Unit – 2: INHERITANCE, PACKAGES AND INTERFACES

UNIT II INHERITANCE, PACKAGES AND INTERFACES

Overloading Methods – Objects as Parameters – Returning Objects – Static, Nested and Inner Classes. Inheritance: Basics – Types of Inheritance – super keyword – Method Overriding – Dynamic Method Dispatch – Abstract Classes – final with Inheritance. Packages and Interfaces: Packages – Packages and Member Access – Importing Packages – Interfaces.

2.1: Overloading Methods

Method Overloading is a feature in Java that allows a class to have **more than one methods having same name**, but with **different signatures** (Each method must have different number of parameters or parameters having different types and orders).

Advantage:

- ✓ Method Overloading increases the readability of the program.
- ✓ Provides the flexibility to use similar method with different parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters. (Different number of parameters in argument list)

For example: This is a valid case of overloading

add(int, int)
add(int, int, int)

2. Data type of parameters. (Difference in data type of parameters)

For example:

add(int, int)
add(int, float)

3. Sequence of Data type of parameters.

For example:

add(int, float)
add(float, int)

Rules for Method Overloading:

1. First and important rule to overload a method in java is to change method signature.
2. Return type of method is never part of method signature, so only changing the return type of method does not amount to method overloading.

Example: To find the Minimum of given numbers:

```
public class OverloadingCalculation1
{
  public static void main(String[] args)
 {
    int a = 11;
    int b = 6;
    int c = 3;
    double x = 7.3;
    double y = 9.4;

    int result1 = minFunction(a, b, c);
     double result2 = minFunction(x, y);
    double result3 = minFunction(a, x);

    System.out.println("Minimum("+a+","+b+","+c+") = " + result1);
    System.out.println("Minimum("+x+","+y+") = " + result2);
    System.out.println("Minimum("+a+","+x+") = " + result3);
  }

  public static int minFunction(int n1, int n2, int n3)
  {
    int min;
    int temp = n1<n2? n1 : n2;
    min = n3 < temp? n3 : temp;
    return min;
  }

  public static double minFunction(double n1, double n2)
  {
   double min;
```

```
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
public static double minFunction(int n1, double n2)
  {
   double min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
}
```

This would produce the following result:

```
Minimum(11,6,3) = 3
Minimum(7.3,9.4) = 7.3
Minimum(11,7.3) = 7.3
```

Note:-

Method overloading is not possible by changing the return type of the method because of ambiguity that may arise while calling the method with same parameter list with different return type.

Example:

```
class Add
{
   static int sum(int a, int b)
  {
      return a+b;
  }
  static float sum(int a, int b)
```

```
 {
    return a+b;
 }
public static void main(String arg[])
{
   System.out.println(sum(10,20));
   System.out.println(sum(15,25));
}
}
```

Output:

Compile by: javac TestOverloading3.java

```
Add.java:7: error: method sum(int,int) is already defined in class Add
 static float sum(int a, int b)
 ^
1 error
```

Method Overloading and Type Promotion

Type Promotion: When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

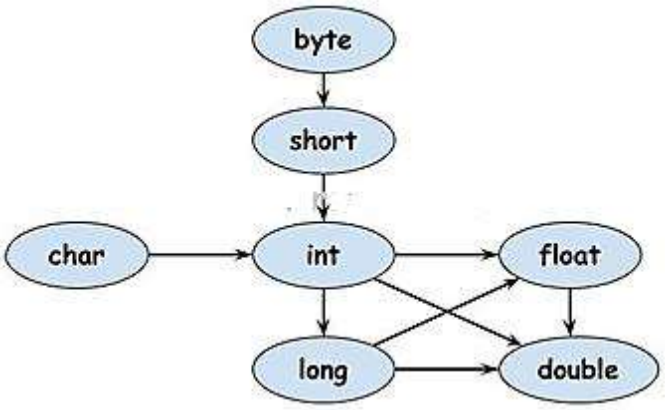
Type Promotion in Method Overloading:

One type is promoted to another implicitly if no matching datatype is found.

Type Promotion Table:

The **data type on the left side can be promoted to the any of the data type present in the right side of it.**

```
byte → short → int → long → double
short → int → long → float → double
int → long → float → double
float → double
long → float → double
char → int → long → float → double
```



Example: Method Overloading with Type Promotion:

```
class Overloading
{
     void sum(int a, float b)
    {
        System.out.println(a+b);
    }
     void sum(int a, int b, int c)
    {
        System.out.println(a+b+c);
    }

  public static void main(String args[])
{
         OverloadingCalculation1 obj=new OverloadingCalculation1();
         obj.sum(20,20);         //now second int literal will be promoted to float
         obj.sum(100,'A');       //Character literal will be promoted to float
         obj.sum(20,20,20);
  }
}
```

OUTPUT:

```
40.0
165.0
60
```

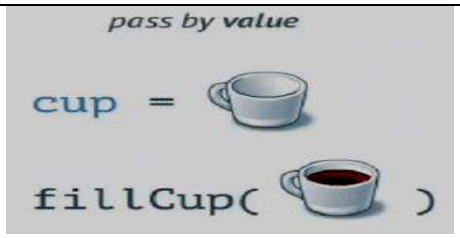
2.2: Objects as Parameters

Java is strictly pass-by-value. But the scenario may change when the parameter passed is of primitive type or reference type.

