ITM SLS BARODA UNIVERSITY
"Think Big... Think Beyond"

SE Semester-III
Object Oriented Programming with Java
Unit-3
Inheritance and Interfaces

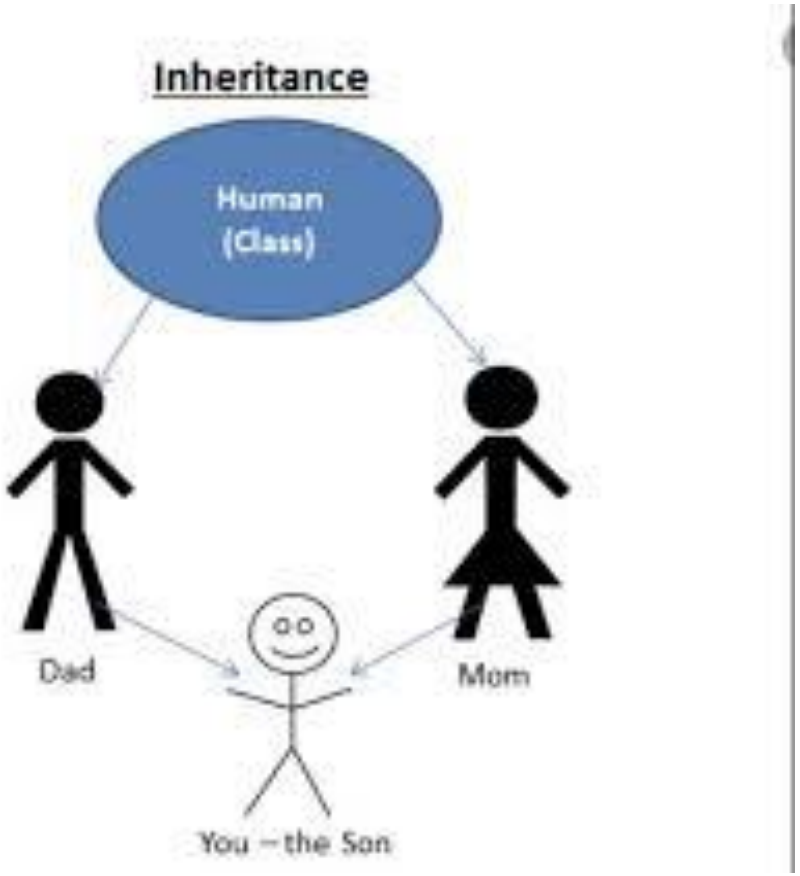
Subject Teacher : Dr. K. V. Metre

https://beginnersbook.com/2013/03/inheritance-in-java/

https://www.geeksforgeeks.org/inheritance-in-java/

www.javatpoint.com/

Inheritance



Inheritance

- **Inheritance** 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).

- 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.

- you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- For Code Reusability.

- Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
- Inheritance allows us to reuse of code, it improves reusability in your java application.
- The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.
- This means that the data members(instance variables) and methods of the parent class can be used in the child class as.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, sub class, extended class, or child class.
- **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.
- **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.

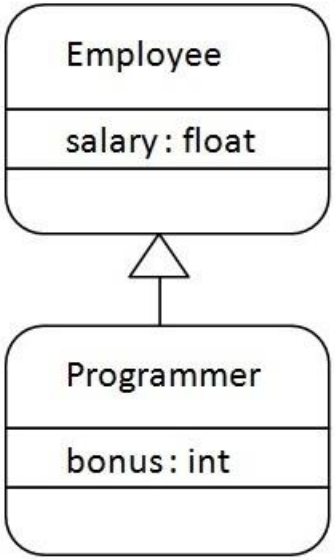
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
  {
    //methods and fields
  }
```

- 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.**

```
class Superclass_name { ..... ..... }

class derived_class_name extends Superclass-name
{
  // body of the derived class.
}
Class Person {
        protected String name;
        protected  int mob;
        protected String address;
}
Class Student extends Person
{
    float per;
    String branch;
    int year;
}
```



Programmer is the subclass

Employee is the superclass.

The relationship between the two classes is **Programmer IS-A Employee**.

It means that Programmer is a type of Employee.

