7.Inheritence in Java

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

Inheritance

- The derivation of one class from another class is called Inheritance.
- A class that is inherited is called a super class.
- The class that does the inheriting is called as subclass.
- A subclass inherits all instance variables and methods from its super class and also has its own variables and methods.
- One can inherit the class using keyword ***extends*.**

Syntax :

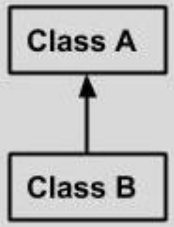
class *subclass-name* extends *super class-name*

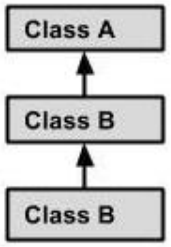
{

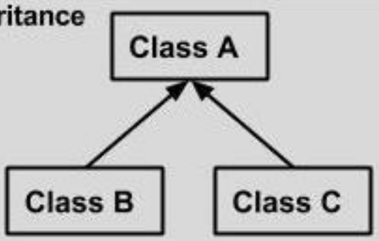
// body of class.

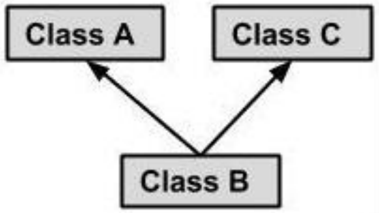
}

Inheritance Types

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

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

Hierarchical Inheritance 	public class A { ....}
	
	public class B **extends** A {...................}
	
	public class C **extends** A {..................... }

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

Inheritance

- In java, a class has only one super class.
- Java does not support Multiple Inheritance.
- One can create a hierarchy of inheritance in which a subclass becomes a super class of another subclass.
- However, no class can be a super class of itself.
- super class is also called parent class or base class
- sub class is also called child class or derived class

```
class A   //superclass
{
   int num1;    //member of superclass
   int num2;    //member of superclass
   void setVal(int no1, int no2)   //method of superclass
   {
      num1 = no1;
      num2 = no2;
   }
}
 class B extends A   //subclass B
{
   int multi;   //member of subclass
   void mul()   //method of subclass
   {
      multi = num1*num2;   //accessing member of superclass from subclass
   }
}
 class inhe2
{
   public static void main(String args[])
   {
      B subob = new B();
      subob.setVal(5,6);  //calling superclass method throgh subclass object
      subob.mul();        System.out.println("Multiplication is  " + subob.multi);
   }}
```

Output :

```
Multiplication is  30
```

Inheritance – Member Access

- Private members of super class is not accessible in subclass
- Protected member of super class is accessible in sub class
- Default access types not be accessible to child class, but may be accessible due to them being in same package

Keyword: super

- super keyword is used to call a superclass constructor and to call or access super class members(instance variables or methods).

  syntax of super : super (arg-list)

- When a subclass calls super() it is calling the constructor of its immediate superclass.
- super() must always be the first statement executed inside a subclass constructor.
- super. member

  Here member can be either method or an instance variables.

- This second form of super is most applicable to situation in which member names of a subclass hide member of superclass due to same name.

```
class A1
{
    public int i;
    A1()
    {
        i=5;
    }
}

class B1 extends A1
{
    int i;
    B1(int a,int b)
    {
        super();    //calling super class constructor
        //now we will change value of superclass variable i
        super.i=a;  //accessing superclass member from subclass
        i=b;
    }
```

```
 void show()
    {
        System.out.println("i in superclass = " + super.i );
        System.out.println("i in subclass = " + i );
    }
}

public class Usesuper
{
        public static void main(String[] args)
        {
            B1 b = new B1(10,12);
            b.show();


        }
}
```

```
Output :

i in superclass = 10
i in subclass = 12
```

Note:

- The instance variable i in B1 hides the i in A, super allows access to the i defined in the superclass.

- super can also be used to call methods that are hidden by a subclass.

Method Overriding

- Defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass is called method overriding.

- Now when the method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass.

```
class Xsuper
{
   int y;
   Xsuper(int y)
   {
      this.y=y;
   }
   void display()
   {
      System.out.println("super y = " +y);
   }
}
class Xsub extends Xsuper
{
   int z;
   Xsub(int z , int y)
   {
      super(y);
      this.z=z;
   }
```

```
 void display()
   {
      System.out.println("super y = " +y);
      System.out.println("sub z = " +z);
   }

}

public class TestOverride {
   public static void main(String[] args)
   {
      Xsub s1 = new Xsub(100,200);
      s1.display();
   }
}
```

Output :

```
super y = 200
sub z = 100
```

Here the method display() defined in the subclass is invoked.

