

Java Programming-III

By:- Jagdish Chandra Pandey
H.O.D(I.T.), G.P. Dwarahat/Officiating
Principal, G.P.Chaunaliya

Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Why use inheritance in java:-
 1. For Method Overriding (so runtime polymorphism can be achieved).
 2. For Code Reusability.

Inheritance

- Terms used in Inheritance:-

1. **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
2. **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
3. **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
4. **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Inheritance

- The syntax of Java Inheritance:-

```
1.  class Subclass-name extends Superclass-name
2.  {
3.    //methods and fields
4.  }
```

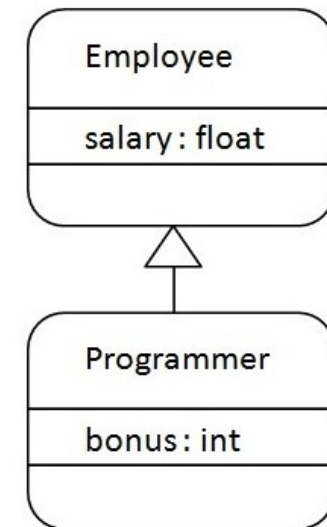
- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a **parent or superclass**, and the new class is called **child or subclass**.

Programmer salary is:40000.0 Bonus of programmer is:10000

Inheritance

```
class Employee{  
    float salary=40000;  
}  
class Programmer extends Employee{  
    int bonus=10000;  
    public static void main(String args[]){  
        Programmer p=new Programmer();  
        System.out.println("Programmer salary is:"+p.salary);  
        System.out.println("Bonus of Programmer is:"+p.bonus);  
    }  
}
```

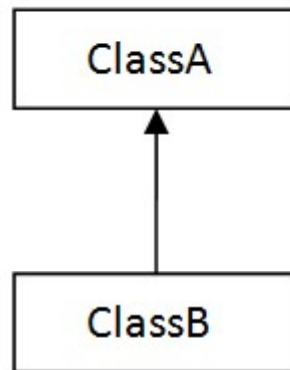
- As displayed in the figure, Programmer is the subclass and Employee is the superclass.



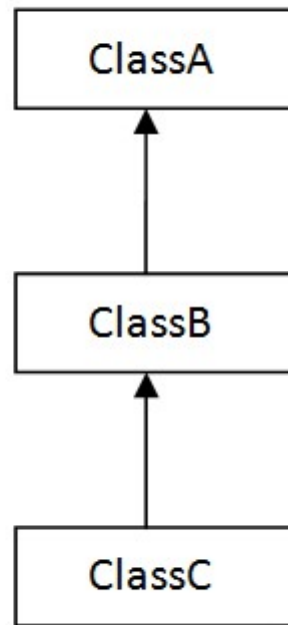
Output:- Programmer salary is:40000.0
Bonus of programmer is:10000

- In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

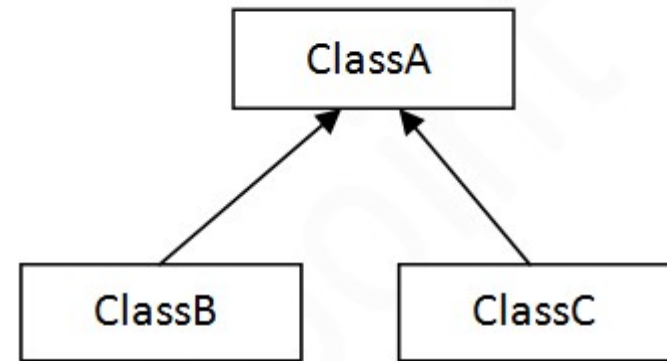
Types of Inheritance



1) Single



2) Multilevel



3) Hierarchical

Single Inheritance

- When a class inherits another class, it is known as a ***single inheritance***. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

- ```
class Animal{
1. void eat(){System.out.println("eating...");}
2. }
3. class Dog extends Animal{
4. void bark(){System.out.println("barking...");}
5. }
6. class TestInheritance{
7. public static void main(String args[]){
8. Dog d=new Dog();
9. d.bark();
10. d.eat();
11. }}
```

- **Output:-**  
1. barking...  
2. eating...



