Unit – III: I Inheritance, Packages & Interfaces

1. Basics of Inheritance, Types of inheritance: single, multiple, multilevel, hierarchical and hybrid inheritance.

Basics of Inheritance:

- **Definition: "The mechanism of deriving a new class from old class is called Inheritance."**
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Also you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- Why use inheritance in java:
    - For Method Overriding (so runtime polymorphism can be achieved).
    - For Code Reusability.
- Terms used in Inheritance:
    - **Class**: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
    - **Sub Class/Child Class/Derived Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
    - **Super Class/Parent Class/Base Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
    - **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Here, the meaning of "extends" is to increase the functionality.

Types of inheritance: single, multiple, multilevel, hierarchical and hybrid inheritance:

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.

Types of inheritance:

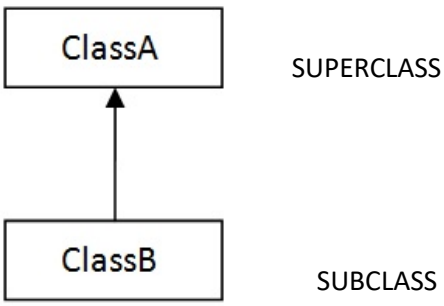
1) **Single** inheritance (only one super class)
2) **Multilevel** inheritance (derived from a derived class)
3) **Hierarchical** inheritance(1 super class, many subclasses)
4) **Multiple** inheritance(several super class)
5) **Hybrid** Inheritance

Single inheritance: When a class inherits another class, it is known as a *single inheritance*.

- Only one super class
- Every class has one and only one direct super class.

Syntax:

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



1) Single

Example:

```
class Art{
        void draw(){
                System.out.println("Nice drawing");
        }
}
```

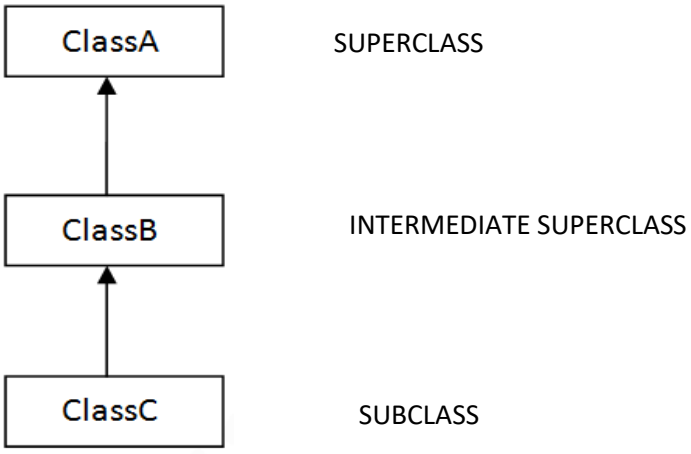
```
class Paint extends Art{
        void color(){
                System.out.println("Nice colors");
        }
}
class TestInheritance{
        public static void main(String args[]){
        Paint p=new Paint();
        p.draw();
        p.color();
}}
```

Multilevel inheritance: When new class is derived from derived class is known as *multilevel inheritance*.

- Java uses mainly in building its class library.
- Ex:

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

Here class C can use properties of class B and class B uses the properties of class A. So class C can use properties of A and B.



2) Multilevel

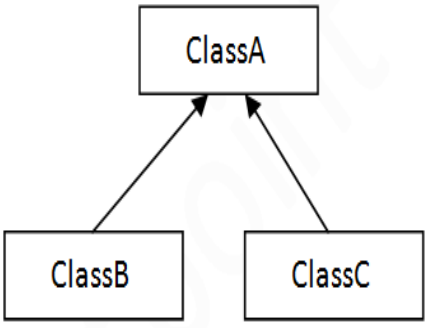
Example:

```
class A   //super class {
          int i;
 }
class B extends A //intermediate superclass which extends A
 {
          int j;
          void display () {
            System.out.println ("i in class B="+i);
          }
 }
 class C extends B //sub class C which extends B
{
          void sum () {
            System.out.println ("sum of i and j in C ="+ (i + j));
          }
}
class Multilevel {
    public static void main (String args[]) {
          B b1=new B ( );
          b1.j=5;
          System.out.println ("j in B="+b1.j);
          C c1=new C ( );
          c1.i=3;
          c1.j=4;
          c1.sum();
          }
```

