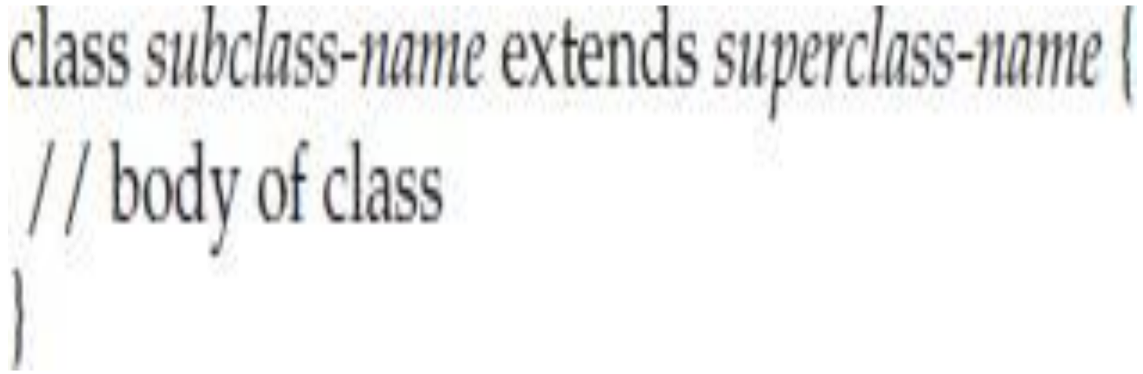
UNIT-III

Inheritance, Interface

Unit – III	**Contact Hours = 8 Hours**
Inheritance: Inheritance basics, member access and inheritance, constructors, and inheritance, using super, multilevel hierarchy, when are constructors executed, superclass reference and subclass objects, method overriding, polymorphism, using abstract classes. **Interfaces:** interface fundamentals, creating, implementing, and using interfaces, implementing multiple interfaces.	

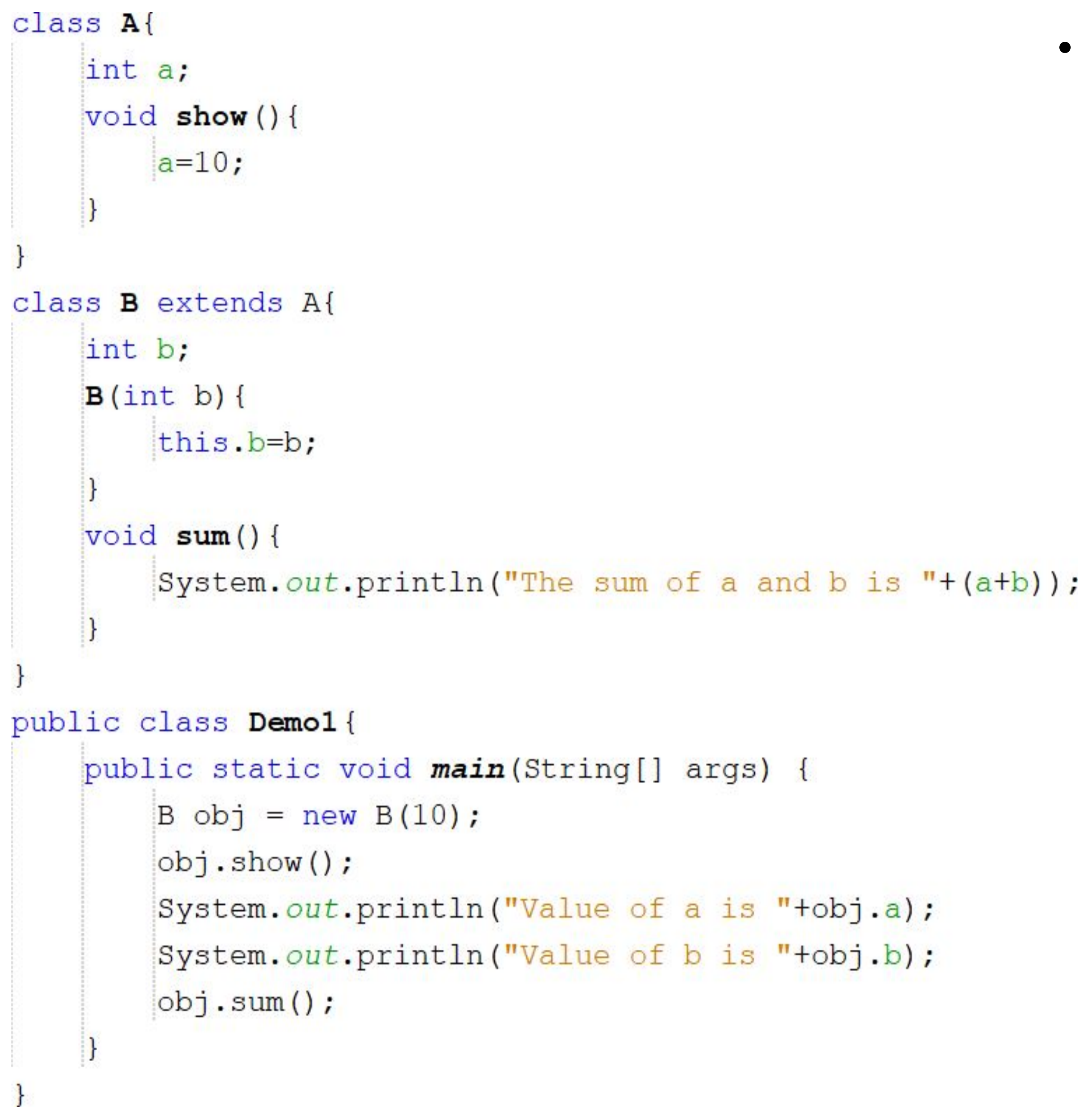
Inheritance basics

- In Java, a class that is inherited is called a superclass and the class that does the inheriting is called a subclass
- A subclass is a specialized version of a superclass which inherits all of the variables and methods defined by the superclass and adds its own elements
- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- The general form of a class declaration that inherits a superclass is shown here:



```
class subclass-name extends superclass-name {
 // body of class
}
```

- Ex: The following program creates a superclass called A and a subclass called B



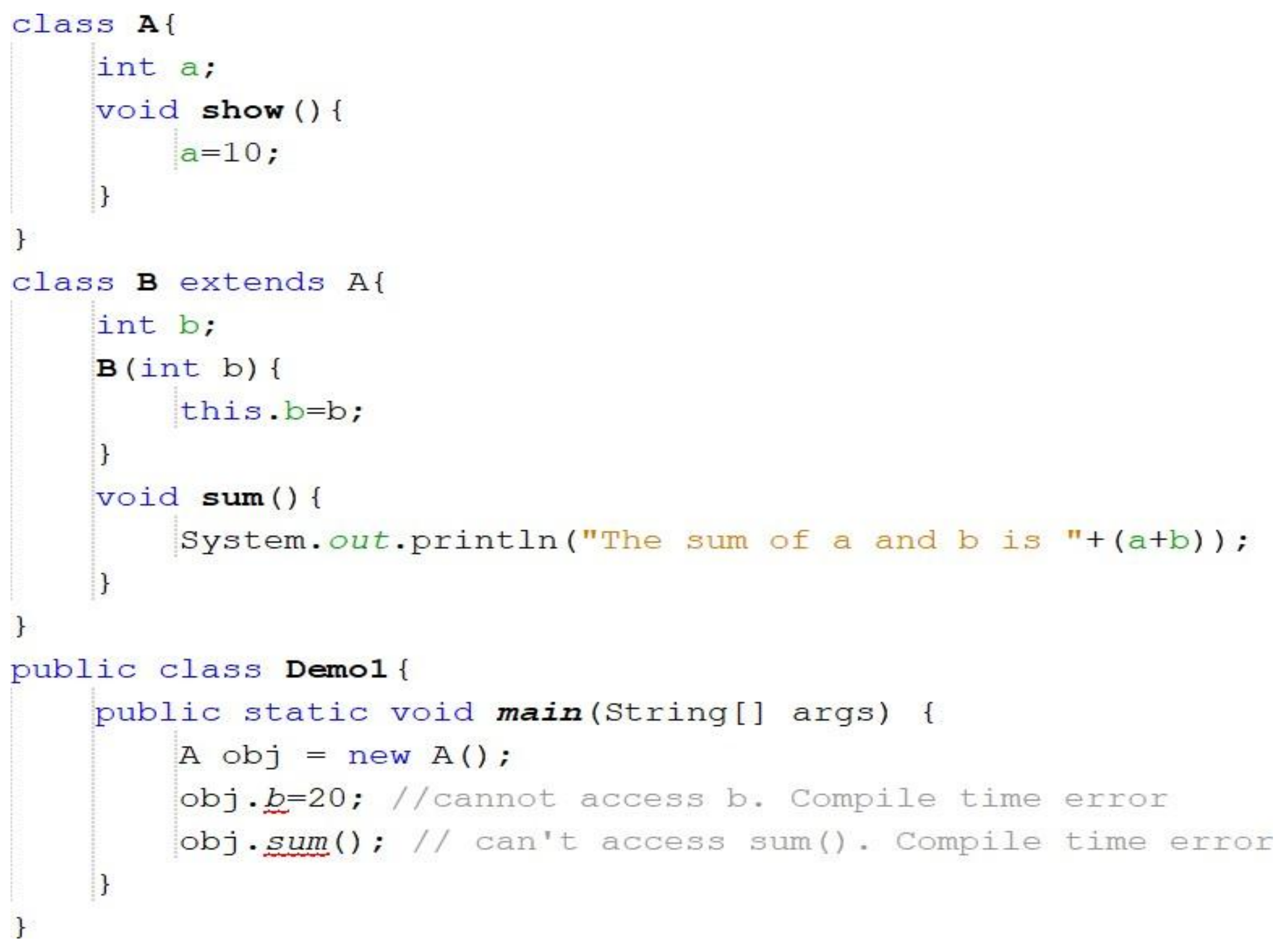
```
class A{
    int a;
    void show(){
        a=10;
    }
}
class B extends A{
    int b;
    B(int b){
        this.b=b;
    }
    void sum(){
        System.out.println("The sum of a and b is "+(a+b));
    }
}
public class Demo1{
    public static void main(String[] args) {
        B obj = new B(10);
        obj.show();
        System.out.println("Value of a is "+obj.a);
        System.out.println("Value of b is "+obj.b);
        obj.sum();
    }
}
```

- The subclass **B** includes all of the members of its superclass, **A**. **B** inherits variable **a** and method **show().** This is why **obj** can access **a** and invoke **show( )** directly as if they were part of **B**. Also, it adds its own variable **b** and method **sum( )**. **sum()** can access variable a directly as if it was a part of class **B**

Output:

```
Value of a is 10
Value of b is 10
The sum of a and b is 20
```

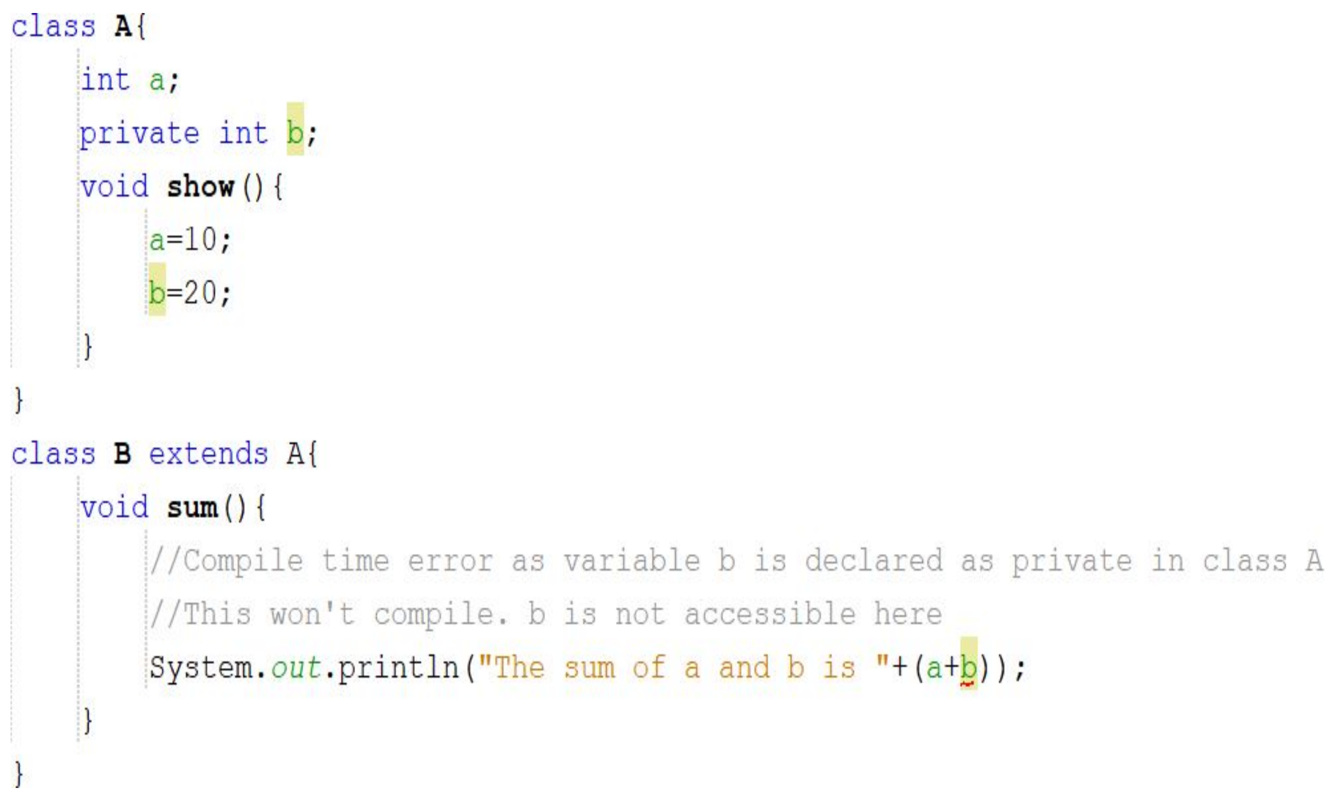
- However, class **A** cannot access variable **b** and method **sum( )** defined in its **subclass B** because the superclass has no knowledge about its subclasses**.** Attempting to do so will result in compile time error.



```
class A{
    int a;
    void show(){
        a=10;
    }
}
class B extends A{
    int b;
    B(int b){
        this.b=b;
    }
    void sum(){
        System.out.println("The sum of a and b is "+(a+b));
    }
}
public class Demo1{
    public static void main(String[] args) {
        A obj = new A();
        obj.b=20; //cannot access b. Compile time error
        obj.sum(); // can't access sum(). Compile time error
    }
}
```

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
- Example:



```
class A{
    int a;
    private int b;
    void show(){
        a=10;
        b=20;
    }
}
class B extends A{
    void sum(){
        //Compile time error as variable b is declared as private in class A
        //This won't compile. b is not accessible here
        System.out.println("The sum of a and b is "+(a+b));
    }
}
```

This program will not compile because the reference to **b** inside the **sum( )** method of **B** causes an access violation. Since **b** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

Example of member access and inheritance in Java

```
// Parent class
class Animal {
  public String name;

  public Animal(String name) {
    this.name = name;
  }

  public void speak() {
    System.out.println("Animal is
speaking.");
  }
}
```

```
// Child class
class Cat extends Animal {
  public int age;

public Cat(String name, int age) {
    super(name);
    this.age = age;
  }
 public void speak() {
    System.out.println("Meow!");
  }

  public void jump() {
    System.out.println("Cat is
jumping.");
  }
}
```

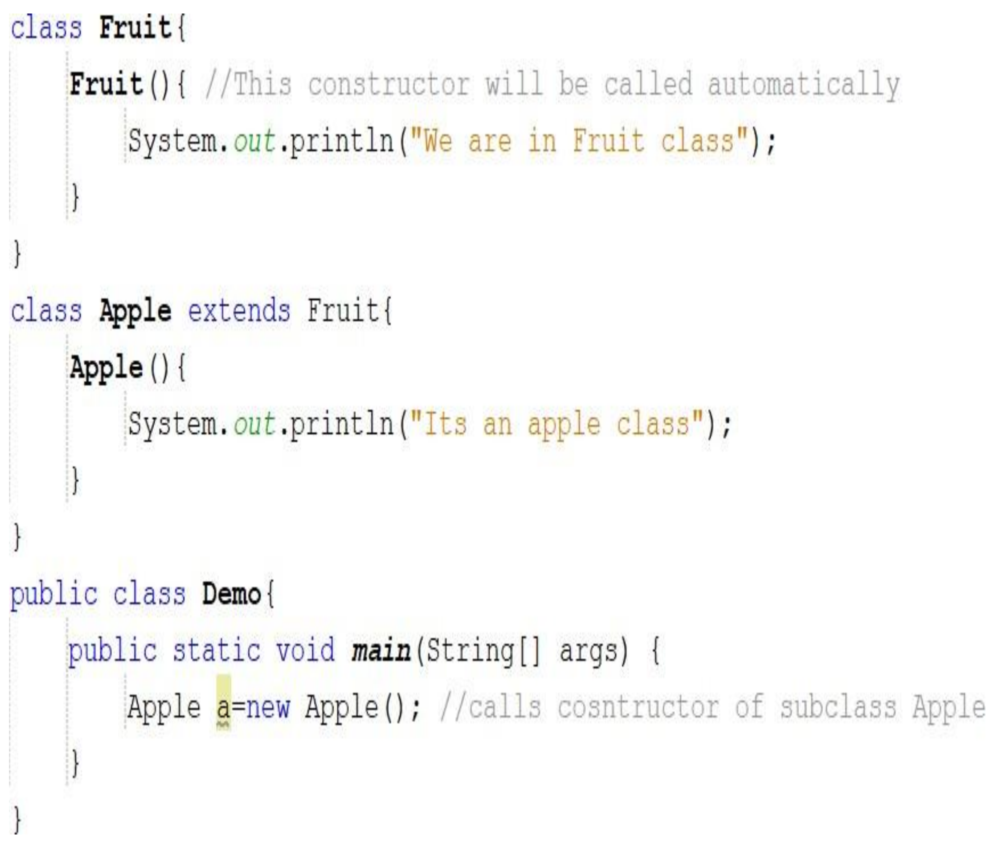
```
// Main class
public class Main {
  public static void main(String[] args) {
    // Create a new Cat object
    Cat cat = new Cat("Fluffy", 3);

    // Access parent class member variable
    System.out.println(cat.name);

    // Access child class member variable
    System.out.println(cat.age);

    // Call parent class method
    cat.speak();

    // Call child class method
    cat.jump();
  }
}
```

- In this example, the **Animal** class is the parent class and the **Cat** class is the child class that inherits from **Animal**.
- The **Cat** class adds an **age** member variable and a **jump()** method to the **Animal** class.
- In the main() method, we create a new Cat object and access its member variables and methods.
- We can access the parent class name member variable using the cat.name syntax, and we can access the child class age member variable using the cat.age syntax.
- We can also call the speak() method on the cat object, which is inherited from the parent class, and the jump() method, which is specific to the child class.