# Multilevel Inheritance

- When there is a chain of inheritance, it is known as ***multilevel inheritance***. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

- ```
class Animal{
1.   void eat(){System.out.println("eating...");}
2.   }
3.   class Dog extends Animal{
4.   void bark(){System.out.println("barking...");}
5.   }
6.   class BabyDog extends Dog{
7.   void weep(){System.out.println("weeping...");}
8.   }
9.   class TestInheritance2{
10.  public static void main(String args[]){
11.  BabyDog d=new BabyDog();
12.  d.weep();
13.  d.bark();
14.  d.eat();
15.  }}
```

- **Output:-**

```
1.  weeping...
2.  barking...
3.  eating...
```


Output:
meowing... eating...

Hierarchical Inheritance

- When two or more classes inherits a single class, it is known as **hierarchical inheritance**. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

- class Animal**

```
1.  {
2.  void eat()
3.  {
4.  System.out.println("eating...");
5.  }
6.  }
7.  class Dog extends Animal
8.  {
9.  void bark()
10. {
11. System.out.println("barking...");
12. }
13. }
14. class Cat extends Animal
15. {
16. void meow()
17. {
18. System.out.println("meowing...");
19. }
20. }
21. class TestInheritance3
22. {
23. public static void main(String args[]){
24. Cat c=new Cat();
25. c.meow();
26. c.eat();
27. //c.bark(); is error because bark is member function of class Dog and there is no such relation between class Dog and class Cat
28. }}
```

Output:
meowing...
eating.....

Q) Why multiple inheritance is not supported in java?

1. To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
2. 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 the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
3. 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.
4. In java programming, multiple and hybrid inheritance is supported through interface only.
5. We can only specify one superclass for any subclass as java does not support the inheritance of multiple superclass into a single subclass.

Inheritance

```
class A
{
    int i,j;
    void show ij()
    {
        System.out.println(" i and j:" +i+ " " +j) ;
    }
}
class B extends A
{
    int k;
    void showk()
    {
        System.out.println("k:" +k);
    }
}
Void sum()
{
    System.out.println("i + j + k :" + (i+j+k));
}
}
class simpleinheritance
{
    public static void main(String args[])
    {
        A superOb = new A();
        B subOb = new B();
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb:");
        SuperOb.showij();
        System.out.println();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println(" contents of subOb:");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("sum of i,j and k in subOb:");
        subOb.sum();
    }
}
```


Inheritance

```
class box
{
    double width;
    double height;
    double depth;
    Box(Box ob)
    {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    Box ()
    {
        width = -1;
        height = -1;
        depth = -1;
    }
    Box(double len)
    {
        width = height = depth = len;
    }

    double volume()
    {
        return width * height * depth;
    }
}

class Boxweight extends Box
{
    double weight;
    Boxweight(double w, double h, double d, double m)
    {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}
```

```
class DemoBoxweight
{
    public static void main(String args[])
    {
        Boxweight mybox1 = new Boxweight(10,20,15, 34.3);
        Boxweight mybox2 = new Boxweight(2,3,4,0.76);
        Double vol;
        Vol mybox1.volume();
        System.out.println("Volume of mybox1 is" +vol);
        System.out.println("Weight of mybox1 is" +mybox1.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is" +vol);
        System.out.println("weight of mybox2 is" +mybox2.weight);
        Boxweight weightbox = new Boxweight(3,5,7,8.37);
        Box plainbox = new Box();
        vol = weightbox.volume();
        System.out.println("Volume of weightbox is" +vol);
        System.out.println("weight of weightbox is" +weightbox.weight);
        System.out.println();
        plainbox = weightbox;
        vol = plainbox.volume();
        System.out.println("volume of plainbox is" +vol);
        /* The following statement is invalid because plainbox does not define a weight member */
        //System.out.println("weight of plainbox is" + plainbox.weight);
    }
}
```

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

- Here, weightbox is a reference to Boxweight objects and plainbox is a reference to Box objects. Since Boxweight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.

