Inheritance

Inheritance is a mechanism in which one object acquires all the properties and behaviors of parent object.

In other words, you can create new classes from existing classes, then you can reuse methods and variables of parent class, and you can add new methods and variables also.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends keyword

The extends keyword indicates that you are making a new class that derives from an existing class.

```
class Subclass extends Superclass{
        //methods and variables
   }
```

where:

```
class Superclass{
        .....
    }
```

Example 1:

```
class Employee{
   int id;
   String  name;
   float salary=500; }

class Programmer extends Employee{
   int bonus=200;

   public static void main(String args[]){
      Programmer p=new Programmer();
      p.name="Ali";
      p.id= 123;
      System.out.print("Programmer Information :"+p.name + "  "+ p.salary);
      System.out.println("  "+p.id+ " "+ p.bonus);}
 }
```

Output:

Example2:

```
class Calculation{
     int z;

     public void add(int x,  int y){
        z = x+y;
       System.out.println("The sum of the given numbers:"+z);}
```

```
public void Sub(int x,  int y){
    z = x-y;
    System.out.println("The difference between the given numbers:"+z);}
}

public class MyCalculation extends Calculation{
public void mult(int x,  int y){
    z = x*y;
    System.out.println("The product of the given numbers:"+z);}

 public static void main(String args[]){
      int a = 24, b = 6;
      MyCalculation d=new MyCalculation();
      d.add(a, b);
      d.Sub(a, b);
      d.mult(a, b);}
}
```

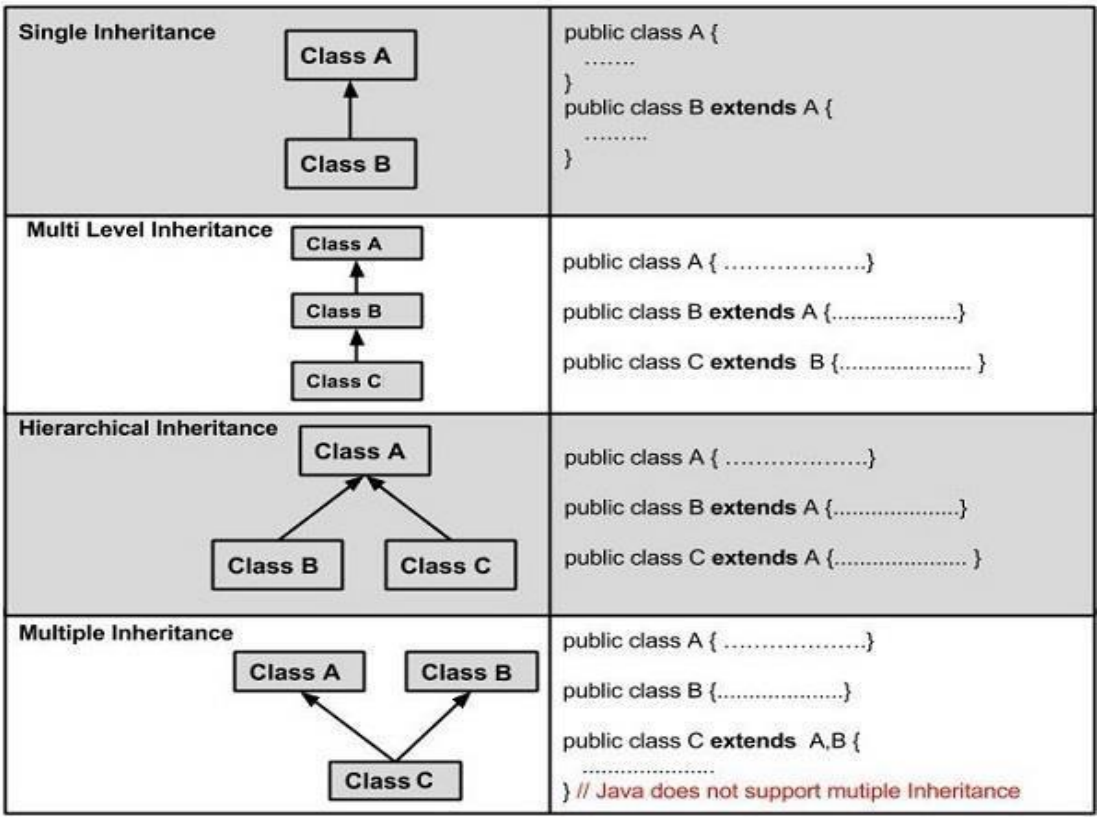
Output:

Notes:

- A subclass inherits **all** the members (variables, methods) from its superclass.
- **Constructors are not members**, so they are not inherited by subclasses, but it can be invoked from the subclass.
- Members declared **private** are not inherited at all.