Constructors and Inheritance

- Constructors cannot be inherited by the subclasses.
- It is possible for both superclass and subclasses to have their own constructors.
- The constructor in the **superclass** constructs the **superclass portion of the object** and the constructor in the **subclass** constructs the **subclass portion of the object**
- When only the subclass defines a constructor, simply construct subclass object. The superclass portion of the object is constructed automatically using its **default constructor.**

- For example,



```
class Fruit{
    Fruit(){ //This constructor will be called automatically
        System.out.println("We are in Fruit class");
    }
}
class Apple extends Fruit{
    Apple(){
        System.out.println("Its an apple class");
    }
}
public class Demo{
    public static void main(String[] args) {
        Apple a=new Apple(); //calls cosntructor of subclass Apple
    }
}
```

Output:

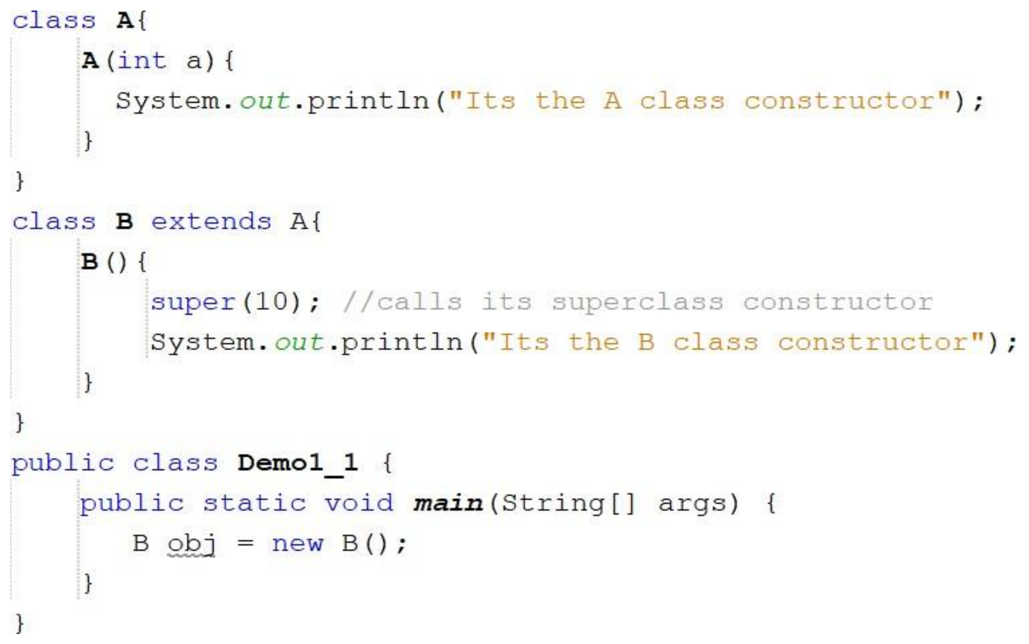
```
We are in Fruit class

Its an apple class
```

- In the above program, **subclass constructor** is invoked through **subclass object** and the **superclass constructor** is called **automatically**

Using super

- **super** has two general forms.
  - The first calls the superclass's constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.
- **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 it
- **super( )** must always be the **first statement** executed inside a **subclass's constructor**

- Example:



```
class A{
    A(int a){
      System.out.println("Its the A class constructor");
    }
}
class B extends A{
    B(){
        super(10); //calls its superclass constructor
        System.out.println("Its the B class constructor");
    }
}
public class Demo1_1 {
    public static void main(String[] args) {
       B obj = new B();
    }
}
```

Output:

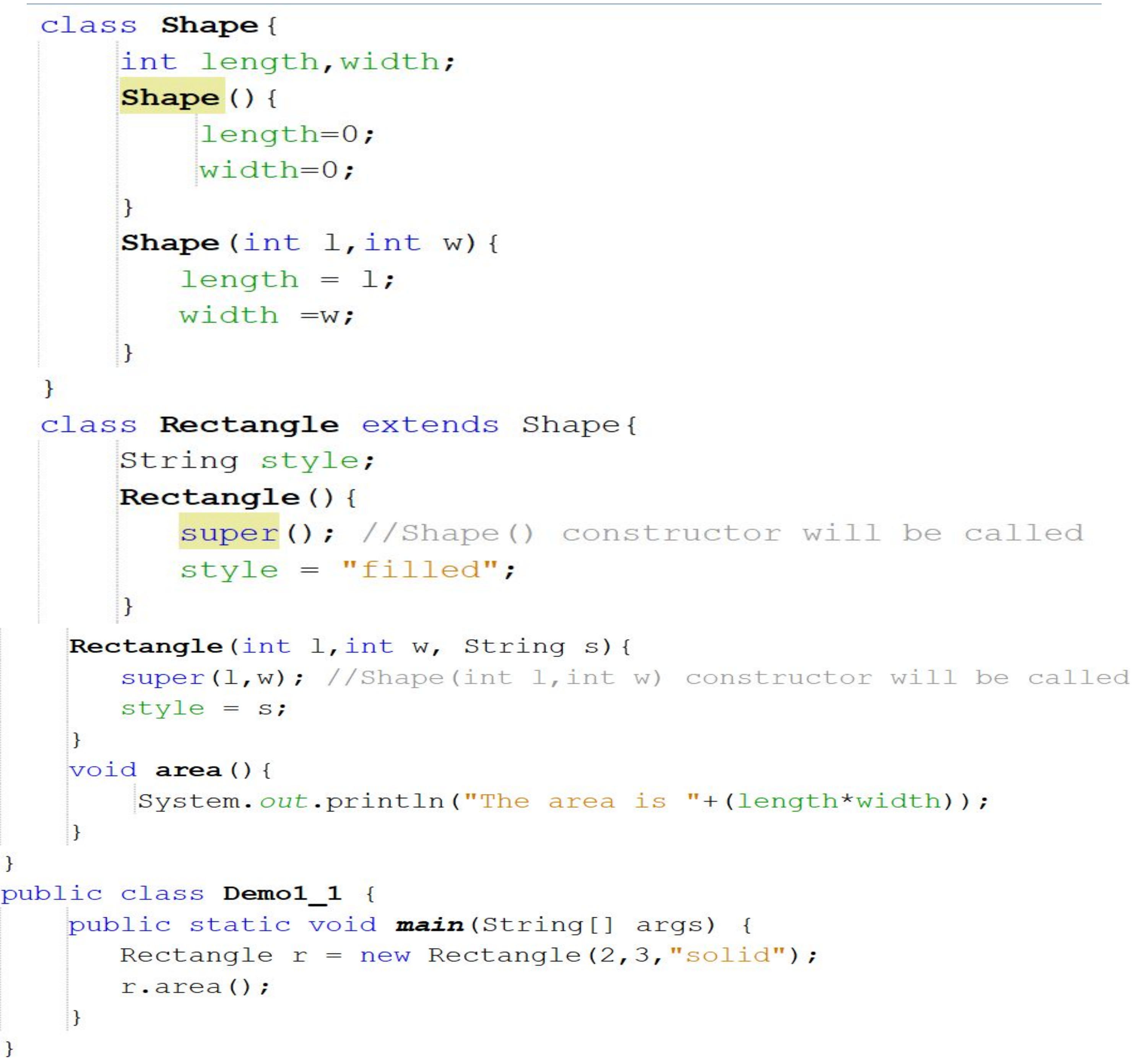
```
  Its the A class constructor

  Its the B class constructor
```

- In the above program, if the **subclass constructor** does not invoke the **superclass constructor** explicitly using **super( )** then a compile time error occurs.
- Because, subclass constructor automatically calls the **default constructor** of its superclass.
- Since the **superclass** defines a parameterized constructor, its **default constructor** will no longer be available which results in **compile time error**

```
class A{
    A(int a){
      System.out.println("Its the A class constructor");
    }
}
class B extends A{
    B(){
      //does not call superclass constructor explicitly
       System.out.println("Its the B class constructor");
    }
}
public class Demo1_1 {
    public static void main(String[] args) {
      B obj = new B();
    }
}
```

Output

```
COMPILATION ERROR :
-------------------------------------------------------------
Demo1 1.java:[8,8] constructor A in class A cannot be applied to given types;
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```



```
class  Shape{
       int  length,width;
       Shape(){
              length=0;
              width=0;
       }
       Shape(int  l,int  w){
            length  =  l;
            width  =w;
       }
}
class  Rectangle  extends  Shape{
       String  style;
       Rectangle(){
            super();  //Shape()  constructor  will  be  called
            style  =  "filled";
       }
    Rectangle(int  l,int  w,  String  s){
       super(l,w);  //Shape(int  l,int  w)  constructor  will  be  called
       style  =  s;
    }
    void  area(){
          System.out.println("The  area  is  "+(length*width));
    }
}
public  class  Demo1_1  {
    public  static  void  main(String[]  args)  {
       Rectangle  r  =  new  Rectangle(2,3,"solid");
       r.area();
    }
}
```

- In this program, Rectangle() constructor calls super() with the parameters l and w. This causes Shape() constructor to be called, which then initializes length and width using these values.

- Rectangle does not initialize these values on its own. It calls the its superclass constructor which does the job. Rectangle need to initialize only the value which is unique to it i.e. “style”

- Any form of constructor defined by the superclass can be called by super(). The constructor executed will be the one that matches the arguments

Using super to Access Superclass Members

- 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.

```
// Using super to overcome name hiding.
class A {
  int i;
}
```

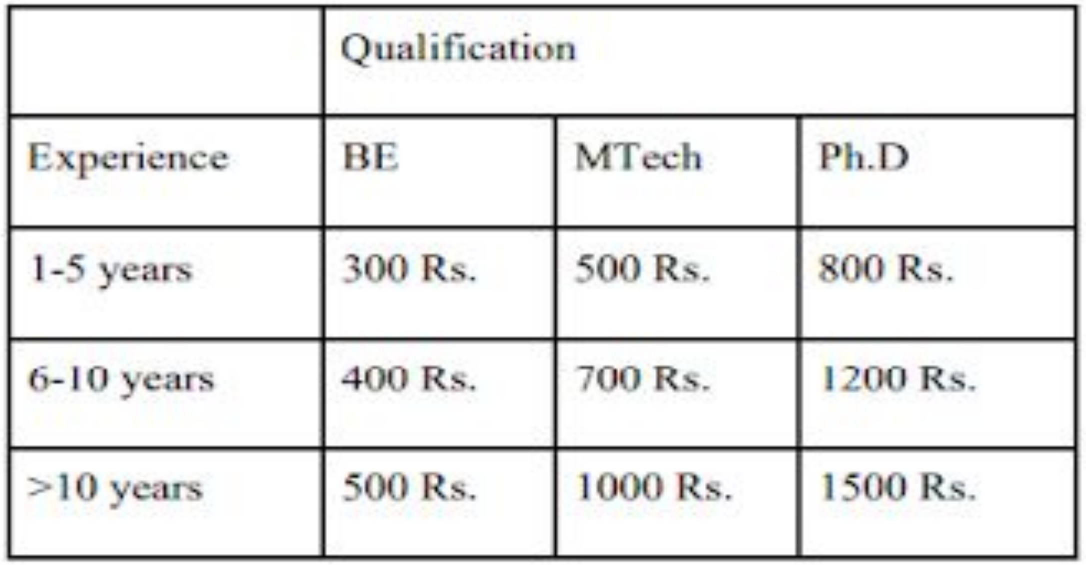
```
// Create a subclass by extending class A.
class B extends A {
  int i; // this i hides the i in A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in 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
```

TW-4

- A company has two types of employees – Full Time and Par time. The company records for each employee his/her name, age, address, salary and gender. Given the basic salary of the Full Time employee the components of his/her gross salary are: Dearness allowance – 75% of basic salary, HRA – 7.5% of basic salary, IT – 10% of basic. The salary of a Part time employee is dependent on the qualification, experience, number of working hours and the rate per hour, as below:



	Qualification		
Experience	BE	MTech	Ph.D
1-5 years	300 Rs.	500 Rs.	800 Rs.
6-10 years	400 Rs.	700 Rs.	1200 Rs.
>10 years	500 Rs.	1000 Rs.	1500 Rs.

```
class Employ{
    String name,address,gender;
    int age;
    double sal;
    Employ(String name,int age,String address,String gender){
        this.name=name;
        this.age=age;
        this.address=address;
        this.gender=gender;
    }
void show(){
        System.out.println("Name:"+name);
        System.out.println("Age:"+age);
        System.out.println("Address:"+address);
        System.out.println("Gender:"+gender);
        System.out.println("Salay:"+sal);
    }

}
```

```
class FTEmploy extends Employ{
    int basSal;
    FTEmploy(String name,int age,String address,String gender,int basSal){
        super(name,age,address,gender);
        this.basSal=basSal;
    }

    void calSal(){
        sal=(basSal+basSal*0.75+basSal*0.075-basSal*0.1);
    }
}
```

```
class PTEmploy extends Employ{
    String qual;
    int exp,numHour;
    PTEmploy(String name,int age,String
address,String gender,String qual,int exp,int
numHour){
        super(name,age,address,gender);
        this.qual=qual;
        this.exp=exp;
        this.numHour=numHour;
    }
```

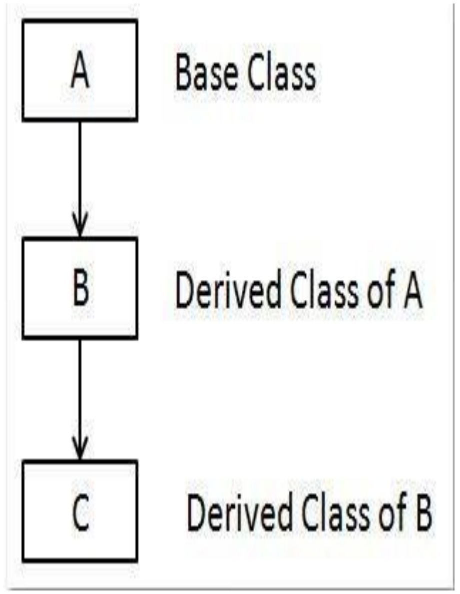
```
void calSal(){
       switch(qual){
           case "BE":
               if(exp<=5) sal=numHour*300;
               else if(exp<=10) sal=numHour*400;
               else sal=numHour*500;
               break;
           case "MTech":
               if(exp<=5) sal=numHour*500;
               else if(exp<=10) sal=numHour*700;
               else sal=numHour*1000;
               break;
```

```
 case "PhD":
                    if(exp<=5) sal=numHour*800;
                    else if(exp<=10) sal=numHour*1200;
                    else sal=numHour*1500;
                    break;
          }
     }
}
```

```
public class Tw4{
    public static void main(String []args){
        FTEmploy f1=new FTEmploy("Arun",25,"Vijayapur","Male",10000);
        f1.calSal();
        System.out.println("Details of Full time Employ");
        f1.show();
        PTEmploy e1=new PTEmploy("Rohit",30,"Belgum","Male","BE",6,10);
        e1.calSal();
        System.out.println("\nDetails of Part time Employ 1:");
        e1.show();
        PTEmploy e2=new PTEmploy("Rohini",26,"Mysore","Female","PhD",10,8);
        e2.calSal();
        System.out.println("\nDetails of Part time Employ 2:");
        e2.show();
    }
}
```

Multilevel hierarchy

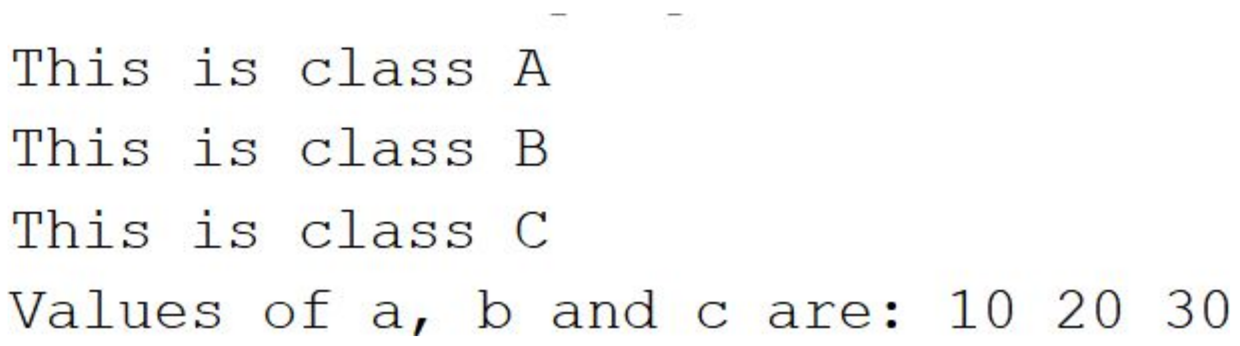
- It is perfectly acceptable to use a subclass as a superclass of another.
- For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the aspects found in all of its superclasses. In this case, C inherits all aspects of B and A.