Output: j in B=5
sum of i and j in C=7

Hierarchical inheritance: When one super class having many subclasses is known as *hierarchical inheritance.*

- Many classes extend one sub class.



3) Hierarchical

Ex:

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

Here, class A extended by class B, C, and D.

Example:

```
class A                                  //super class
{
        int i;
}
class B extends A                 // sub class of A
{
        int j;
        void display()
        {
          System.out.println("sum of i and j in B="+ (i +j));
        }
}
class C extends A                 //subclass of B
{
        int k;
        void sum()
        {
        System.out.println("sum of i and k in C ="+ (i+k));
```

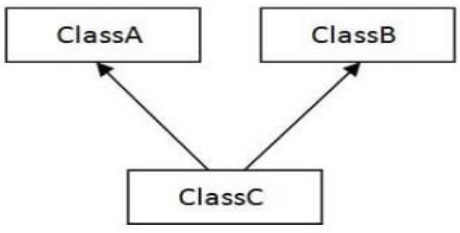
```
        }
}
class H
{
        public static void main(String args[])
        {
        B b1=new B();
        b1.i=2;
        b1.j=3;
        b1.display();

        C c1=new C();
        c1.i=2;
         c1.k=4;
        c1.sum();
        }
        }
```

Multiple inheritance: One class extends properties of Several super classes is known as Multiple inheritance.

- Java does not support this directly.
- Here class C uses properties of class A and class B at same level.
- By using interfaces it is possible to implement multiple inheritance in java.

Example:

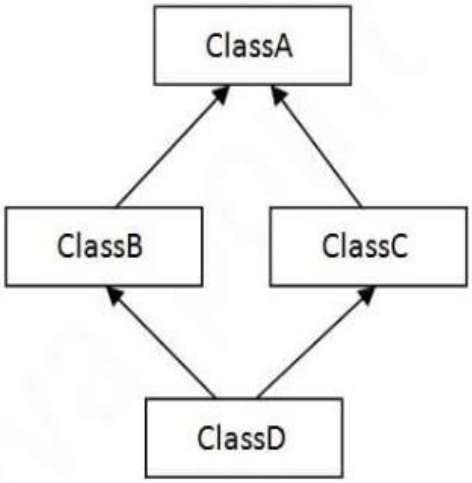
```
class A {
   --------- ---------
}
class B {
   --------- ---------
}
class C extends A ,B       //Invalid and Not supported in java
{
  --------- --------
}
```



4) Multiple

Hybrid inheritance: We can inherit properties of several classes at same level is known as Hybrid Inheritance.

- If you are using only classes then this is not allowed in java.
- But using interfaces it is possible to have implement hybrid inheritance in java.



5) Hybrid

- In general, the meaning of hybrid (mixture) is made of more than one thing. In Java, the hybrid inheritance is the composition of two or more types of inheritance.
- Here, if in above figure B and C are classes then this inheritance is not allowed beacuse a single class cannot extend more than one class (Class D is extending both B and C).
- If we write program of hybrid inheritance as shown below then it gives error.
- Example:

```
class A
{
    public void methodA()
    {
        System.out.println("Class A methodA");        }
 }
 class B extends A
{
     public void methodA()
     {
        System.out.println("Child class B is overriding inherited method A");        }
    public void methodB()
    {
        System.out.println("Class B methodB");            }
  }
```

```
class C extends A
{
    public void methodA()
    {
        System.out.println("Child class C is overriding the methodA");
    }
    public void methodC()
    {
        System.out.println("Class C methodC");
    }
}
class D extends B,C                    // it generates error
{
    public void methodD()
    {
        System.out.println("Class D methodD");
    }
}
}
class multiple
{
    public static void main(String args[])
    {
        D obj1= new D();
        obj1.methodD();
        obj1.methodA();
    }
}
```