```
class Employee{
 protected 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);

}
}
```

Output :

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

Types of inheritance

- On the basis of class, there can be different types of inheritance

1) Single (Java & C++)

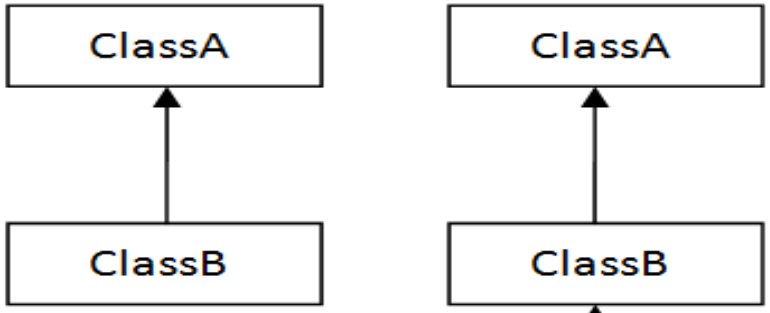
2) Multilevel (Java & C++)

3) Hierarchical. (Java & C++)

4) Multiple (C++)

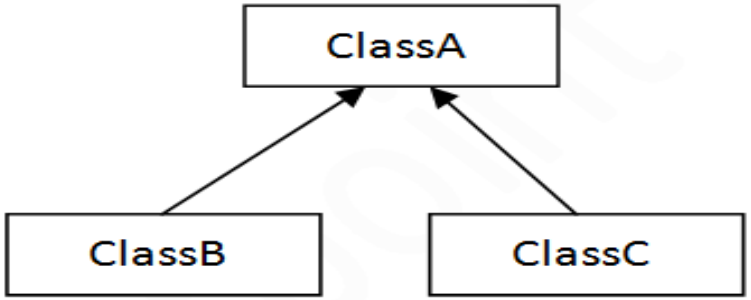
5) Hybrid (C++)

- In Java programming, multiple and hybrid inheritance is supported through interface only.

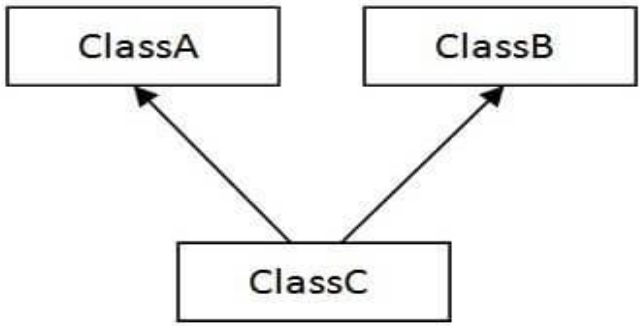


1) Single

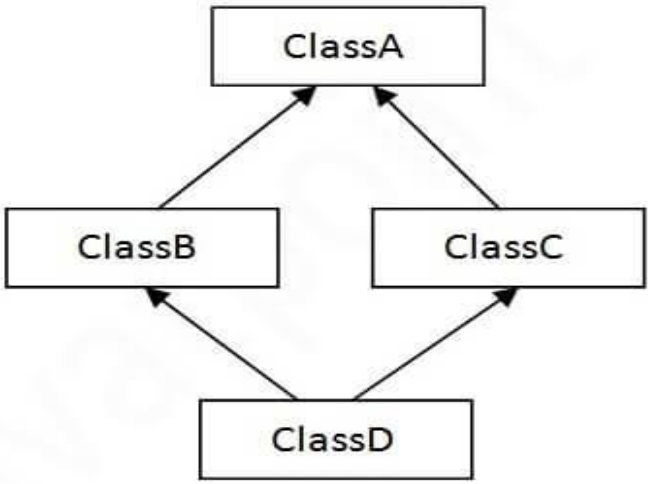
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

Single Inheritance Example

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