```
class A {
    int a;
    A()
    {
        a=10;
    }
   void funcA() {
      System.out.println("This is class A");
   }
}
class B extends A {
    int b;
    B(){
        super(); //calls its superclass constructor(class A)
        b=20;
    }
   void funcB() {
      System.out.println("This is class B");
   }
}
```

```
class C extends B {
    int c;
    C(){
        super(); //calls its superclass constructor(class B)
        c=30;
    }
   void funcC() {
      System.out.println("This is class C");
   }
}
public class Demo1 {
   public static void main(String args[]) {
      C obj = new C();
      obj.funcA();
      obj.funcB();
      obj.funcC();
      System.out.println("Values of a, b and c are: "+obj.a+" "+obj.b+" "+obj.c);
   }
}
```

Out Put

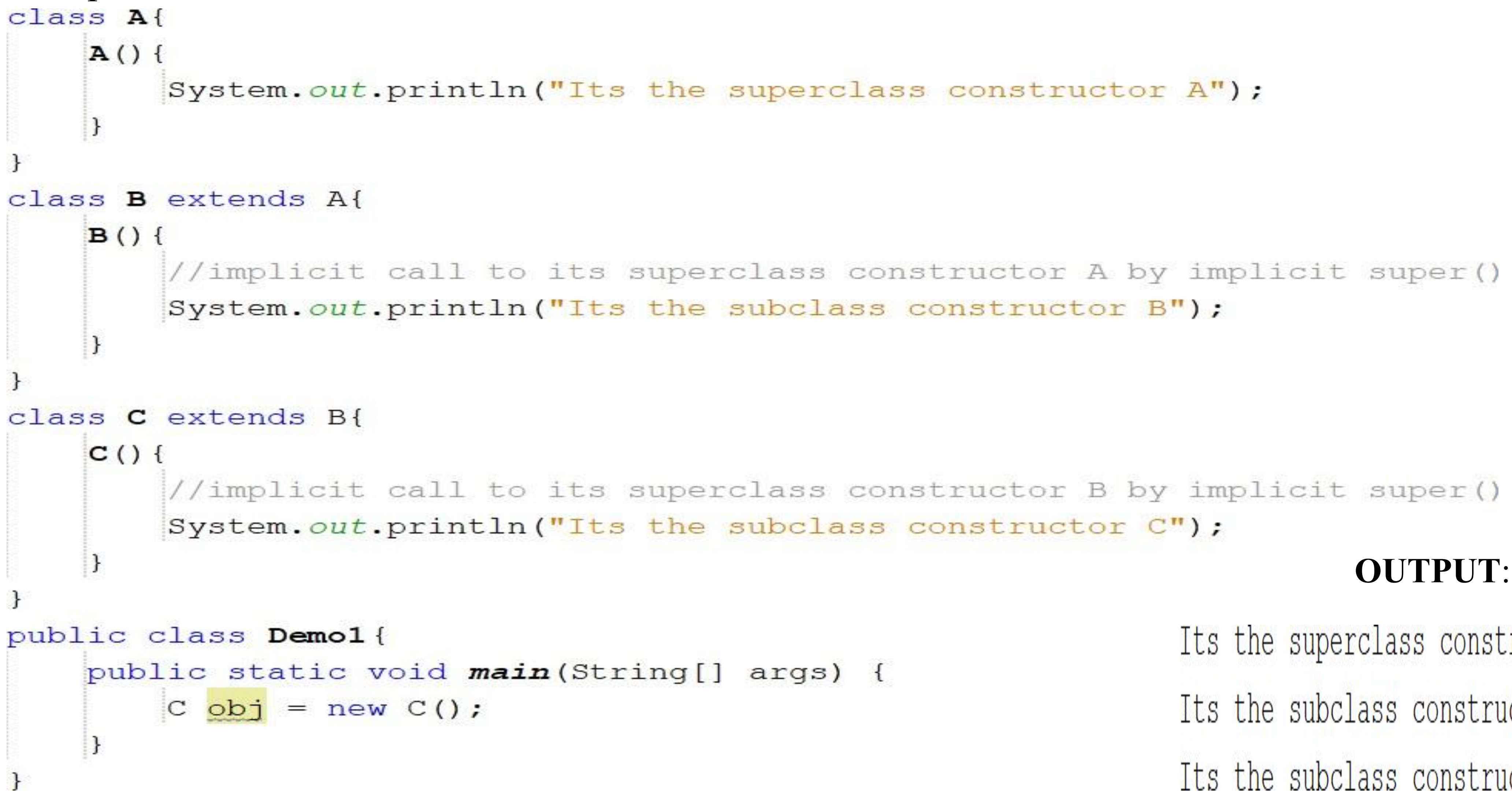


```
This is class A
This is class B
This is class C
Values of a, b and c are: 10 20 30
```

When Constructors Are Called?

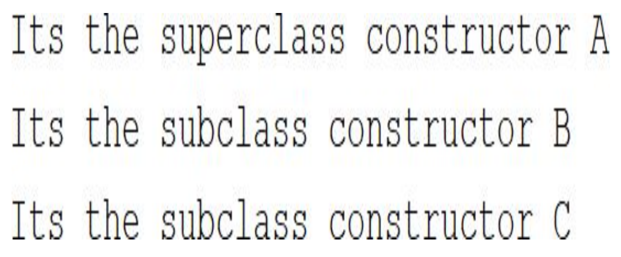
- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass

For example,



```
class A{
    A(){
        System.out.println("Its the superclass constructor A");
    }
}
class B extends A{
    B(){
        //implicit call to its superclass constructor A by implicit super()
        System.out.println("Its the subclass constructor B");
    }
}
class C extends B{
    C(){
        //implicit call to its superclass constructor B by implicit super()
        System.out.println("Its the subclass constructor C");
    }
}
public class Demo1{
    public static void main(String[] args) {
        C obj = new C();
    }
}
```

OUTPUT:



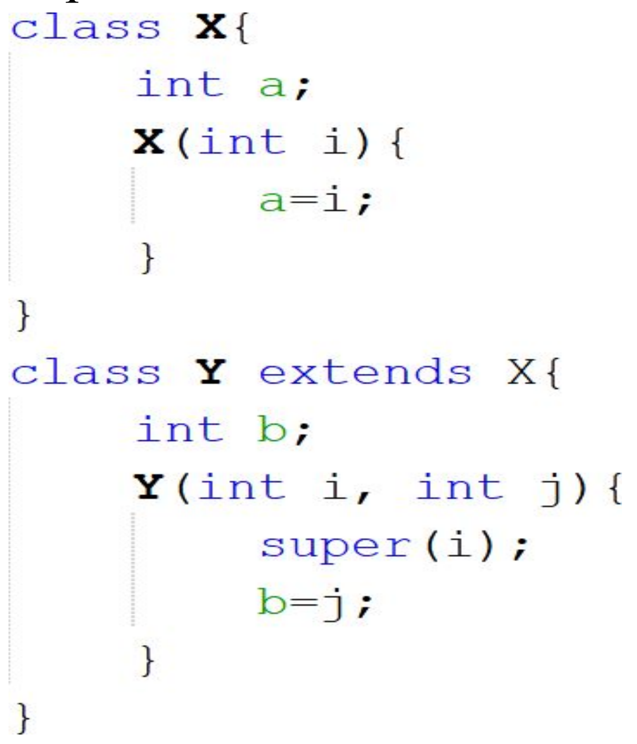
```
Its the superclass constructor A
Its the subclass constructor B
Its the subclass constructor C
```

- Always superclass constructors will be called before their subclass constructors. This is because each subclass constructor will **implicitly** call its superclass default constructor
- Hence, the constructors are called in **order of derivation.**

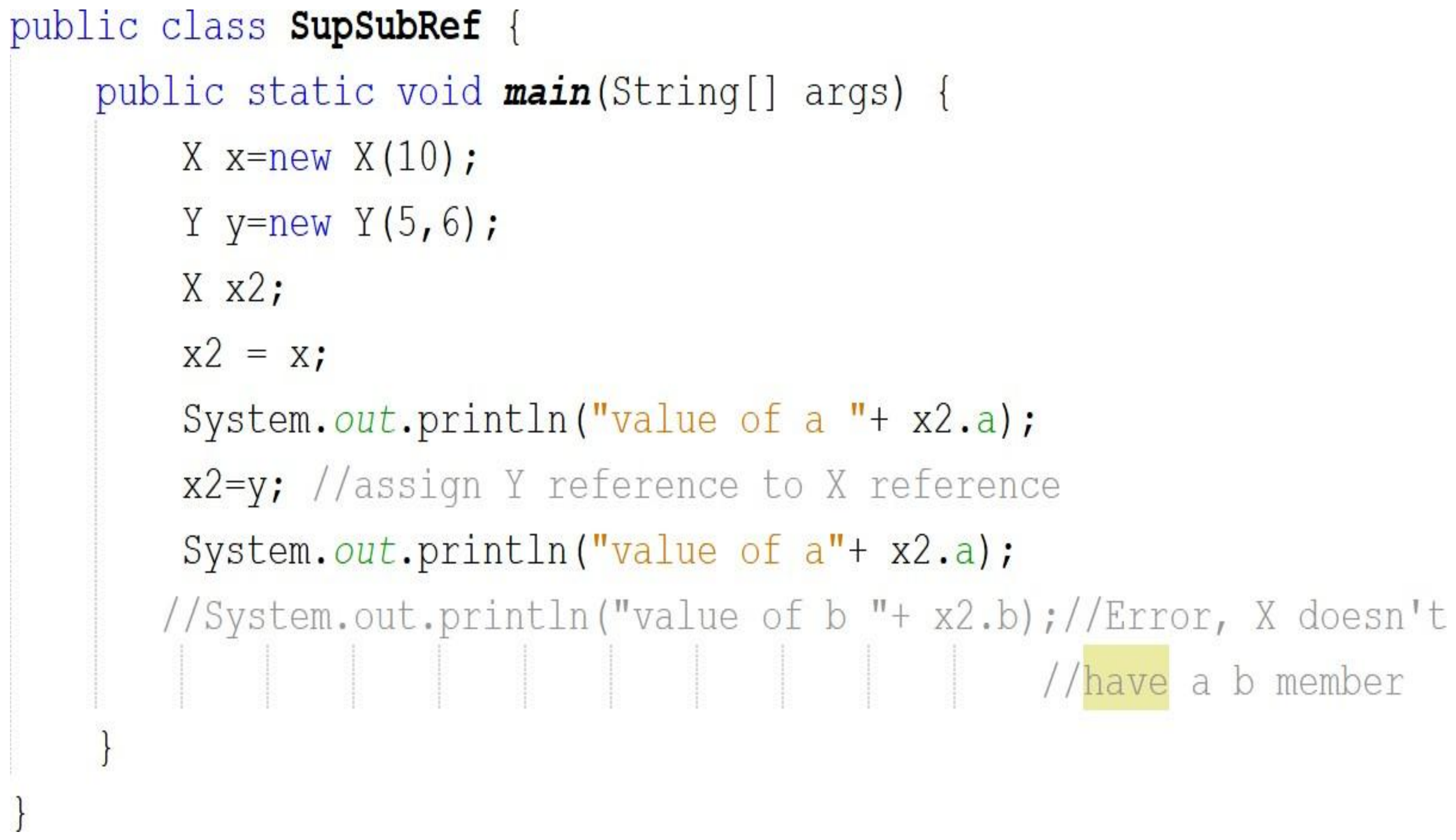
A Superclass Variable Can Reference a Subclass Object

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

• Example:



```
class X{
      int a;
      X(int i){
            a=i;
      }
}
class Y extends X{
      int b;
      Y(int i, int j){
            super(i);
            b=j;
      }
}
```

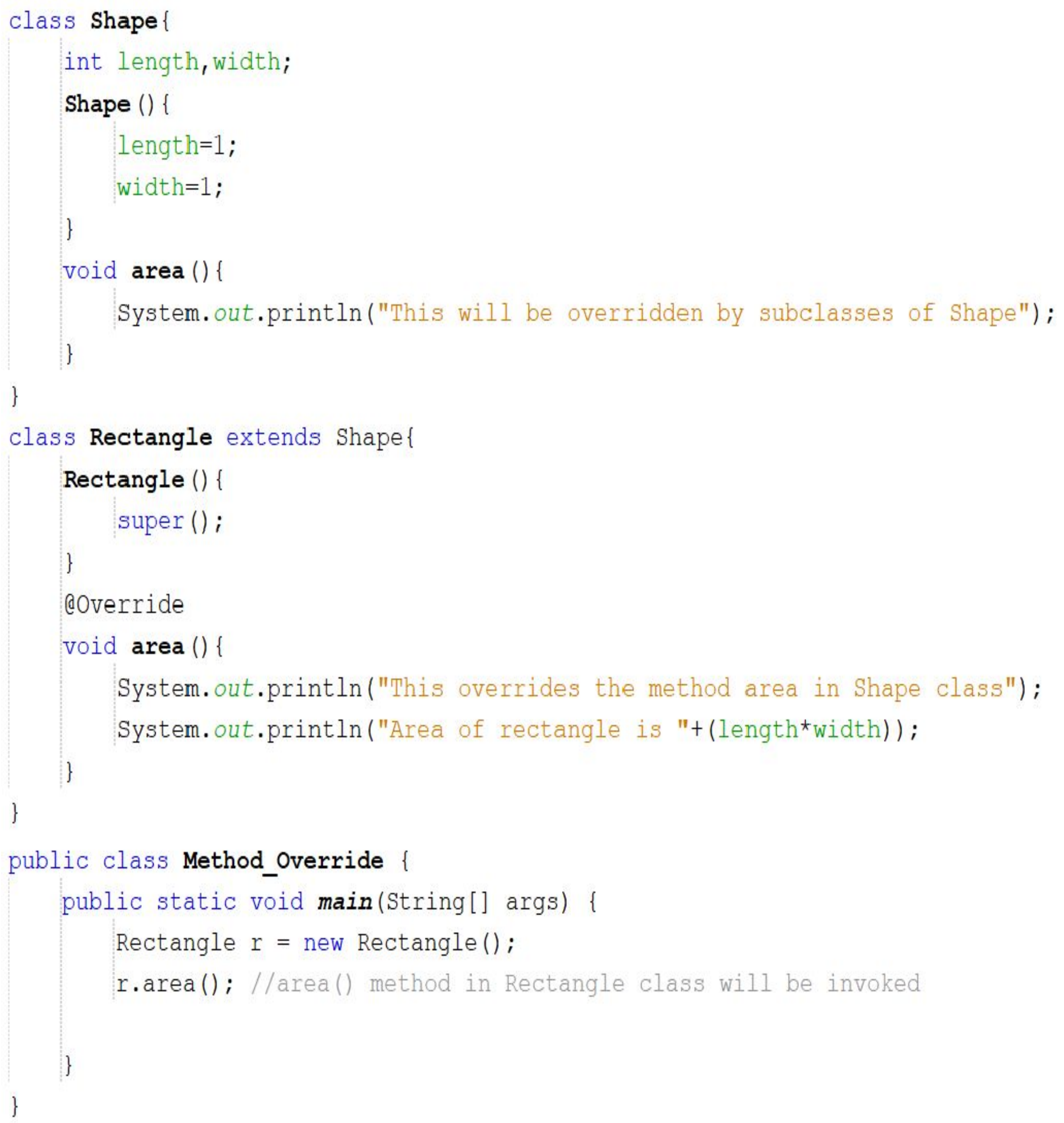


```
public class SupSubRef {
    public static void main(String[] args) {
        X x=new X(10);
        Y y=new Y(5,6);
        X x2;
        x2 = x;
        System.out.println("value of a "+ x2.a);
        x2=y; //assign Y reference to X reference
        System.out.println("value of a"+ x2.a);
       //System.out.println("value of b "+ x2.b);//Error, X doesn't
                                                //have a b member
    }
}
```

- 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 **x2** can’t access **b** even when it refers to a **Y** object.

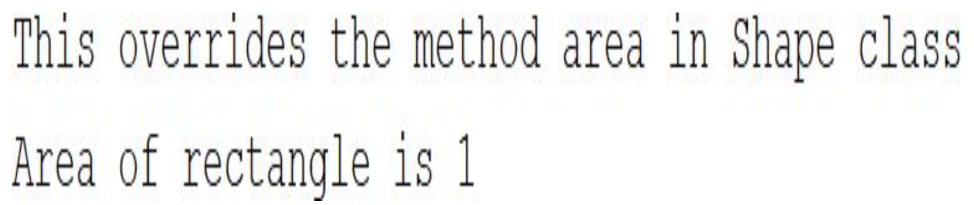
Method Overriding

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



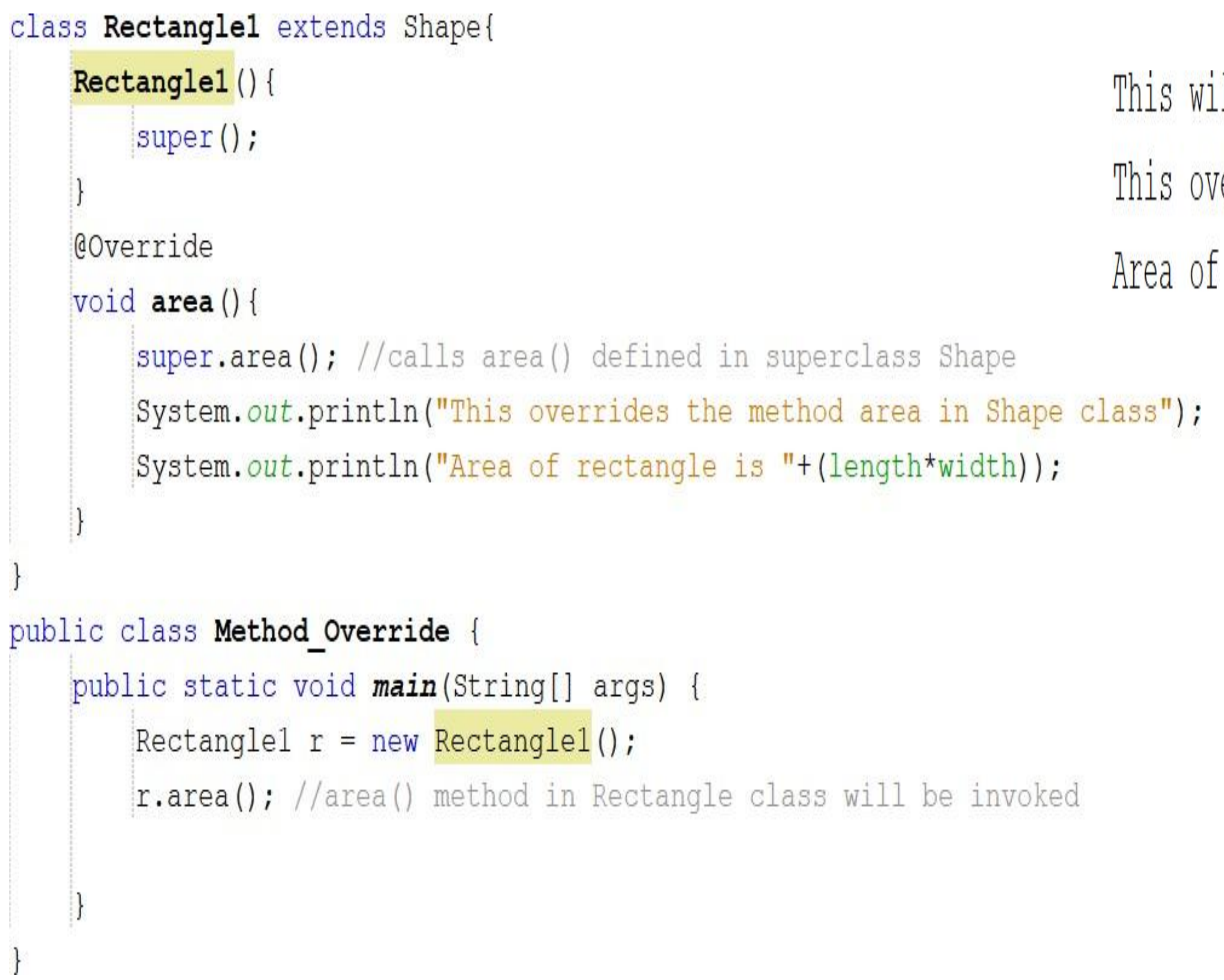
```
class Shape{
    int length,width;
    Shape(){
        length=1;
        width=1;
    }
    void area(){
        System.out.println("This will be overridden by subclasses of Shape");
    }
}
class Rectangle extends Shape{
    Rectangle(){
        super();
    }
    @Override
    void area(){
        System.out.println("This overrides the method area in Shape class");
        System.out.println("Area of rectangle is "+(length*width));
    }
}
public class Method_Override {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.area(); //area() method in Rectangle class will be invoked