Types of inheritance

There are various types of inheritance as demonstrated below.



Q. Why multiple inheritance is not supported in java?

Ans.: **To reduce the complexity and simplify the language**. For Example, consider 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.

Example:

```
class A{
   void printMsg(){
       System.out.println("Hello");}
 }

class B{
   void printMsg(){
       System.out.println("Welcome");}
 }

 class C extends A,B{
   Public Static void main(String args[]){
      C obj=new C();
      obj.printMsg();}            //Now which msg() method would be invoked?
   }
```

super keyword

In the following usage of super keyword:

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

Example without super keyword

```
 class A{
    int x=10;}

 class B extends A{
    int x=50;
    void display(){
      System.out.println(x);     // print x of B
  }

Public class Example{
   public static void main(String args[]){
       B b=new B();
       b.display();}
}
```

Output:

Example with super keyword

```
class A{
    int x=10;}

 class B extends A{
    int x=50;

  void display(){
     System.out.println(super.x);}// print x  of A

Public class Example{
     public static void main(String args[]){
```

Output:

2) super is used **to invoke parent class constructor**.

Example:

```
class A{
    A(){
       System.out.println("In class A");}
}

class B extends A{
    B(){
      super();   //will invoke parent class constructor
       System.out.println("In class B");}
}

Public class Example{
   public static void main(String args[]){
      B b=new B();}
}
```

Output:

Notes:

- In constructor super() must be the **first** statement.
- Always the **default constructor(if found)** for the superclass is executed **(before the default constructor for subclass)** when the object create from the subclass.

```
class  A{
   A(){
      System.out.println("In class A");}
}

class B extends A{
   int x;
   B(int x){
      this.x=x;
      System.out.println(x);}
}

Public class Example{
   public static void main(String args[]){
     B b=new B(50);}
}
```

Q: Show the output of the following programs:

1.

```
class  A{
    A(int x){
       System.out.println(x+ "In class A");}
 }

 class B extends A{
    int x;
    B(int x){
       this.x=x;
       System.out.println(x);}
}

Public class Example{
    public static void main(String args[]){
      B b=new B(50);} }
```

2.

```
class  A{
   int  x=10;
    A(){
       System.out.println(x + "  From  class A");}
 }
 class B extends A{
    int  x=50;
    B(){
        System.out.println(x+ "  From class B );}
}

Public class Example{
    public static void main(String args[]){
      B b=new B();
      System.out.println(b.x + " is winner");} }
```

3) super can be used to invoke **parent class member**, It should be used in case subclass contains the same member as parent class as shown below:

```
super.variable
super.method();
```

```
class A{
   void message(){
     System.out.println("welcome");}
 }

class B extends A{
    void message(){
        System.out.println("welcome to java");}

    void display(){
       message();          // current class message() method because priority is given to local.
       super.message(); //will invoke parent class message() method
     }

   public static void main(String args[]){
       B  s=new B();
       s.display();}
 }
```

Output:

Q: Why default constructor is required(explicitly) in a parent class if it has an Parameterized constructor ?

Ans.:

The following codes will be an error.

```
class A {
  A(int x){
}

class B extends A{
}

public static void main (String args[]){
  B b = new B();
}
```

```
class A {
  A(int x){
}

class B extends A{
    super();
}

public static void main (String args[]){
  B b = new B();
}
```

There are a few things to be noted when using constructors and how you should declare them in your base class and subclass.