```
class Animal{
    void eat(){       System.out.println("eating...");     }
 }
class Dog extends Animal{
    void bark(){    System.out.println("barking...");   }
 }
class Test{
    public static void main(String args[]){
            Dog d=new Dog();
            d.bark();
            d.eat();
            }}
```

Output:

barking...

eating..

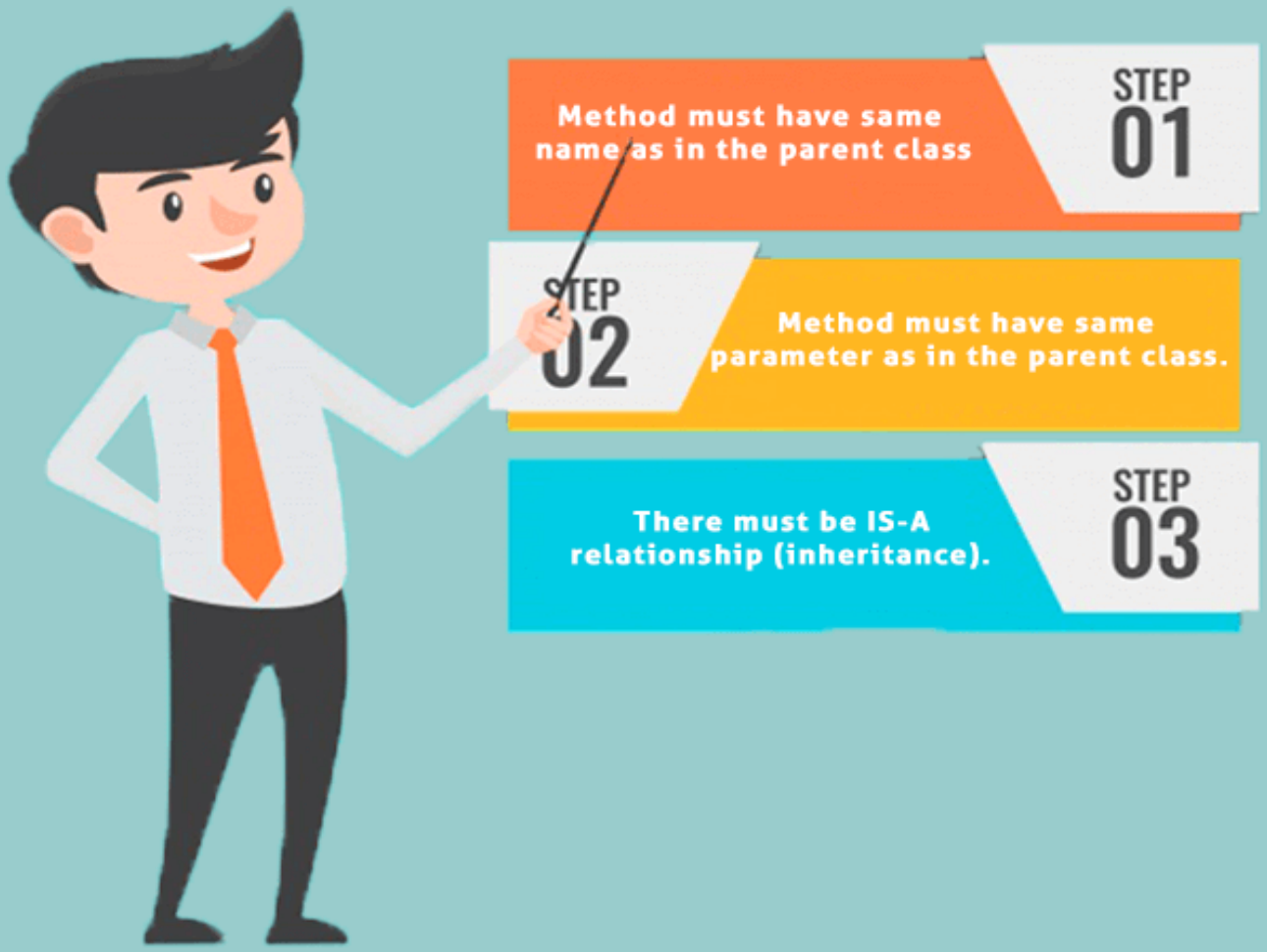
Method Overriding/Polymorphism

- If subclass (child class) has the same method as declared in the super class, it is known as **method overriding in Java**.

- Method signature should be same in super class and sub class.

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding



Method Overriding

```
class Vehicle{
  public void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){

        Bike obj = new Bike(); //creating an instance of child class

        obj.run();   //calling the method with child class instance
  }  // here method overriding is not done
}
```

Output :

```
Vehicle is running
```

```
//Java Program to illustrate the use of Java Method Overriding
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}  }

        //Creating a child class
class Bike extends Vehicle{
        //defining the same method as in the parent class ,method overriding
  void run(){System.out.println("Bike is running ");}

  public static void main(String args[]){
      Bike obj = new Bike();
     obj.run();// here method overriding is done   }
}
```

Output :

```
Bike is running
```

Without method overriding

```
class Person {
   protected String name;
   protected int mob;
  public void  getdata( String n, int m)
     {
         name = n;
          mob = m; }
  public void show()
   {
     System.out.println("Name: " + name + "    Mobile
     No : " + mob);
 }
}
class Student extends Person
{
      String branch;
      int year;
 public void getdata( String n, int m, String b, int y)
   {
name = n;
       mob = m;
      branch = b;      year = y;  }
 public void show()
  {
      System.out.println("Name: " + name + "
      Mobile No : " + mob);
System.out.println("Branch: " + branch + "   Year :
      " + year);
  }

}
public class Main {
  public static void main(String [] args)
  {
    Person p1 = new Person();

    p1.getdata("AAAA", 11111111);
    System.out.println("Person Details \n");
    p1.show();

    Student s1 = new Student();
    s1.getdata("BBBBB", 222222, "IT", 2);
    System.out.println("\n Student Details \n");
    s1.show();

