# Inheritance

- Inheritance is one of the feature of Object-Oriented Programming.
- Inheritance is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.
- **Super Class:** The class whose features are inherited is known as 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 keyword used for inheritance is **extends**.
- Inheritance is also known as the IS-A relationship.

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
   .....
}
```

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.

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

By creating the object of the subclass you can access the members of a superclass and subclass.

To access the members of both classes it is recommended to always create reference variable to the subclass.

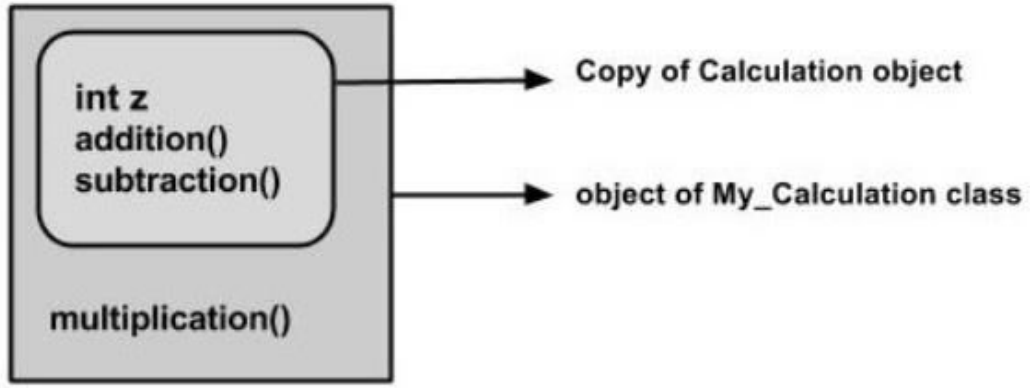
```java
class Calculation {
   int z;

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

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

public class My_Calculation extends Calculation {
   public void multiplication(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 = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.Subtraction(a, b);
      demo.multiplication(a, b);
   }
}
```




```
javac My_Calculation.java
java My_Calculation
```

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

# Types of Inheritance in Java

- The different types of Inheritance are:
  - **Single Inheritance**
  - **Multiple Inheritance**
  - **Multi-Level Inheritance**
  - **Hierarchical Inheritance**
  - **Hybrid Inheritance**
- **Single Inheritance**
- Creating subclasses from a single base class is called single inheritance.

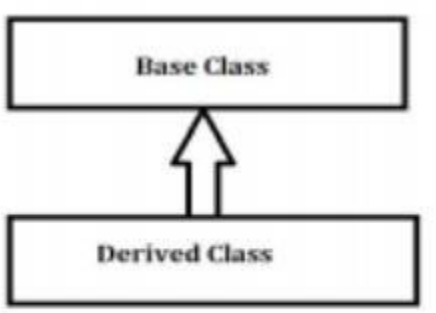



*Figure: Single Inheritance*