- If you explicitly define constructors in any class(base class/subclass), the Java compiler will not create any constructor for you in that respective class.
- If you don't explicitly define constructors in any class(base class/subclass), the Java compiler will create a no-argument **(default)** constructor for you in that respective class.
- If your class is a subclass inheriting from a super class and you don't not explicitly define constructors in that subclass, not only will a no-argument constructor be created for you by the compiler, but it will also implicitly call the no-argument constructor from the super class.
- If you do not explicitly type super(), (or super(parameters)), the compiler will put in the super() for you in your code.
- If super() is being called (explicitly or implicitly by the compiler) , the compiler will expect your superclass to have a constructor without parameters. If it does not find constructor in your superclass without parameters but find parameterized constructor , it will give you a **compiler error.**
- Similarly if super(parameters) is called, the compiler will expect your superclass to have a constructor with parameters(number and type of parameters should match). If it does not find such a constructor in your superclass, it will give you a compiler error. (Super(parameters) can never be called implicitly by the compiler. It has to be explicitly put in your code if one is required.

overriding

If subclass has the same method(same name and same parameter) as declared in the parent class, it is known as method overriding.

Example without method overriding

```
class A{
  void display(){
    System.out.println("In class A");}
}

class B extends A{
  public static void main(String args[]){
    B b = new B();
    b.display();}
}
```

Example with method overriding

```
class  A{
  void display(){
      System.out.println("In class A");}
}

class B extends A{
  void display(){
    System.out.println("In class B "); }

  public static void main(String args[]){
    A a = new A();
    B b = new B();
```

Difference Between Method Overloading and Method Overriding

	Method Overloading	Method Overriding
Definition	Method Overloading means more than one method shares the same name in the same class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Inheritance	It may or may not need **inheritance** in Method Overloading.	It always requires inheritance in Method Overriding.
Relationship of Methods	relationship is there between methods of same class.	relationship is there between methods of super class and sub class(mean different class).
No. of Classes	does not require more than one class for overloading.	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) {
		System.out.println(num); }
		}
		
		//Class B inherits Class A
		Class B {
		void display(int num) {
		System.out.println(num);}
		}

Notes:

- Instance methods can be overridden only if they are inherited by the subclass.
- **Constructors** cannot be overridden.
- 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.
- When **super** keyword is used for method in superclass, this method can be overridden.

Example:

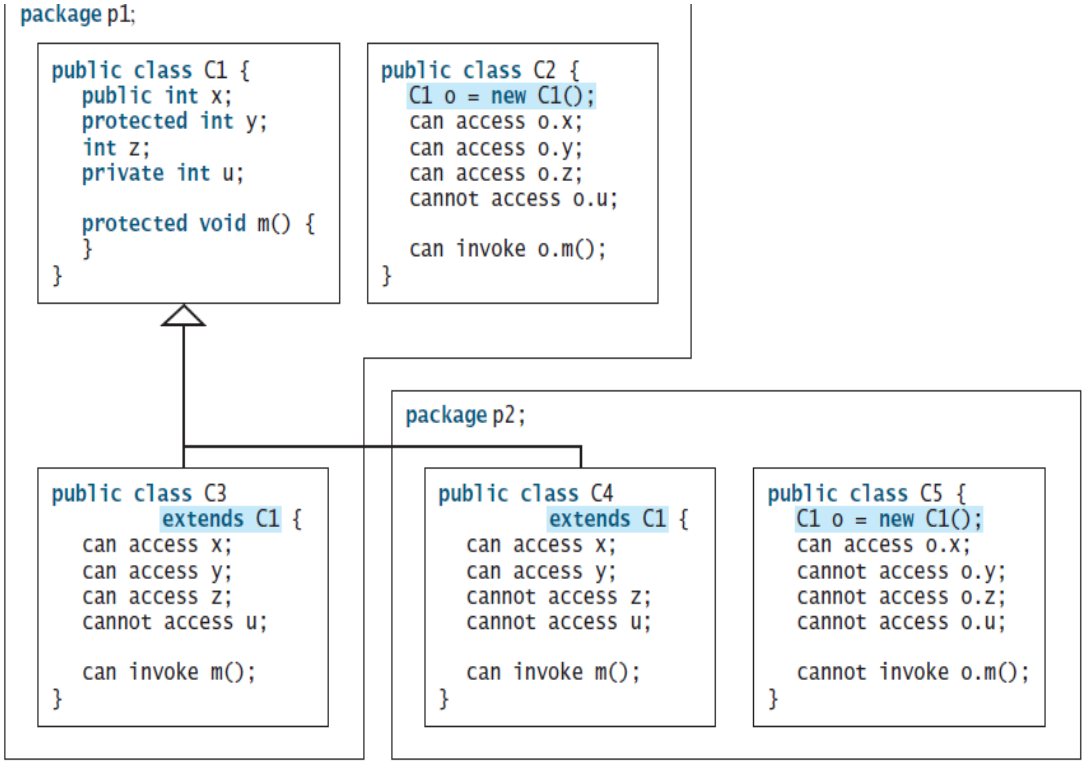
```
class A{
    public void display(){ System.out.println("in  class A");}
 }


 class B extends A{
    public void display(){
       super.display();
       System.out.println("In class B ");}
    }

public class Example{
   public static void main(String args[]){
      B b = new B();
      b.display(); }
}
```