}
}
```

With method overriding

```
class Person {
   protected String name;
   protected int mob;
  public void  getdata( String n, int m)
     {
         name = n;
          mob = m; }
  public void show()
   {
     System.out.println("Name: " + name + "    Mobile
     No : " + mob);
 }
}
class Student extends Person
{
      String branch;
      int year;
 public void getdata( String n, int m, String b, int y)
   {
     super.getdata(n, m);
      branch = b;      year = y;  }
 public void show()
  {
     super.show();
      System.out.println("Branch: " + branch + "
     Year : " + year);
  }  }
public class Main {
  public static void main(String [] args)
  {
    Person p1 = new Person();

    p1.getdata("AAAA", 11111111);
    System.out.println("Person Details \n");
    p1.show();

    Student s1 = new Student();
    s1.getdata("BBBBB", 222222, "IT", 2);
    System.out.println("\n Student Details \n");
    s1.show();

}}
```

Here getdata() and show() methods are overridden. Super keyword is used to call method of super class.

- Output :

Person Details

Name: AAAA Mobile No : 11111111

Student Details

Name: BBBBB Mobile No : 222222

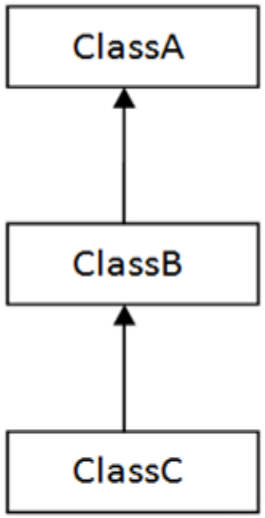
Branch: IT Year : 2

Multilevel Inheritance Example

- 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{
    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 Test{
    public static void main(String args[]){
          Dog d1=new Dog();
           d1.eat();  d1.bark(); //d1.weep(); //error

BabyDog d=new BabyDog();
      d.eat();   d.bark();    d.weep();              }
}
```



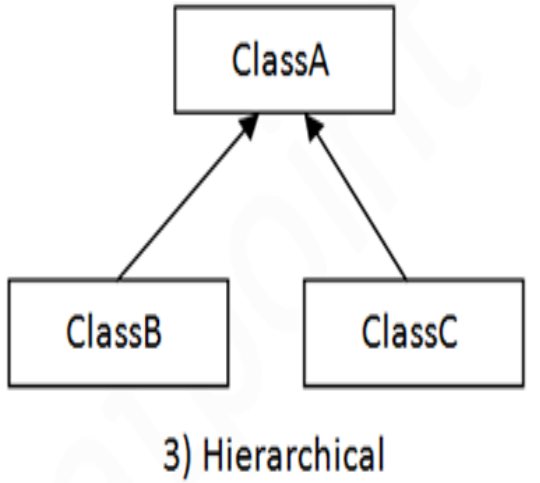
2) Multilevel

- **Hierarchical Inheritance Example**
- 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{
    void eat(){    System.out.println("eating");    }
}
class Dog extends Animal{

     void sound(){  System.out.println("barking");  }
}
class Cat extends Animal{
     void sound(){   System.out.println("meowing");   }
}
class Test{
      public static void main(String args[]){
         Cat c=new Cat();
         c.sound();
         c.eat();
        Dog d=new Dog();    d.sound();
}}
```

Output:
meowing
eating



3) Hierarchical

```
 class Super{
        public void msgsuper() { System.out.println("Executing Superclass"); }
    }
class Sub1 extends Super {
        public void msgsub1() { System.out.println("Executing Subclass one"); }
}
class Sub2 extends Super {
        public void msgsub2() { System.out.println("Executing Subclass two"); }}
Public class Main{
public static void main(String[] args)  {
        Sub1 c = new Sub1();
    c.msgsuper();
    c.msgsub1();
    System.out.println();
    Sub2  j = new Sub2 ();
     j.msgsuper()
    j.msgsub2();
    }}
}
```

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with public visibility are inherited; those with private visibility are not inherited.
- But public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: protected.
- The protected modifier allows a member of a base class to be inherited into a child
- Protected visibility provides more encapsulation than public visibility does
- However, protected visibility is not as tightly encapsulated as private visibility

Access Specifiers

```
/* private members remain private to their class. This program will not compile. */
 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; // ERROR, j is not accessible here
 }
}
class Access {
     public static void main(String args[]) {
            B b = new B();
            b.setij(10, 12);
            b.sum();
            System.out.println("Total is " + b.total);
}}
```

Output :

Access.java:10: error: j has private access in A
total = i + j; // ERROR, j is not accessible here

Access Specifiers