```
class A
{
 int a, b;
 void display()
 {
 System.out.println("Inside class A values ="+a+" "+b);
 }
}
class B extends A
{
 int c;
 void show()
 {
 System.out.println("Inside Class B values="+a+" "+b+" "+c);  }
}
class SingleInheritance
{
 public static void main(String args[])
{
 B obj = new B(); //derived class object
 obj.a=10;
 obj.b=20;
 obj.c=30;
 obj.display();
 obj.show();
 }
}
```

Output:

Inside class A values =10 20

Inside Class B values =10 20 30

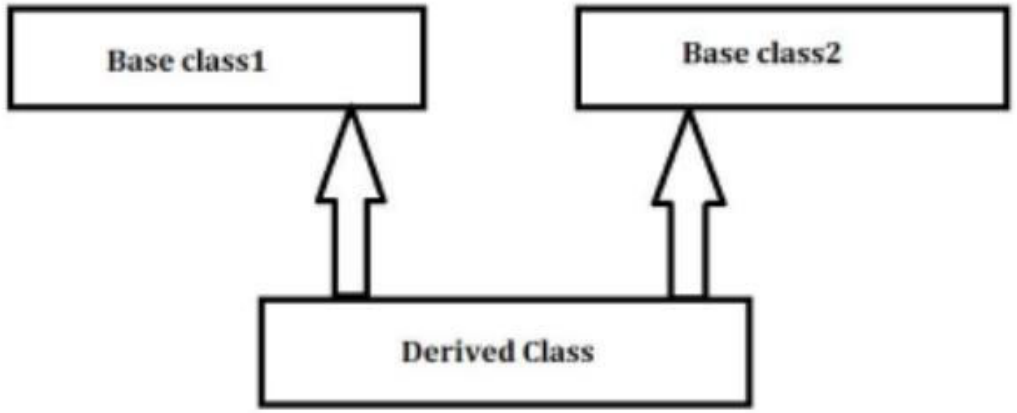
- **Multiple Inheritance in Java:**

Defining derived class from numerous base classes is known as ‘Multiple Inheritance’. In this case, there is more than one superclass, and there can be one or more subclasses. Multiple inheritances are available in object-oriented programming with C++, but it is not available in Java.

Java developers want to use multiple inheritances in some cases. Fortunately, Java developers have interface concepts expecting the developers to achieve multiple inheritances by using multiple interfaces.

Ex: class Myclass implements interface1, interface2,….

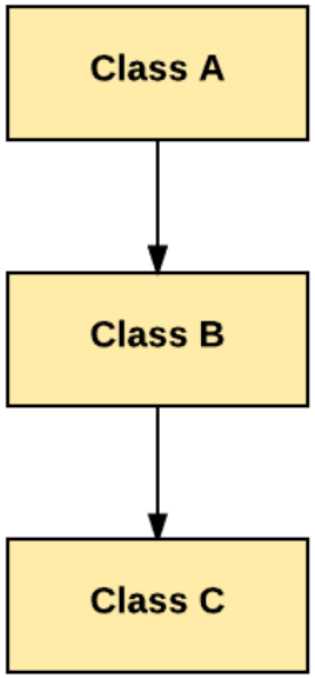
Consider a class A, class B and class C. Now, let class C extend class A and class B. Now, consider a method read() in both class A and class B. The method read() in class A is different from the method read() in class B. But, while inheritance happens, the compiler has difficulty in deciding on which read() to inherit. So, in order to avoid such kind of ambiguity, multiple inheritance is not supported



