Java - Method Overloading

Method Overloading

f a class has multiple methods having same name but different in parameters, it is known as Method Overloading

Different ways to overload the method
There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

```
public class OverloadDemo {
      // Method to add two integers
       private int add(int a, int b) {
           return a + b;
       }
       // Overloaded method to add two double values
       private double add(double a, double b) {
           return a + b;
       }
       // Overloaded method to add three integers
       private int add(int a, int b, int c) {
           return a + b + c;
       }
```

```
 public static void main(String[] args) {
      OverloadDemo example = new OverloadDemo();

          // Using the first add method
      System.out.println("Sum of two integers: " + example.add(10, 20));

          // Using the second add method
     System.out.println("Sum of two doubles: " + example.add(10.5, 20.5));

          // Using the third add method
     System.out.println("Sum of three integers: " + example.add(10, 20, 30));
      }
}
```

Java - Inheritance

●●●

Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance gives you the opportunity to add some functionality that does not exist in the original class.
- The subclass and the superclass has an "is-a" relationship / parent-child relationship.
  - Vehicle is super class of Car.
  - Car is sub class of Vehicle.

Inheritance

- Within a subclass you can access its superclass's public and protected methods and fields, but not the superclass's private methods.
- If the subclass and the superclass are in the same package, you can also access the superclass's default methods and fields.

- simpleinheritance.java

Inheritance - Why?

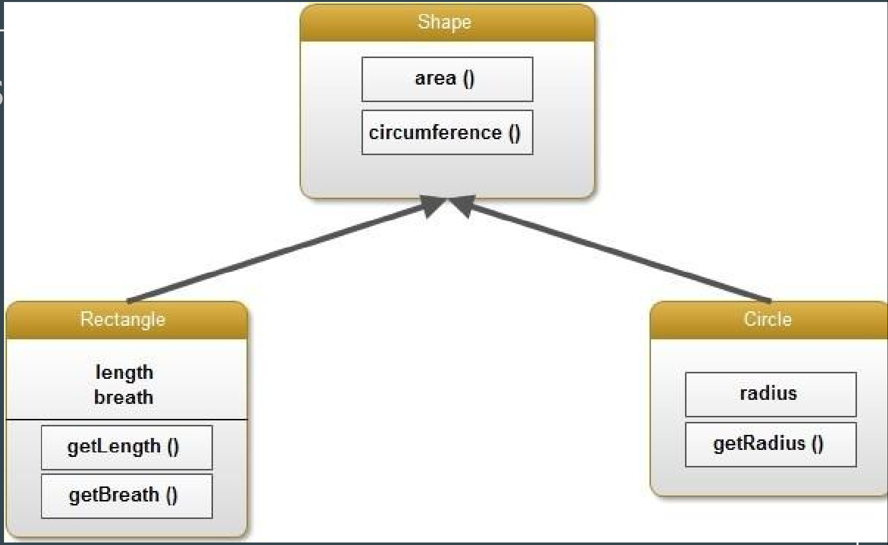
1. For Method Overriding (so runtime polymorphism can be achieved).
2. It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived 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

Inheritance - extends keyword

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



- “extends” indicates that you are making a new class that derives from an existing class. ["extends" means to increase the functionality].
- A class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Inheritance - extend keyword

```
class Animal{

  private String type;
  public Animal(String aType)
  { type = aType;
  }
  public String toString(){
  return "This is a " + type;
  }

}
```

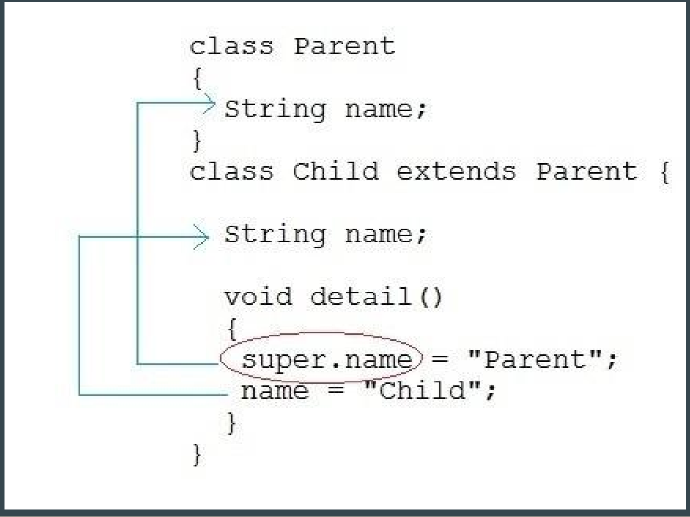
```
class Dog extends Animal{

  private String breed;



  public Dog(String name)
  { super(name);
  }


}
```