```
class Person {
   private String
    name;
   protected int
    mob;
  public void
    getdata( String n,
    int m)
{      name = n;
       mob = m; }
  public void show()
   {
    System.out.printl
    n("Name: " +
    name + "
    Mobile No : " +
    mob);  }}
```

```
class Student extends
     Person
{   String branch;
    int year;
  public void getdata(
     String n, int m, String
     b, int y)   {
      super.getdata(n, m);
    branch = b;   year = y;  }
  public void show(){
 super.show();
System.out.println("Branc
     h: " + branch + "   Year
     : " + year);
  }
  }
```

```
public class Main {
  public static void
main(String [] args)
  {
    Person p1 = new Person();

    p1.getdata("AAAA",
11111111);

System.out.println("Person
Details \n");
    p1.show();

    Student s1 = new
Student();
    s1.getdata("BBBBB",
222222, "IT", 2);
    System.out.println("\n
Student Details \n");
    s1.show();

}
}
```

Multilevel Inheritance

```
Class Rect {
Protected int len;
Protected int breadth;
public Rect(){
   len=breadth=0;   }
public Rect(int l, int b){
   len=l;
  breadth=b;}
Public void show(){
System.out.println("Length="+len+"
    Breadth=" +breadth);} }
class Box extends Rect {
    double depth;
          // construct clone of an object
     public Box() {
     super();
       depth=0;}
```

```
class BoxWeight extends Box {
double weight; // weight of box
BoxWeight(double w, double h, double d, double m)
 {
    super(w,h,d);
     weight = m; } }
Void show()
    { super.show();}}
class DemoBoxWeight {
public static void main(String args[]) {
     BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
     BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
      System.out.println("Weight of mybox1 is " +
      ybox1.weight);
     System.out.println();
               vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " +
      mybox2.weight);
}}
```

Output :

```
 Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

A Superclass Variable Can Refer a Subclass Object

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

```
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);
        System.out.println(); // assign BoxWeight reference to Box reference
        plainbox = weightbox;
        vol = plainbox.volume(); // OK, volume() defined in Box
        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); } }
```

Using super

- This usage has the following general form: super.member
- 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

```
class Super
{    int value = 111;
 }
class Sub1 extends Super{
     int value = 222;
public void print()
{    System.out.println("From superclass:" + super.value);
     System.out.println("From subclass:" + value);
} }
public class Main{
public static void main(String s[])
{    System.out.println("Using super:");
     Sub1 ob = new Sub1();
     ob.print();
}   }
```

Output : Using super:

From superclass:111

From subclass:222

Using super keyword

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The **`super`** reference can be used to refer to the parent class, and often is used to invoke the parent's constructor.
- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the **`super`** reference to call the parent's constructor
- The **`super`** reference can also be used to reference other variables and methods defined in the parent's class

Constructor using Super

- Using super to Call Superclass Constructors
- A subclass can call a constructor defined by its superclass by use of the following form of super:
- super(arg-list);
- Here, arg-list specifies any arguments needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass' constructor.

Without using super keyword

```
class Box {
      double width;
      double height;
      double depth;
// construct clone of an object
 Box(Box ob) { // pass object to constructor
       width = ob.width;
      height = ob.height;
      depth = ob.depth; }
Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d; }
Box() {
      width = height =  depth = 10;}
Box(double len) {
      width = height = depth = len; }
volume double volume() {
      return width * height * depth; }
}
```

```
class BoxWeight extends Box {
double weight; // weight of box
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, 0.076);
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);
}}
```

Output :

```
 Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

using super keyword

```
class Box {
        double width;
        double height;
        double depth;
// construct clone of an object
 Box(Box ob) { // pass object to constructor
         width = ob.width;
        height = ob.height;
        depth = ob.depth; }
Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d; }
Box() {
        width = height =  depth = 10;}
Box(double len) {
        width = height = depth = len; }
volume double volume() {
        return width * height * depth; }
}
```

```
class BoxWeight extends Box {
double weight; // weight of box
BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight; } // are specified
 BoxWeight(double w, double h, double d, double m) {
         super(w, h, d); // call superclass constructor
        weight = m; } // default constructor
BoxWeight() {
     super();
     weight = -1; }
// constructor used when cube is created
 BoxWeight(double len, double m) {
      super(len);
      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, 0.076);
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);
}}
```

Output :