- If we pass a primitive type to a method, then it is called **pass-by-value** or call-by-value.
- If we pass an object to a method, then it is called **pass-by-reference** or call-by-reference.

Object as a parameter is a way to establish communication between two or more objects of the same class or different class as well.

Pass-by-value vs. Pass-by-reference:

Pass-by-value (Value as parameter)	Pass-by-reference (Object as parameter)
Only values are passes to the function parameters. So any modifications done in the formal parameter will not affect the value of actual parameter	Reference to the object is passed. So any modifications done through the object will affect the actual object.
Caller and Callee method will have two independent variables with same value.	Caller and Callee methods use the same reference for the object.
Callee method will not have any access to the actual parameter	Callee method will have the direct reference to the actual object
Requires more memory	Requires less memory
	

class CallByVal
{
void Increment(int count)
{
count=count+10;
}
}
public class CallByValueDemo {
public static void main(String arg[])
{
CallByVal ob1=new CallByVal();
int count=100;
System.out.println("Value of Count before method call = "+count);
	class CallByRef
	{
	int count=0;
	CallByRef(int c)
	{
	count=c;
	}
	static void Increment(CallByRef obj) {
	obj.count=obj.count+10;
	}
	
	public static void main(String arg[]) {
	CallByRef ob1=new CallByRef(10);
	System.out.println("Value of Count (Object 1) before

```
    ob1.Increment(count);
    System.out.println("Value of Count after
                        method call = "+count);
  }
}
```

OUTPUT:

```
Value of Count before method call = 100
Value of Count after method call = 100
```

```
                        method call = "+ob1.count);
    Increment(ob1);
    System.out.println("Value of Count (Object 1) after
                        method call = "+ob1.count);

  } }
```

OUTPUT:

```
Value of Count (Object 1) before method call = 10
Value of Count (Object 1) after method call = 20
```

Returning Objects:

In Java, a method can return any type of data. Return type may any primitive data type or class type (i.e. object). As a method takes objects as parameters, it can also return objects as return value.

Example:

```
class Add
{
  int num1,num2,sum;

  static Add calculateSum(Add a1,Add a2)
  {
    Add a3=new Add();
    a3.num1=a1.num1+a1.num2;
    a3.num2=a2.num1+a2.num2;
    a3.sum=a3.num1+a3.num2;
    return a3;
  }

  public static void main(String arg[])
  {
    Add ob1=new Add();
    ob1.num1=10;
    ob1.num2=15;

    Add ob2=new Add();
    ob2.num1=100;
    ob2.num2=150;

    Add ob3=calculateSum(ob1,ob2);
    System.out.println("Object 1 -> Sum = "+ob1.sum);
```

```
        System.out.println("Object 2 -> Sum = "+ob2.sum);
        System.out.println("Object 3 -> Sum = "+ob3.sum);
    }
}
```

OUTPUT:

```
Object 1 -> Sum = 0
Object 2 -> Sum = 0
Object 3 -> Sum = 275
```

2.3: STATIC, NESTED and INNER CLASSES

Definition:

An inner class is a class that is defined inside another class.
Inner classes let you make one class a member of another class. Just as classes have member variables and methods, a class can also have member classes.

Benefits:

1. Name control
2. Access control
3. Code becomes more readable and maintainable because it locally group related classes in one place.

Syntax: For declaring Inner classes

```
[modifier] class OuterClassName
{
      ---- Code -----
       [modifier] class InnerClassName
       {
       ---- Code ----
       }
}
```

- **Instantiating an Inner Class:**

Two Methods:

1. **Instantiating an Inner class from outside the outer class:**
   To instantiate an instance of an inner class, you must have an instance of the outer class.
   Syntax:
   OuterClass.InnerClass objectName=OuterObj.new InnerClass();

2. **Instantiating an Inner Class from Within Code in the Outer Class:**
   From inside the outer class instance code, use the inner class name in the normal way:

Syntax:

InnerClassName obj=new InnerClassName();

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested class **can access all the members (data members and methods) of outer class** including private.
2) Nested classes are used **to develop more readable and maintainable code**.
3) **Code Optimization**: It requires less code to write.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

1. Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer
{
 //code
    class Inner
    {
       //code
    }
}
```

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
1.  class TestMemberOuter1
2.  {
3.   private int data=30;
4.   class Inner
5.  {
6.    void msg()
7.  {
8.  System.out.println("data is "+data);
9.  }
10.  }
11.  public static void main(String args[])
12.  {
13.   TestMemberOuter1 obj=new TestMemberOuter1();
14.   TestMemberOuter1.Inner in=obj.new Inner();
15.   in.msg();
16.  }
17. }
```

Output:

data is 30

2. Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

```
1.  abstract class Person
2.  {
3.    abstract void eat();
4.  }
5.  class TestAnonymousInner
6.  {
7.   public static void main(String args[])
8.  {
9.    Person p=new Person()
10. {
11.   void eat()
12. {
13.     System.out.println("nice fruits");
14. }
15.   };
16.   p.eat();
17.  }
18. }
```

Output:

nice fruits

Java anonymous inner class example using interface

```
1.  interface Eatable
2.  {
3.   void eat();
```

```
4.  }
5.  class TestAnnonymousInner1
6.  {
7.   public static void main(String args[])
8.  {
9.   Eatable e=new Eatable()
10.  {
11.  public void eat(){System.out.println("nice fruits");
12.  }
13. };
14. e.eat();
15. }
16. }
```