Here, when we call method() then compiler cannot decide which method() shall be call from class B or C as they both extends class A. So here method() is overridden in class B and C. So Hybrid inheritance is not possible in this way. We can implement hybrid inheritance using interfaces.

2. method overriding, Object class and overriding its methods : equals(), toString(), finalize(), hashCode()

Method overriding (Function Overriding):

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
class Vehicle{
    void run(){System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output:

Vehicle is running safely

Definition: "If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**."

Usage of Java Method Overriding

- It is used to provide the specific implementation of a method which is already provided by its superclass.
- It is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example:

```
class Vehicle
{
    void run()
    {
        System.out.println("Vehicle is running");
    }
}

class Bike2 extends Vehicle
{
    void run()
    {
        System.out.println("Bike is running safely");
    }

    public static void main(String args[])
    {
        Bike2 obj = new Bike2();
        obj.run();
    }
}
```

Output: Bike is running safely

Object class and overriding its methods : equals(), toString(), finalize(), hashCode()

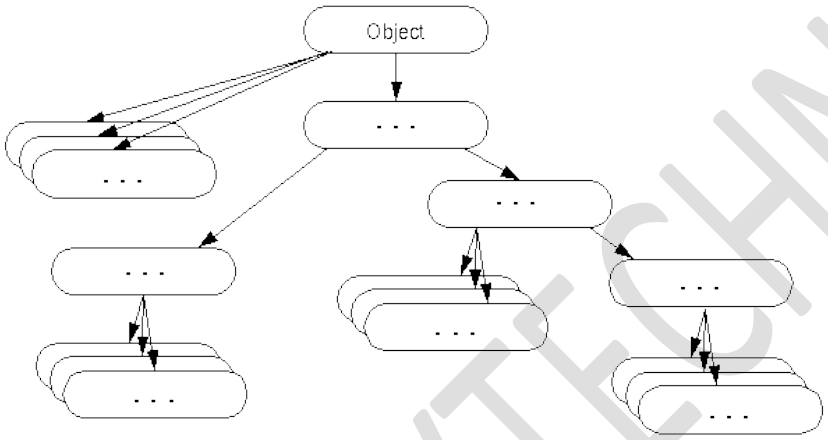
- In java object class is the root of class hierarchy. All other class in Java inherit directly or indirectly from the object class.
- Object class provides a number of useful methods that can be used by all Java classes.
- **Object class declaration:** Declaration for java.lang.Object class is….

  Syntax: **Public class Object**

- Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws	causes the current thread to wait, until another thread notifies (invokes

InterruptedException	notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.



OBJECT CLASS

Methods of Object class

The Object class provides many methods. They are as follows:

Java toString() Method:

- If you want to represent any object as a string, **toString() method** comes into existence.
- The toString() method returns the String representation of the object.
- **Advantage of Java toString() method:** By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.
- **Example:**

```
class Student
{
    int rollno;
    String name;

    Student(int rollno, String name)
    {
        this.rollno=rollno;
        this.name=name;
    }
     public String toString()
     {
```

```
            return rollno+" "+name;
        }
        public static void main(String args[])
        {
          Student s1=new Student(101,"Jia");
          Student s2=new Student(102,"Jeni");

          System.out.println(s1);//compiler writes here s1.toString()
          System.out.println(s2);//compiler writes here s2.toString()
        }
    }
```

Output:

101 Jia
102 Jeni

Java equals() Method:

- equals() is the method of Object class.
- This method is used to compare the given objects.
- By default, it compares memory addresses of objects.
- **Example:**

```
class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}
// Driver class to test the Complex class
public class Test{
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(10, 15);
        if (c1 == c2) {
            System.out.println("Equal ");
        } else {
            System.out.println("Not Equal ");
        }
    }
}
```

Output: Not Equal

Java hasCode() Method:

- For every object, JVM generates a unique number which is a hashcode.
- It returns distinct integers for distinct objects.
- A common misconception about this method is that the hashCode() method returns the address of the object, which is not correct.
- It converts the internal address of the object to an integer by using an algorithm.
- ***Use of hashCode() method***
    - It returns a hash value that is used to search objects in a collection.
    - JVM uses the hashcode method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc.
    - The main advantage of saving objects based on hash code is that searching becomes easy.
- **Example:**

```
public class Student {
        static int last_roll = 100;
        int roll_no;

        // Constructor
        Student()
        {
                roll_no = last_roll;
                last_roll++;
        }

        // Overriding hashCode()
        @Override public int hashCode() { return roll_no; }

        public static void main(String args[])
        {
                Student s = new Student();
                Student s2 = new Student();


                System.out.println(s);
                System.out.println(s.toString());
                System.out.println(s2);
                System.out.println(s2.toString());
        }
}
```

Output:
```
System@64
System@64
System@65
System@65
```

3. Defining interface, implementing interface, multiple inheritance using 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.*
- **Use :** There are mainly three reasons to use interface.
  - It is used to achieve abstraction.
  - By interface, we can support the functionality of multiple inheritance.
  - It can be used to achieve loose coupling.

Syntax to define interface:

```
access interface interfaceName
{
        variable declaration;
        method declaration;
}
```

- Interface is keyword**.**
- Access: is access modifier, it must be either public or not used any other(friendly).

Declaration of variable:

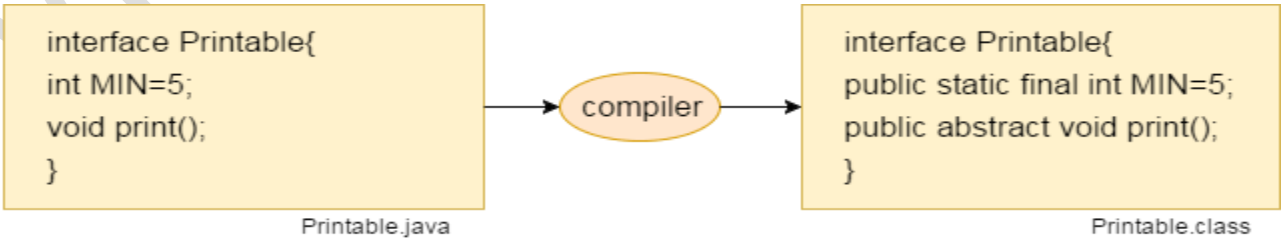
static final *type* variablename = **value**;

Note:

- All variables are declared as constants.
- It is allowed to declared variable without **final** and **static**, because allvariables in interface are treated as constants.

Rules for using interface:

- Methods inside interface must not be static, final, native or strict.
- All variables declared inside interface are implicitly public static final variables(constants).
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.



Interface fields are public, static and final by default, and the methods are public and abstract.

Implementing Interface:

- Interface is a prototype or template for a class so we must implement an interface to use it.
- For this, we requires to implement all methods defined in interface.
- Use implement keyword to implement interface.
- Syntax:

```
class classname implements InterfaceName
{
        body of class name
}
```