```
 Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

```
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20,
    15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4,
    0.076);
 BoxWeight mybox3 = new BoxWeight();
 BoxWeight myclone = new
    BoxWeight(mybox1);
 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);
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();
 vol = myclone.volume();
    System.out.println("Volume of myclone is
    " + vol);
System.out.println("Weight of myclone is " +
    myclone.weight);
 System.out.println();
 vol = mycube.volume();
    System.out.println("Volume of mycube is "
    + vol);
System.out.println("Weight of mycube is " +
    mycube.weight);
 System.out.println(); } }
```

output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
 Weight of mybox3 is -1.0
 Volume of myclone is 3000.0
 Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

Method overriding

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
- A parent method can be invoked explicitly using the **`super`** reference
- If a method is declared with the **`final`** modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Method Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same name
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

- 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. The version of the method defined by the superclass will be hidden.

```
class A {
    int i, j;
 A(int a, int b) {
    i = a; j = b; } // display i and j
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; }
// display k – this overrides  show() in A
void show() {
    System.out.println("k: " + k); }
}
```

```
class Override {
public static void main(String args[]) {
      B b1 = new B(1, 2, 3);

       b1.show();
      // this calls show() in B
}
}
```

Output
k: 3

final keyword

- Final keyword can be used for instance variable, method and class.

final variable Example

```
class Bike{
 final int speedlimit=90;//final variable
 void run(){
      speedlimit=400; //  }  It can't be changed because final variable

                                once assigned a value can never be changed.

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

Test it Now

Output:

Compile Time Error

Java final method

- If you make any method as final, 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");
}  // can not override final method

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

  public static void main(String args[]){
  Honda h1= new Honda();
  h1.run();
  }
}
```

Output:
Compile Time Error

Stop inheritance

Stopping Inheritance with Final Keyword

- **Definition:**
  - When a class is declared as final, it cannot be subclassed /inherited

Example:

```
final class FinalClass { }

Class A extends FinalClass {
}  // not possible
```

Interface in Java

Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.
- Syntax:

interface <interface_name>{

// declare constant fields

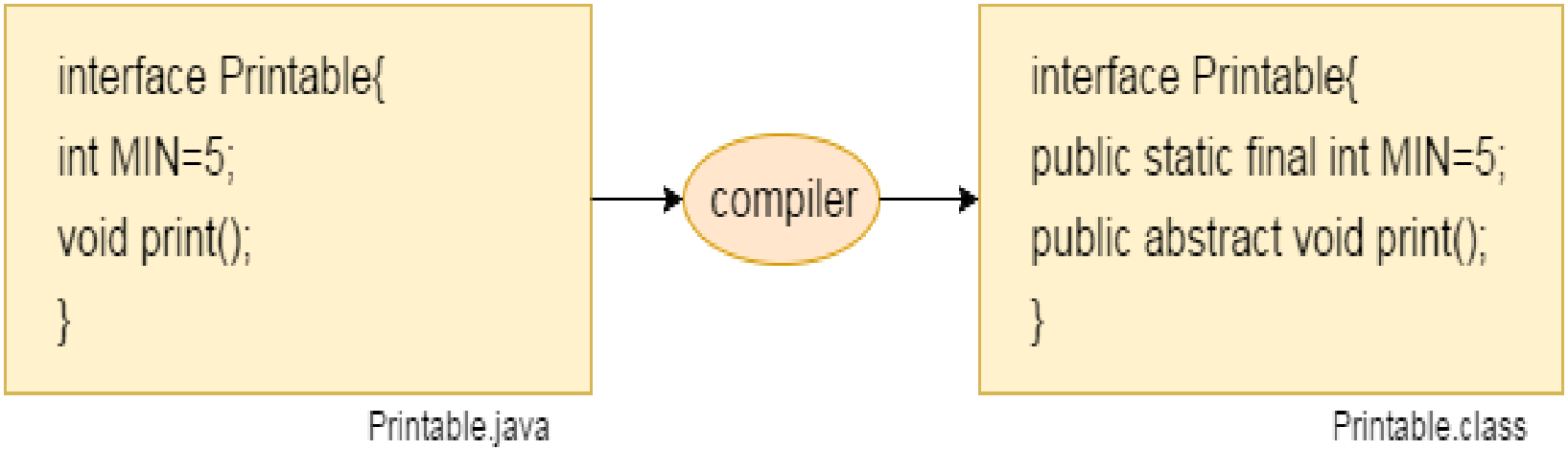
// declare methods that abstract

// by default.