    }
}
```

OUTPUT:



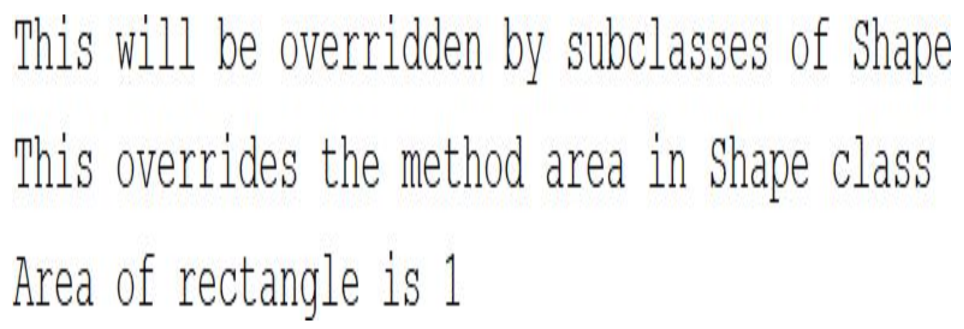
```
This overrides the method area in Shape class
Area of rectangle is 1
```

- When **area( )** is invoked on an object of type **Rectangle**, the version of **area( )** defined within **Rectangle** is used. That is, the version of **area( )** inside **Rectangle** overrides the version declared in **Shape**.
- To access the superclass version of an overridden method, you can do so by using **super** keyword.

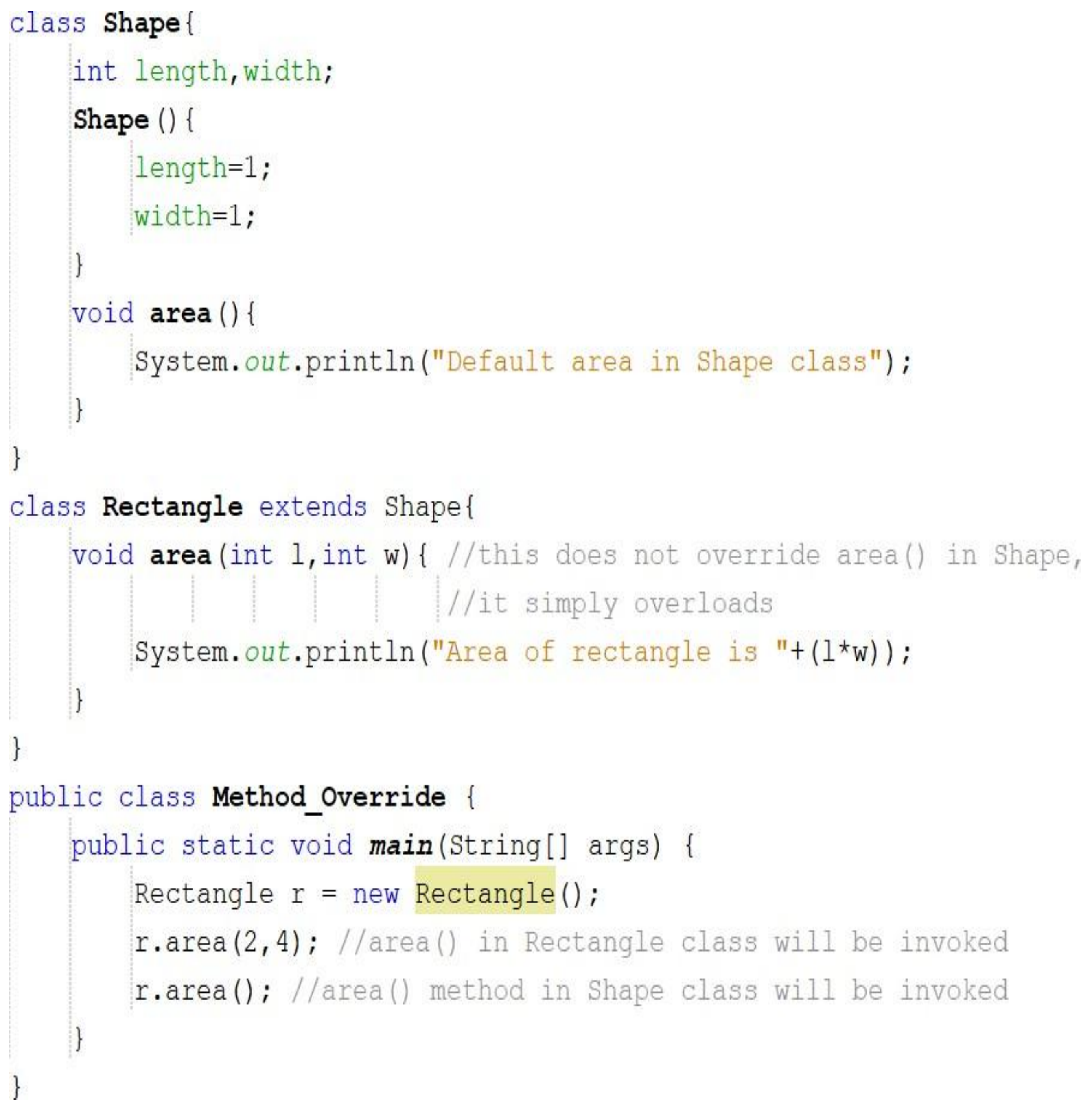


```java
class Rectangle1 extends Shape{
    Rectangle1(){
        super();
    }
    @Override
    void area(){
        super.area(); //calls area() defined in superclass Shape
        System.out.println("This overrides the method area in Shape class");
        System.out.println("Area of rectangle is "+(length*width));
    }
}
public class Method_Override {
    public static void main(String[] args) {
        Rectangle1 r = new Rectangle1();
        r.area(); //area() method in Rectangle class will be invoked


    }
}
```

OUTPUT

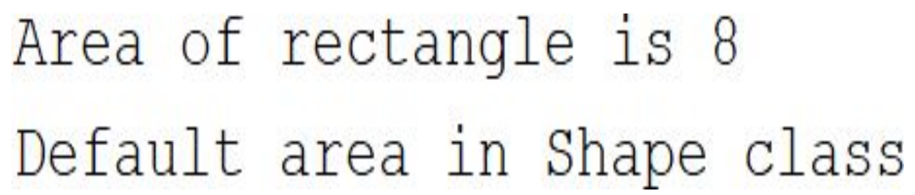


```
This will be overridden by subclasses of Shape
This overrides the method area in Shape class
Area of rectangle is 1
```



```java
class Shape{
    int length,width;
    Shape(){
        length=1;
        width=1;
    }
    void area(){
        System.out.println("Default area in Shape class");
    }
}
class Rectangle extends Shape{
    void area(int l,int w){ //this does not override area() in Shape,
                            //it simply overloads
        System.out.println("Area of rectangle is "+(l*w));
    }
}
public class Method_Override {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.area(2,4); //area() in Rectangle class will be invoked
        r.area(); //area() method in Shape class will be invoked
    }
}
```

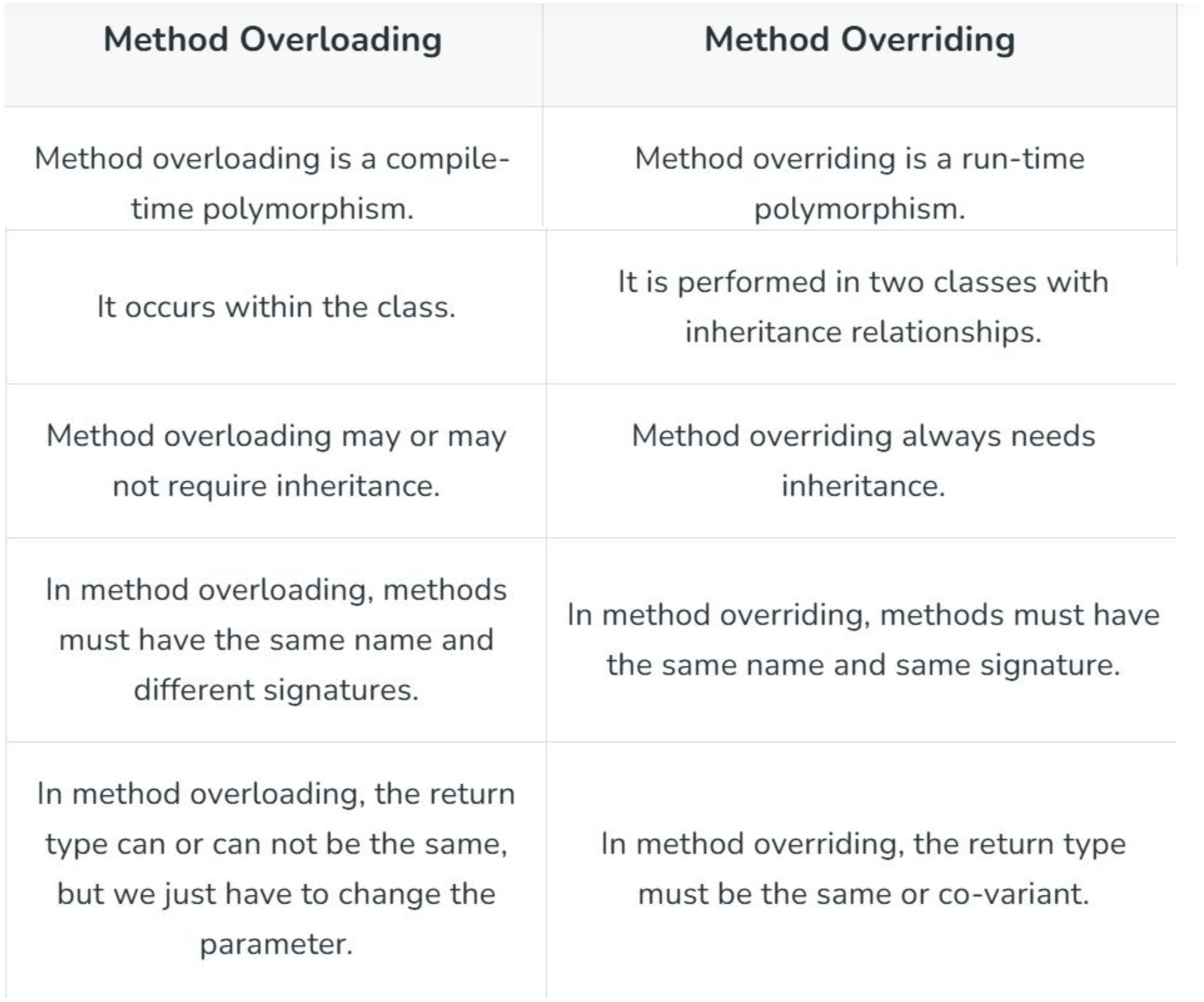
OUTPUT:



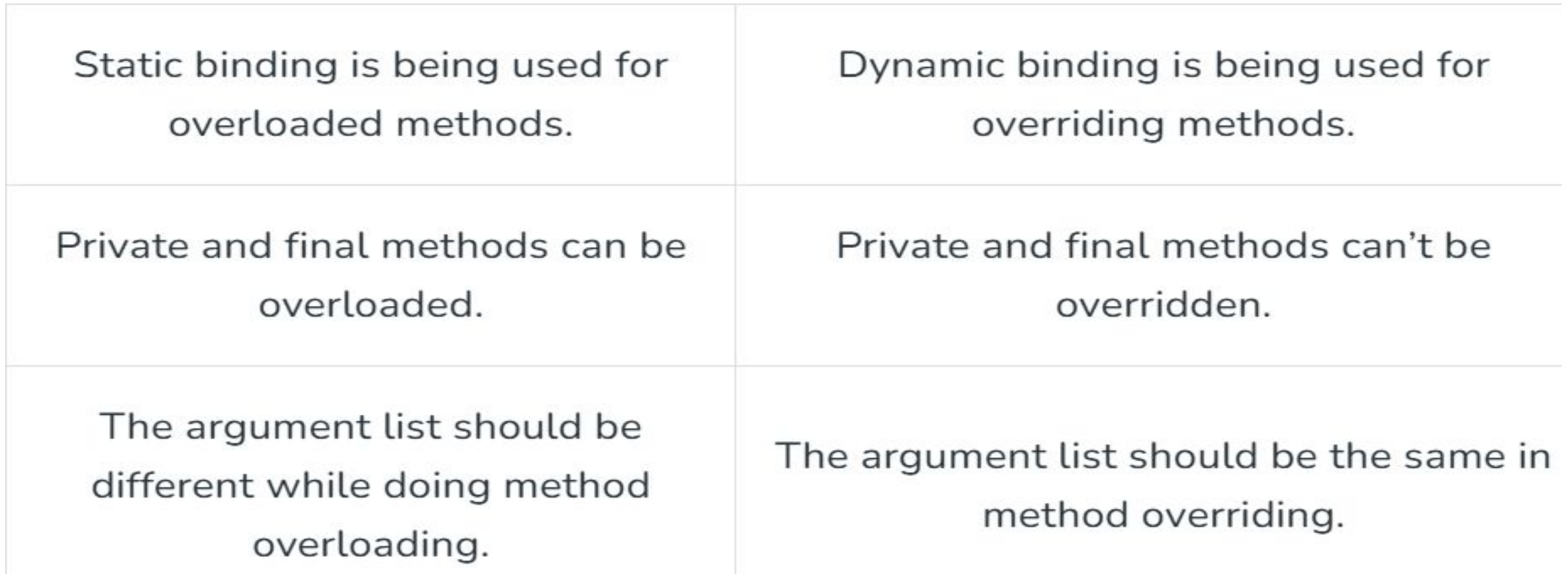
```
Area of rectangle is 8
Default area in Shape class
```

- The version of **area( )** in **Rectangle** takes two integer parameters.
- This makes its type signature different from the one in **Shape**, which takes no parameters.
- Therefore, no overriding (or name hiding) takes place. Instead, the version of **area( )** in **Rectangle** simply overloads the version of **area( )** in **Shape**.

The differences between Method Overloading and Method Overriding in Java are as follows:



Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.

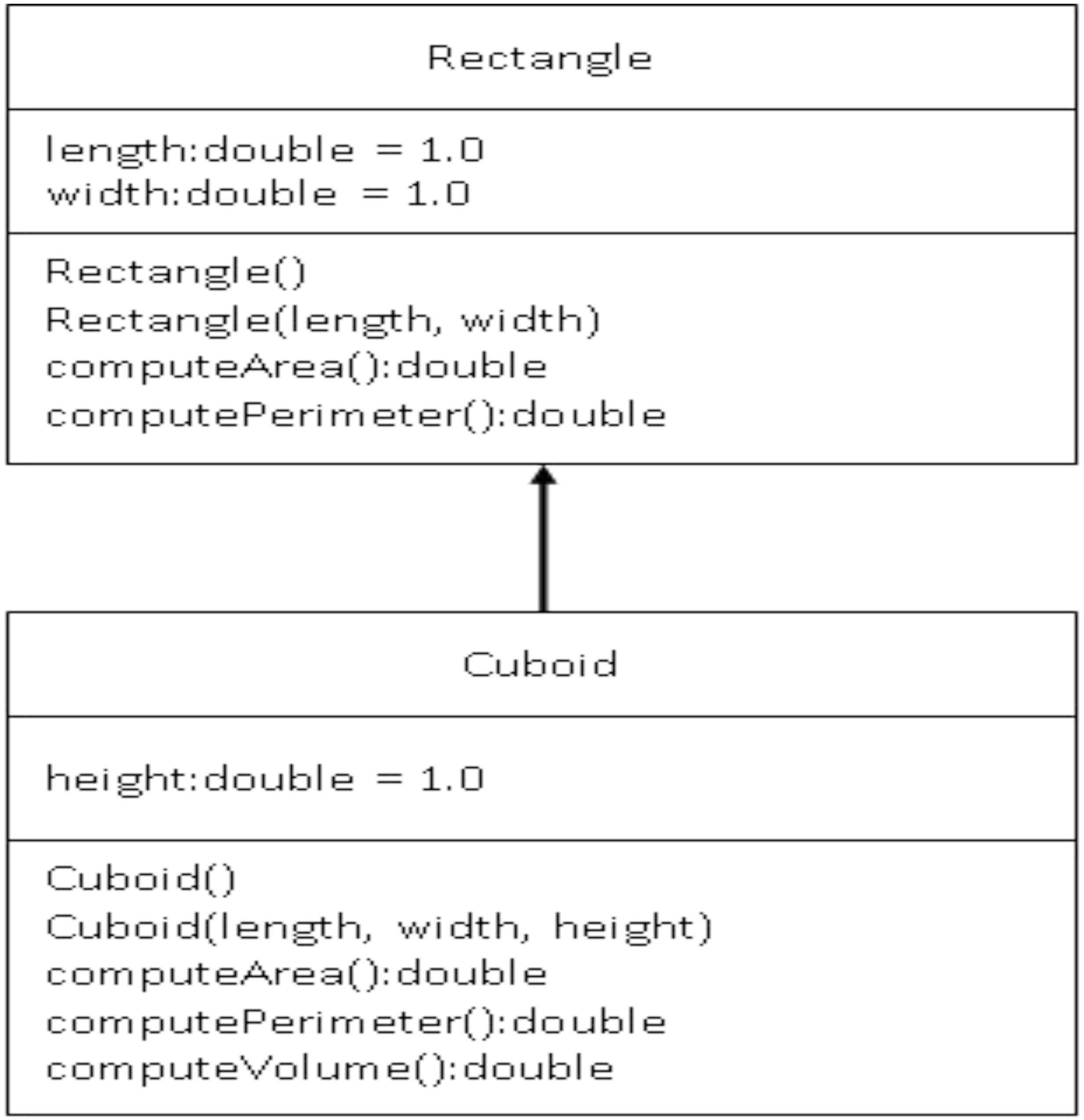


Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
The argument list should be different while doing method overloading.	The argument list should be the same in method overriding.

5 b) Overriding.

5 b)Implement the following class hierarchy. In the Cuboid class, override the method computeArea() and computePerimeter() of Rectangle class to compute the surface area and perimeter of a rectangle cuboid. Add a method computeVolume() in Cuboid class to compute volume of the cuboid. Assuming length, width and height as l, w and h respectively,

- formula to find the surface area = 2(lw) + 2(hl) + 2(hw)
- formula to find the perimeter = 2l + 2w
- formula to find the volume = l x w x h


Rectangle
length:double = 1.0
width:double = 1.0
Rectangle()
Rectangle(length, width)
computeArea():double
computePerimeter():double
Cuboid
height:double = 1.0
Cuboid()
Cuboid(length, width, height)
computeArea():double
computePerimeter():double
computeVolume():double