Multilevel Inheritance

```
class student
{
   int rollno;
   String name;
    student(int r, String n)
   {
      rollno = r;
      name = n;
   }
   void dispdatas()
   {
      System.out.println("Rollno = " + rollno);
      System.out.println("Name = " + name);
   }}
 class marks extends student
•    {
•       int total;
•       marks(int r, String n, int t)
•       {
•          super(r,n);   //call super class (student) constructor
•          total = t;
•       }
```

```
   void dispdatam()
   {
      dispdatas();    // call dispdatap of student class
      System.out.println("Total = " + total);
   }
}
      class percentage extends marks
{
   int per;

   percentage(int r, String n, int t, int p)
   {
      super(r,n,t);  //call super class(marks) constructor
      per = p;
   }
   void dispdatap()
   {
      dispdatam();    // call dispdatap of marks class
      System.out.println("Percentage = " + per);
   }
}
```

```
class Multi_Inhe
{
   public static void main(String args[])
   {
      percentage stu = new percentage(1912, "SAM", 350, 50); //call constructor percentage
      stu.dispdatap();  // call dispdatap of percentage class
   }
}
```

```
Output :

Rollno = 1912
Name = SAM
Total = 350
Percentage = 50
```

It is common that a class is derived from another derived class.

The class student serves as a base class for the derived class marks, which in turn serves as a base class for the derived class percentage.

The class marks is known as intermediated base class since it provides a link for the inheritance between student and percentage.

The chain is known as inheritance path.

When this type of situation occurs, each subclass inherits all of the features found in all of its super classes. In this case, percentage inherits all aspects of marks and student.

To understand the flow of program read all comments of program.

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?

```
class X
{
   X()
   {
      System.out.println("Inside X's constructor.");
   }
}
 class Y extends X   // Create a subclass by extending class A.
{
   Y()
   {
      System.out.println("Inside Y's constructor.");
   }
}
class Z extends Y   // Create another subclass by extending B.
{
   Z()
   {
      System.out.println("Inside Z's constructor.");
   }
}
```

```
public class CallingCons
{
    public static void main(String args[])
    {
        Z z = new Z();
    }
}
```

Output:

```
Inside X's constructor.
Inside Y's constructor.
Inside Z's constructor.
```

The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.

Further, since super( ) must be the first statement executed in a subclass' 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.

As you can see from the output the constructors are called in order of derivation.

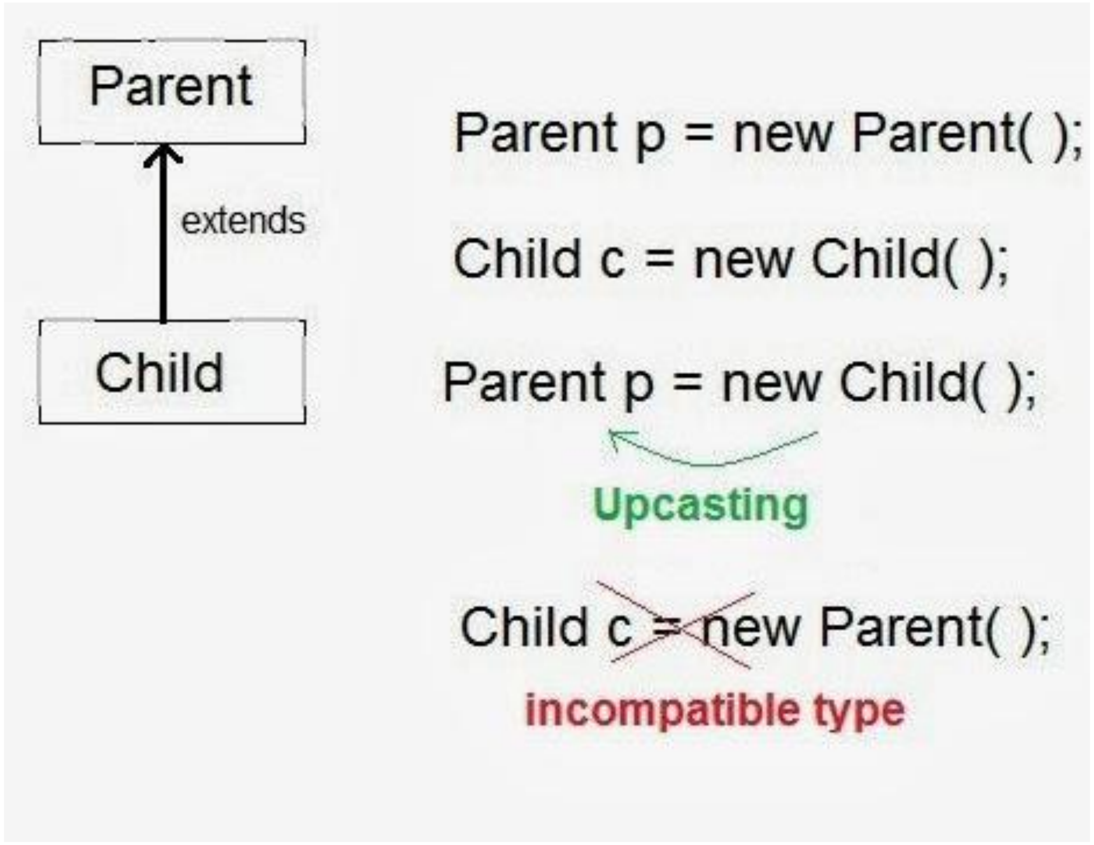
If you think about it, it makes sense that constructors are executed in order of derivation.