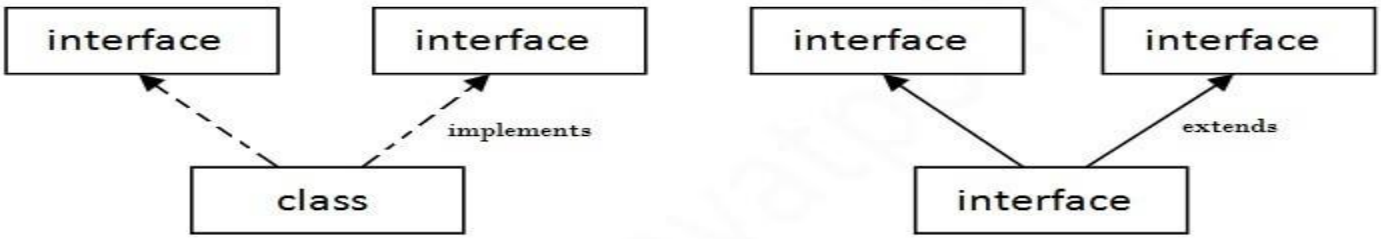
- Example:

```
public interface Shape
{
    double getArea();
    double getPerimeter();

}
public class Rectangle implements Shape
{
    private double l;
    private double w;

    public Rectangle(double l, double w)
    {
        this.l = l;
        this.w = w;

    }
    @Override
    public double getArea()
    {
        return l * w;
    }

    @Override
    public double getPerimeter ()
    {
        return 2 * ( l + w );
    }
}
public class InterfaceEx
{
    public static void main(String[] args)
    {
```

```
            Rectangle r = new Rectangle();
            System.out.println("Rectangle area:"+ r.getArea());
            System.out.println("Rectangle perimeter:"+ r. getPerimeter());
        }
    }
```

Multiple inheritance using interface

- One interface can inherit from another interface by use of "**extends"** keyword.
- This is known as interface inheritance as interfaces support multiple inheritance.
- Example:
    - Public interface Hockey extends Sports, Event
- If a class implements multiple interfaces or interface extends multiple interfaces is known as multiple inheritance.



Multiple Inheritance in Java

Example: Multiple inheritance with interfaces:

```
//Interface 1
interface Bird
{
        public void eat();
}
//Interface 2
interface Fly
{
        public void canFly();
}
Class Bird implements Bird, Fly
{
        public void eat()
        {
                System.out.println("I can eat");
        }

        public void canFly()
```

```
        {
                System.out.println("I can fly");
        }
}
public class multipleInheritanceEx
{
        pubic static void main(String[] args)
        {
                Bird b = new Bird();
                b.eat();   //inherited from Bird interface
                b.canFly();  //inherited from Fly interface
        }

}
```

Output:

```
I can eat
I can fly
```

Here, we have two interfaces, Bird and Fly each with method. eat() method is inherited from Bird interface and canFly() method is inherited from Fly interface. So Bird class can inherit from multiple interfaces, achieving multiple inheritance in Java.

4. Abstract class and final class

Abstract class:

- **Abstraction** is a process of hiding the implementation details and showing only functionality to user.
- A class which is declared with the abstract keyword is known as an abstract class in Java.
- If we want method must always be redefined in subclass, to make overriding compulsory then abstract (modifier) keyword is used to make overriding compulsory.
- **abstract** keyword can be used with class and methods.Ex:

```
abstract class Shape
{
  --------
  abstract void draw( );
  --------
}
```

- Abstract methods are referred as sub class responsibility because they have no implementation specified in the super class. Subclass must override them,it cannot simply use the version defined in super class.
- Method in super class does not have body.

  Syntax: abstract type ***name*** (parameter-list);

- If a class contains one or more abstract methods, it should be declared asabstract class.

- Any subclass of an abstract class must either implement all of the abstract methods in the super class, or be itself declared abstract.
- Abstract class has both abstract (without body) and concrete methods (normal method which have body).
- Conditions to use abstract:
  - We cannot use abstract classes to initiate objects directly.
    Shape s=new **Shape**( );
  - The abstract methods of an abstract class must be redefined in its subclass.
  - We cannot declare abstract constructor or abstract static methods.

