2.1 Packages

2.1.1 Introduction to packages

A **java package** is a group of similar types of classes, interfaces and sub-packages.

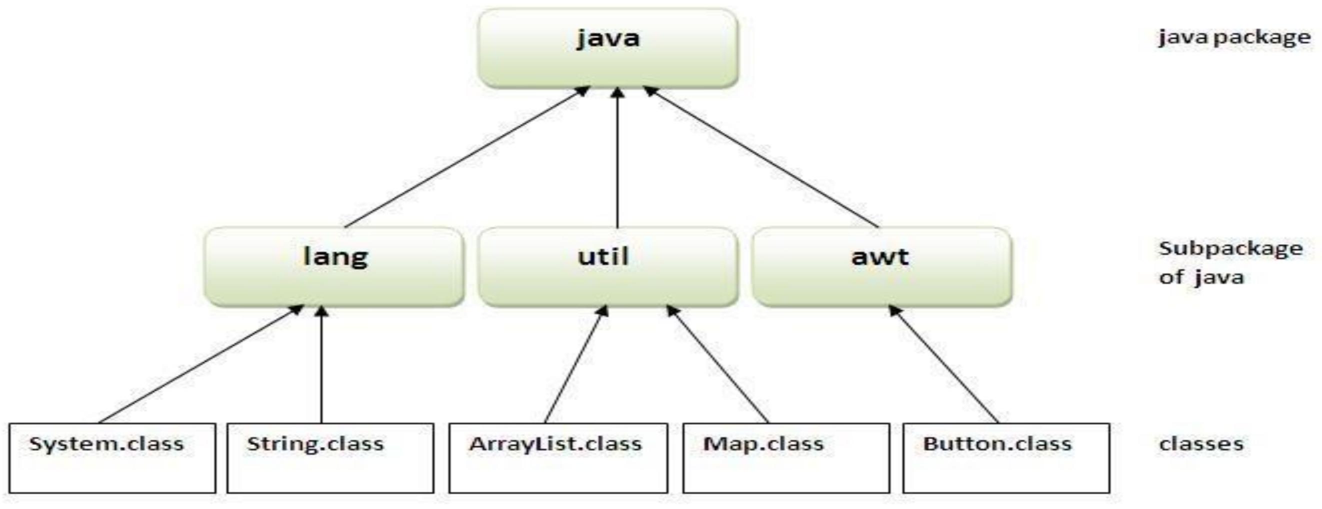
Package in java can be categorized in two forms,

i) built-in package and
ii) user-defined package.

Built-in package:

Collection of classes & interfaces which are already defined are called as Build-in packages.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



user-defined packages:

The collection of classes and interfaces for which definition is provided by the developer or programmer are called as user-defined packages.

2.1.2 Defining a user-defined package

To create a user defined package in java we use a keyword “package”

Syntax:

package packagename[.subpackage1][.subpackage2]…[.subpackageN];

Sample program on userdefined package:

//save as Simple.java

```
package mypack;

public class Simple

{

 public static void main(String args[])

       {

    System.out.println("Welcome to package");

        }

}
```

compile java package:

If you are not using any IDE, you need to follow the syntax given below:

Syntax:

```
javac -d directory javafilename
```

For example

1. javac -d . Simple.java

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

run java package program:

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To 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 . represents the current folder.

2.1.3 CLASSPATH Setting

CLASS PATH environment variable setting is used by java run time system to know where to look for package that is created.

For example, consider the following package specification:

package MyPack

*the class path **<u>must not</u>** include MyPack, itself.*

It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is

C:\MyPrograms\Java\MyPack

Then the class path to MyPack is

C:\MyPrograms\Java

Example:

```
package MyPack;
class Demo
{
…….
…….
}
class classpathdemo
{
public static void main(String args[])
{
Demo obj=new Demo();
………
}
}
```

Call this file **classpathdemo.java** and put it in a directory called **MyPack.**

Next, compile the file. Make sure that the resulting **.class** file is also in the MyPack directory.

Then, try executing the classpathdemo class, using the following command line:

java MyPack.**classpathdemo**

Remember, you will need to be in the directory above MyPack when you execute this command.

2.1.4 Importing packages

There are three ways to access the package from outside the package.

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

1) 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 of package that import the packagename.**

//let us save this program in D:/pack folder as A.java

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

Compile as: D:/pack>javac –d . A.java

//save it D floder as B.java

```
import pack.*;
  class B
{
  public static void main(String args[])
        {
   A obj = new A();
   obj.msg();
        }

}
```

Compile as: D:\>javac B.java

Run as :D:\> java B

Output:Hello

---

2) Using packagename.classname

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

Example of package by import package.classname

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

```
import pack.A;
class B
{
  public static void main(String args[])
                    {
   A obj = new A();
   obj.msg();
                     }
}
```

Output:Hello

3) Using fully qualified name

- If you 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.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

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

```
class B
{
  public static void main(String args[])
        {
   pack.A obj  = new pack.A();//using fully
qualified name
   obj.msg();
         }
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Implicit and Explicit import statement

Imports can be categorized as explicit (for example import java.util.List;) or implicit (also known as 'on-demand', for example import java.util.*;):

Implicit imports give access to all visible types in the type (or package) that precedes the ".*"; types imported in this way never shadow other types.

Explicit imports give access to just the named type; they can shadow other types that would normally be visible through an implicit import, or through the normal package visibility rules.

Example

The following example uses implicit imports. This means that it is not clear to a programmer where the List type on line 5 is imported from.