Output:

```
nice fruits
```

3. Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
1.  public class localInner1
2.  {
3.   private int data=30;//instance variable
4.   void display()
5.  {
6.  int value=50;
7.    class Local
8.  {
9.     void msg()
10.      {
11.       System.out.println(data);
12.       System.out.println(value);
13.      }
14.  }
```

```
15.    Local l=new Local();
16.    l.msg();
17.   }
18.   public static void main(String args[])
19.  {
20.    localInner1 obj=new localInner1();
21.    obj.display();
22.   }
23.  }
```

Output:

```
30
50
```

Rules for Java Local Inner class

1. Local inner class cannot be invoked from outside the method.
2. Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.
3. Local variable can't be private, public or protected.

Properties:

1. Completely hidden from the outside world.
2. Cannot access the local variables of the method (in which they are defined), but the local variables has to be declared final to access.

4. Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
1.    class TestOuter1
2.    {
3.      static int data=30;
4.      static class Inner
5.    {
6.       void msg()
```

```
7.        {
8.          System.out.println("data is "+data);
9.        }
10.      }
11.      public static void main(String args[])
12.     {
13.      TestOuter1.Inner obj=new TestOuter1.Inner();
14.      obj.msg();
15.      }
16.     }
```

Output:

```
data is 30
```

Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
1.  class TestOuter2{
2.    static int data=30;
3.    static class Inner
4.  {
5.     static void msg()
6.      {
7.       System.out.println("data is "+data);
8.      }
9.    }
10.   public static void main(String args[])
11. {
12.   TestOuter2.Inner.msg();//no need to create the instance of static nested
      class
13.   }
14. }
```

Output:

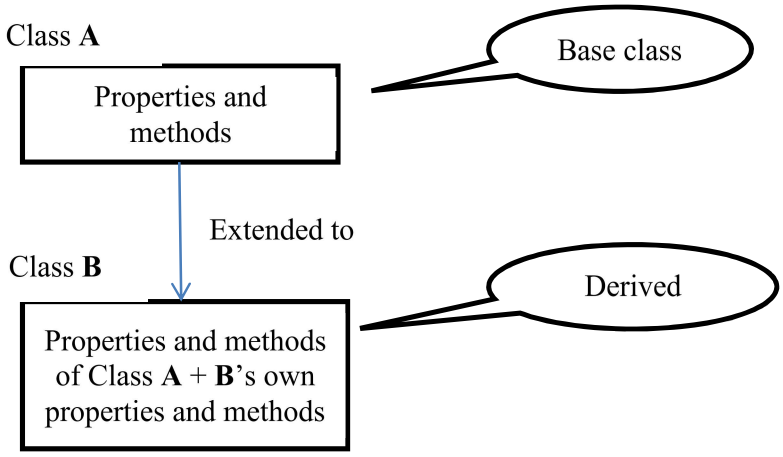
```
data is 30
```

2.4: Inheritance

Definition:

Inheritance is a process of deriving a new class from existing class, also called as "extending a class". When an existing class is extended, the new (inherited) class has all the properties and methods of the existing class and also possesses its own characteristics.

- ✓ The class whose property is being inherited by another class is called **"base class"** (or) **"parent class"** (or) **"super class"**.
- ✓ The class that inherits a particular property or a set of properties from the base class is called **"derived class"** (or) **"child class"** (or) **"sub class"**.



- ✓ Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

➢ **ADVANTAGES OF INHERITANCE:**

- **Reusability of Code:**
  - ✓ Inheritance is mainly used for code reusability (Code reusability means that we can add extra features to an existing class without modifying it).
- **Effort and Time Saving:**
  - ✓ The advantage of reusability saves the programmer time and effort. Since the main code written can be reused in various situations as needed.
- **Increased Reliability:**
  - ✓ The program with inheritance becomes more understandable and easily maintainable as the sub classes are created from the existing reliably working classes.

➢ **"extends" KEYWORD:**

- ✓ Inheriting a class means creating a new class as an extension of another class.
- ✓ The **extends** keyword is used to inherit a class from existing class.
- ✓ The general form of a **class** declaration that inherits a superclass is shown here:
- ✓ **Syntax:**

```
[access_specifier] class subclass_name extends superclass_name
{
        // body of class
}
```

Characteristics of Class Inheritance:

1. A class cannot be inherited from more than one base class. Java does not support the inheritance of multiple super classes into a single subclass.
2. Sub class can access only the non-private members of the super class.
3. Private data members of a super class are local only to that class. Therefore, they can't be accessed outside the super class, even sub classes can't access the private members.
4. Protected features in Java are visible to all subclasses as well as all other classes in the same package.

✓ **Example:**