# Multilevel Inheritance in Java:

In Multi-Level Inheritance in Java, a class extends to another class that is already extended from another class. For example, if there is a class A that extends class B and class B extends from another class C, then this scenario is known to follow Multi-level Inheritance.

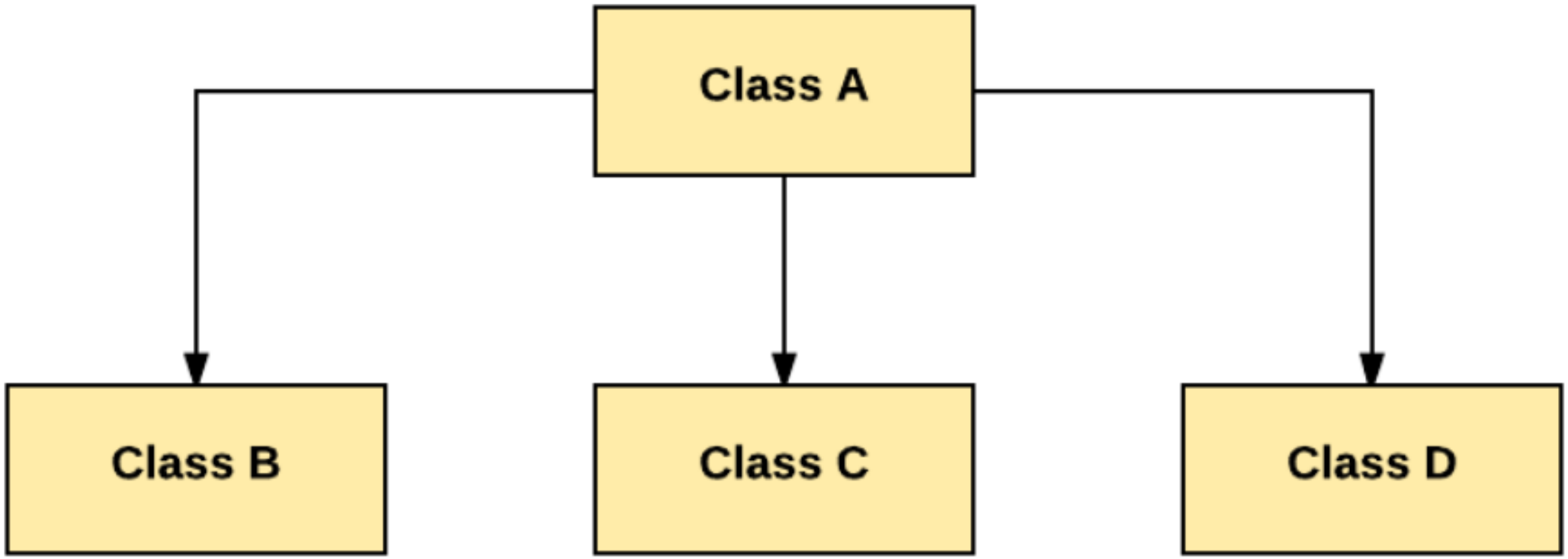
We can take an example of three classes, class Vehicle, class Car and class SUV. Here, tl cle is the grandfather class. The class Car extends class Vehicle and cla ıds class Car.




Multilevel Inheritance

## Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes.

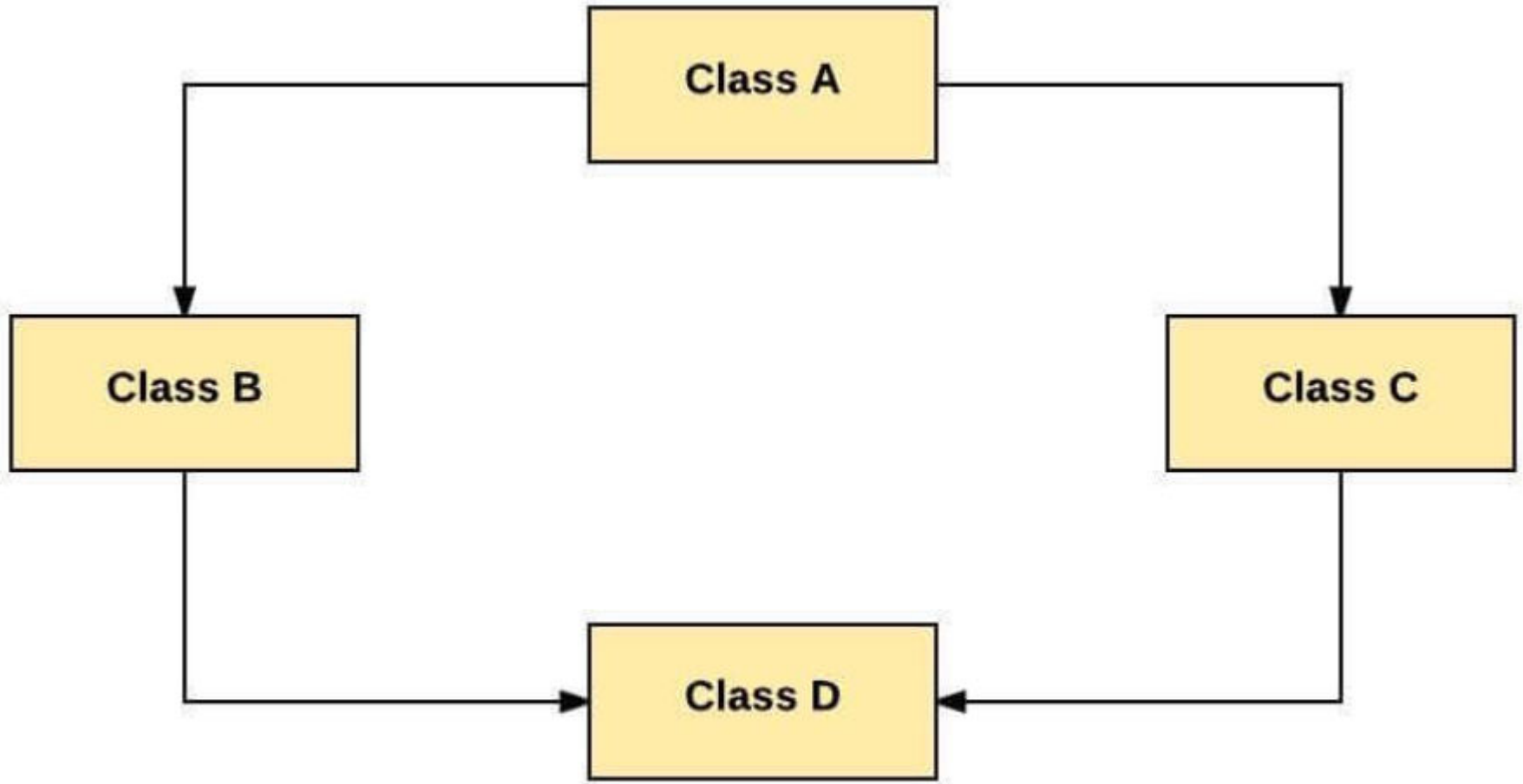



Hierarchical Inheritance

As per above example, Class B, C, and D inherit the same class A.

**Hybrid Inheritance:**

Hybrid inheritance is one of the inheritance types in Java which is a combination of Single and Multiple inheritance.




Hybrid Inheritance in Java

Java doesn't support hybrid/Multiple inheritence

# Super keyword in Java:

- The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.
  - It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
  - It is used to **invoke the superclass** constructor from subclass.

# differentiate the members