```
import java.util.*;  // AVOID: Implicit import statements
import java.awt.*;

public class Customers {
    public List getCustomers() {  // Compiler error: 'List' is ambiguous.
        ...
    }
}
```

To improve readability, the implicit imports should be replaced by explicit imports. For example, import java.util.*; should be replaced by import java.util.List; on line 1.

2.1.5 Access protection or Access Modifiers

There are two types of modifiers in java: **access modifier** and **non-access modifier**. The access modifiers specifies accessibility (scope) of a datamember, method, constructor or class.

There are 4 types of 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.

1) private access modifier:

The private datamembers, method ,contructor are accessible only within class in which their are declared and out of the class the doesn't have scope.

Example of private access modifier:

In this example, we have created two classes 'A' and 'Simple'. 'A' class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A
{
private int data=40;
private void msg()
      {
System.out.println("Hello java");
      }
}
  public class Simple
{
 public static void main(String args[])
      {
      A obj=new A();
      System.out.println(obj.data);//Compile Time Error
      obj.msg();//Compile Time Error
   }
}
```

Role of Private Constructor:

If you make any class constructor private, you cannot create the instance of that class from outside the class.

For example:

```
class A
{
private A( )
        {
        }//private constructor
void msg( )
        {
System.out.println("Hello java");
        }
}
public class Simple
{
public static void main(String args[])
        {
  A obj=new A( );//Compile Time Error
        }
}
```

Note: A class cannot be private or protected except nested class

2) default access modifier:

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package.

Example of default access modifier:

In this example, We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

```
package pack;
class A
{
void msg( )
        {
System.out.println("Hello");
        }
}
```

//save by B.java

```
import pack.*;
 class B
{
 public static void main(String args[])
        {
 A obj = new A();//Compile Time Error
 obj.msg();//Compile Time Error
        }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) protected access modifier:

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages “pack” and “ mypack”. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
package pack;
public class A
{
            protected void msg()
            {                       System.out.println("Hello");
            }
}
```

```
import pack.*;
class B extends A
{
 public static void main(String args[])
            {
 B obj = new B();
  obj.msg();
            }
}
```

Output:Hello

4) public access modifier:

The public access modifier is accessible everywhere.

It has the widest scope among all other modifiers.

Example of public access modifier

//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

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

2.2 INTERFACES

2.2.1 Introduction to Interfaces

Question: How multiple inheritance is achieved in java?

(or)

Explain about interface concept?

(or)

How a Is-A relationship is possible in java

An **interface in java** is a blueprint of a class. It has static final 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.
- Java Interface also **represents IS-A relationship**.
- It cannot be instantiated just like abstract class.

They are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

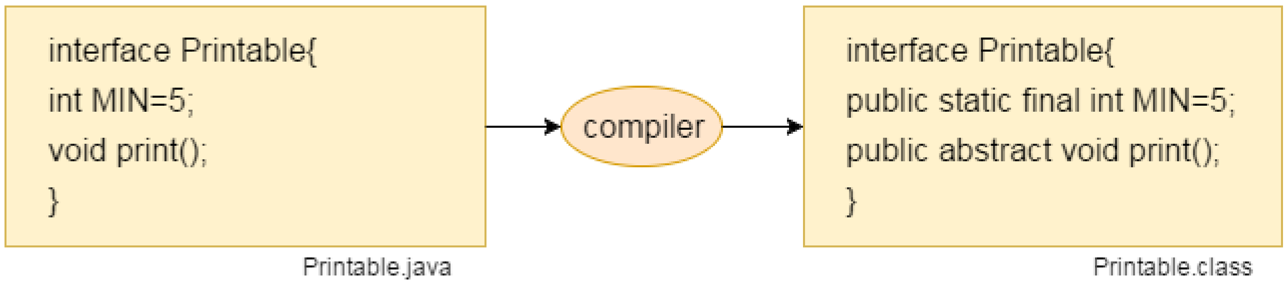
- It can be used to achieve loose coupling.