```
class Vehicle
{
   String brand;
   String color;
}
class Car extends Vehicle
{
   int totalDoor;
}

class Bike extends Vehicle
{
}
```

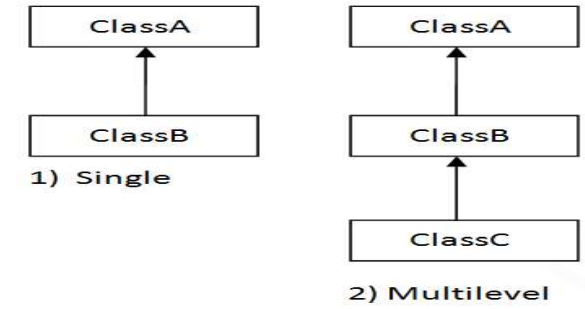
In the above example, Vehicle is the **super class** or base class that holds the common property of Car and Bike. Car and Bike is the **sub class** or derived class that inherits the property of class Vehicle **extends** is the keyword used to inherit a class.

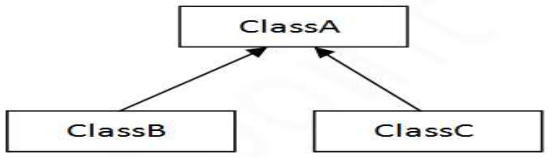
➢ TYPES OF INHERITACE:

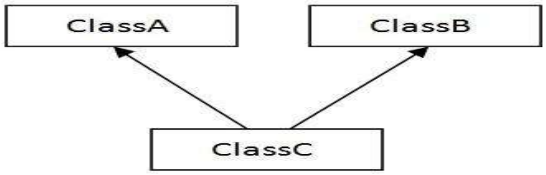
1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance

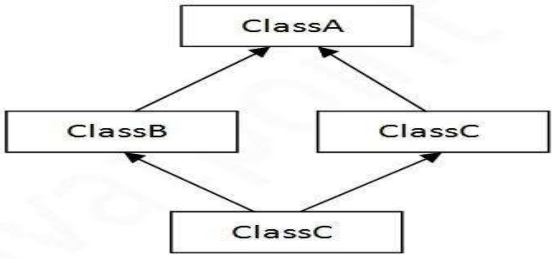
Note: The following inheritance types are not directly supported in Java.

4. Hierarchical Inheritance
5. Hybrid Inheritance









.

Single Inheritance
Class A
Class B
	public class A {
	…….
	}
	public class B **extends** A {
	……..
	}

Multi Level Inheritance Class A Class B Class C	public class A { ……..} public class B **extends** A {……..} public class C **extends** B {…….. }
Hierarchical Inheritance Class A Class B Class C	public class A { ……..} public class B **extends** A {……..} public class C **extends** A {…….. }

Multiple Inheritance
Class A
Class B
Class C
	public class A { ……..}
	public class B {……..}
	public class C **extends** A,B {
	……..
	} // Java does not support mutiple Inheritance

18

1. SINGLE INHERITANCE

The process of creating only one subclass from only one super class is known as **Single Inheritance.**

- ✓ Only two classes are involved in this inheritance.
- ✓ The subclass can access all the members of super class.

Example: Animal → Dog

```
1. class Animal
2. {
3.     void eat()
4.    {
5.          System.out.println("eating...");
6.     }
7. }
8. class Dog extends Animal
9. {
10.   void bark()
11.   {
12.        System.out.println("barking...");
13.    }
14.}
15.class TestInheritance
16.{
17.    public static void main(String args[])
18.    {
19.        Dog d=new Dog();
20.        d.bark();
21.        d.eat();
22.    }
23.}
```

Output:

```
$java TestInheritance
barking...
eating...
```

2. MULTILEVEL INHERITANCE:

- ✓ The process of creating a new sub class from an already inherited sub class is known as **Multilevel Inheritance**.
- ✓ Multiple classes are involved in inheritance, but one class extends only one.
- ✓ The lowermost subclass can make use of all its super classes' members.
- ✓ Multilevel inheritance is an indirect way of implementing multiple inheritance.
- ✓ **Example: Animal → Dog → BabyDog**

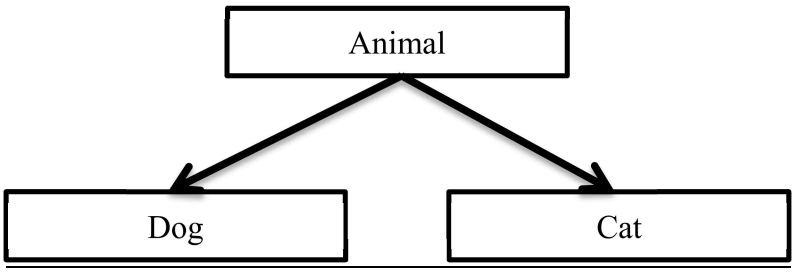
```
1.  class Animal
2.  {
3.    void eat()
4.    {
5.        System.out.println("eating...");
6.    }
7.  }
8.  class Dog extends Animal
9.  {
10.     void bark()
11.    {
12.       System.out.println("barking...");
13.    }
14.  }
15. class BabyDog extends Dog
16. {
17.    void weep()
18.   {
19.       System.out.println("weeping...");
20.   }
21. }
22. class TestInheritance2
23. {
24.   public static void main(String args[])  {
25.       BabyDog d=new BabyDog();
26.       d.weep();
27.       d.bark();
28.       d.eat();
29.   }
30. }
```

Output:

```
$java TestInheritance2
weeping...
barking...
eating..
```

3. HIERARCHICAL INHERITANCE

✓ The process of creating more than one sub classes from one super class is called **Hierarchical Inheritance.**



✓ **Example:**

```
1.  class Animal
2.  {
3.  void eat()
4.  {
5.  System.out.println("eating...");
6.  }
7.  }
8.  class Dog extends Animal
9.  {
10. void bark()
11. {
12. System.out.println("barking...");
13. }
14. }
15. class Cat extends Animal
16. {
17. void meow()
18. {
19. System.out.println("meowing...");
20. }
```