```
class Superclass
{
 int i =20;
void display()
{
 System.out.println(“Superclass display method”);
 }
}
class Subclass extends Superclass
{
 int i = 100;
 void display()
 {
 super.display();
 System.out.println(“Subclass display method”);
 System.out.println(“ i value =”+i);
 System.out.println(“superclass i value =”+super.i);
 }
}
class SuperUse
{
 public static void main(String args[])
{
 Subclass obj = new Subclass();
 obj.display();
}
}
```

```
Superclass display method
Subclass display method
 i value =100
superclass i value =20
```

## **invoke the superclass** constructor

super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation.

```
/* 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
```

```java
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}
```

```
1 ankit 45000
```

# Final keyword

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

1. Variable (stop value change of a variable)
2. Method (stop method overriding)
3. Class (stop inheritance)

## Java final variable

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

```
class Bike
{
  final int speedlimit=90;//final variable
  void run()
{
   speedlimit=400;
  }
  public static void main(String args[])
{
  Bike obj=new  Bike();
  obj.run();
  }
 }//end of class
```

- Output: error: cannot assign a value to final variable speedlimit
speedlimit=400;
^

# Java final method:

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

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Output:Compile Time Error
```

# Java final class:

If you make any class as final, you cannot extend it.

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

# Object class in Java

- **Object** class is present in **java.lang** package.
- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- Every class in Java is directly or indirectly derived from the **Object** class.
- If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived.
- The Object class provides some common behaviors to all the objects
- The Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.
- **toString()** :toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Whenever we try to print any Object reference, then internally toString() method is called.

```
Student s = new Student();

// Below two statements are equivalent
System.out.println(s);
System.out.println(s.toString());
```

```java
class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

```
Complex@19821f
```

The default toString() method in Object prints “class name @ hash code”. We can override toString() method in our class to print proper output. For example, in the following code toString() is overridden to print “Real + i Imag” form.

```java
class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    /* Returns the string representation of this Complex number.
       The format of string is "Re + iIm" where Re is real part
       and Im is imagenary part.*/
    @Override
    public String toString() {
        return String.format(re + " + i" + im);
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

```
10.0 + i15.0
```

In general, it is a good idea to override toString() as we get get proper output when an object is used in print() or println().

**hashCode()** : For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find address of an object

Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example,for a Student class we can return roll no. of student from hashCode() method as it is unique.

```
public class Student
{
    static int last_roll = 100;
    int roll_no;

    // Constructor
    Student()
    {
        roll_no = last_roll;
        last_roll++;
    }

    // Overriding hashCode()
    @Override
    public int hashCode()
    {
        return roll_no;
    }

    // Driver code
    public static void main(String args[])
    {
        Student s = new Student();

        // Below two statements are equivalent
        System.out.println(s);
        System.out.println(s.toString());
    }
}
```

```
Student@64
Student@64
```

**getClass()** : Returns the class object of “this” object and used to get actual runtime class of the object.

```
public class Test
{
    public static void main(String[] args)
    {
        Object obj = new String("GeeksForGeeks");
        Class c = obj.getClass();
        System.out.println("Class of Object obj is : "
                           + c.getName());
    }
}
```

```
Class of Object obj is : java.lang.String
```

**finalize()** method : This method is called just before an object is garbage collected. It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks.

```
public class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;

        // calling garbage collector
        System.gc();

        System.out.println("end");
    }

    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

```
366712642
finalize method called
end
```

**clone()** : It returns a new object that is exactly the same as this object.

```
class Test {
    int x, y;
    Test()
    {
        x = 10;
        y = 20;
    }
}

// Driver Class
class Main {
    public static void main(String[] args)
    {
        Test ob1 = new Test();

        System.out.println(ob1.x + " " + ob1.y);

        // Creating a new reference variable ob2
        // pointing to same address as ob1
        Test ob2 = ob1;

        // Any change made in ob2 will
        // be reflected in ob1
        ob2.x = 100;

        System.out.println(ob1.x + " " + ob1.y);
        System.out.println(ob2.x + " " + ob2.y);
    }
}
```

```
10 20
100 20
100 20
```

```
class Test implements Cloneable{

//write clone method in Test class
public Object clone()throws CloneNotSupportedExceptio
n
{
return super.clone();
}
```