```
class Rectangle {

    double length;
    double width;

    Rectangle() {
        length = 1.0;
        width = 1.0;
    }

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double computeArea() {
        return length * width;
    }

    double computePerimeter() {
        return 2 * (length + width);
    }
}
```

```
class Cuboid extends Rectangle {
    double height;
    Cuboid( ) {
        super( );
        height = 1.0;
    }
    Cuboid(double length, double width, double height) {
        super(length, width);
        this.height = height;
    }
    @Override
    double computeArea() {
        return 2 * ((length * width) + (width * height) + (length *
height));
    }
    @Override
    double computePerimeter() {
        return 4 * (length + width + height);
    }
    double computeVolume() {
        return length * width * height;
    }
}
```

```
public class TW5b{
     public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
        System.out.println("Rectangle 1:");
        System.out.println("Area:" + r1.computeArea());
        System.out.println("Perimeter:" + r1.computePerimeter());
        Rectangle r2 = new Rectangle(10,30);
        System.out.println("\nRectangle 2:");
        System.out.println("Area:" + r2.computeArea());
        System.out.println("Perimeter:" + r2.computePerimeter());
        Cuboid c1=new Cuboid();
        System.out.println("\nCuboid 1:");
        System.out.println("Area:" + c1.computeArea());
        System.out.println("Perimeter:" + c1.computePerimeter());
        System.out.println("Volume:" + c1.computeVolume());
        Cuboid c2=new Cuboid(10,30,40);
        System.out.println("\nCuboid 2:");
        System.out.println("Surface area:" + c2.computeArea());
        System.out.println("Perimeter:" + c2.computePerimeter());
        System.out.println("Volume:" + c2.computeVolume());
    }
}
```

Design a base class where Hospital provides no. of patients admitted in it. Number of patients varies with the different hospitals, For example Health India hospital has 1657 patients, IVY hospital has 2965 patients and Apollo Hospital has 1631 patients. Hospital parent class which has one method getNumberOfPatients() and sub class HealthIndia, IVY and Apolo class which extends parent class & override its method.

```
// Base class: Hospital
class Hospital {
    private int numberOfPatients;

    public Hospital(int numberOfPatients) {
        this.numberOfPatients = numberOfPatients;
    }

    public int getNumberOfPatients() {
        return numberOfPatients;
    }
}

// Subclass: HealthIndia
class HealthIndia extends Hospital {
    public HealthIndia() {
        // Health India hospital has 1657 patients
        super(1657);
    }
}
```

```
// Subclass: IVY
class IVY extends Hospital {
    public IVY() {
        // IVY hospital has 2965 patients
        super(2965);
    }
}

// Subclass: Apollo
class Apollo extends Hospital {
    public Apollo() {
        // Apollo Hospital has 1631 patients
        super(1631);
    }
}
```

```
// Main class for testing
public class HospitalMain {
    public static void main(String[] args) {
        HealthIndia healthIndiaHospital = new HealthIndia();
        IVY ivyHospital = new IVY();
        Apollo apolloHospital = new Apollo();

        // Using the overridden method to get the number of patients for each hospital
        System.out.println("Health India Hospital has " + healthIndiaHospital.getNumberOfPatients() + " patients.");
        System.out.println("IVY Hospital has " + ivyHospital.getNumberOfPatients() + " patients.");
        System.out.println("Apollo Hospital has " + apolloHospital.getNumberOfPatients() + " patients.");
    }
}
```

Abstract class in java

- An abstract class is a class that cannot be instantiated on its own and is typically used as a base class for other classes.

- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.

- Abstract class may contain **abstract methods**, (which are methods declared without a body), and concrete methods or **non abstract methods** , (which have an implementation.)
- To declare an abstract method, use this **general form**:
  - **abstract type name(parameter-list);**
- The primary purpose of an abstract class is to provide a **common interface for a group of related subclasses.** By defining a set of abstract methods that the subclasses must implement, the abstract class ensures that all of the subclasses have a common set of features and behaviors.

- Additionally, **an abstract class can have instance variables and constructors**, just like any other class. However, **since an abstract class cannot be instantiated, its constructors are typically used to initialize the instance variables of the subclasses.**

- Overall, **abstract classes are an important concept in Java** programming and are often used in combination with interfaces to create hierarchies of related classes with a **common set of behaviors.**

```
abstract class Shape {
    public abstract void draw();
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Main {
    public static void main(String[] args) {
        // This will result in a compilation error
        // Shape shape = new Shape();

        // We can create an object of a subclass
        Circle c= new Circle();
        c.draw();
    }
}
```

let me provide an example to explain abstract class in Java:

```
abstract class Shape {
    protected String name;
    public Shape(String name) {
        this.name = name;
    }
    // Abstract method for calculating area
    public abstract double area();
    public void display() {
        System.out.println("This is a " + name);
    }
}
// Circle subclass
class Circle extends Shape {
    private double radius;
    public Circle(String name, double radius) {
        super(name);
        this.radius = radius;
    }
// Implementation of abstract method for calculating area
    public double area() {
        return Math.PI * radius * radius;
    }
}
// Rectangle subclass
class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(String name, double width, double height) {
        super(name);
        this.width = width;
        this.height = height;
    }
 // Implementation of abstract method for calculating area
    public double area() {
        return width * height;
    }
}
```

```
// Triangle subclass
class Triangle extends Shape {
   private double base;
   private double height;
   public Triangle(String name, double base, double height)
{
      super(name);
      this.base = base;
      this.height = height;
   }
   // Implementation of abstract method for calculating
area
   public double area() {
      return 0.5 * base * height;
   }
}
```

```
// Main program to test the shapes
public class Main {
   public static void main(String[] args) {
    Shape circle;
   Circle=new Circle("circle", 5);
      //Shape circle = new Circle("circle", 5);
      circle.display();
      System.out.println("Area of circle: " + circle.area());

      Shape rectangle = new Rectangle("rectangle", 4, 5);
      rectangle.display();
      System.out.println("Area of rectangle: " +
rectangle.area());

      Shape triangle = new Triangle("triangle", 4, 5);
      triangle.display();
      System.out.println("Area of triangle: " + triangle.area());
   }
}
```

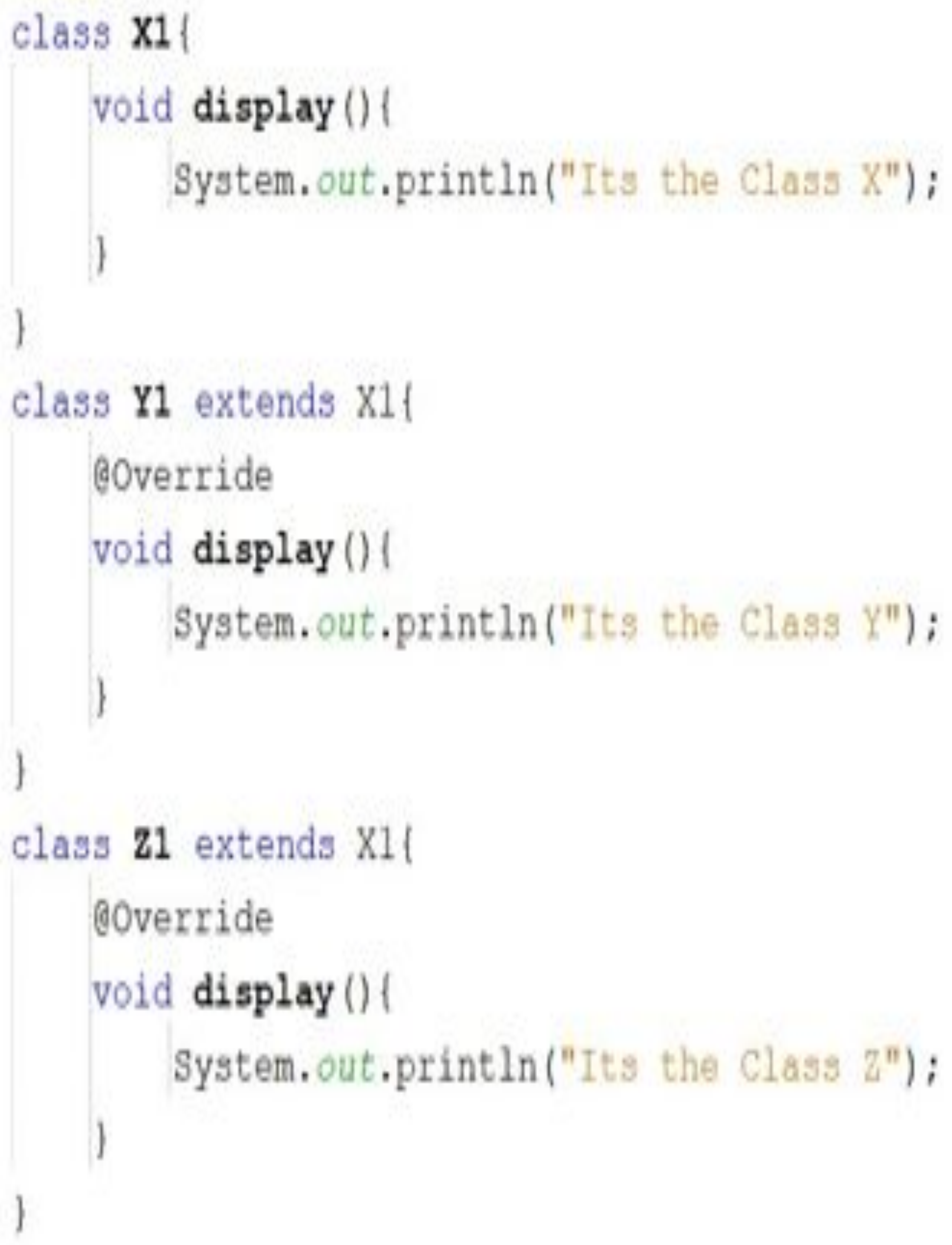
- In this example, we have defined an abstract class Shape that has a protected instance variable name and an abstract method area().
- We have also defined three subclasses Circle, Rectangle, and Triangle that inherit from Shape and provide their own implementation for the area() method.
- Since Shape is an abstract class, we cannot create an instance of it directly. However, we can create instances of its subclasses and call their methods, including the overridden area() method. When we call the area() method on a shape object, the appropriate implementation for that shape is executed, and we get the correct area value.

Run Time Polymorphism(also known as dynamic polymorphism .)

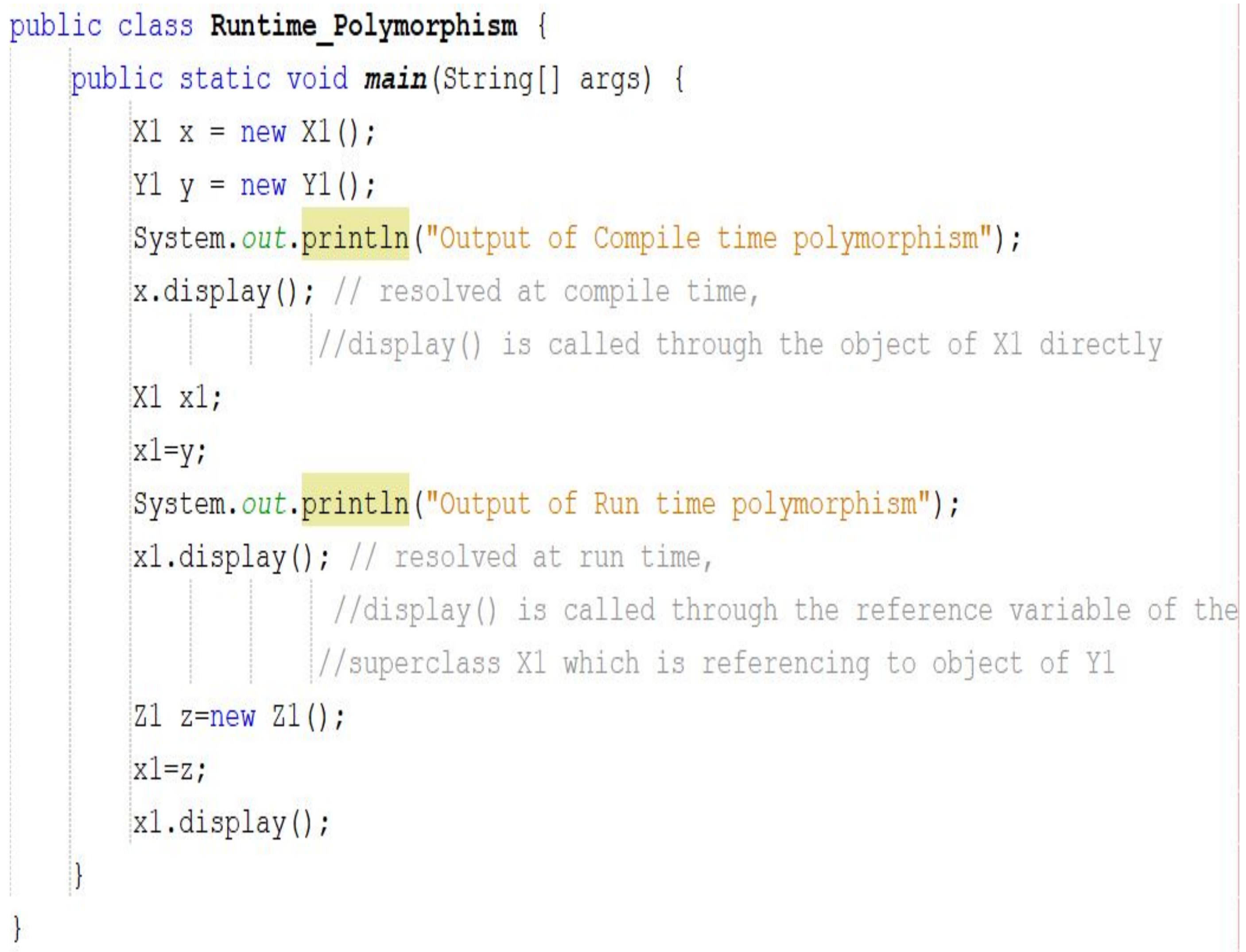
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.
- 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.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
// Base class: Animal
class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}

// Subclass: Dog
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

// Subclass: Cat
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

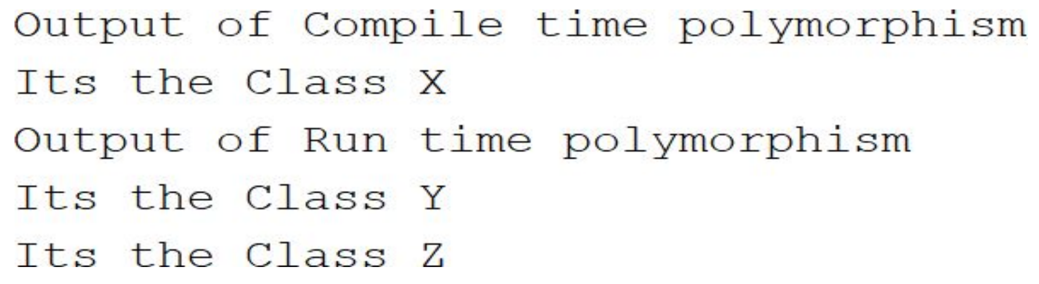
```
// Main class for testing
public class PolymorphismExample {
    public static void main(String[] args) {
        // Creating objects of different types
        Animal a1 = new Dog();
        Animal a2 = new Cat();

        // Calling the makeSound method on different objects
        a1.makeSound(); // This will call the overridden method in Dog class
        a2.makeSound(); // This will call the overridden method in Cat class
    }
}
```



```java
class X1{
    void display(){
        System.out.println("Its the Class X");
    }
}
class Y1 extends X1{
    @Override
    void display(){
        System.out.println("Its the Class Y");
    }
}
class Z1 extends X1{
    @Override
    void display(){
        System.out.println("Its the Class Z");
    }
}
```



```java
public class Runtime_Polymorphism {
    public static void main(String[] args) {
        X1 x = new X1();
        Y1 y = new Y1();
        System.out.println("Output of Compile time polymorphism");
        x.display(); // resolved at compile time,
                    //display() is called through the object of X1 directly
        X1 x1;
        x1=y;
        System.out.println("Output of Run time polymorphism");
        x1.display(); // resolved at run time,
                     //display() is called through the reference variable of the
                    //superclass X1 which is referencing to object of Y1
        Z1 z=new Z1();
        x1=z;
        x1.display();
    }
}
```



```
Output of Compile time polymorphism
Its the Class X
Output of Run time polymorphism
Its the Class Y
Its the Class Z
```

TW-6: Write a program to demonstrate Run-time Polymorphism

- Design an abstract class Car to have carName, chassiNum, modelName as member variables and add two abstract methods, startCar and operateSteering . Inherit MarutiCar and BmwCar from Car class and override the two abstract methods in their own unique way. Design a driver class to have driver name, gender and age as data members and add a method driveCar with abstract class reference variable as argument and invoke the two basic operations namely, startCar and operateStearing and demonstrate run-time polymorphism.