```
21.     }
22.     class TestInheritance3
23.     {
24.     public static void main(String args[])
25.     {
26.     Cat c=new Cat();
27.     c.meow();
28.     c.eat();
29.     //c.bark();//C.T.Error
30.     }
31.     }
```

Output:

```
meowing...
eating...
```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritances is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A  {
    void msg()
   {
       System.out.println("Hello");
   }
}
class B  {
   void msg()
   {
       System.out.println("Welcome");
   }
}
class C extends A,B   // this is multiple inheritance which is ERROR
  {
       Public Static void main(String args[])
       {
          C obj=new C();
          obj.msg();//Now which msg() method would be i
nvoked?
       }
}
```

Output

```
Compile Time Error
```

Multiple Inheritance using Interface

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

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

Output:

```
Hello
Welcome
```

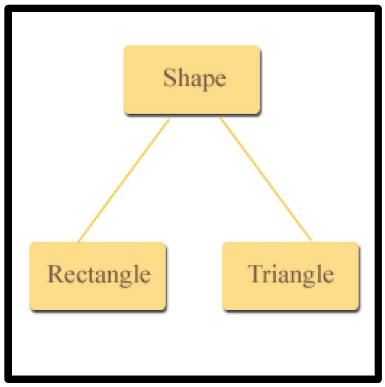
2.4.1: PROTECTED MEMBER

The private members of a class cannot be directly accessed outside the class. Only methods of that class can access the private members directly. However, sometimes it may be necessary for a subclass to access a private member of a superclass. If you make a private member public, then anyone can access that member. So, if a member of a superclass needs to be (directly) accessed in a subclass then you must declare that member **protected**.

Following table describes the difference

Modifier	Class	Package	subclass	World
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No

Following program illustrates how the methods of a subclass can directly access a protected member of the superclass.



Consider two kinds of shapes: **rectangles and triangles**. These two shapes have certain common properties height and a width (or base).

This could be represented in the world of classes with a **class Shapes** from which we would derive the two other ones : Rectangle and Triangle

Program : (Shape.java)

```
public class Shape
{
    protected double height;  // To hold height.
    protected double width;  //To hold width or base
```

```
    public void setValues(double height, double width)
   {
         this.height = height;
         this.width = width;
   }
}
```

Program : (Rectangle.java)

```
public class Rectangle extends Shape
{
     public double getArea()
    {
       return height * width; //accessing protected members
    }
}
```

Program : (Triangle.java)

```
public class Triangle extends Shape
{
            public double getArea()
           {
                 return height * width / 2; //accessing protected members
           }
}
```

Program : (TestProgram.java)

```
public class TestProgram
{
       public static void main(String[] args)
      {
           //Create object of Rectangle.
            Rectangle rectangle = new Rectangle();

          //Create object of Triangle.
          Triangle triangle = new Triangle();

          //Set values in rectangle object
         rectangle.setValues(5,4);
```

```
        //Set values in trianlge object
        triangle.setValues(5,10);

       // Display the area of rectangle.
        System.out.println("Area of rectangle : " +
        rectangle.getArea());

     // Display the area of triangle.
      System.out.println("Area of triangle : " +
      triangle.getArea());
    }
}
```

Output :

Area of rectangle : 20.0
Area of triangle : 25.0

2.4.2: CONSTRUCTORS IN SUB – CLASSES

In Java, constructor of base class with no argument gets automatically called in derived class constructor.

When Constructors are Called?

Constructors are called in order of derivation, from superclass to subclass. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Example:

```
class A
{
   A()
   {     System.out.println(" Inside A's Constructor");    }
}

class B extends  A
{
   B()
```

```
    {    System.out.println(" Inside B's Constructor");    }
}
class C  extends  B
{
    C()
    {    System.out.println(" Inside C's Constructor");    }
}
class CallingCons
{
    public static void main(String args[])
   {
        C objC=new C();
    }
}
```

Output:

```
Inside A's Constructor
Inside B's Constructor
Inside C's Constructor
```

Program Explanation:

In the above program, we have created three classes A, B and C using multilevel inheritance concept. Here, constructors of the three classes are called in the order of derivation. Since **super()** must be the first statement executed in subclass's constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. When inheriting from another class, super() has to be called first in the constructor. If not, the compiler will insert that call. This is why super constructor is also invoked when a Sub object is created.

After compiler inserts the super constructor, the sub class constructor looks like the following:

```
B()
{
        super();
        System.out.println("Inside B's Constructor");
}
C()
{
        super();
```

```
        System.out.println("Inside C's Constructor");
}
```

2.5: "super" keyword

- ✓ Super is a special keyword that directs the compiler to invoke the superclass members. It is used to refer to the parent class of the class in which the keyword is used.
- ✓ **super keyword is used for the following three purposes**:
  1. To invoke superclass constructor.
  2. To invoke superclass members variables.
  3. To invoke superclass methods.

1. Invoking a superclass constructor:

- ✓ **super** as a standalone statement(ie. super()) represents a call to a constructor of the superclass.
- ✓ A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

```
super();
or
super(parameter-list);
```

- ✓ Here, parameter-list specifies any parameters needed by the constructor in the superclass.
- ✓ **super( )** must always be the first statement executed inside a subclass constructor.
- ✓ The compiler implicitly calls the base class's no-parameter constructor or default constructor.
- ✓ If the superclass has parameterized constructor and the subclass constructor does not call superclass constructor explicitly, then the Java compiler reports an error.