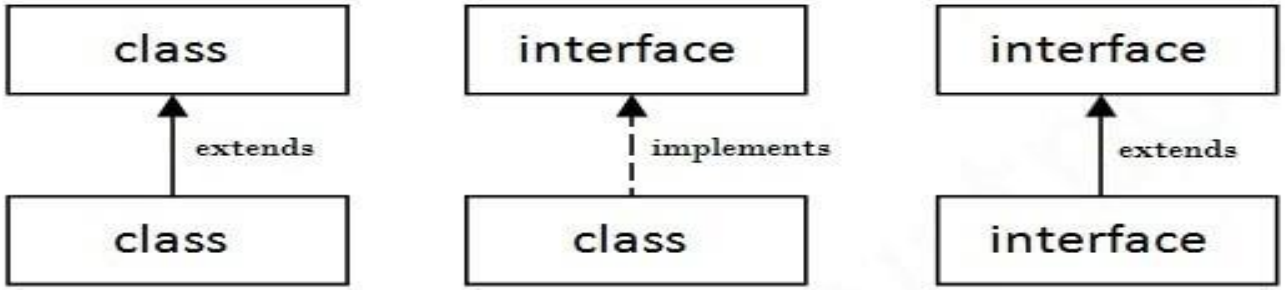
}

e.g.

```
interface Animal {
 void sound();
    }
```



The relationship between classes and interfaces



```
interface printable{
   void print();
}
class A implements printable{  //method overriding
      public void print(){System.out.println("Hello");}

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

Output:

Hello

Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes.

```
interface Drawable{
    void draw();    }
class Rectangle implements Drawable{
    public void draw(){
            System.out.println("drawing rectangle");}
    }
class Circle implements Drawable{
    public void draw(){      System.out.println("drawing circle");}  }
class TestInterface1{
    public static void main(String args[]){
    Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
    d.draw();
Drawable d=new  Rectangle();// object is provided by method e.g. getDrawable()
    d.draw();        }}
```

Output: drawing circle
drawing rectangle

Implementing class from interface

```
interface Bank{
      float rateOfInterest();
}
class SBI implements Bank{
            public float rateOfInterest() {return 9.15f;}
     }
class PNB implements Bank{
     public float rateOfInterest() {   return 9.7f;}
}
class TestInterface{
     public static void main(String[] args){
            Bank b=new SBI();
               System.out.println("ROI: "+b.rateOfInterest());
           Bank b=new PNB();
               System.out.println("ROI: "+b.rateOfInterest());}
}
```

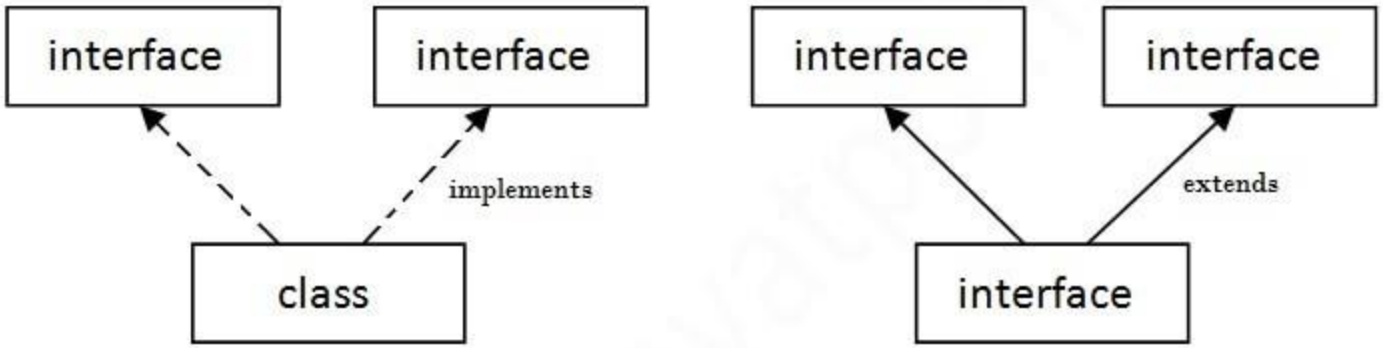
Output:

ROI: 9.15

ROI: 9.7

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

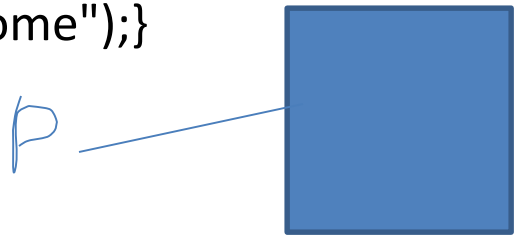


Multiple Inheritance in Java

```
interface Printable{
    void print();
}
interface Showable{
    void show();
}
class A implements Printable, Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

public static void main(String args[]){

        A obj = new A();
        obj.print();
        obj.show();
    }
}
```



Output:

Hello
Welcome

Multiple inheritance is not supported through class in java, but it is possible by an interface,
multiple inheritance is not supported in the case of class because of ambiguity.
However, it is supported in case of an interface because there is no ambiguity.
It is because its implementation is provided by the implementation class.