```
abstract class Car{
    String carName,modelName;
    int chassiNum;
    Car(String carName,int chassiNum,String modelName){
        this.carName=carName;
        this.chassiNum=chassiNum;
        this.modelName=modelName;
    }
    abstract void startCar();
    abstract void operateSteering();
    void display(){
        System.out.println("Car Name:"+carName);
        System.out.println("Chassi number:"+chassiNum);
        System.out.println("Model Name:"+modelName);
    }
}
```

```
class MarutiCar extends Car{

    MarutiCar(String carName, int
chassiNum, String modelName) {
        super(carName, chassiNum,
modelName);
    }
    void startCar(){
        System.out.println("Starting
a Maruti car....");
    }
    void operateSteering(){
        System.out.println("This car
is manually steered.....");
    }
}
```

```
class BmwCar extends Car{

    BmwCar(String carName, int
chassiNum, String modelName) {
        super(carName, chassiNum,
modelName);
    }
    void startCar(){
        System.out.println("Starting a
BMW car....");
    }
    void operateSteering(){
        System.out.println("This car
is automatically steered.....");
    }
}
```

```
class Driver{
    String name,gender;
    int age;
    Driver(String name,int age,String gender){
        this.name=name;
        this.age=age;
        this.gender=gender;
    }
    void driveCar(Car obj){
     System.out.println("Driver:"+name);
        System.out.println("Age:"+age);
        System.out.println("Gender:"+gender);
        obj.display();
        obj.startCar();
        obj.operateSteering();
    }
}
```

```
public class Tw6 {
    public static void main(String []args){
MarutiCar m=new MarutiCar("Suzuki",1253,"A21s");
        BmwCar b=new BmwCar("BMW5",4596,"S5");
        Driver d1=new Driver("Vishal",25,"Male");
        d1.driveCar(m);
        System.out.println();
        Driver d2=new Driver("Priya",23,"Female");
        d2.driveCar(b);
    }
}
```

Using final

In Java, the final keyword is used to create a variable, method, or class that cannot be changed or overridden.

The keyword **final** has three uses.

1. **Using final to Prevent Overriding(i.e using final keyword with method)**

Final method: A final method is a method that cannot be overridden by a subclass. This is useful when you want to ensure that a method always behaves the same way in all subclasses.

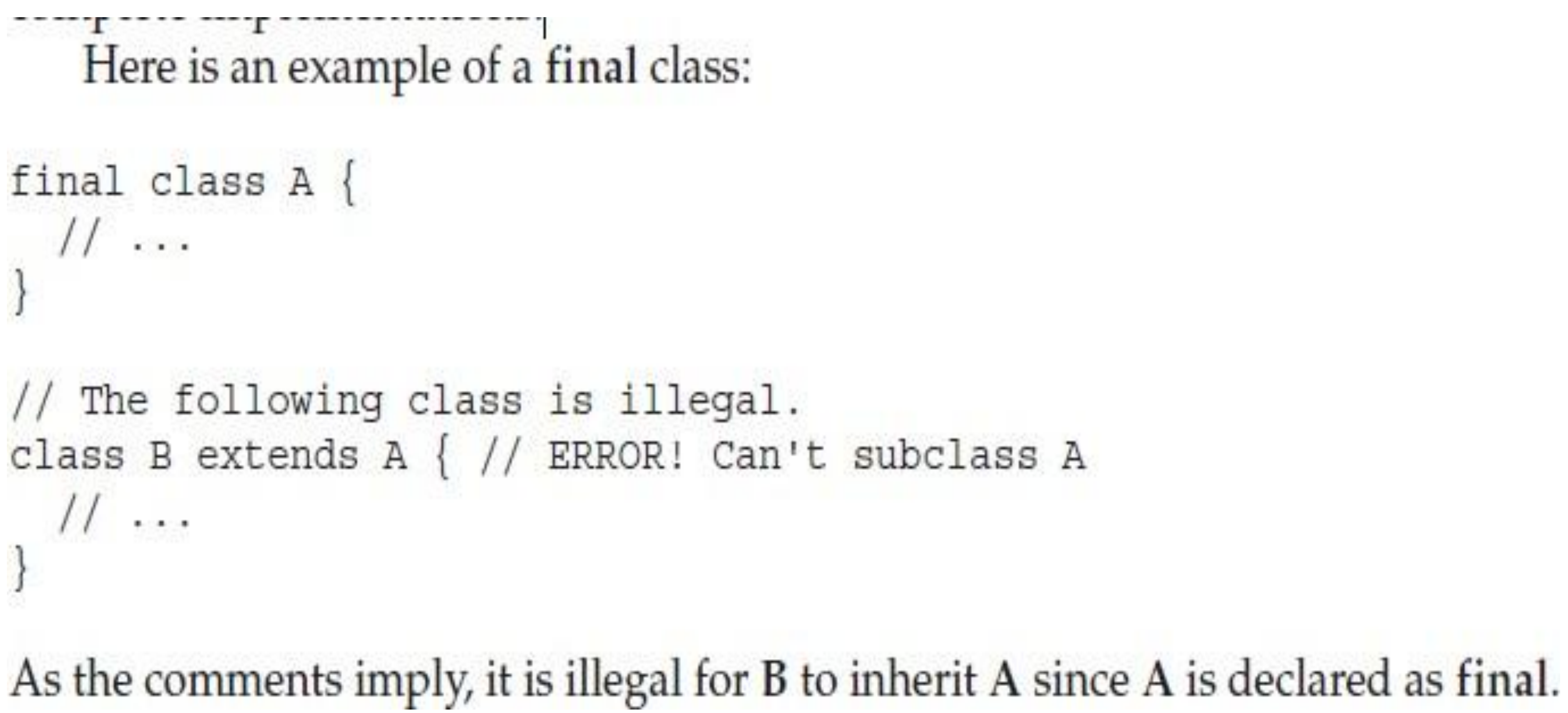
- To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as final cannot be overridden.
- The following fragment illustrates final

```
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

2. **Using final to Prevent Inheritance(final used with class)**

- You can prevent a class from being inherited by preceding the class declaration with **final**.
- Declaring a class as **final** implicitly declares all of its methods as **final**, too
- 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.



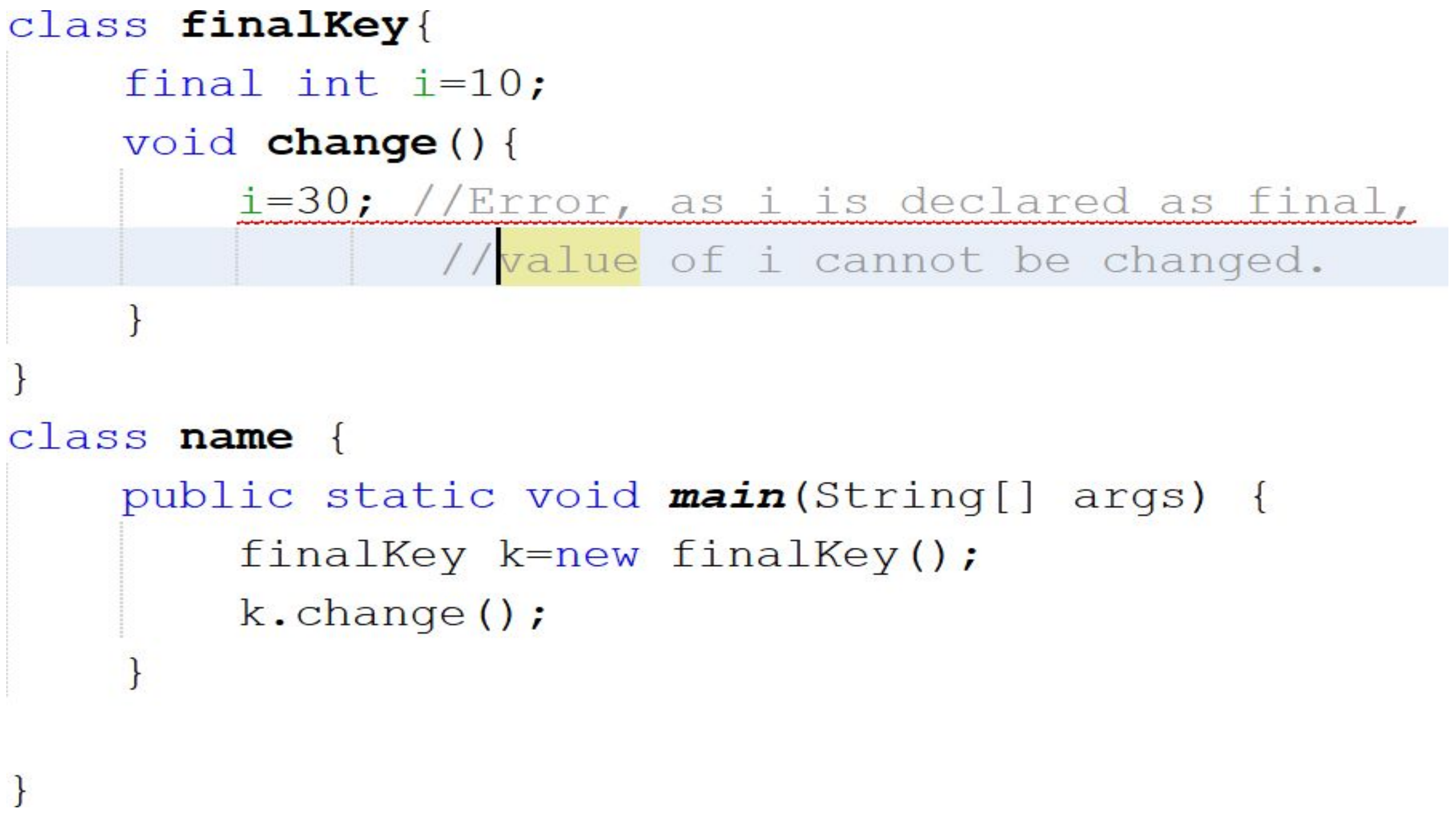
Here is an example of a final class:

```
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.

3. Using final with Data Members(variables)

- If a class variable's name is preceded with **final**, its value cannot be changed throughout the lifetime of the program.
- Final variable simply means making constants.
- Initial value can be assigned.



```
class finalKey{
	final int i=10;
	void change(){
		i=30; //Error, as i is declared as final,
			//value of i cannot be changed.
	}
}
class name {
	public static void main(String[] args) {
		finalKey k=new finalKey();
		k.change();
	}

}
```

1. Explain with examples the effect of the keyword ‘final’ with respect to:

    class

    methods

    variable

Interfaces

- **Interface Fundamentals**
- An interface can only **have abstract methods and constants** (final fields).
- An interface **cannot have instance variables (fields) other than constants.**
- All methods in an interface are **implicitly public and abstract**.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- **Interfaces cannot have constructors**, as they cannot be instantiated on their own.
- To implement an interface, a class must provide bodies (implementations) for the methods declared by the interface.

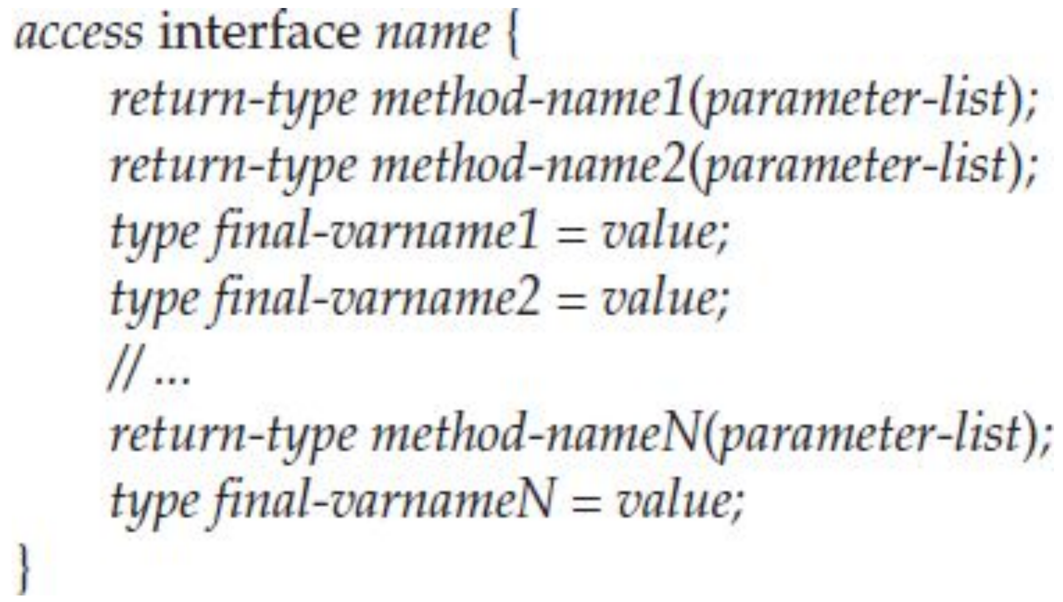
- Interfaces are used to achieve **total abstraction** which was not achieved through abstract classes//This is because an abstract class can still contain some implementation details, while an interface only contains method signatures without any implementation
- By providing the interface keyword, Java allows you to fully utilize the **“one interface, multiple methods”** aspect of **polymorphism**.

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

1. **It is used to achieve abstraction** i.e **Abstraction**: Interfaces allow you to separate the definition of a behavior or capability from its implementation. This allows you to create more flexible and reusable code that can work with different implementations of the same interface.
2. **Multiple Inheritance**: Java doesn’t support multiple inheritance of classes, but a class can implement multiple interfaces. This allows you to inherit the behavior of multiple interfaces, which can be useful in certain cases.
3. **It can be used to achieve loose coupling**. **i.e Polymorphism**: Interfaces allow you to use polymorphism in your code, which means you can treat objects of different classes that implement the same interface as interchangeable. This allows you to write more generic code that can work with a wider range of objects.

Creating an Interface

- An interface is defined much like a class. **This is the general form of an interface:**



```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

- **When no access specifier is included**, then default access results, and the **interface is only available to other members of the package in which it is declared.**
- When it is declared as public, the interface can be used by any other code. **In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface**

```
interface Callback {
  void callback(int param);
}
```

- It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter

<u>Implementing Interfaces</u>

- Once an **interface** has been defined, one or more classes can implement that interface

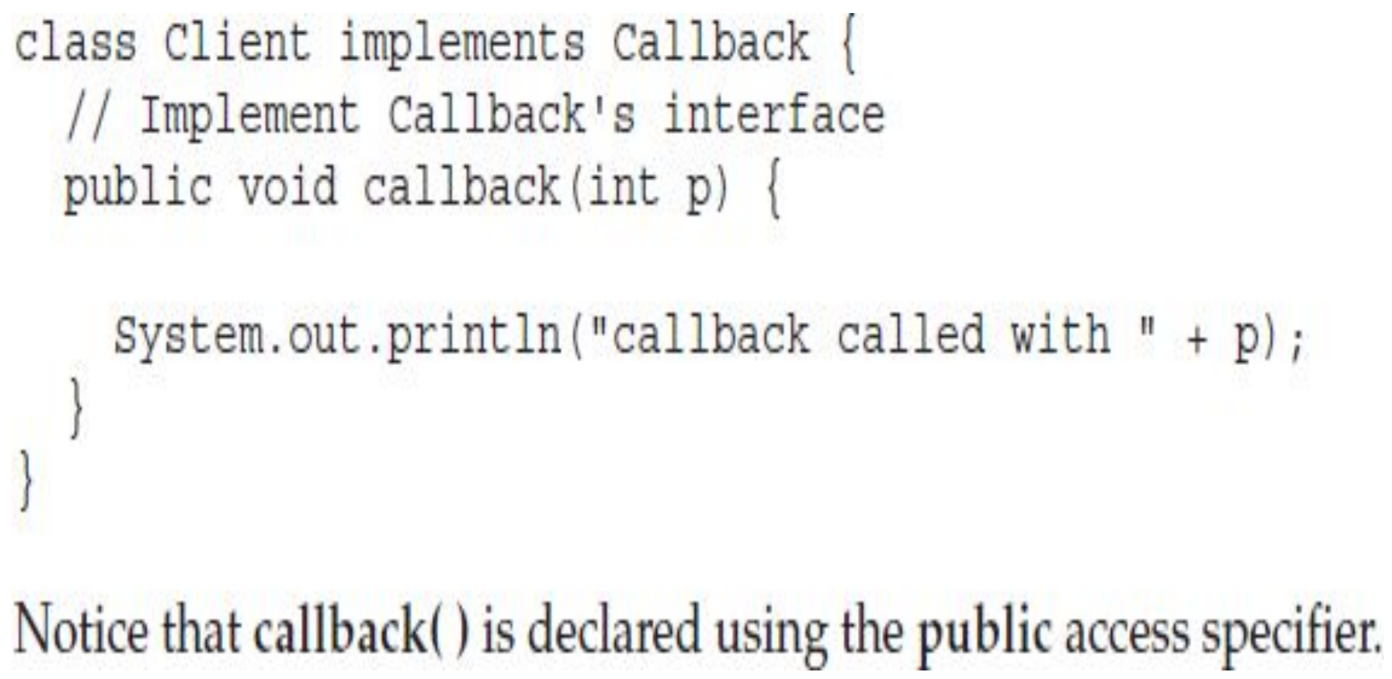
  To implement an interface, include the **implements** clause in a class definition, and Then create the methods defined by the interface .

- <u>The general form of a class that includes the **implements** clause looks like this:</u>

  class *classname* implements *interface_name*{
    // class-body
  }

- If a class implements more than one interface, the interfaces are separated with a comma.

