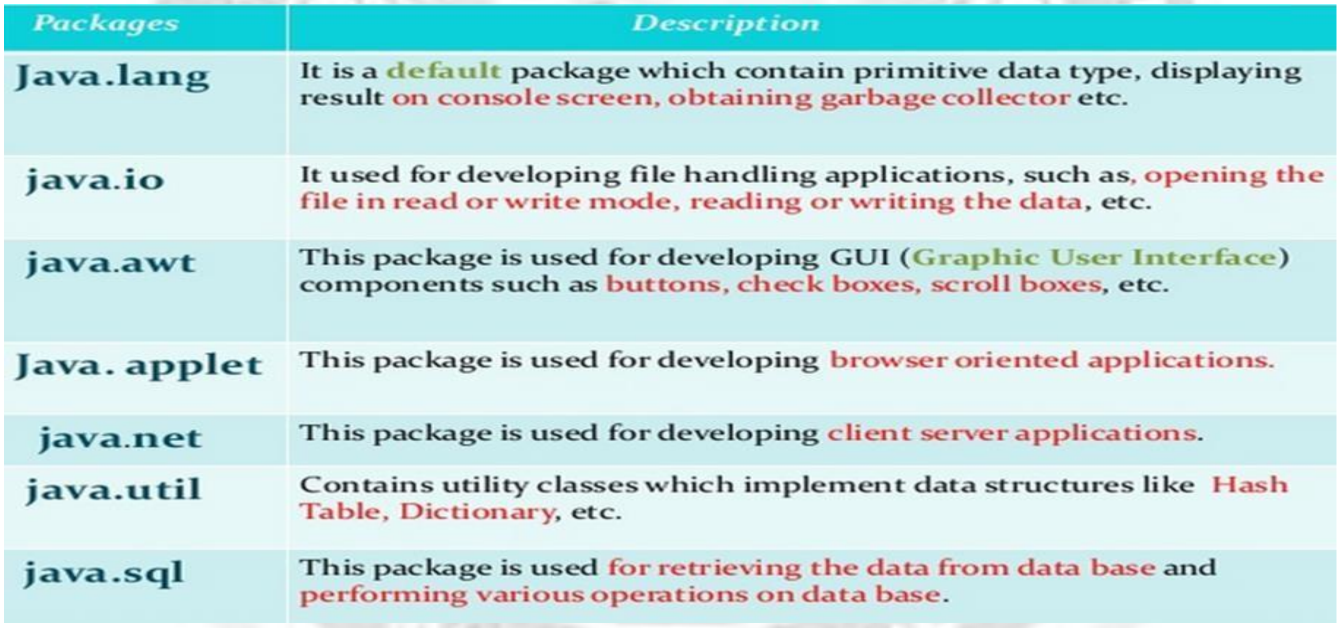
2.10: PACKAGES

Definition:

A Package can be defined as a collection of classes, interfaces, enumerations and annotations, providing access protection and name space management.

✓ Package can be categorized in two form:
1. Built-in package
2. user-defined package.



Packages	Description
Java.lang	It is a default package which contain primitive data type, displaying result on console screen, obtaining garbage collector etc.
java.io	It used for developing file handling applications, such as, opening the file in read or write mode, reading or writing the data, etc.
java.awt	This package is used for developing GUI (Graphic User Interface) components such as buttons, check boxes, scroll boxes, etc.
Java. applet	This package is used for developing browser oriented applications.
java.net	This package is used for developing client server applications.
java.util	Contains utility classes which implement data structures like Hash Table, Dictionary, etc.
java.sql	This package is used for retrieving the data from data base and performing various operations on data base.

Table: List of Built-in Packages

<u>Advantage of Package:</u>

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.
- To bundle classes and interface
- The classes of one package are isolated from the classes of another package
- Provides reusability of code
- We can create our own package or extend already available package

<u>CREATING USER DEFINED PACKAGES:</u>

Java package created by user to categorize their project's classes and interface are known as user-defined packages.

✓ When creating a package, you should choose a name for the package.

✓ Put a **package** statement with that name at the top of every source file that contains the classes and interfaces.

- ✓ The **package** statement should be the first line in the source file.
- ✓ There can be only one package statement in each source file
- ✓ **Syntax:**

```
package package_name.[sub_package_name];
public class classname
{  ……..
   ……..
}
```

- ✓ Steps involved in creating user-defined package:
  1. Create a directory which has the same name as the package.
  2. Include package statement along with the package name as the first statement in the program.
  3. Write class declarations.
  4. Save the file in this directory as “name of class.java”.
  5. Compile this file using java compiler.