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.

Dynamic Method Dispatch

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.**
- A method to execution based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- Dynamic Method Dispatch is related to a principle that states that an super class reference can store the reference of subclass object. However, it can't call any of the newly added methods by the subclass but a call to an overridden methods results in calling a method of that object whose reference is stored in the super class reference.



```
Parent p = new Parent( );
Child c = new Child( );
Parent p = new Child( );
Child c = new Parent( );
```

In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

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

```
public class Dynamic_disp
{
    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 callme
        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme

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

Output:

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

Here reference of type A, called r, is declared.

The program then assigns a reference to each type of object to r and uses that reference to invoke callme( ).

As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call.

Final keyword :

- final keyword can be used with variables, methods and classes.
- final variables.
  - When we want to declare constant variable in java we use final keyword.
  - Syntax : final variable name = value;
- final method
  - Syntax: final methodname(arg)
  - To prevent overriding of method final keyword is used, means final method cant be override.

- final class
  - **A class that can not be sub classed/inherited is called a final class.**
  - This is achived in java using the keyword final as follow.
  - Syntax : final class class_name {   ...   }
  - Any attempt to inherit this class will cause an error and compiler will not allow it.

```
final class aa
{
    final int a=10;
    public final void ainc()
    {
        a++;   // The final field aa.a cannot be assigned
    }

}

class bb extends aa    // The type bb cannot subclass the final class aa
{
    public void ainc()    //Cannot override the final method from aa
    {
        System.out.println("a = " + a);
    }
}

public class Final_Demo
{
    public static void main(String[] args)
    {
        bb b1 = new bb();
        b1.ainc();
    }
}
```

Here no output will be there because all the comments in above program are errors. Remove final keyword from class than you will get error like final method can not be override.

Abstract Classes

- When the keyword abstract appears in a class definition, it means that zero or more of it's methods are abstract.
- An abstract method has no body.
- Some of the subclass has to override it and provide the implementation.
- Objects cannot be created out of abstract class.
- Abstract classes basically provide a guideline for the properties and methods of an object.

- In order to use abstract classes, they have to be subclassed.

- There are situations in which you want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

- That is, sometimes you want to create a superclass that only defines generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method

Syntax :

```
abstract type name(parameter-list);
```

- As you can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
- Any subclass of an abstract class must either implement all of the abstract methods of the superclass, or be itself declared abstract.

```
abstract class A1
{
   abstract void displayb1();
   void displaya1()
   {
      System.out.println("This is a concrete method");
   }
 }
 class B1 extends A1
{
   void displayb1()
   {
      System.out.println("B1's implementation");
   }
}
 public class Abstract_Demo
{
   public static void main(String args[])
   {
      B1 b = new B1();
      b.displayb1();
      b.displaya1();
   }
}
```

Output:

```
B1's implementation
This is a concrete method
```

```
abstract class shape
{
   double dim1;
   double dim2;
   shape(double a, double b)
   {
      dim1 = a;
      dim2 = b;
   }
   abstract double area();
}
 class rectangle extends shape
{
   rectangle(double a, double b)
   {
      super(a,b);
   }
   double area()
   {
      System.out.println("Area of rectangle.");
      return dim1*dim2;
   }}
```