- If a class implements **two interfaces** that declare the **same method**, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**. Also, the **type signature of the implementing method must match exactly the type signature specified in the interface definition**



```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {

    System.out.println("callback called with " + p);
  }
}
```

Notice that callback( ) is declared using the public access specifier.

```
public interface Vehicle {

   int getNumWheels();

   String getFuelType();

   void startEngine();

}
```

```
public class Car implements Vehicle {
   private int numWheels;
   private String fuelType;

   public Car(int numWheels, String fuelType) {
      this.numWheels = numWheels;
      this.fuelType = fuelType;
   }

   public int getNumWheels() {
      return numWheels;
   }

   public String getFuelType() {
      return fuelType;
   }

   public void startEngine() {
      System.out.println("Starting car engine");
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Vehicle car = new Car(4, "gasoline");
      System.out.println("Number of wheels: 
" + car.getNumWheels());
      System.out.println("Fuel type: " + 
car.getFuelType());
      car.startEngine();
   }
}
```

Using Interface References

- An interface declaration creates a new reference type.
- In other words, you can create an interface reference variable but not its object just like abstract classes.
- However, it can refer to any object that implements the interface.
- When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed.
- The following example calls the **display( ) and show( )** method via an interface reference variable:

```
interface In {
    // implicitly public and abstract
    void display();
    void show();
}
class TestClass implements In { //Needs to prvide implementation for both
                                //display and show methods
    // Implementing the capabilities of interface.
    public void display(){
      System.out.println("This is overriding display in interface In");
    }
    public void show(){
      System.out.println("This is the implementation for show in interface In");
    }
}
public class Interface_ex{
    public static void main(String[] args){
        In i; //reference variable if interface In.
              //object cannot be created as interface is not a complete entity
        i= new TestClass(); //interface reference variable is
                            //referencing to TestClass object
        i.display(); //display in TestClass is called through
                     //interface reference variable
        i.show();
    }
}
```

Another Example….

In Java, you can use interface references to achieve loose coupling between objects, which makes your code more flexible and reusable. Here's an example that demonstrates the use of **interface references:**

```
public interface Animal {
   public void makeSound();
}
public class Dog implements Animal {
   public void makeSound() {
      System.out.println("Woof!");
   }
}
public class Cat implements Animal {
   public void makeSound() {
      System.out.println("Meow!");
   }
}
public class AnimalApp {
   public static void main(String[] args) {
      Animal animal1 = new Dog(); // Use interface reference to create a Dog object
      Animal animal2 = new Cat(); // Use interface reference to create a Cat object
      animal1.makeSound(); // Output: Woof!
      animal2.makeSound(); // Output: Meow!
   }
}
```

- In the AnimalApp class, we create two objects animal1 and animal2 using the Dog and Cat classes respectively, but we use interface references to create them.

- This means that we can treat both objects as Animal objects, even though they are of different concrete types (Dog and Cat).

- Using interface references in this way makes your code more flexible and reusable because you can swap out different implementations of an interface without changing the code that uses the interface. In other words, you can use polymorphism to allow different objects to be treated interchangeably based on a common interface.

Example of an interface in Java with detailed description:

```
// Define an interface named "Drawable"
public interface Drawable {
   // Declare a method signature for "draw"
   void draw();
}
```

//In this example, we define an interface named "Drawable". The //interface contains a method signature for the draw() method.

```
// Define a class named "Circle" that implements "Drawable" interface
public class Circle implements Drawable {
   // Implement the "draw" method
   public void draw() {
      System.out.println("Drawing a circle");
   }
}
```

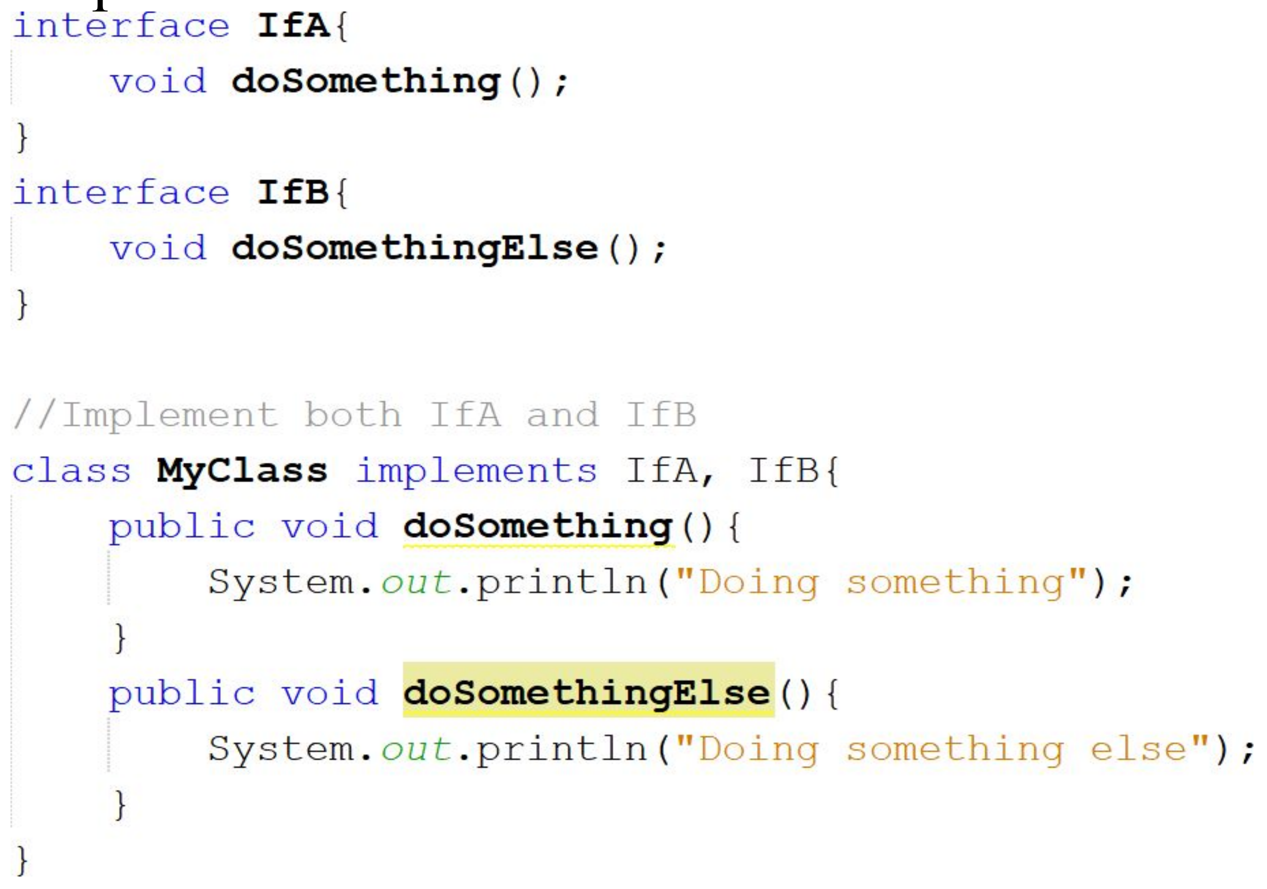
//In this example, we define a class named "Circle" that implements //the "Drawable" interface. The Circle class provides an //implementation for the draw() method that simply outputs //"Drawing a circle" to the console.

```
// Define a class named "Rectangle" that
implements "Drawable" interface
public class Rectangle implements Drawable {
   // Implement the "draw" method
   public void draw() {
      System.out.println("Drawing a rectangle");
   }
}
```

```
// Define a class named "DrawingApp" that uses "Drawable"
interface
public class DrawingApp {
   // Define a static method to draw a shape using "Drawable"
interface
void drawShape(Drawable shape) {
      shape. public static draw();
   }

   // Main method
   public static void main(String[] args) {
      // Create instances of "Circle" and "Rectangle" classes
      Circle myCircle = new Circle();
      Rectangle myRectangle = new Rectangle();

      // Draw the circle and rectangle using the "drawShape"
method
      drawShape(myCircle); // Output: Drawing a circle
      drawShape(myRectangle); // Output: Drawing a rectangle
   }
}
```

- In this example, we define a class named "DrawingApp" that uses the "Drawable" interface.
- **We define a static method named "drawShape()" that takes an object that implements the "Drawable" interface as an argument, and calls the draw() method on that object.**
- In the main() method, we create instances of the Circle and Rectangle classes, and call the drawShape() method on each of them.
- This results in the draw() method being called on each object, which outputs "Drawing a circle" and "Drawing a rectangle" to the console.
- Overall, this example demonstrates how an interface can be used to define a common behavior that can be implemented by different classes in different ways, allowing for more flexible and modular code.
- **It also shows how the interface can be used to create polymorphic behavior, where the same method can be called on different objects that implement the same**

- **Implementing Multiple Interfaces**
  - A class can implement more than one interface.
  - The class must implement all of the methods specified by each interface.
- For example

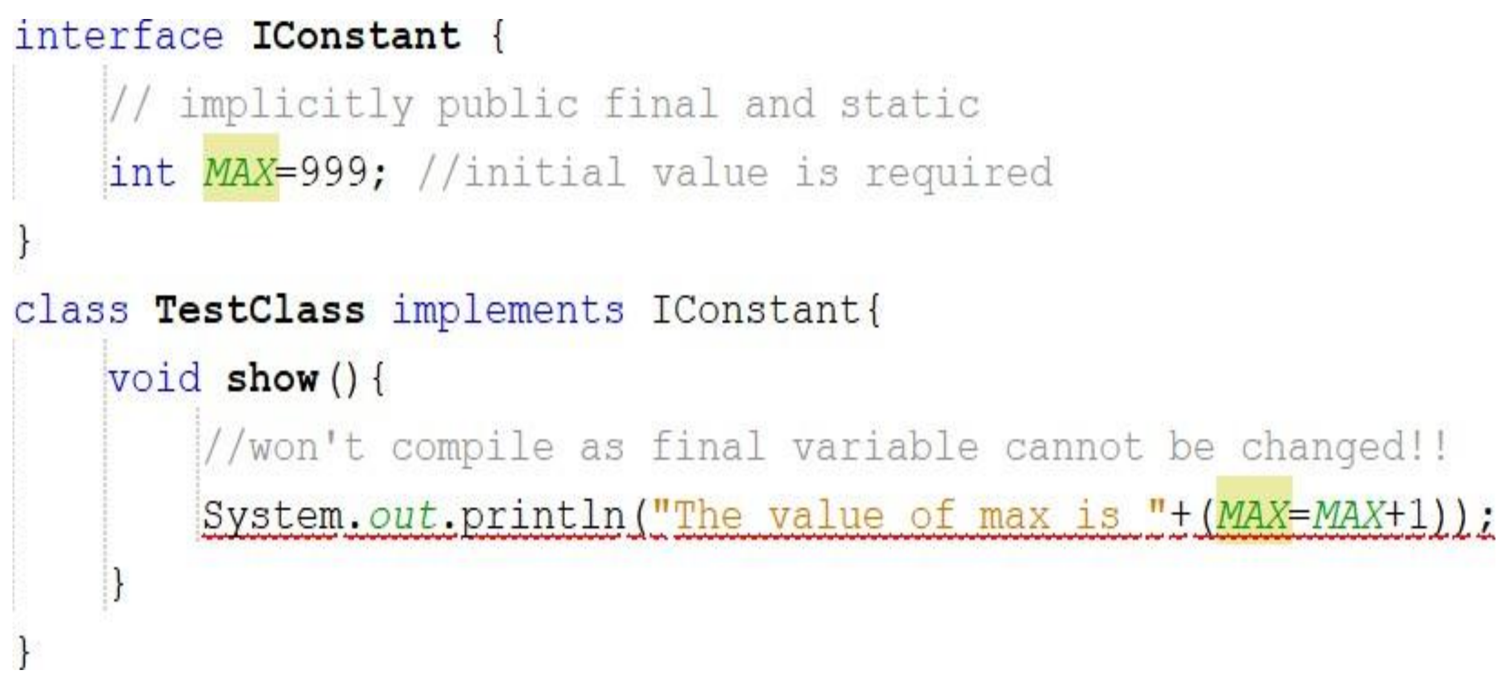


```
interface IfA{
    void doSomething();
}
interface IfB{
    void doSomethingElse();
}

//Implement both IfA and IfB
class MyClass implements IfA, IfB{
    public void doSomething(){
        System.out.println("Doing something");
    }
    public void doSomethingElse(){
        System.out.println("Doing something else");
    }
}
```

In this example, **MyClass** specifies both **IfA** and **IfB** in its implements clause

- **Constants in Interfaces**

- Interface can also include variables. Such variables are not instance variables. Instead, they are implicitly **public, final and static** and must be initialised.
- Hence, they are constants



```
interface IConstant {
    // implicitly public final and static
    int MAX=999; //initial value is required
}
class TestClass implements IConstant{
    void show(){
        //won't compile as final variable cannot be changed!!
        System.out.println("The value of max is "+(MAX=MAX+1));
    }
}
```

- IConstant inteface defines **MAX** constant. As required, they are given initial value.
- The above program won’t compile because once the final variable in initialized, its value cannot be altered.

Here are the key differences between an abstract class and an interface class in Java:

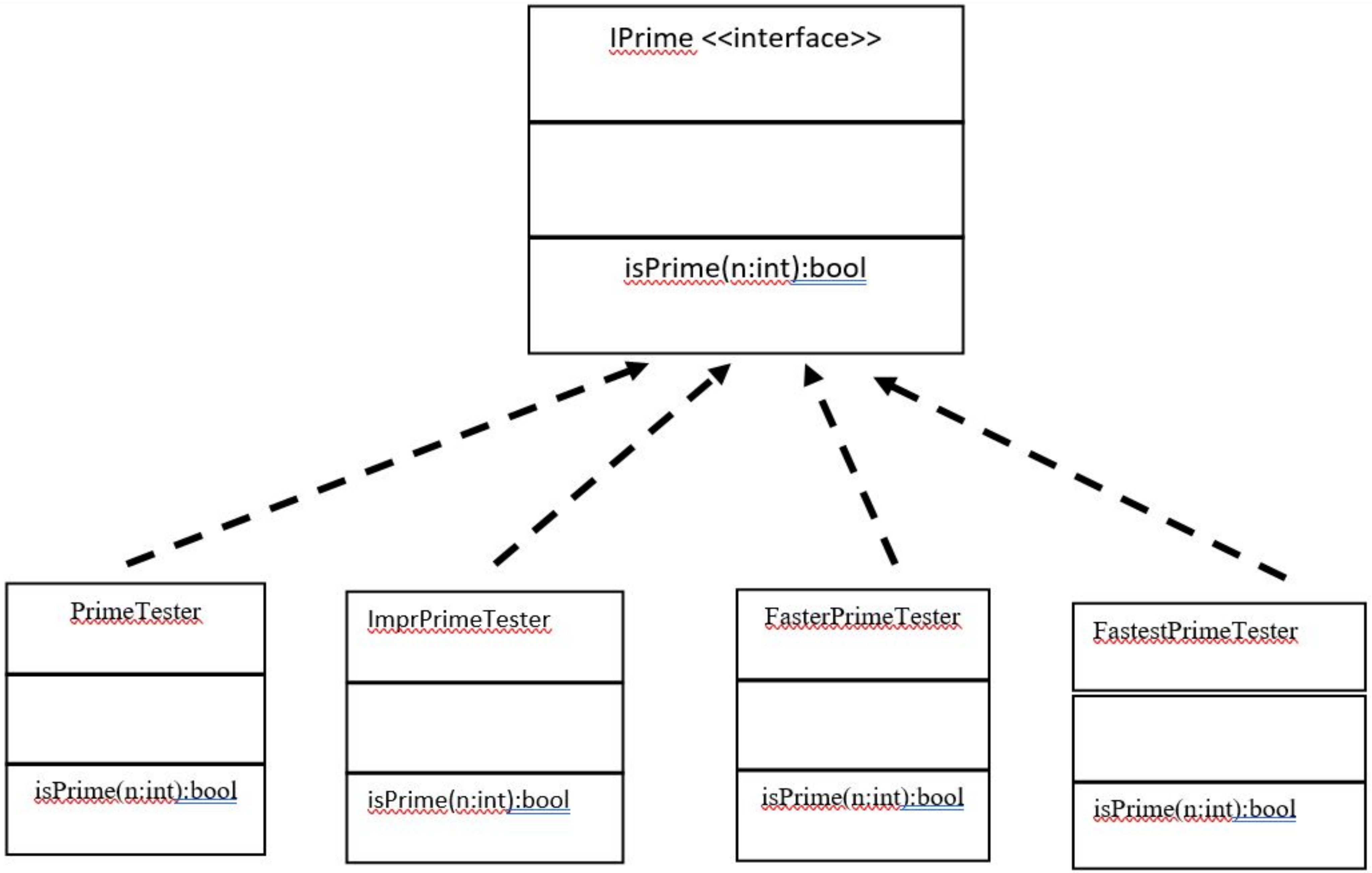
1. **Implementation**: An abstract class can contain both abstract and non-abstract methods, while an interface only contains method signatures without any implementation.
2. **Multiple inheritance**: A class can only inherit from one abstract class, but it can implement multiple interfaces. This is because an abstract class defines some implementation details, which can lead to conflicts when multiple abstract classes are inherited.
3. **Access modifiers**: An abstract class can have **abstract and non-abstract methods** with any access modifier, while methods in an interface are by **default public and abstract.**
4. **Constructors:** An abstract class can have a constructor, while an interface cannot.
5. **Variables**: An abstract class can have instance variables, while an interface cannot. However, an interface can have constants (i.e., static final variables).

TW #7 Write a program to demonstrate the implementation of interfaces

Write a Java application to implement the following UML diagram.