- ✓ Example:

```
package pack;
public class class1 {
public static void greet()
{  System.out.println(“Hello”);   }
}
```

To create the above package,

1. Create a directory called pack.
2. Open a new file and enter the code given above.
3. Save the file as class1.java in the directory.
4. A package called pack has now been created which contains one class class1.

<u>ACCESSING A PACKAGE (using “import” keyword):</u>

- The import keyword is used to make the classes and interface of another package accessible to the current package.

<u>Syntax:</u>

```
import package1[.package2][.package3].classname or *;
```

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.

✓ **Using packagename.classname**

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

✓ **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.

Example :

greeting.java (create a folder named "pack" in **F:\** and save **)**

```
package pack;
public class greeting{
public static void greet()
{  System.out.println("Hello! Good Morning!");   }
}
```

FactorialClass.java (create a folder named "Factorial" inside **F:\pack** and save**)**

```
package Factorial;
public class FactorialClass
{
public int fact(int a)
{
if(a==1)
return 1;
else
return a*fact(a-1);
}
}
```

ImportClass.java (save the file in F:\ **)**

```
import java.lang.*;  // using import package.*
import pack.Factorial.FactorialClass; // using import package.subpackage.class;
import java.util.Scanner;
public class ImportClass
{
public static void main(String[] arg)
{
int n;
Scanner in=new Scanner(System.in);
System.out.println("Enter a Number: ");




n=in.nextInt();
pack.greeting p1=new pack.greeting();  // using fully qualified name
p1.greet();
FactorialClass fobj=new FactorialClass();
System.out.println("Factorial of "+n+" = "+fobj.fact(n));
System.out.println("Power("+n+",2) = "+Math.pow(n,2));
}
}
```

Output:

```
F:\>java ImportClass
Enter a Number:
5
Hello! Good Morning!
Factorial of 5 = 120
Power(5,2) = 25.0
```

PACKAGES AND MEMBER ACCESS:

Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
There are two levels of access control:

- **At the top level— public,** or package-private (no explicit modifier).
- **At the member level—public, private, protected,** or package-private (no explicit modifier).

Top Level access control:

- ✓ A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- ✓ If a class has no modifier (the default, also known as package-private), it is visible only within its own package.

Member Level access control:

- ✓ **public** – if a member is declared with public, it is visible and accessible to all classes everywhere.
- ✓ **private** - The private modifier specifies that the member can only be accessed in its own class.
- ✓ **protected** - The protected modifier specifies that the member can only be accessed within its own package and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The following figure shows the four classes in this example and how they are related.

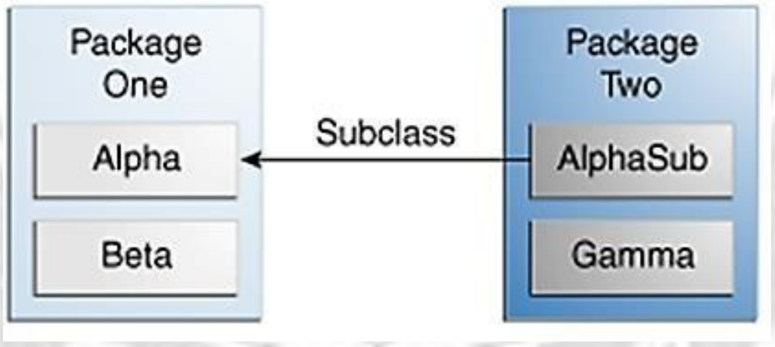


Figure: Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Visibility				
Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Example:

Z:\MyPack\FirstClass.java

```
package MyPack;

public class FirstClass
{
  public String i="I am public variable";
  protected String j="I am protected variable";
  private String k="I am private variable";
  String r="I dont have any modifier";
}
```

Z:\MyPack2\SecondClass.java

```
package MyPack2;
import MyPack.FirstClass;
class SecondClass extends FirstClass {
   void method()
   {
      System.out.println(i);     // No Error: Will print "I am public variable".
     System.out.println(j);    // No Error: Will print "I am protected variable".
     System.out.println(k); // Error: k has private access in FirstClass
     System.out.println(r);  // Error: r is not public in FirstClass; cannot be accessed
                                  //           from outside package
   }

   public static void main(String arg[])
   {
      SecondClass obj=new SecondClass();
      obj.method();
   }
}
```

Output:

```
I am public variable
I am protected variable

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - k
has private access in MyPack.FirstClass
```

Visibility of the variables i,j,k and r in MyPack2				
Accessibility	**i**	**j**	**k**	**R**
Class	Y	Y	Y	Y
Package	Y	Y	N	N
Subclass	Y	Y	N	N
world	Y	N	N	N

Table: Accessibility of variables of MyyPack/FirstClass in MyPack2/SecondClass

2.11: INTERFACES

“interface” is a keyword which is used to achieve full abstraction. Using interface, we can specify what the class must do but not how it does.