Access Modifiers

Modifiers are keywords that you add to those definitions to change their meanings. There are two categories of modifiers:

- **Access Modifiers:**
  - **default**: visible to the package(no modifiers are needed).
  - **private**: visible to the class only.
  - **public**: visible to the world.
  - **protected:** visible to the package and all subclasses..

- **Non-access Modifiers:**
  - **static** modifier for creating class methods and variables.
  - **final** modifier for finalizing the implementations of classes, methods, and variables.
  - **abstrac**t modifier for creating abstract classes and methods.



Default Access Modifier **(No Keyword**) means we do not explicitly declare an access modifier for a class, method, etc.

Example1:

```
String name = "Ali";
boolean processOrder(){
   return true;}
```

Example2:

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

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

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

Private access modifier

- Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
- Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class, in this case **getter and setter methods** are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example
```
public classExample {
    private int age;
     public int getAge() {
       return age;}

     public void setAge(int ag) {
        age = ag;}
 }
```

Note: the age variable of the example class is private, so there's no way for other classes to retrieve or set its value directly.

```
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
  }
 }
```

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

Example:

```
class A{
  private A(){}  //private constructor
  void msg(){System.out.println("Hello java");}
}
```

```
public class Simple{
  public static void main(String args[]){
    A obj=new A();              //Compile Time Error
  }
}
```

Public access modifier

- A class, method, constructor, interface, etc. declared **public** can be accessed from any other class. Therefore, variables, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.
- If the public class and we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.
- The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example

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

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

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter to run the class.

Protected access modifier

- Variables, methods, and constructors, which are declared **protected** in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.
- The protected access modifier cannot be applied to class and interfaces. Methods, variables can be declared protected but methods and variables in a interface cannot be declared protected.
- Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.
- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example

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

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

static keyword

The static keyword in java is used for memory management mainly.

The static can be:

1. variable
2. method
3. block
4. nested class

static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects.
- The static variable gets memory only once in class area at the time of class loading.
- Advantage of static variable: It makes your program memory efficient, where static property is shared to all objects .

Example: In following class**,** college refers to the common property of all objects. If we make it static, this field will get memory only once.

```
class Student{
   int id;
   String name;
   String college="IS";}
```

Suppose there are 500 students in my college, now all instance will get memory each time when object is created. All student have its unique id and name so instance variable is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once:

```
class Student{
   int id;
   String name;
   static String college ="IS";
```

```
   Student(int r, String n){
       id = r;
       name = n;}

   void display (){
      System.out.println(id+" "+name+" "+college);}

 public static void main(String args[]){
   Student e1 = new Student(343,"Sarah");
   Student e2 = new Student(292,"Zain");
   e1.display();
   e2.display();}
}
```

Output: ?