- PrimeTester class implements isPrime() method by iterating from 2 to n-1 for a given number n
- ImprPrimeTester class implements isPrime() method by iterating from 2 to n/2
- FasterPrimeTester class implements isPrime() method by iterating from 2 to sqrt(n)
- FastestPrimeTester class implements isPrime() method using Fermat's Little theorem.
    - Fermat's Little Theorem:
    - If n is a prime number, then for every a, $1 < a < n-1$,
        - $a^{n-1} \% n = 1$

Note : Fermat's Little Theorem -The theorem states that if **p** is a prime number and **a** is any integer not divisible by **p**, then:
a^(p-1) ≡ 1 (mod p)


IPrime <<interface>>
isPrime(n:int):bool
PrimeTester
isPrime(n:int):bool
ImprPrimeTester
isPrime(n:int):bool
FasterPrimeTester
isPrime(n:int):bool
FastestPrimeTester
isPrime(n:int):bool

```java
interface IsPrime{
    boolean isPrime(int n);
}
class PrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        boolean flag=true;
        for(int i=2; i<=n-1; i++){
            if((n%i)==0)
            {
                flag=false;
                break;
            }
        }
        return flag;
    }
}
```

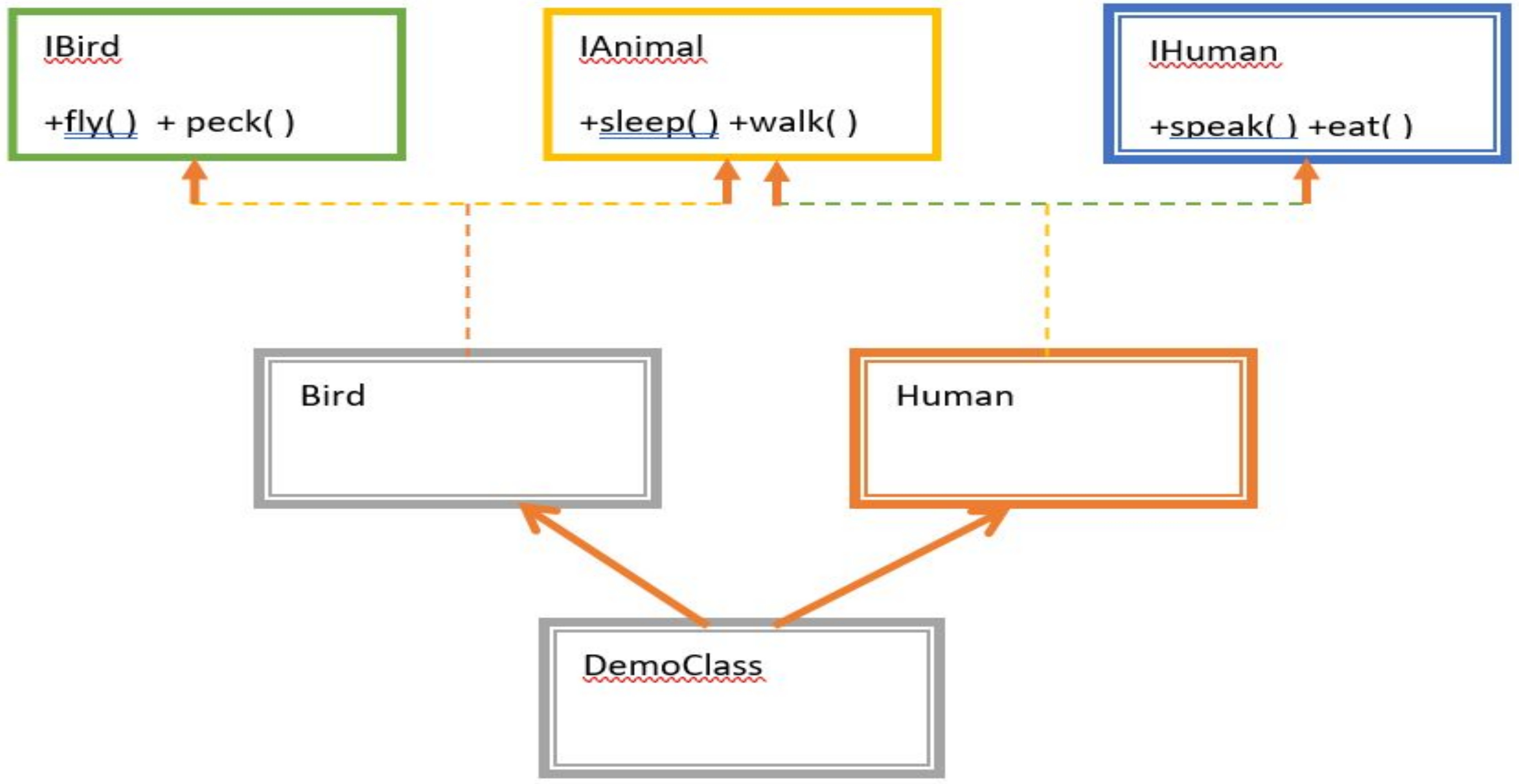
```java
class ImprPrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        boolean flag=true;
        for(int i=2; i<=n/2; i++){
            if((n%i)==0)
            {
                flag=false;
                break;
            }
        }
        return flag;
    }
}
```

```java
class FasterPrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        boolean flag=true;
        for(int i=2; i<Math.sqrt(n); i++){
            if((n%i)==0)
            {
                flag=false;
                break;
            }
        }
        return flag;
    }
}
```

```java
class FastestPrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        int a=2;
        return Math.pow(a,n-1)%n==1;
    }
}
```

```
public class TW7 {
    public static void main(String[] args) {
        PrimeTester p1=new PrimeTester();
        ImprPrimeTester p2=new ImprPrimeTester();
        FasterPrimeTester p3=new FasterPrimeTester();
        FastestPrimeTester p4=new FastestPrimeTester();
        System.out.println("32 is Prime? "+p1.isPrime(32));
        System.out.println("17 is Prime? "+p1.isPrime(17));
        System.out.println("32 is Prime? "+p2.isPrime(32));
        System.out.println("17 is Prime? "+p2.isPrime(17));
        System.out.println("32 is Prime? "+p3.isPrime(32));
        System.out.println("17 is Prime? "+p3.isPrime(17));
        System.out.println("32 is Prime? "+p4.isPrime(32));
        System.out.println("17 is Prime? "+p4.isPrime(17));
    }
}
```

- Design an interface IAnimal that has walk, and sleep methods, an interface IBird that has fly, and peck methods, an interface IHuman that has eat and speak methods. Construct a Bird class that implements IAnimal and IBird interfaces and also construct Human class that implements IAnimal and IHuman interfaces. Demonstrate the working of these methods by invoking the methods using appropriate reference variables.



```
// IAnimal interface
interface IAnimal {
   void walk();
   void sleep();
}

// IBird interface
interface IBird {
   void fly();
   void peck();
}

// IHuman interface
interface IHuman {
   void eat();
   void speak();
}
```

```
// Bird class implementing IAnimal and IBird interfaces
class Bird implements IAnimal, IBird {
   public void walk() {
      System.out.println("The bird is walking.");
   }

   public void sleep() {
      System.out.println("The bird is sleeping.");
   }

   public void fly() {
      System.out.println("The bird is flying.");
   }

   public void peck() {
      System.out.println("The bird is pecking.");
   }
}
```

```
// Human class implementing IAnimal and IHuman interfaces
class Human implements IAnimal, IHuman {
    public void walk() {
        System.out.println("The human is walking.");
    }

    public void sleep() {
        System.out.println("The human is sleeping.");
    }

    public void eat() {
        System.out.println("The human is eating.");
    }

    public void speak() {
        System.out.println("The human is speaking.");
    }
}
```

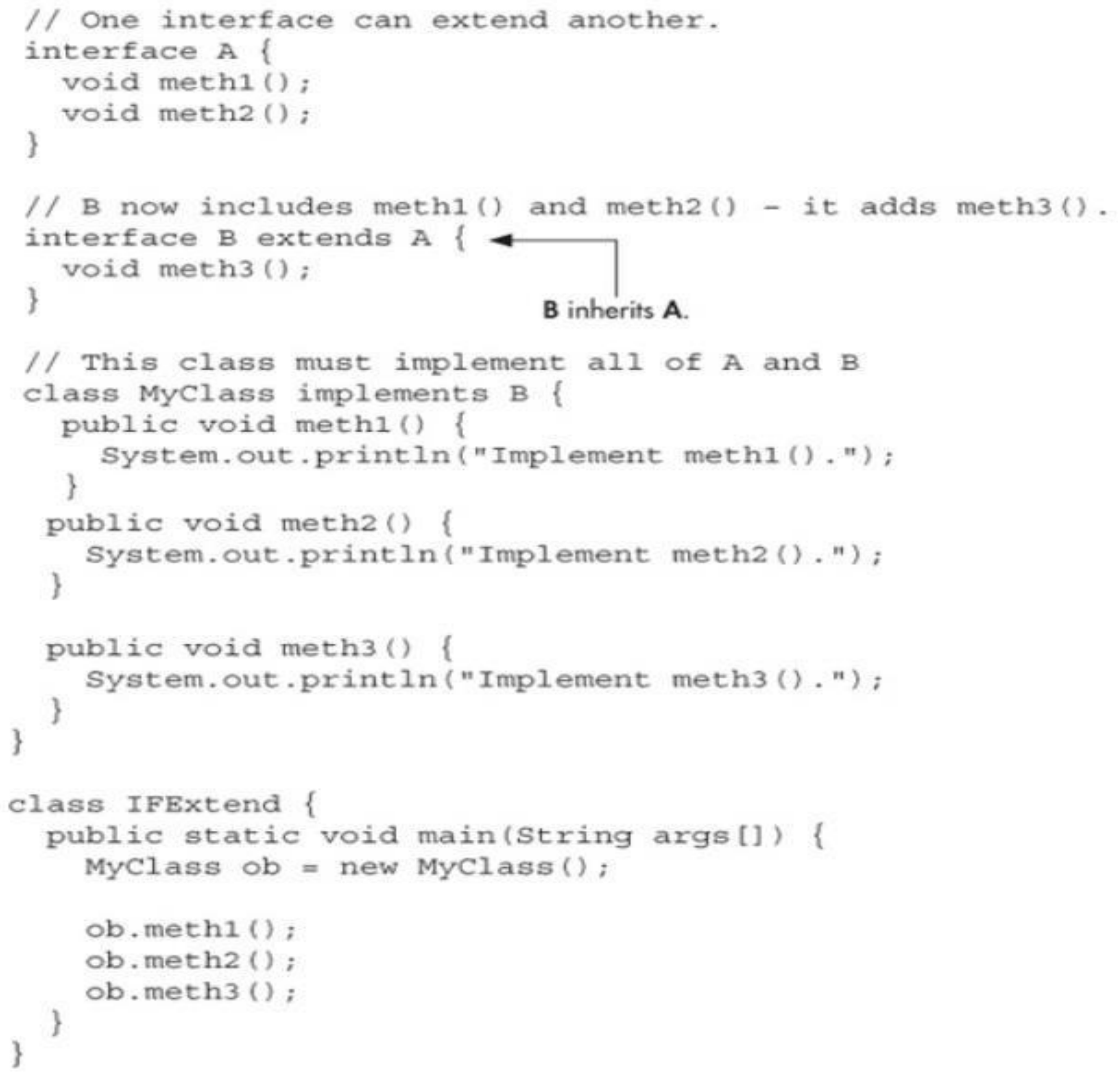
```
// Demo code
public class Demo {
    public static void main(String[] args) {
        // Create instances of Bird and Human classes
        Bird bird = new Bird();
        Human human = new Human();

        // Invoke methods using appropriate
reference variables
        bird.walk();
        bird.fly();
        bird.peck();
        bird.sleep();

        human.walk();
        human.eat();
        human.speak();
        human.sleep();
    }
}
```

OUTPUT

```
The bird is walking.
The bird is flying.
The bird is pecking.
The bird is sleeping.
The human is walking.
The human is eating.
The human is speaking.
The human is sleeping.
```

Interfaces can be Extended (FYI)

- One interface can inherit another by use of the keyword **extends**
- When class implements an interface that inherits another interface. It must provide implementations for all methods defined within the interface inheritance chain.

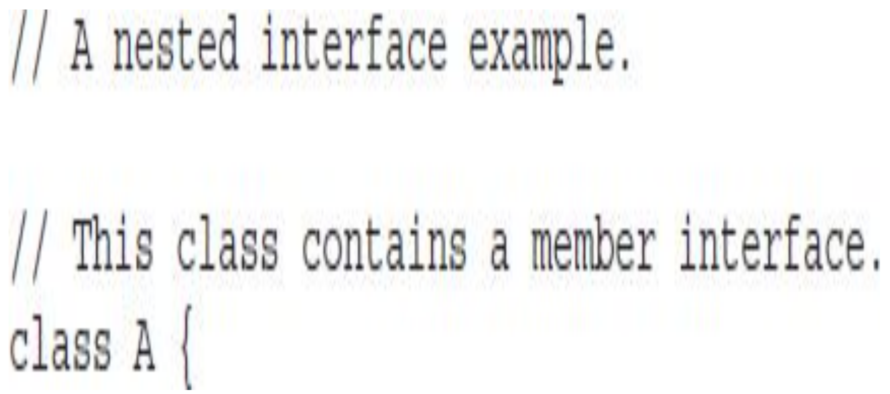


```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() - it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }
 public void meth2() {
   System.out.println("Implement meth2().");
 }

 public void meth3() {
   System.out.println("Implement meth3().");
 }
}

class IFExtend {
 public static void main(String args[]) {
   MyClass ob = new MyClass();

   ob.meth1();
   ob.meth2();
   ob.meth3();
 }
}
```

- In this example, interface A is extended by interface B
- MyClass implements B. This means that MyClass must implement all of the methods defined by both interfaces A and B.

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as public, private, or protected
- An interface nested in another interface is implicitly public
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member



```
// A nested interface example.

// This class contains a member interface.
class A {
    // this is a nested interface
    public interface NestedIF {
      boolean isNotNegative(int x);
    }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
  }
}

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

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

- **A** defines a member interface called **NestedIF** and that it is declared **public**.
- Next, **B** implements the nested interface by specifying
  implements A.NestedIF
- The name is fully qualified by the enclosing class name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**
- Since **B** implements only the nested interface NestedIF, it does not need to implement
  doSomething( ) method.

Sample Questions:

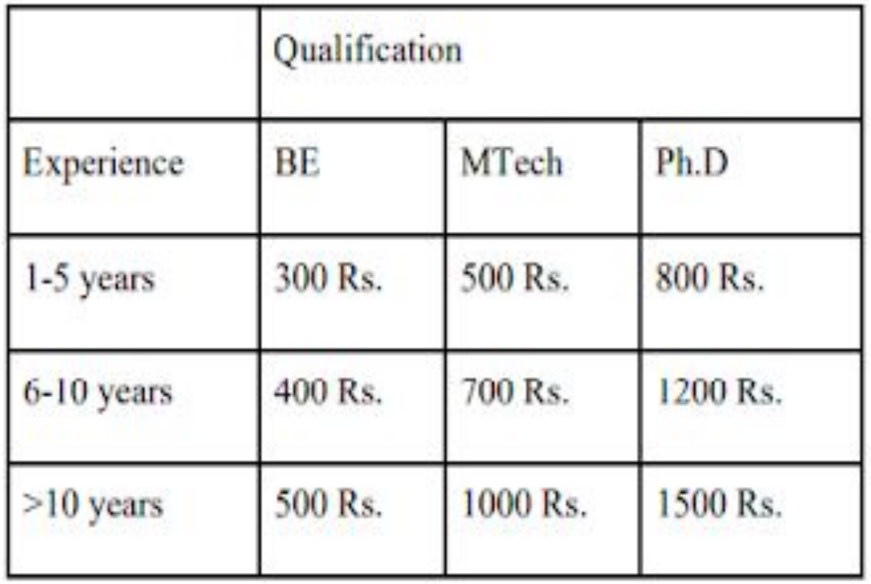
1. Define inheritance. And describe in brief the different forms of inheritance.L1,CO3,PO1,M-7
2. Compare and contrast method overloading and method overriding with examples
3. Explain with examples the effect of the keyword ‘final’ with respect to:

    - class
    - methods
    - variable

4. Define interface. Explain how to define and implement an interface by taking an example

TW – 4

- A company has two types of employees – Full Time and Part Time. The company records for each employee his/her name, age, address, salary and gender. Given the basic salary of the Full Time employee the components of his/her gross salary are: Dearness allowance – 75% of basic salary, HRA – 7.5% of basic salary, IT – 10% of basic. The salary of a Part time employee is dependent on the qualification, experience, number of working hours and the rate per hour, as below:



	Qualification		
Experience	BE	MTech	Ph.D
1-5 years	300 Rs.	500 Rs.	800 Rs.
6-10 years	400 Rs.	700 Rs.	1200 Rs.
>10 years	500 Rs.	1000 Rs.	1500 Rs.

```
class Employ{
    String name,address,gender;
    int age;
    double sal;
    Employ(String name,int age,String address,String gender){
        this.name=name;
        this.age=age;
        this.address=address;
        this.gender=gender;
    }
    void show(){
        System.out.println("Name:"+name);
        System.out.println("Age:"+age);
        System.out.println("Address:"+address);
        System.out.println("Gender:"+gender);
        System.out.println("Salay:"+sal);
    }   }
```

```
class FTEmploy extends Employ{
    int basSal;
    FTEmploy(String name,int age,String address,String gender,int basSal){
        super(name,age,address,gender);
        this.basSal=basSal;
    }

    void calSal(){
        sal=(basSal+basSal*0.75+basSal*0.075-basSal*0.1);
    }
}
```

```
class PTEmploy extends Employ{
    String qual;
    int exp,numHour;
    PTEmploy(String name, int age, String address, String gender, String
qual, int exp, int numHour){
        super(name,age,address,gender);
        this.qual=qual;
        this.exp=exp;
        this.numHour=numHour;
    }
```

```
void calSal(){
        switch(qual){
            case "BE":
                if(exp<=5) sal=numHour*300;
                else if(exp<=10) sal=numHour*400;
                else sal=numHour*500;
                break;
            case "MTech":
                if(exp<=5) sal=numHour*500;
                else if(exp<=10) sal=numHour*700;
                else sal=numHour*1000;
                break;
case "PhD":
                    if(exp<=5) sal=numHour*800;
                    else if(exp<=10) sal=numHour*1200;
                    else sal=numHour*1500;
                    break;
          }
     }
  }
```

```
public class Tw4{
    public static void main(String []args){
        FTEmploy f1=new FTEmploy("Arun",25,"Vijayapur","Male",10000);
        f1.calSal();
        System.out.println("Details of Full time Employ");
        f1.show();
        PTEmploy e1=new PTEmploy("Rohit",30,"Belgum","Male","BE",6,10);
        e1.calSal();
        System.out.println("\nDetails of Part time Employ 1:");
        e1.show();
        PTEmploy e2=new PTEmploy("Rohini",26,"Mysore","Female","PhD",10,8);
        e2.calSal();
        System.out.println("\nDetails of Part time Employ 2:");
        e2.show();
    }
}
```