```
class triangle extends shape
{
   triangle(double a, double b)
   {
      super(a,b);
   }
   double area()
   {
      System.out.println("Area of triangle.");
      return dim1*dim2/2;
   }
}
 public class Abstract_D2
{
   public static void main(String args[])
   {
      rectangle r = new rectangle(10,20);
      triangle t = new triangle(10,6);

      System.out.println(r.area());
      System.out.println(t.area());
   }}
```

Output:

```
Area of rectangle. 200.0
Area of triangle.    30.0
```

INTERFACE

Interface

- The interface keyword takes the concept of abstractness one step further.
- An interface is a group of related methods with empty bodies
- An interface declaration can contain method signatures, default methods, static methods and constant definitions
- The only methods that have implementations are default and static methods
- In interfaces, no other of the methods are implemented means interfaces defines methods without body.
- All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.
- An interface can contain constant declarations.
- All constant values defined in an interface are implicitly public, static, and final
- Once again, you can omit these modifiers

Declaring interfaces in java with syntax

```
[Access-specifier] interface interface-name
{
return-type method-name(parameter-list);
type var1=value;
}
```

- Where, Access-specifier is either public or it is not given.
- When no access specifier is used, it results into default access specifier and if interface has default access specifier then it is only available to other members of the same package.
- When it is declared as public, the interface can be used by any other code of other package.

Declaring interfaces in java with syntax

- *Interface-Name: name of an interface, it can be any valid identifier.*
- The methods which are declared having no bodies they end with a semicolon after the parameter list. Actually they are abstract methods

- Variables can be declared inside interface declarations.
- They are implicitly final and static, means they can not be changed by implementing it in a class.
- They must also be initialized with a constant value.

Extending Inheritance

- An interface can extend other interfaces
- An interface can extend any number of interfaces
- The interface declaration includes a comma-separated list of all the interfaces that it extends
- Eg.

```
public interface Group1 extends Interface1, Interface2, Interface3
```

Interface vs Abstract Class

- Interfaces are similar to abstract classes, but differ in their functionality.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

EX :

```
interface Item
{
      static final int code = 100;
      static final String name = "Fan";
      void display ( );
}

interface Area
{
      static final float pi = 3.14F;
      float compute ( float x, float y );
      void show ( );
}
```

Implementing Interfaces

- Once an interface has been defined, one or more classes can implement that interface.
- To implement an interface, include the *implements* clause in a class definition, and then create the methods declared by the interface.
- The general form of a class that includes the implements clause looks like this:
- Access-specifier class classname [extends superclass] [implements interface, [, interface..]]

```
 {
       // class body
  }
```

Implementing Interfaces

- If we are implementing an interface in a class we must implement all the methods defined in the interface as well as a class can also implement its own methods
- But if a class implements an interface but does not fully implement the method defined by that interface, then that class must be declared as abstract.
- Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in C++.

Implementing Interfaces

- The methods that implement an interface must be declared as public.
- The type-signature of implementing method must match exactly the type signature specified in the interface.
- If a class implements from more than one interface, names are separated by comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.

Ex:1

```
interface religion
{
    String city = new String("Amritsar");
    void greet();
    void pray();
}
 class gs implements religion
{
    public void greet()
    {
        System.out.println("We greet - ABCD");
    }
    public void pray()
    {
        System.out.println("We pray at " + city + " XYZ ");
    }
}
```

```
class iface1
{
    public static void main(String args[])
    {
        gs sikh = new gs();
        sikh.greet();
        sikh.pray();
    }
}
```

Output:

```
We greet - ABCD
We pray at Amritsar XYZ
```

Ex: 2

```
interface i1
{
    void dispi1();
}
 interface i2
{
    void dispi2();
}

class c1 implements i1
{
    public void dispi1()
    {
        System.out.println("This is display of i1");
    }
}
 class c2 implements i2
{
public void dispi2()
    {
        System.out.println("This is display of i2");
    }}
    class c3 implements i1, i2
{
    public void dispi1()
    {
        System.out.println("This is display of i1");
    }
    public void dispi2()
    {
        System.out.println("This is display of i2");
    }
}

class iface2
{
    public static void main(String args[])
    {
        c1 c1obj = new c1();
        c2 c2obj = new c2();
        c3 c3obj = new c3();
         c1obj.dispi1();
        c2obj.dispi2();
        c3obj.dispi1();
        c3obj.dispi2();
    }}
```

Output:

```
This is display of i1
This is display of i2
This is display of i1
This is display of i2
```

// Implementing interface having common function name

```
interface i1
{
   void disp();
}
 interface i2
{
   void disp();
}
 class c implements i1, i2
{
   public void disp()
   {
      System.out.println("This is display .. ");
   }}
 class iface7
{
   public static void main(String args[])
   {
      c cobj = new c();
      cobj.disp();
   }}
```

Output:

```
This is display ……
```

Note : When implementing an interface method, it must be declared as public. It is possible for classes that implement interfaces to define additional members of their own.

Partial Implementation of Interface :

- If we want to implement an interface in a class we have to implement all the methods defined in the interface
- But if a class implements an interface but does not fully implement the method defined by that interface, then that class must be declared as abstract.