Q: Show the output of each following programs:

```
class Counter2{
   static int count=0;

   Counter2(){
      count++;
      System.out.println(count);
   }

 public static void main(String args[]){
   Counter2 c1=new Counter2();
   Counter2 c2=new Counter2();
   Counter2 c3=new Counter2();}
 }
```

Output: ?

```
class Counter{
    int count=0;
    Counter(){
        count++;
        System.out.println(count);
    }

 public static void main(String args[]){
     Counter c1=new Counter();
     Counter c2=new Counter();
     Counter c3=new Counter();
  }
```

Notes: while the static variable get the memory only once, any object changes the value of the static variable, it will retain its value.

static method

If you declare any method as static, it is known static method.

- A static method can be invoked without the need for creating an instance of a class.
- static method can access static variable and can change the value of it.

Example :

```
class Student{
   int id;
   String name;
   static String college = "IS";
   Student(int i, String n){
      id = i;
      name = n;}

   static void change(){
     college = "CS";}

void display (){
    System.out.println(id+" "+name+" "+college);}

public static void main(String args[]){
  Student.change();
  Student e1 = new Student(343,"Sarah");
  Student e2 = new Student(292,"Zain");
  Student e3 = new Student(199,"Ali");
  e1.display();
  e2.display();
  e3.display();}
}
```

Output: ?

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method cannot use non static variable.
2. this and super cannot be used in static context.

Q: The output of the following program "Compile Time Error", Why?

```
class A{
   int a=40;
public static void main(String args[]){
   System.out.println(a);}
}
```

Q: Why java main method is static?

Ans.: because object is not required to call static method if it were nonstatic method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

static block

- Is used to initialize the static variable.
- It is executed before main method at the time of class loading.

Example:

```
class A2{
  static{
    System.out.println("static block is invoked");}

 public static void main(String args[]){
    System.out.println("Hello main");}
}
```

Output: ?

Q: Can we execute a program without main() method?

Ans.: Yes, one of the way is static block but in any version of JDK(not JDK7).

```
class A3{
  static{
    System.out.println("static block is invoked");
    System.exit(0);}
}
```

Final Keyword

The final keyword in java is used to restrict the user, it can be:

1. variable
2. method
3. class

final variable

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

Example:

```
public class Test{
  final int VALUE = 10;
    public static final int BOXWIDTH = 6;
    static final String TITLE = "Manager";
    public void changeValue(){
        VALUE = 12; //will give an error
    }
}
```

final method

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

Syntax:

```
public class Test{
        public final void methodName(){
            // body of method
         }
}
```

Example: Output: Compile Time Error

```
class A{
   final void show(){
       System.out.println("In Class A");}
}

class B extends A{
  void show(){
     System.out.println("In Class B ");

public static void main(String args[]){
   B b= new B();
   b.show();}
}
```

final class

If a class is marked as final then no class can inherit any feature from the final class.

Syntax

```
public final class Test {
      // body of class
 }
```

Example: Output: Compile Time Error

```
final class A{}

  class B extends A{
     void show(){
         System.out.println("In class B");}
class Example{
 public static void main(String args[]){
     B b= new B();
      b.show();}
}
```

Q: Is final method inherited?

Ans.: Yes, final method is inherited but you cannot override it. For example:

```
class A {
   final void show(){
        System.out.println("In class B");}
}

class B extends A{
    void show(){
        System.out.println("In class B");}

 public static void main(String args[]){
     B b= new B();
      b.show();}
}
```

Q: What is blank or uninitialized final variable?

Ans.: A final variable that is not initialized at the time of declaration is known as blank final variable.

Q: Can we initialize blank final variable?

Ans.: Yes, but only in constructor.

Example:

```
class A{
   final int x;//blank final variable
   A(){
      x=10;
      System.out.println("x="+x);}
}
```

Q: What is static blank final variable

Ans.: A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example :

```
 class A{
    static final int data;//static blank final variable
    static{ data=50;}
      public static void main(String args[]){
    System.out.println(A.data);}
}
```

Q: What is final parameter?

Ans: If you declare any parameter as final, you cannot change the value of it.

Example: Output: Compile Time Error

```
class A{
   int sum(final int n){
   n=n+2;//n can't be changed
 }

  public static void main(String args[]){
     A a=new A();
      a.sum(3);}
 }
```

Q: Can we declare a constructor final?

No, because constructor is never inherited

Nested Classes

Java inner class or nested class is a class i.e. declared inside the class or interface.