2. Invoking a superclass members (variables and methods):

(i) Accessing the instance member variables of the superclass:

Syntax:

```
super.membervariable;
```

(ii) Accessing the methos of the superclass:

Syntax:

super.methodName();

This call is particularly necessary while calling a method of the super class that is overridden in the subclass.

✓ If a parent class contains a finalize() method, it must be called explicitly by the derived class's finalize() method.

super.finalize();

Example:

```
class A       // super class
{
  int i;
  A(String str)      //superclass constructor
  {
    System.out.println(" Welcome to "+str);
  }
  void show()       //superclass method
  {
    System.out.println(" Thank You!");
  }
}
class B extends A
{
  int i;     // hides the superclass variable 'i'.
  B(int a, int b)     // subclass constructor
  {
    super("Java Programming");     // invoking superclass constructor
    super.i=a;     //accessing superclass member variable
    i=b;
  }
  // Mehtod overriding
  @Override
  void show()
  {
    System.out.println(" i in superclass : "+super.i);
    System.out.println(" i in subclass : "+i);
    super.show();      // invoking superclass method
```

```
    }
}
public class UseSuper {
   public static void main(String[] args) {
      B objB=new B(1,2);     // subclass object construction
      objB.show();       // call to  subclass method show()
   }
}
```

Output:

```
Welcome to Java Programming
        i in superclass : 1
        i in subclass : 2
        Thank You!
```

Program Explanation:

In the above program, we have created the base class named **A** that contains a instance variable **'i'** and a method **show().** Class A contains a parameterized constructor that receives string as a parameter and prints that string. Class **B** is a subclass of **A** which contains a instance variable 'i' (hides the superclass variable 'i') and overrides the superclass method **show().** The subclass defines the constructor with two parameters a **and b.** The subclass constructor invokes the superclass constructor **super(String)** by passing the string "Java Programming" and assigns the value **a** to the superclass variable**(super.i=a)** and **b** to the subclass variable. The show() method of subclass prints the values of 'i' form both superclass and subclass & invokes the superclass method as **super.show().**

In the main class, object for subclass **B** is created and the object is used to invoke **show()** method of subclass.

2.6: METHOD OVERRIDING

The process of a subclass redefining a method contained in the superclass (with the same method signature) is called Method Overriding.

- ✓ When a method in a subclass has the same name and type signature as a method in its superclass, then the method in subclass is said to override a method in the superclass.

Example:

```
class Bank
{
int getRateOfInterest()// super class method
{
return 0;
}
}
class Axis extends Bank// subclass of bank
{
int getRateOfInterest()// overriding the superclass method
{
return 6;
}
}
class ICICI extends Bank// subclass of Bank
{
        int getRateOfInterest()// overriding the superclass method
{
return 15;
}
}
// Mainclass
class BankTest
{
public static void main(String[] a)
{
Axis a=new Axis();
ICICI i=new ICICI();
// following method call invokes the overridden method of subclass AXIS
System.out.println("AXIS: Rate of Interest = "+a.getRateOfInterest());

// following method call invokes the overridden method of subclass ICICI
System.out.println("ICICI: Rate of Interest = "+i.getRateOfInterest());
}
}
```

Output:

Z:\> java BankTest

AXIS: Rate of Interest = 6
ICICI: Rate of Interest = 15

- **RULES FOR METHOD OVERRIDING:**

  - ✓ The method signature must be same for all overridden methods.
  - ✓ Instance methods can be overridden only if they are inherited by the subclass.
  - ✓ A method declared final cannot be overridden.
  - ✓ A method declared static cannot be overridden but can be re-declared.
  - ✓ If a method cannot be inherited, then it cannot be overridden.
  - ✓ Constructors cannot be overridden.

- **ADVANTAGE OF JAVA METHOD OVERRIDING**

  - ✓ Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
  - ✓ Method Overriding is used for Runtime Polymorphism

2.7: DYNAMIC METHOD DISPATCH

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Example that illustrate dynamic method dispatch:

```
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}

class B extends A {
 //override callme()
void callme() {
System.out.println("Inside B's callme method");
} }
```

```
class C extends A
{
 //override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}

class Dispatch
{
public static void main(String args[])
{
A a=new A();            //object of type A
B b=new B();           //object of type B
C c=new C();           //object of type C
A r;// obtain a reference of type A

r = a;      // r refers to an A object    // dynamic method dispatch
r.callme();// calls A's version of callme()

r = b;// r refers to an B object
r.callme();// calls B's version of callme()

r = c;// r refers to an C object
r.callme();// calls C's version of callme()
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

- **DIFFERENCE BETWEEN METHOD OVERLOADING AND METHOD OVERRIDING IN JAVA:**

	Method Overloading	**Method Overriding**
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to "add" or "extend" more to method's behavior.	Method Overriding is to "Change" existing behavior of method.
Overloading and Overriding is a kind of polymorphism. Polymorphism means "one name, many forms".		
Polymorphism	It is a **compile time polymorphism**.	It is a **run time polymorphism**.
Inheritance	It may or may not need **inheritance** in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have **different signature**.	In Method Overriding, methods must have **same signature**.
Relationship of Methods	In Method Overloading, relationship is there between methods of same class.	In Method Overriding, relationship is there between methods of super class and sub class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

Example	Class Add
	{
	int sum(int a, int b)
	{
	return a + b;
	}
	int sum(int a)
	{
	return a + 10;
	}
	}
		Class A // Super Class
		{
		void display(int num)
		{
		print num ;
		}
		}
		//Class B inherits Class A
		Class B //Sub Class
		{
		void display(int num)
		{
		print num ;
		}
		}

2.8: ABSTRACT CLASSES

- **Abstraction:**

Abstraction is a process of hiding the implementation details and showing only the essential features to the user.

- ✓ For example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- ✓ Abstraction lets you focus on what the object does instead of how it does it.

- **Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

- **Abstract Classes:**

A class that is declared as abstract is known as **abstract class**. Abstract classes cannot be instantiated, but they can be subclassed.

✓ **Syntax to declare the abstract class:**