2.2.2 Defining an Interface

Syntax:

```
public interface NameOfInterface

{

  // Any number of final, static fields

  // Any number of abstract method declarations\

}
```

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



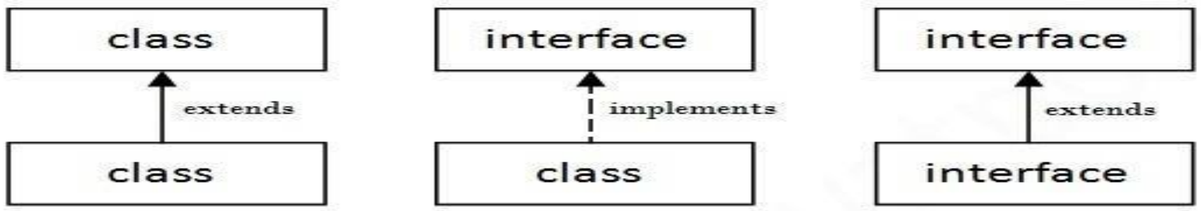
```
interface Printable{
int MIN=5;
void print();
}
```

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

2.2.3 Implementation of Interface

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**



Example on class implementing interface

```
interface MyInterface
                {
   public void method();
                }
class Demo implements MyInterface
{
  public void method()
  {
      System.out.println("Implementation of method");
  }
  public static void main(String arg[])
  {

      Demo obj=new Demo();
      obj. method();
  }
}
```

Output: Implementation of method

Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

For example**:**

```
interface Printable
        {
void print();
        }
interface Showable
        {
void show();
        }

class TestInterface  implements Printable, Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
    public void show()
        {
    System.out.println("Welcome");
          }
        public static void main(String args[])
                {
        TestInterface obj = new TestInterface ();
        obj.print();
        obj.show();
                }
}
```

Output:

```
Hello
```

Welcome

Difference between abstract class and interface

- Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods.
- Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
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 provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

6) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

2.2.4 Nested Interfaces

- An interface which is declared within another interface or class is known as nested interface.
- The nested interfaces are used to group related interfaces so that they can be easy to maintain.
- The nested interface must be referred by the outer interface or class.
- It can't be accessed directly.

Points to remember for nested interfaces

• Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

• Nested interfaces are declared static implicitely.

Syntax of nested interface which is declared within the interface:

```
 interface interface_name
 {
         ...
  public interface nested_interface_name
        {
         ...
        }
}
```

-Example of nested interface which is declared within the interface

```
interface Outerinterface
{
void show();
public interface innerinterface
{
public void message();
}
}
class Test implements Outerinterface.innerinterface
{
void show()
{
System.out.println(" implementation of outerinterface show method");
}
public void message()
{
System.out.println("implementation of nestedinterface message method");
}
public static void main(String args[])
{
Test obj=new Test();
obj.show();
obj.message();
}
}
```

Output:

```
implementation of outerinterface show method
implementation of nestedinterface message method
```

Syntax of nested interface which is declared within the class

```
class class_name
{
        ...
        interface nested_interface_name
                {
                ...
                }
}
```

Example of nested interface which is declared within the class

```
class A
{
public interface innerinterface
{
public void message();
}
}
class NestedTest implements A.innerinterface
{
public void message()
{
System.out.println("implementation of nestedinterface message method");
}
public static void main(String args[])
{
NestedTest obj=new NestedITest();
obj.message();
}
}
```

Output:

implementation of nestedinterface message method

Example2:

If class have any members and methods ,then to access this members and methods

- First extend the class and implement the interfaces

So that you will have access to members and methods of class also.

```
class A
{
void show()
{
System.out.println("Hello");
}
public interface innerinterface
{
public void message();
}
}
class NestedTest extends A implements A.innerinterface
{
public void message()
{
System.out.println("implementation of nestedinterface message method");
}
public static void main(String args[])
{
NestedTest obj=new NestedTest();
obj.show();
obj.message();
}
}
```

Output:

Hello

implementation of nestedinterface message method");

NOTE:

If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
interface interface_name
{
 class  class_name
          {
          …..
          }
}
```

2.2.5 Applying Interfaces

A sample example on Applying *of interface*

```
interface MyInterface
                {
   public void method();
                }
class Demo implements MyInterface
{
  public void method()
  {
      System.out.println("Implementation of method");
  }
  public static void main(String arg[])
  {
            Demo obj=new Demo();
      obj. method();
  }
}
```

Output: Implementation of method

2.2.6 Variables in Interface

- An interface can also contain variables just as classes but the variable in interface must be of
- Public,static and final access modifiers.
- If the variable are not specified with access specifier then during compilation process compiler will provide this access specifier.

Syntax for declaring variables in interface:

```
interface  interface_name
{
public static final  datatype variablename=value;
....
}
```

Example for declaring a variable in interface:

```
interface hai
{
public static final min=5;
}
class demo impliments hai
{
public static void main(string args[])
        {
        demo obj=new demo();
        System.out.println (“minimum value is”);
        System.out.println (obj.min);
        }
}
```

2.2.7 Extension of Interface

Example on interface extending interface:

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable
          {
void print();
          }
interface Showable extends Printable
          {
void show();
          }
class TestInterface  implements Showable
{
public void print()
          {
System.out.println("Hello");
          }
public void show()
          {
System.out.println("Welcome");
          }

public static void main(String args[])
          {
TestInterface  obj = new TestInterface();
obj.print();
obj.show();
           }
}
```

Output:

```
Hello

Welcome
```