . Simple.java`
  To run: `D:\packages>javac mypack.Simple`

- Suppose if we want to compile the above program in some other directory (Assume `C:\packages`)

  To compile: `D:\packages> javac -d C:\packages Simple.java`
  To run: `C:\packages>java mypack.Simple`

- To run this program from "`D:\packages`" directory, you need to set classpath of the directory where the class file resides.

  To set class path: `D:\packages>set classpath=c:\packages;.;`
  To run: `D:\packages>java mypack.Simple`

- Another way to run this program by `–classpath` switch of java:

```
D:\packages> java -classpath c:\packages mypack.Simple
```

Output: `Welcome to package`

****