```
interface Printable{
    void print();
}
interface Showable{
    void print();
}

class TestInterface implements Printable, Showable {
    public void print(){
        System.out.println("Hello");}
    public static void main(String args[]){
        TestInterface3 obj = new TestInterface3();
        obj.print();
 } }
```

Output:
Hello

Static Method in Interface

Since Java 8, we can have static method in interface.

```
interface Drawable{
    void draw();
    static int cube(int x){return x*x*x;}  }
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}
  class TestInterfaceStatic{
            public static void main(String args[]){
            Drawable d=new Rectangle();
            d.draw();
            System.out.println(Drawable.cube(3));  //can be called using
    //interface only
} }
```

Output:

```
drawing rectangle
27
```

Polymorphism in Java

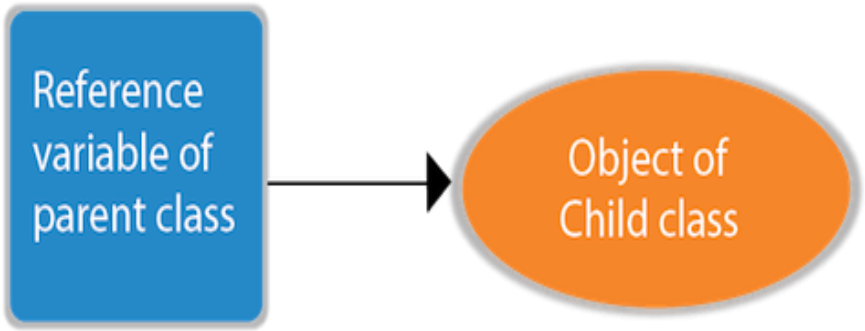
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.
- We can perform polymorphism in java by method overloading and method overriding.

Runtime polymorphism or Dynamic method dispatch

- 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.



class A{}

class B **extends** A{}

A a=**new** B();//upcasting

For upcasting, we can use the reference variable of class type or an interface type.

Dynamic dispatch method program on next slide

```
interface Shape {
void area ();
double pi = 3.14;
}
class Circle implements Shape {
double  r;
Circle (double radius)
{r = radius; }
public void area () {
System.out.println ("Area of a
    circle is : " + pi*r*r );
} }
class Rectangle implements
    Shape {
double l,b;
Rectangle (double length, double
    breadth)
{ l = length; b = breadth;. }
public void area () {
System.out.println ("Area of a Rectangle is
    : " + l*b );
} }
class InterfaceDemo {
public static void main (String args[])
{
Circle ob1 = new Circle (10);
ob1.area (); //dynamic method dispatch
Rectangle ob2 = new Rectangle (10,10);
ob2.area (); //dynamic method dispatch


} }
```

Output :

Area of a circle is : 314

Area of a Rectangle is : 100

instanceof operator

- **instanceof' operator** is used to test whether an object is an instance of a specified type (class or sub - class or interface).
- It returns true or false.

```
class Student{  }
class Test{
  public static void main( String args[ ] )
  {
    // declaring an object 's' of the student class

    student s = new student( ) ;

    // checking whether s is an instance of the student class
    Boolean str = s instanceof student;
    // printing the string value
    System.out.println( str ) ;
}
```

Output : true

- An object of subclass type is also a type of parent class.

```
class Teacher { }
public class Student extends Teacher
{
   public static void main( String args[ ] )
   {
     // declaring the object of the class 'Student'

     Student s = new Student( ) ;

     // checking whether the object s is the instance of the parent class ' Teacher '

     Boolean str = s instanceof Teacher ;

     // printing the boolean value

     System.out.println( str ) ;
   }
}
```

Output:
true

Difference between Abstract class and interface

S.N	Abstract class	interface
1	Abstract class can **have abstract and non-abstract** methods	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2	Abstract class **doesn't support multiple inheritance**.	interface **supports multiple inheritance**.
3	Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4	Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5	The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface
6	An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7	A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Inheritance and Substitutability

- Idealization of inheritance:instance of "subclass" can substitute for instance of parent In Java, substitutability by implementing interfaces as well as subclass
- Abstract concept captured with two rules of thumb:
- ***is-a* relation** "A Car *is-a* Vehicle" sounds right Natural for Car to inherit from mammal
- ***has-a* relation**"A car *is-a(n)* engine" sounds wrong Not natural to use inheritance But "a car *has-a(n)* engine" sounds right
- Can use composition (aggregation)

Benefits of Inheritance :

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

Conclusion

- Importance of Inheritance and Interfaces in code organization.
- Enhancements in code reusability and maintenance.