Example:

```
abstract class A
{
        abstract void callme ( );          //abstract method(without body)
        // concrete method are still allowed in abstract class
        void callmeToo( )       //concrete method(with body)
        {
                System . out .println( "this is concrete method");
        }
}
class B extends A
{
        void callme( )
        {
                System. out .println("B's implementation of call me");
        }
}
class AbstractDemo
{
public static void main(String args[])
{
        B b1=new B( );
        //A a1=new A( );  // invalid, because we cannot create object of abstract class.
        A a1;
        a1=b1;
        b1.callme( );
        b1.callmeToo( );
        a1.callme( ) );
        }
}
```

Output:

```
B's implementation of call me
this is concrete method
B's implementation of call me
```

final class

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) Java final variable
If you make any variable as final, you cannot change the value of final variable(It will be constant).

2) Java final method
If you make any method as final, you cannot override it.

3) Java final class
If you make any class as final, you cannot extend it.

Example:

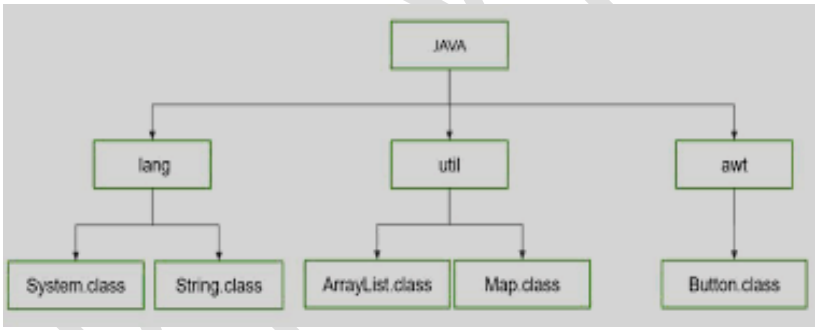
```
final class A
{
        final void display()
        {
          System.out.println("this is display method");
        }
}
class B extends A
{
        void display()
        {
         System.out.println("this is display method");
        }
}
class P10
{
        public static void main( String args[])
        {

         A a1 = new A();
         B b1 = new B();
         a1.display();
         b1.display();

        final int MAX=50;
        System.out.println( "MAX="+MAX);
        MAX=100;
        }
}
```

Output:

```
p10.java:8: error: cannot inherit from final A
class B extends A
                ^
p10.java:10: error: display() in B cannot override display() in A
 void display()
      ^
  overridden method is final
p10.java:27: error: cannot assign a value to final variable MAX
MAX=100;
^
3 errors
```

5. Creating package, importing package, access rules for packages

- Main feature of OOP is its reusability of the code.
- A Package is a collection of logically related classes, Interfaces or subpackages.
- Packages act as "containers" for classes.
- Java packages are classified into two types
    1. Java API packages (system defined package) :Existing java packages Ex. Java.lang
    2. User defined packages: Created by user Ex.



Advantage of package

- Provide common classes and interfaces for any program.
- It provides access protection.
- It removes naming collision.
- Application development time is reduced, because of reuse of the code.
- Application execution time is less.
- Improve performance.
- Reduce redundancy(duplication).

Creating package

- To create a package first declares the name of package using packagekeyword.
  Syntax:

  package packagename;

- This must be first statement in source file.

Example:

```
package StudentPackage; //package declaration
public class Student //class declaration
{
    //body of class
}
```

- Save that file as Student.java in directory StudentPackage.

How to compile java package: If you are not using any IDE, you need to follow the syntax:

Syntax: java –d directory javafilename

Ex: java –d . Simple.java

Here, -d specify destination where to put the generated class file. If you want to keep the package within same directory, you can use dot.

How to run java package: Use fully qualified name eg. myPacke.Simple to run the class.

Ex. Java mypack.Simple

Importing Package:

Following are the syntax to include/use Package in program:

Including Package

Import package_name.*;

Including Package Member

Import package_name.member_name;

Accessing Package Member in program:

package_name.subpackage_name.member_name;

How to access package from another package?