- It is type of reference variable not the type of the object that it refers to, that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, we will have access only to those parts of object defined by superclass. This is why plainbox can't access even when it refers to a Boxweight object.

Using super

1. In previous examples, classes derived from Box were not implemented as efficiently, for ex:- the constructor for Boxweight explicitly initializes the width, height and depth fields of Box(). Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. But there will be times when we will want to create a superclass that keeps the details of its implementation to itself (i.e., that keeps its data member's private), so whenever a subclass needs to refer to its immediate superclass it can do so by use of keyword super.
2. super has two general forms:-
 - The first calls the superclass constructor.
 - The second is used to access a member of superclass that had been hidden by a member of a subclass.
3. A subclass can call a constructor defined by its superclass by use of super as:-

```
super(arg_list);
```

Here `arg_list` specifies any arguments needed by constructor in superclass. `super()` must always be the first statement executed inside a subclass constructor.

Using super to call superclass constructors

```
// Boxweight uses super to initialize Box(superclass) attributes
Class Boxweight extends Box
{
double weight;
Boxweight(double w, double h, double d, double m)
{
super(w,h,d);
weight = m;
}
}
```

- The above code leaves Box(superclass) free to make values (width, height, depth) private if desired.

Using Super to call superclass constructors

```
class box
{
    private double width;
    private double height;
    private double depth;
    Box(Box ob)
    {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
    Box()
    {
        width = -1;
        height = -1;
        depth = -1;
    }
    Box(double len)
    {
        width = height = depth = len;;
    }
    double volume()
    {
        return width*height*depth;
    }
}
class Boxweight extends Box
{
    double weight;
    Boxweight(Boxweight ob)
    {
        super(ob);
        weight = ob.weight;
    }
    Boxweight (double w, double h, double d, double m)
    {
        super(w,h,d);
        weight = m;
    }
}
```

```
Boxweight()
{
    super();
    Weight = -1;
}
Boxweight(double len, double m)
{
    super(len);
    weight = m;
}
class Demosuper
{
    public static void main(String args[])
    {
        Boxweight mybox1 = new Boxweight(10,20,15,34.3);
        Boxweight mybox2 = new Boxweight(2,3,4,.076);
        Boxweight mybox3 = new Boxweight();
        Boxweight mycube = new Boxweight(3,2);
        Boxweight myclone = new Boxweight(mybox1);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("weight of mybox2 is" + mybox2.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("volume of mybox2 is" + vol);
        System.out.println("weight of mybox2 is" +mybox2.weight);
        System.out.println();
        vol = mybox3.volume();
        System.out.println("volume of mybox3 is" +vol);
        System.out.println("weight of mybox3 is" +mybox3.weight);
        System.out.println();
    }
}
```

In this program the constructor:-

```
Boxweight(Boxweight ob)
{
    super(ob);
    weight = ob.weight;
}
```

`super()` is passed an object of type `Boxweight`-not of type `Box`. This still involves the constructor `Box(Box ob)` as a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a `Boxweight` object to the `Box` constructor. But, `Box` only has knowledge of its own members.

A second use for super

```
class A
{
    int i;
}
class B extends A
{
    int i;
    B(int a, int b)
    {
        super.i = a; // i in A
        i = b;
    }
    void show()
    {
        System.out.println(" i in superclass : " +super.i);
        System.out.println("i in subclass: " +i);
    }
}
class usesuper
{
    public static void main(String args[])
    {
        B subOb = new B(1,2);
        subOb.show();
    }
}
```

super.member

This second form of super is most applicable to situations in which member names of a subclass hide members by same name in superclass.