```
Test ob2 = (Test)ob1.clone();
```

# Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- An individual can have different relationships with different people. A woman can be a mother, a daughter, a sister, a friend, all at the same time, i.e. she performs other behaviours in different situations.
- There are two types of polymorphism in Java:
  1. static or compile-time polymorphism
  2. Dynamic or runtime polymorphism
- **Compile-Time Polymorphism**
- **Compile-Time polymorphism** in java is also known as **Static Polymorphism.** In this process, the call to the method is resolved at compile-time. Compile-Time polymorphism is achieved through **Method Overloading**.

Method Overloading is when a class has multiple methods with the same name, but the number, types and order of parameters and the return type of the methods are different. Java allows the user freedom to use the same name for various functions as long as it can distinguish between them by the type and number of parameters.

```
public class Addition
{
void sum(int a, int b)
{
int c = a+b;
System.out.println(" Addition of two numbers :" +c); }
void sum(int a, int b, int e)
{
int c = a+b+e;
System.out.println(" Addition of three numbers :" +c); }
public static void main(String[] args)
{
Addition obj = new Addition();
obj.sum ( 30,90);
obj.sum(45, 80, 22);
}
}
```

*Sum of two numbers: 120*

*Sum of three numbers: 147*

```java
class MultiplyFun {

    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 2 double parameter
    static double Multiply(double a, double b)
    {
        return a * b;
    }
}

class Main {
    public static void main(String[] args)
    {

        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(5.5, 6.3));
    }
}
```

```
8
34.65
```

## Runtime Polymorphism

**Runtime polymorphism** in java is also known as **Dynamic Binding or Dynamic Method Dispatch.** In this process, the call to an overridden method is resolved dynamically at runtime rather than at compile-time. Runtime polymorphism is achieved through **Method Overriding**.

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
class Parent {

    void Print()
    {
        System.out.println("parent class");
    }
}

class subclass1 extends Parent {

    void Print()
    {
        System.out.println("subclass1");
    }
}

class subclass2 extends Parent {

    void Print()
    {
        System.out.println("subclass2");
    }
}
```

```
class TestPolymorphism3 {
    public static void main(String[] args)
    {

        Parent a;

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

**Output:**

```
subclass1
subclass2
```

```
class Animal{
  void eat(){
System.out.println("Animals Eat");
}
}
class herbivores extends Animal{
  void eat(){
System.out.println("Herbivores Eat Plants");
}
  }
class omnivores extends Animal{
  void eat(){
System.out.println("Omnivores Eat Plants and meat");
}
  }
class carnivores extends Animal{
  void eat(){
System.out.println("Carnivores Eat meat");
}
  }
class main{
  public static void main(String args[]){
    Animal A = new Animal();
    Animal h = new herbivores(); //upcasting
    Animal o = new omnivores(); //upcasting
    Animal c = new carnivores(); //upcasting
    A.eat();
    h.eat();
    o.eat();
    c.eat();

  }
```

**Output:**

Animals eat

Herbivores Eat Plants

Omnivores Eat Plants and meat

Carnivores eat meat

```
 class Car
{
void run()
{
System.out.println(" running");
}
}
class innova extends Car
{
void run();
{
System.out.println(" running fast at 120km");
}
public static void main(String args[])
{
Car c = new innova();
c.run();
}
}
```

*Running fast at 120 km.*

```
 class grandfather
{
void swim()
{
System.out.println(“ Swimming”);
}
}
class father extends grandfather
{
void swim()
{
System.out.println(“ Swimming in river”);
}
}
class son extends father
{
void swim()
{
System.out.println(“ Swimming in pool”);
}
public static void main(String args[])
{
grandfather f1,f2,f3;
f1 =new grandfather();
f2 = new father();
f3 = new son();
f1.swim();
f2.swim();
f3.swim():
}
}
```

***The output of the following program will be:***

*Swimming*

*Swimming in river*

*Swimming in pool*

# Abstract classes

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- **Abstract class**: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract class can have abstract(method with no body) and non-abstract methods (method with the body).
- It needs to be extended and its method implemented. It cannot be instantiated.
- It can have constructors and static methods also.

## Abstract Method in Java

- A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract class and method**

```
abstract class nameoftheclass{ }
abstract void printStatus();//no method body and abstract
```

- There are two ways to achieve abstraction in java
  1. Abstract class
  2. Interface

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

```
running safely
```

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
}
}
```

```
drawing circle
```

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```