There are 3 ways to access package from outside package:

a. Using packagename
b. Using packagename.classname
c. Using fully qualified name

a. Using packagename:

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- Use the keyword **import** to import a package.
- Example:

```
//save by Student1.java
package pack;
public class Student1
{
        public void msg() {System.out.println("Whatsapp");}
```

```
}
//save by Student2.java
paclage mypack;
import pack.*;
class Student2
{
        public static void main(String args[])
        {
                Student1 obj = new Student1();
                obj.msg();
        }
}
```

Output: Whatsapp

b. Using packagename.classname:

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

```
//save by Student1.java
package pack;
public class Student1
{
        Public void msg() {System.out.println("Whatsapp me");}
}
//save bu Student2.java
package mypack;
import pack.Student1;
class Student2
{
        public static void main(String args[])
        {
                Student1 obj = new Student1();
                obj.msg();
        }
}
```

Output: Whatsapp

c. Using fully qualified name:

- If you use fully qualified name then only declared class of this package will be accessible.
- No need to import but use fully qualified name everytime when you are accessing the class of interface.
- It is used when two packages have same class name like java.util and java.sql packages contains Date class.
- Example:

```
//save by Student1.java
package pack;
public class Student1
{
        Public void msg() {System.out.println("Whatsapp me");}
```

```
}
//save bu Student2.java
package mypack;
class Student2
{
	public static void main(String args[])
	{
		pack.Student1 obj = new pack.Student1(); //using fully qualified name
		obj.msg();
	}
}
```

Output: Whatsapp

Access rules for packages

The class defined in the package are affected by access control.
There are 4 categories of visibility for class members.

- Sublcass in the same package
- Non-subclass in the same package
- Subclass in different packages
- Classes that are not in same package or in the subclass

There are four access specifiers: private, public, protected and default or no specifier

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Location ----- Access Specifier:	**Private**	**Default**	**Protected**	**Public**
Same class	Y	Y	Y	Y
Same package – Different Class	N	Y	Y	Y
Same package – Subclass	N	Y	Y	Y
Different package – Subclass	N	N	Y	Y
Different package – Different Class	N	N	N	Y

Example1: private

```
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
   }
}
```

Example2: default

```
//save by A.java
package pack;
class A{
  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();//Compile Time Error
           obj.msg();//Compile Time Error
          }
        }
```

Example3: protected

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

   class B extends A{
     public static void main(String args[]){
```

```
        B obj = new B();
        obj.msg();
       }
      }
```

Output: Hello

Example4:public

```
//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

Extra:

Super Keyword in Java

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

1**) super is used to refer immediate parent class instance variable.**

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
        String color="white";
}
class Dog extends Animal{
        String color="black";
```

```
        void printColor(){
            System.out.println(color);//prints color of Dog class
            System.out.println(super.color);//prints color of Animal class
        }
    }
    class TestSuper1{
        public static void main(String args[]){
            Dog d=new Dog();
        d.printColor();
        }
    }
```

Output:

```
    black
    white
```

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
    class Animal{
    void eat(){System.out.println("eating...");}
    }
    class Dog extends Animal{
        void eat(){System.out.println("eating bread...");}
        void bark(){System.out.println("barking...");}
        void work(){
        super.eat();
        bark();
    }
    }
    class TestSuper2{
        public static void main(String args[]){
            Dog d=new Dog();
            d.work();
    }}
```

Output:

```
    eating...
    barking...
```

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
    class Animal{
        Animal(){System.out.println("animal is created");}
    }
    class Dog extends Animal{
        Dog(){
        super();
```

```
        System.out.println("dog is created");
    }
    }
    class TestSuper3{
        public static void main(String args[]){
        Dog d=new Dog();
    }}
```

Output:

```
        animal is created
        dog is created
```