Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

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. Moreover, you can add new methods and fields in your current class also.

Terms:

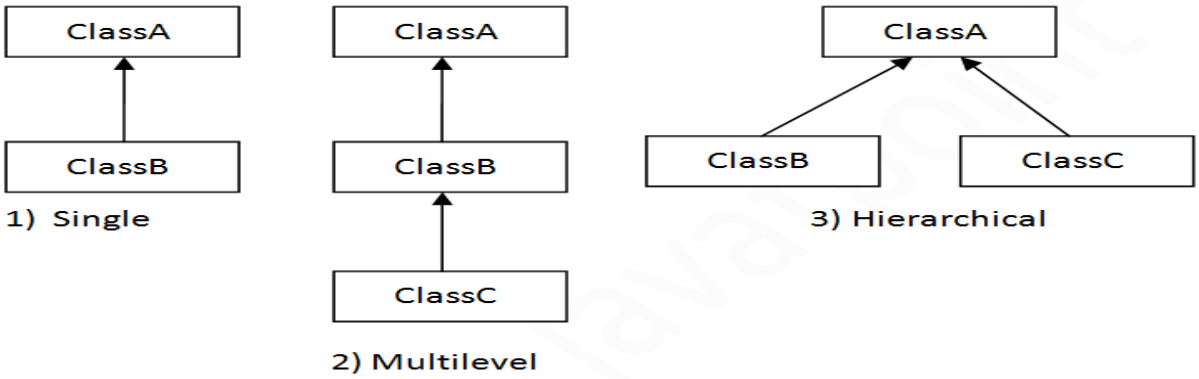
- **Super Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

The syntax of Java Inheritance

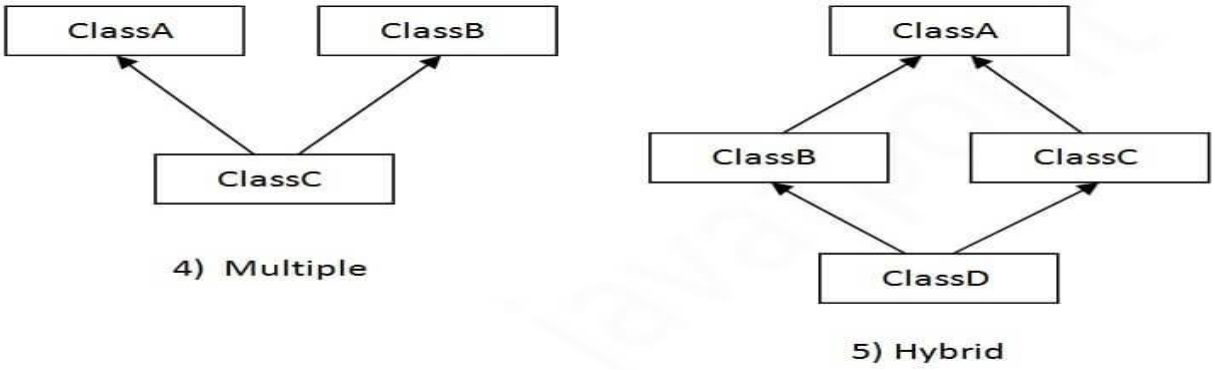
```
1.  class Subclass-name extends Superclass-name
2.  {
3.    //methods and variables
4.  }
```

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

Types of Inheritance



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

Note: Multiple inheritance is not supported in Java through class.

Single Inheritance

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

Example Simple Inheritance

```
class A
{
int i, j;
void showij() {
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();

subOb.i = 7;
subOb.j = 8;
subOb.k = 9;

System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();

System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

Member Access and Inheritance

```
// This program results in error.

class A
{
int i;
private int j;

void setij(int x, int y)
{
i = x;
j = y;
}

}

class B extends A
{
int total;
void sum() {
total = i + j;  // As j is private so it cannot be accessed here. This is an error.
}
}

class Access
{
```

```
public static void main(String args[])
{
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

```
class Animal
{
void eat()
{
System.out.println("eating...");
}
}

class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}

class BabyDog extends Dog
{
void weep()
{
System.out.println("weeping...");
}
}

class TestInheritance
{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

A Superclass Variable Can Reference a Subclass Object

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

```
class Box {
double width;
double height;
double depth;

Box() {
width = 1;
height = 1;
depth = 1;
}

Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}

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 RefDemo {
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
plainbox = weightbox;
vol = plainbox.volume();
System.out.println("Volume of plainbox is " + vol);

// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

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 important to understand that it is the type of the 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, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object.