```
abstract class <class_name>
{
        Member variables;
        Concrete methods { }
        Abstract methods();
}
```

✓ Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

Properties of abstract class:

- abstract keyword is used to make a class abstract.
- Abstract class can't be instantiated.
- If a class has abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- Abstract classes can have both concrete methods and abstract methods.
- The subclass of abstract class must implement all the abstract methods unless the subclass is also an abstract class.
- A constructor of an abstract class can be defined and can be invoked by the subclasses.
- We can run abstract class like any other class if it has main() method.

Example:

```
abstract class GraphicObject {
   int x, y;
   ...
   void moveTo(int newX, int newY) {
      ...
   }
   abstract void draw();
   abstract void resize();
}
```

- **Abstract Methods:**

A method that is declared as abstract and does not have implementation is known as **abstract method.** It acts as placeholder methods that are implemented in the subclasses.

✓ **Syntax to declare a abstract method:**

```
abstract class classname
{

abstract return_type <method_name>(parameter_list);//no braces{}
        // no implementation required
                ……..
}
```

✓Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Properties of abstract methods:

- The abstract keyword is also used to declare a method as abstract.
- An abstract method consists of a method signature, but no method body.
- If a class includes abstract methods, the class itself must be declared abstract.
- Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
public abstract class Employee {
  private String name;
  private String address;
  private int number;
  public abstract double computePay();
  //Remainder of class definition
}
```

- Any child class must either override the abstract method or declare itself abstract.

Write a Java program to create an abstract class named Shape that contains 2 integers and an empty method named PrintArea(). Provide 3 classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contain only the method PrintArea() that prints the area of the given shape.

```
abstract class shape
{
    int x, y;
    abstract void printArea();
}
```

```
class Rectangle extends shape
{
void printArea()
{
      System.out.println("Area of Rectangle is " + x * y);
}
}
class Triangle extends shape
{
void printArea()
{
System.out.println("Area of Triangle is " + (x * y) / 2);
}
}
class Circle extends shape
{
void printArea()
{
        System.out.println("Area of Circle is " + (22 * x * x) / 7);
}
}
class abs
{
public static void main(String[] args)
{
Rectangle r = new Rectangle();
r.x = 10;
r.y = 20;
r.printArea();

System.out.println("-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -");

Triangle t = new Triangle();
t.x = 30;
t.y = 35;
t.printArea();

System.out.println("-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -");
```

```
Circle c = new Circle();
c.x = 2;
c.printArea();

System.out.println("-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -");
}
}
```

Output:

```
D:\>javac abs.java
D:\>java abs
Area of Rectangle is 200
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
Area of Triangle is 525
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
Area of Circle is 12
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
```

2.9: final WITH INHERITANCE

What is final keyword in Java?

Final is a keyword or reserved word in java used for restricting some functionality. It can be applied to member variables, methods, class and local variables in Java.

- ✓ **final** keyword has three uses:
  1. For declaring variable **– to create a named constant.** A final variable cannot be changed once it is initialized.
  2. For declaring the methods **– to prevent method overriding**. A final method cannot be overridden by subclasses.
  3. For declaring the class – **to prevent a class from inheritance.** A final class cannot be inherited.

1. Final Variable:

Any variable either member variable or local variable (declared inside method or block) modified by final keyword is called final variable.

✓The final variables are equivalent to **const qualifier** in C++ and **#define** directive in C.

✓Syntax:

✓final data_type variable_name = value;

✓Example:

final int MAXMARKS=100;
final int PI=3.14;

✓The final variable can be assigned only once.

✓The value of the final variable will not be changed during the execution of the program. If an attempt is made to alter the final variable value, the java compiler will throw an error message.