Dynamic Method Dispatch

```
class A
{
void callme()
{
System.out.println("Inside A's call me method");
}
}
class B extends A
{
void callme()
{
System.out.println("Inside B's call me method");
}
}
class C extends A
{
void callme()
{
System.out.println("Inside C's call me 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
r.callme();// calls A's version of call me
r = b; // r refers to a B object
r.callme() // calls B's version of call me
r = c; // r refers to a C object
r.callme();// calls c's version of call me
}
}
```

1. It is mechanism by which a call to an overridden method is resolved at runtime , rather than compile time so it is example of run time polymorphism .
2. A superclass reference variable can refer to a subclass object and Java uses this fact to resolve calls to overridden methods at run time.
3. When an overridden method is called through a superclass reference , java determines which version of that method to execute based upon type of object being referenced to at the time the call occurs. Thus this determination is made at run time. In other words, it is the type of object being referred to(not the type of reference variable) that determines which version of an overridden method will be executed.

Final Keyword to prevent Overriding

```
class A
{
final void meth()
{
System.out.println("This is a final method");
}
}
class B extends A
{
void meth() // Error ! Cann't override
{
System.out.println("Illegal !");
}
}
```

The keyword final has 3 uses:-

1. First it can be used to create equivalent of a named constant
2. While other two uses of final apply to inheritance

- Because meth() is declared as final, it cannot be overridden in B
- Methods declared as final can provide a performance enhancement. The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small final method is called, often the java compiler can copy the bytecode for the subroutine directly inline with the compiled code of calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods.
- Normally, Java resolve calls to methods dynamically, at run time. This is called **late binding**. However since final methods cannot be overridden, a call to one can be resolved at compile time. This is called **early binding**.

Using final to prevent Inheritance

- To prevent a class from being inherited precede the class declaration with final.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

eg:-

```
final class A
```

```
{
```

```
    //...
```

```
}
```

```
// The following class is illegal.
```

```
class B extends A
```

```
{
```

```
    .....
```

```
}
```

Using abstract class

- Sometimes we want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. In Java this is done through abstract method.
- Here certain methods overridden by subclass by specifying abstract type modifier.

Syntax:-

`abstract type name(Parameter-list);`

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract use the abstract keyword in the front of class keyword.
- There can be no objects of an abstract class, also you cannot declare abstract constructors or abstract static methods.
- Any subclass of an abstract class must either implement all of abstract methods in the superclass or be itself declared abstract.

Using abstract class


```
abstract class A
{
    abstract void callme();
    //concrete methods are still allowed in abstract classes
    void callmetoo()
    {
        System.out.println("this is a concrete method.");
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo
{
    public static void main(String args[])
    {
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}
```

Although abstract classes cannot be used to instantiate objects, they can be used to create object references , because java's approach to run time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

The object class

1. There is one special class, `object` defined by java. All other classes are subclasses of `object`. That is, `object` is a superclass of all other classes. This means that a reference variable of type `object` can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type `object` can also refer to any array.
2. `Object` defines following methods, which means that they are available in every object:-
 - i. `object clone()` → creates a new object that is same as the object being cloned.
 - ii. `boolean equals(object object)` → determines whether one object is equal to another.
 - iii. `void finalize()` → called before an unused object is recycled.
 - iv. `class getClass()` → obtains the class of an object at run-time.
 - v. `int hashCode()` → returns the hash code associated with the invoking object.
 - vi. `void notify()` → resumes execution of a thread waiting on the invoking object.
 - vii. `string toString()` → returns a string that describes the object.
 - viii.

`void wait()`
`void wait(long milliseconds)`
`void wait(long milliseconds, int nanoseconds)`



`waits on another thread of execution.`

The object class

- The methods `getClass()`, `notify()`, `notifyall()` and `wait()` are declared as final while we may override the others.
- The `toString()` method returns a string that contains a description of object on which it is called. Also, this method is automatically called when an object is output using `println()`.

Continued.....