Interfaces are syntactically similar to classes but they lack instance variable and their methods are declared without body.

Definition:

An interface is a collection of method definitions (without implementations) and constant values. It is a blueprint of a class. It has static constants and abstract methods.

➢ **<u>Why use Interface?</u>**

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

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.
- Writing flexible and maintainable code.
- Declaring methods that one or more classes are expected to implement.

➢ **<u>An interface is similar to a class in the following ways:</u>**

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

➢ **<u>Defining Interfaces:</u>**

An interface is defined much like a class. The keyword **“interface”** is used to define an interface.

<u>Syntax to define interface:</u>

```
[access_specifier] interface InterfaceName
{
Datatype VariableName1=value;
Datatype VariableName2=value;
.
.
Datatype VariableNameN=value;
returnType methodName1(parameter_list);
returnType methodName2(parameter_list);
.
.
returnType methodNameN(parameter_list);
}
```

Where,

Access_specifer : either **public** or none.

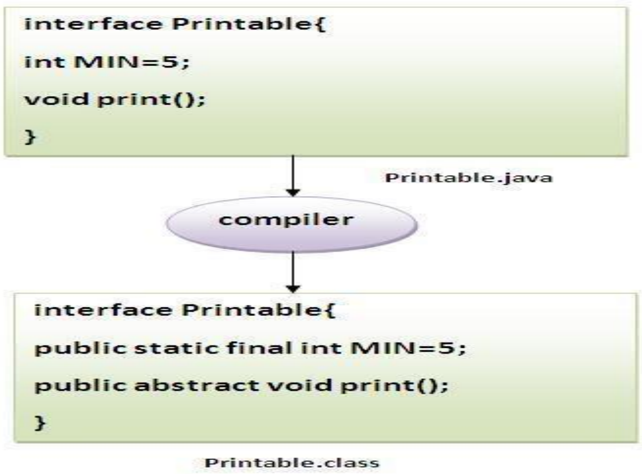
Name: name of an interface can be any valid java identifier.

Variables: They are implicitly **public, final and static**, meaning that they cannot be changed by the implementing class. They must be initialized with a constant value.

Methods: They are implicitly **public and abstract**, meaning that they must be declared without body and defined only by the implementing class.

Note: The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

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

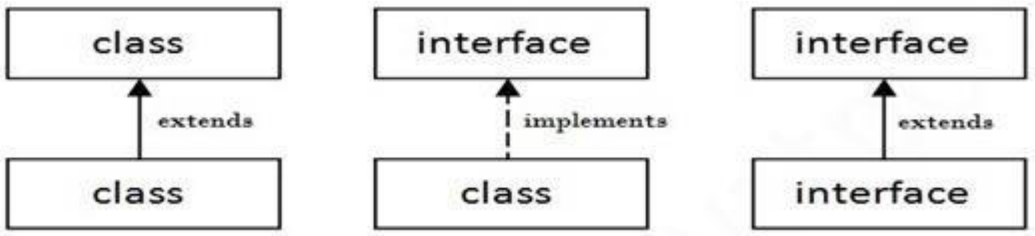


```
interface Printable{
int MIN=5;
void print();
}
```

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

➢ **<u>Understanding relationship between classes and interfaces</u>**

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



Implementing Interfaces ("implements" keyword):

- ✓ Once an interface has been defined, one or more classes can implement that interface.
- ✓ A class uses the **implements** keyword to implement an interface.
- ✓ The implements keyword appears in the class declaration following the extends portion of the declaration.

- ✓ **Syntax:**

```
[access_specifier] class class_name [extends superclassName] implements
interface_name1, interface_name2…
{
//implementation code and code for the method of the interface
}
```

Rules:

1. If a class implements an interface, then it must provide implementation for all the methods defined within that interface.
2. A class can implement more than one interfaces by separating the interface names with comma(,).
3. A class can extend only one class, but implement many interfaces.
4. An interface can extend another interface, similarly to the way that a class can extend another class.
5. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