Super Keyword in Java

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

Usage of Java super Keyword

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

Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

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.

```
/* superclass Person */
class Person
{
   Person()
   {
      System.out.println("Person class Constructor");
   }
}

/* subclass Student extending the Person class */
class Student extends Person
{
   Student()
   {
      // invoke or call parent class constructor
      super();

      System.out.println("Student class Constructor");
   }
}

/* Driver program to test*/
class Test
{
   public static void main(String[] args)
   {
      Student s = new Student();
   }
}
```

Output:

Person class Constructor

Student class Constructor

Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

// Using super to overcome name hiding.

```
class A {
int i;
}

class B extends A {
int i;

B(int a, int b) {
super.i = 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();
}

}
```

This program displays the following:
i in superclass: 1
i in subclass: 2

Use of super with methods

This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.

```
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}

class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();
         // calling display() of Student
        s.display();
    }
}
```

Output:
This is student class
This is person class
In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

Important points:

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*.

When Constructors Are Called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.

```
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 c = new C();
}
}
```

The output from this program is shown here:
Inside A's constructor
Inside B's constructor
Inside C's constructor

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}

void show() {
System.out.println("i and j: " + i + " " + j);
}
}

class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}

void show() {
System.out.println("k: " + k);
}
}

class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show();
}
}
```

Output:
k: 3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden function, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}

void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
```

```
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}

void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show();
}
}
```

The output is:

```
i and j: 1 2
k: 3
```

Dynamic Method Dispatch

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

```
class A{}
class B extends A{}
A a=new B();//upcasting
```

// A Java program to illustrate Dynamic Method Dispatch using hierarchical inheritance

```
class A
{
   void m1()
   {
      System.out.println("Inside A's m1 method");
   }
}

class B extends A
{
   // overriding m1()
   void m1()
   {
      System.out.println("Inside B's m1 method");
   }
}

class C extends A
{
   // overriding m1()
   void m1()
   {
      System.out.println("Inside C's m1 method");
   }
}

class Dispatch
{
   public static void main(String args[])
   {
      A a = new A();
      B b = new B();
      C c = new C();

      // obtain a reference of type A
      A ref;

      // ref refers to an A object
      ref = a;

      // calling A's version of m1()
```

```
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

Output:
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method

// Using run-time polymorphism.

```
class Figure {
double dim1;
double dim2;

Figure(double a, double b) {
dim1 = a;
dim2 = b;
}

double area() {
System.out.println("Area for Figure is undefined.");
return 0;
}
}

class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}

double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}

class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}

double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
```

```
class FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body). It needs to be extended and its abstract methods are to be implemented in the subclass.

In java, the following some *important observations* about abstract classes are as follows:

1. An instance of an abstract class cannot be created.
2. Constructors are allowed. But abstract constructors are not allowed.
3. We can have an abstract class without any abstract method.
4. There can be a **final method** in abstract class but any abstract method in class (abstract class) cannot be declared as final or in simpler terms final method cannot be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final"
5. We can define static methods in an abstract class
6. If a **class** contains at least **one abstract method** then compulsory should declare a class as abstract
7. If the **Child class** is unable to provide implementation to all abstract methods of the **Parent class** then we should declare that **Child class as abstract** so that the next level Child class should provide implementation to the remaining abstract method.

```
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();
}
}
```

Ouput:

```
B's implementation of callme.
This is a concrete method.
```

Final keyword in Java

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

Final can be:

- variable
- method
- class

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}

class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, 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.

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.