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1.  class Bike
2.  {
3.       final int speedlimit=90;//final variable
4.       void run( )
5.      {
6.          speedlimit=400;
7.      }
8.      public static void main(String args[])
9.     {
10.         Bike obj=new  Bike();
11.         obj.run();
12.    }
13.}
```

Output: Compile Time Error

NOTE: Final variables are by default read-only.

2. Final Methods:

- ✓ Final keyword in java can also be applied to methods.
- ✓ **A java method with final keyword is called final method and it cannot be overridden in sub-class.**
- ✓ If a method is defined with final keyword, it cannot be overridden in the subclass and its behaviour should remain constant in sub-classes.
- ✓ **Syntax:**

```
final return_type function_name(parameter_list)
{
// method body
}
```

✓ **Example of final method in Java:**

```
1.  class Bike
2.  {
3.     final void run()
4.    {
5.          System.out.println("running");
6.    }
7.  }
8.  class Honda extends Bike
9.  {
10.   void run()
11.  {
12.       System.out.println("running safely with 100kmph");
13.  }
14.  public static void main(String args[])
15. {
16.       Honda honda= new Honda();
17.       honda.run();
18.  }
19.}
```

Output:

```
D:\>javac Honda.java
Honda.java:9: error: run() in Honda cannot override run() in Bike
        void run()
             ^
  overridden method is final
  1  error
```

3. Final Classes:

✓ **Java class with final modifier is called final class in Java and they cannot be sub-classed or inherited.**

✓ **Syntax:**

```
final class class_name
{
    // body of the class
}
```

✓ Several classes in Java are final e.g. String, Integer and other wrapper classes.

✓ **Example of final class in java:**

```
1. final class Bike
2. {
3. }
4. class Honda1 extends Bike
5. {
6.      void run()
7.     {
8.        System.out.println("running safely with 100kmph");
9.     }
10.    public static void main(String args[])
11.   {
12.       Honda1 honda= new Honda1();
13.       honda.run();
14.   }
15.   }
```

Output:

```
D:\>javac Honda.java
Honda.java:4: error: cannot inherit from final Bike class Honda extends Bike
                                ^
1 error
```

Points to Remember:

1) A constructor cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an interface are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6) A final class cannot be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.
9) final, finally and finalize are three different terms. finally is used in exception handling and
10) finalize is a method that is called by JVM during garbage collection.

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

Advantage of Package:

- 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

2.10.1: CREATING USER DEFINED PACKAGES:

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.

2.10.2: ACCESSING A PACKAGE (using "import" keyword):

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

Syntax:

```
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
```

2.10.3: 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.

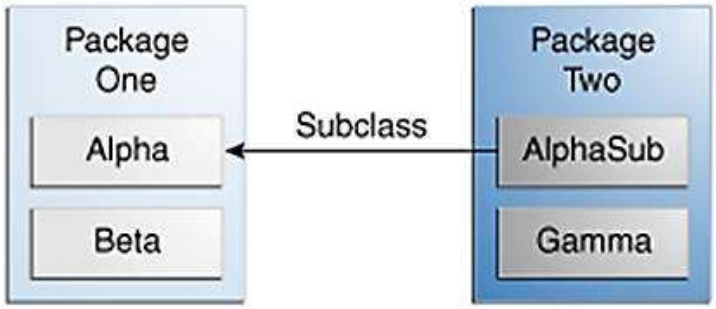


Figure: Classes and Packages of the Example Used to Illustrate Access Levels

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.

- **Why use Interface?**

  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.

➢ **An interface is similar to a class in the following ways:**

- 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.

➢ **Defining Interfaces:**

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

Syntax to define interface:

```
[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.

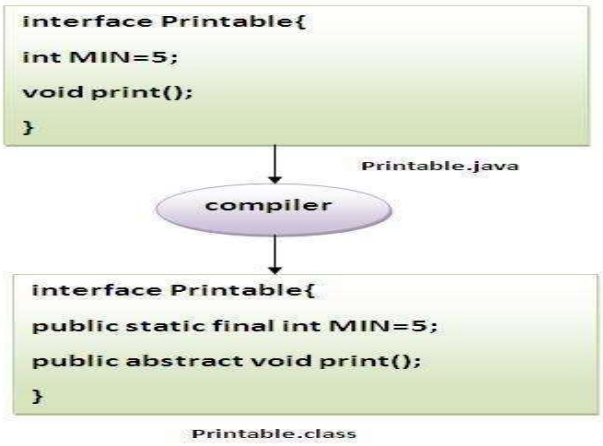
Name: name of an interface can be any valid java identifier.

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.

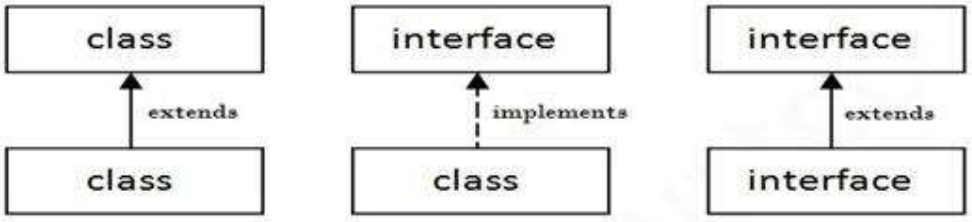
Note: The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

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



➢ 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**.



➢ 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 sample
X = 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.**

- **Extending Interfaces:**
  - ✓ 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 :

```
1.  interface Bank
2.  {
3.      float rateOfInterest();
4.  }
5.  class SBI implements Bank
6.  {
7.      public float rateOfInterest()
8.      {
9.          return 9.15f;
10.     }
11. }
12. class PNB implements Bank
13. {
14.     public float rateOfInterest()
15.     {
16.         return 9.7f;
17.     }
18. }
19. class TestInterface2
20. {
21.     public static void main(String[] args)
22.     {
23.         Bank b=new SBI();
24.         System.out.println("ROI: "+b.rateOfInterest());
25.     }
26. }
```

Output:

```
ROI: 9.15
```