✓ **Example:**

```
/* File name : Super.java */
interface Super
 {
   final int x=10;
   void print();
}
/* File name : Sub.java */
class Sub implements Super
{
   int y=20;
  x=100                //ERROR; cannot change modify the value of final variable

  // defining the method of interface
   public void print()
   {
              System.out.println(“X = “+x);
              System.out.println(“Y = “+y);
   }
}
class sample
{
   public static void main(String arg[])
{
   Sub SubObj=new Sub();
   SubObj.print();
   Super SupObj=new Sub(); // interface variable referring to class object
   SupObj.print();
```

```
        }
    }
```

Output:

```
$java sampleX = 10
Y = 20
X = 10
Y = 20
```

➢ **The rules for interfaces:**

Member variables:

- Can be only public and are by default.
- By default are static and always static
- By default are final and always final

Methods:

- Can be only public and are by default.
- Cannot be static
- Cannot be Final

➢ **When overriding methods defined in interfaces there are several rules to be followed:**

- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.

➢ **Properties of Interfaces:**

1. Interfaces are not classes. So the user can never use the new operator to instantiate an interface.
   Example: interface super {}
   ```
       X=new Super() // ERROR
   ```
2. The interface variables can be declared, even though the interface objects can't be constructed.
   ```
           Super x; // OK
   ```
3. An interface variable must refer to an object of a class that implements the interface.
4. The instanceOf() method can be used to check if an object implements an interface.
5. A class can extend only one class, but implement many interfaces.
6. An interface can extend another interface, similarly to the way that a class can extend another class.
7. All the methods in the interface are **public** and **abstract.**
8. All the variable in the interface are **public, static** and **final.**

- **<u>Extending Interfaces:</u>**
  - ✓ An interface can extend another interface, similarly to the way that a class can extend another class.
  - ✓ The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
  - ✓ **Syntax:**

```
[accessspecifier] interface InterfaceName extends interface1, interface2,.....
{
Code for interface
}
```

Rule: When a class implements an interface that inherits another interface it must provide implementation for all the methods defined within the interface inheritance chain.

Example:

```
interface A
{
        void method1();
}
/* One interface can extend another interface. B now has two abstract methods */
interface B extends A
{
void method2();
}
// This class must implement all the methods of A and B

class MyClass implements B
{
public void method1() // overriding the method of interface A
{
System.out.println("—Method from interface: A—");
}
public void method2() // overriding the method of interface B
{
System.out.println("—Method from interface: B—");
}
public void method3() // instance method of class MyClass
{
System.out.println("—Method of the class : MyClass—");
}
public static void main(String[] arg)
{
MyClass obj=new MyClass();
Obj.method1();
Obj.method2();
Obj.method3();
}
}
```

Output:

```
F:\> java MyClass
--Method from Interface: A—
--Method from Interface: B—
--Method of the class: MyClass--
```

- **Difference between Class and Interface:**

Class	Interface
The class is denoted by a keyword **class**	The interface is denoted by a keyword **Interface**
The class contains data members and methods. but the methods are defined in the class implementation. thus class contains an executable code	The interfaces may contain data members and methods but the methods are not defined. the interface serves as an outline for the class
By creating an instance of a class the class members can be accessed	you cannot create an instance of an interface
The class can use various access specifiers like public, private or protected	The interface makes use of only public access specifier
The members of a class can be constant or final	The members of interfaces are always declared as final

- **Difference between Abstract class and interface**

Abstract Class	Interface
Multiple inheritance is not possible; the class can inherit only one abstract class	Multiple inheritance is possible; The class can implement more than one interfaces
Members of abstract class can have any access modifier such as **public, private and protected**	Members of interface are **public** by default
The methods in abstract class may be **abstract method or concrete method**	The methods in interfaces are **abstract** by default
The method in abstract class **may or may not have implementation**	The methods in interface have **no implementation at all.** Only declaration of the method is given
Java abstract class is extended using the keyword **extends**	Java interface can be implemented by using the keyword **implements**
The member variables of abstract class can be **non-final**	The member variables of interface are **final** by default
Abstract classes **can have constructors**	Interfaces **do not have any constructor**
Only abstract methods need to be overridden.	All the method of an interface must be overridden.
Non-abstract methods can be static.	Methods cannot be static.

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

Example for Interface :

```
interface Bank
{
    float rateOfInterest();
}
class SBI implements Bank
{
     public float rateOfInterest()
    {
        return 9.15f;
     }
}
class PNB implements Bank
{
     public float rateOfInterest()
    {
        return 9.7f;
    }
}
class TestInterface2
{
    public static void main(String[] args)
   {
        Bank b=new SBI();
        System.out.println("ROI: "+b.rateOfInterest());
   }
}
```

Output:

```
ROI: 9.15
```