```
interface i1
{
   void disp1();
   void disp2();
}
 abstract class c1 implements i1
{
   public void disp1()
   {
      System.out.println("This is display of 1");
   }}
 class c2 extends c1
{
   public void disp2()
   {
      System.out.println("This is display of 2");
   }}
class iface
{
   public static void main(String args[])
   {
      c2 c2obj = new c2();
      c2obj.disp1();
      c2obj.disp2();
   }}
```

Output:

```
This is display of 1
This is display of 2
```

Using an Interface as a Data Type

- When you define a new interface, you are defining a new reference data type.
- You can use interface names anywhere you can use any other data type name.
- If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

```
interface AreaCal
{
    final double pi = 3.14;
    double areacalculation(double r);
}

class Circle implements AreaCal
{
    public double areacalculation(double r)
    {
        double ar;
        ar = pi*r*r;
        return ar;
    }
}
 class iface3
{
    public static void main(String args[])
    {
        double area;
        AreaCal ac = new Circle();
        area = ac.areacalculation(10.25);
        System.out.println("Area of Circle is : " + area);
    }
}
```

Output:

```
Area of Circle is :
329.89625
```

Note : Here variable ac is declared to be of the interface type AreaCal, it was assigned an instance of circle. Although ac can be used to access the areacalculation() method, it cannot access any other members of the client class. An interface reference variable only has knowledge of the method declared by its interface declaration.

Extending interfaces :

- One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface,
- It must provide implementation of all methods defined within the interface inheritance.

- **Note :** Any class that implements an interface must implement all methods defined by that interface, including any that inherited from other interfaces.

```
interface if1
{
   void dispi1();
}
interface if2 extends if1
{
   void dispi2();
}
class cls1 implements if2
{
   public void dispi1()
   {
      System.out.println("This is display of i1");
   }
   public void dispi2()
   {
      System.out.println("This is display of i2");
   }}
public class Ext_iface
{
   public static void main(String args[])
   {
      cls1 c1obj = new cls1();
      c1obj.dispi1();
      c1obj.dispi2();       }

}
```

Output:

```
This is display of i1
This is display of i2
```

Note : We have to define disp1() and disp2() in cls1.

Multiple inheritance using interface..

```
class stu
{
    int rollno;
    String name = new String();
    int marks;
    stu(int r, String n, int m)
    {
        rollno = r;
        name = n;
        marks = m;
    }
}

interface i
{
    void display();
}

class studerived extends stu implements i
{
    studerived(int r, String n, int m)
    {
        super(r,n,m);
    }
    public void display()
    {
        System.out.println("Displaying student details .. ");
        System.out.println("Rollno = " + rollno);
        System.out.println("Name = " + name);
        System.out.println("Marks = " + marks);
    }
}

public class Multi_inhe_demo
{
    public static void main(String args[])
    {
        studerived obj = new studerived(1912, "Ram", 75);
        obj.display();
    }

}
```

Output:

```
Displaying student details
..
Rollno = 1912
Name = Ram
Marks = 75
```

Object Class

- There is one special class, Object, defined by Java. All other classes are sub classes of Object.
- That is, Object is a super class of all other classes.
- This means that a reference variable of type Object can refer to an object of any other class.
- Every class in java is descended from the **java.lang.Object** class.
- If no inheritance is specified when a class is defined, the super class of the class is Object by default.

EX :

- public class circle { ... }

is equivalent to

- public class circle extends Object { ... }

Methods of Object class

METHOD	PURPOSE
Void finalize()	Called before an unused object is recycled
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long milliseconds) void wait(long milliseconds, int nanoseconds)	Waits on another thread of execution.

Methods of Object class:

The methods getclass(), notify(), notifyall() and wait() are declared as final.You may override the others.

tostring()

public String tostring()

it returns a String that describe an object.

It consisting class name, an at (@) sign and object memory address in hexadecimal.

EX :

Circle c1 = new Circle();

System.out.println(c1.tostring());

It will give O/P like Circle@15037e5

We can also write System.out.println(c1);