***Q*: What are the advantage of inner classes?**

Ans.:

1) It can access all the members of outer class including private data members and methods.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Q: ***What difference between nested class and inner class in Java?***

Ans.: Inner class is a part of nested class.

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.

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

Member inner class is class **created inside a class:**

Syntax of Inner class

```
class Java_Outer_class{
    //code
   class Java_Inner_class{
      //code
  }
}
```

Example1:

```
class Outer{
 int data;

 private class Inner{
    public void show(){
       System.out.println("In inner class ");}
  }
}

void display(){
 Inner in = new Inner();
 in.show();}
 }

public class MyClass{
     public static void main(String args[]){
        Outer out = new Outer();
        out.display();}
  }
```

Example2:

```
class Outer{
    private int data=30;
    class Inner{
        void display(){
            System.out.println("data is "+data);}
 }

 public static void main(String args[]){
     Outer obj=new Outer();
     Outer.Inner in=obj.new Inner();
      in.dispaly();}
}
```

Java Anonymous inner class, in this type of inner class we declare and instantiate them at the same time and it declared without a class name.
The syntax of an anonymous inner class is as follows:

Example:

```
abstract class AnonymousInner{
   public abstract void display();}

  public class Outer {
    public static void main(String args[]){
       AnonymousInner inner = new AnonymousInner(){
         public void display(){
           System.out.println("In anonymous inner class");}
         };
         inner.display();}
  }
```

Local inner class, It is class created inside a method. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.
Example:

```
 public class localInner1{
   private int data=30;

   void display(){
       class Local{
         void show(){
           System.out.println(data);}
      }

      Local l=new Local();
      l.show();}

  public static void main(String args[]){
     localInner1 obj=new localInner1();
     obj.display();}
}
```

Static nested class, it is a nested class which is a static member of the outer class.

Notes:

- **It cannot access non-static data members and methods.**
- **It can be accessed by outer class name, without instantiating the outer class.**
- **It can access static data members of outer class including private.**

The syntax of static nested class is as follows:

```
class MyOuter {
  static class NestedClass{
   ....  }
}
```

Example:

```
class Outer{
  static int data=30;

  static class Inner{
    void show(){
      System.out.println("data is "+data);}
   }

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

```
//no need to create the instance of static nested class
   Outer.Inner.show(); }
```

Encapsulation

Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Advantage of Encapsulation

- By providing only setter or getter method, you can make the class read only or write only.
- It provides you the control over the data. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

Example :

```
public class Student{
    private  String name;
    private  String id;
    private  int age;
```

```
public int getAge(){ return age;}

public String getName(){ return name;}

public String getId(){
    return id;}

public void setAge( int newAge){ age = newAge;}

public void setName(String newName){ name = newName;}

public void setId( String newId){
   id = newId;}
}

public class ExampleEncap{
   public static void main(String args[]){
       Student stud = new student();
       stud.setName("Zain");
       stud.setAge(25);
        stud.setId("123IS");
        System.out.print("Name : " + stud.getName() + " Age : " + stud.getAge()+ "  ");
        System.out.print("Id: " + stud.getId());}
}
```

instanceof operator

The instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface), it is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Syntax: objectName **instanceof** className

Example:

```
class Example{
    public static void main(String args[]){
       Example  e=new Example();
       Example d = null;
       System.out.println(e  instanceof  Example);
       System.out.println(d  instanceof  Example); }
}
```

Polymorphism

Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

In computer science the term polymorphism means “a method the same as another in name but with different behavior.” The computer differentiates between (or among) methods depending on either the method signature (after compile) or the object reference (at run time).

Why Polymorphism?

To reducing complexity, where the* polymorphism can only be achieved through the behavior (methods)**, the method ***"println()", *single name many form.* It is an example of polymorphism.

println(10);
println("HelloJava");
println(23.4);

Hence you can see only one method name to print all the different values instead of having different functions to print the values of different data types.

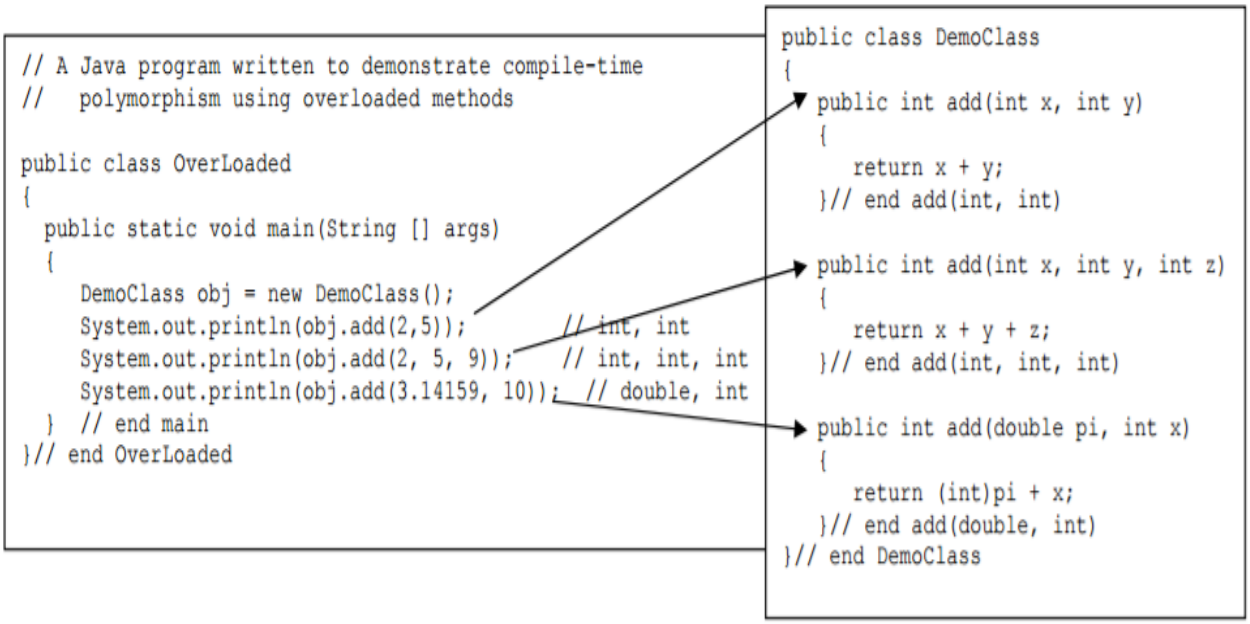
Types of Polymorphism

There are two types:

1. Compile time polymorphism
2. Runtime polymorphism.

Compile time polymorphism

In the example below polymorphism is demonstrated by the use of multiple add methods. The computer differentiates among them by the method signatures (the list of parameters: their number, their types, and the order of the types.)



```
// A Java program written to demonstrate compile-time
//   polymorphism using overloaded methods

public class OverLoaded
{
  public static void main(String [] args)
  {
     DemoClass obj = new DemoClass();
     System.out.println(obj.add(2,5));        // int, int
     System.out.println(obj.add(2, 5, 9));    // int, int, int
     System.out.println(obj.add(3.14159, 10)); // double, int
  }  // end main
}// end OverLoaded
```

```
public class DemoClass
{
   public int add(int x, int y)
   {
      return x + y;
   }// end add(int, int)

   public int add(int x, int y, int z)
   {
      return x + y + z;
   }// end add(int, int, int)

   public int add(double pi, int x)
   {
      return (int)pi + x;
   }// end add(double, int)
}// end DemoClass
```

This form of polymorphism is called compile-time polymorphism because the computer knows after the compile which of the add methods it will execute. Methods whose headings differ in the number and type of formal parameters are said to be overloaded methods.

Runtime polymorphism

Another form of polymorphism called run-time polymorphism because the computer does not know at compile time which of the methods are to be executed. It will not know that until "run time." Runtime polymorphism is achieved through what are called overridden methods.

Run-time polymorphism comes in three different forms:

1. use upcasting, this the most common use of polymorphism in OOP.
2. run-time polymorphism with abstract base classes
3. run-time polymorphism with interfaces.

Q: What is Upcasting?

Ans.: When reference variable of **Parent** class refers to the object of **Child** class, it is known as **upcasting.**

Example:

```
class A{
   void display(){
        System.out.println("In class A");}
 }

 class B extends A{
    void display(){
        System.out.println("In class B");}

 public static void main(String args[]){
   A a = new B();   //upcasting
   a.display(); }
 }
```

Abstraction

Abstract is a process of hiding the implementation details and showing only functionality to the user.

Abstract class

A class that is declared with **abstract** keyword is known as abstract class. An abstract class can have data member, abstract method, non abstract method and constructor.

Syntax:

```
abstract ClassName{
  .... // here must be at least one abstract method
}
```

Abstract method that is declared with abstract keyword and contains a method signature, but no method body with a semoi colon (;) at the end.

Syntax:

```
abstract returnType  methodName(parameters);
```

Rules:

- If there is any abstract method in a class, that class must be abstract.

```
class A{  ←  // class A must be abstract
   abstract void display();
 }
```

- If a class is declared abstract, it cannot be instantiated.
- If you are extending any abstract class that have abstract method, you must provide the implementation of the method(override the abstract method) or declare itself as abstract.

Example1:

```
abstract class A{
   abstract void display();
}

class B extends A{
   void display(){
         System.out.println("In abstract class..");}

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

Example2:

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

class Rectangle extends Shape{
  void draw(){
      System.out.println("drawing rectangle");}
}

class Circle extends Shape{
  void draw(){
       System.out.println("drawing circle");}
}
class TestAbstraction{
  public static void main(String args[]){
    Shape s=new Circle();
    s.draw(); }
}
```

Example3:

```
abstract class Employee{
  private String name;
  private String address;
  private int number;

  Employee(String name, String address, int number){
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;}

   double computePay(){
     System.out.println("Inside Employee computePay");
     return 0.0;}
    String toString(){
      return name + " " + address + " " + number;}
    String getName(){
     return name;}
   String getAddress(){
     return address;}
```

```
    void setAddress(String newAddress){
        address = newAddress;}
    int getNumber(){
        return number;}
}
```

- **To instantiate the Employee class as in the following way(output:Error):**

```
public class AbstractDemo{
public static void main(String [] args){
Employee  e = new Employee("Zain ", "Iraq", 23);
... }
}
```

- **you cannot instantiate the Employee class, but you can instantiate the Salary Class, then you can access all the members of Employee class .**

```
public class AbstractDemo{
  public static void main(String [] args){
     Salary s = new Salary("zain", "Basrah", 3, 2000.00);
     Employee e = new Salary("Ahmed", "Baghdad", 2, 50000.00);
```

Interface

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class **implements** an interface, inheriting the abstract methods of the interface and all these methods need to be defined in the class.

Class VS. Interface

An interface is similar to a class in several ways: including:

- it can contain any number of methods.
- Writing an interface is similar to writing a class.
- An interface can extend another interface, in a similar way as a class can extend another class.

An interface is different from a class in several ways, including:

- a class describes the attributes and behaviors and an interface contains behaviors that a class implements.
- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance variables, only variables that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is **implemented** by a class.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.
- A class can extend only one class, but implement many interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface.

Syntax :

```
public interface NameOfInterface{
        //Any number of final, static variables
       //Any number of abstract method declarations
}
```

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
interface Animal {
   public void eat();
   public void travel();
}
```

Implementing Interfaces

A class uses the **implements** keyword to implement an interface.

Syntax: class ClassName **implements** interfaceName{

```
                ..... // here define all methods in interfaceName
       }
```

Example:

```
public class Mammal implements Animal{
      public void eat(){ System.out.println("Mammal eats");}
      public void travel(){ System.out.println("Mammal travels");}
       public int noOfLegs(){ return 0;}
public static void main(String args[]){
     MammalInt m = new MammalInt();
     m.eat();
     m.travel();}
}
```

Extending Interfaces

An interface can extend another interface in the same 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.

Example:

```
public interface Sports{
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

public interface Football extends Sports{
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

public interface Hockey extends Sports{
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class(Multiple inheritance is not allowed),but interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event{
  .....
}
```