Inheritance - super Keyword

- The keyword super represents an instance of the direct superclass of the current object.
- You can explicitly call the parent's constructor from a subclass's constructor by using the super keyword.
- 'super' must be the first statement in the constructor.

```
class Parent {
      public Parent(){

       }
     }
```

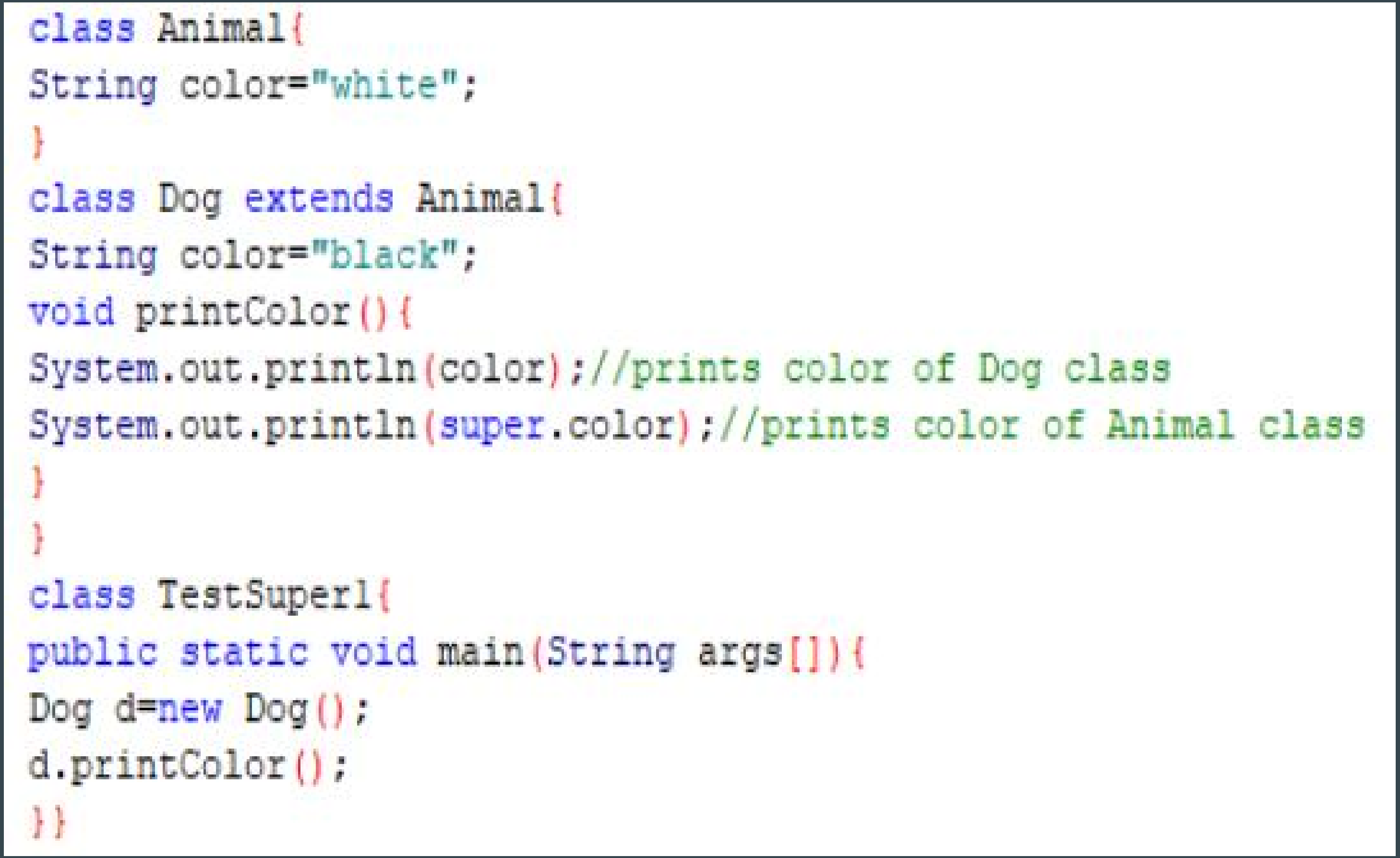
```
public class Child extends Parent
        { public Child () {
          super();
        }
    }
```



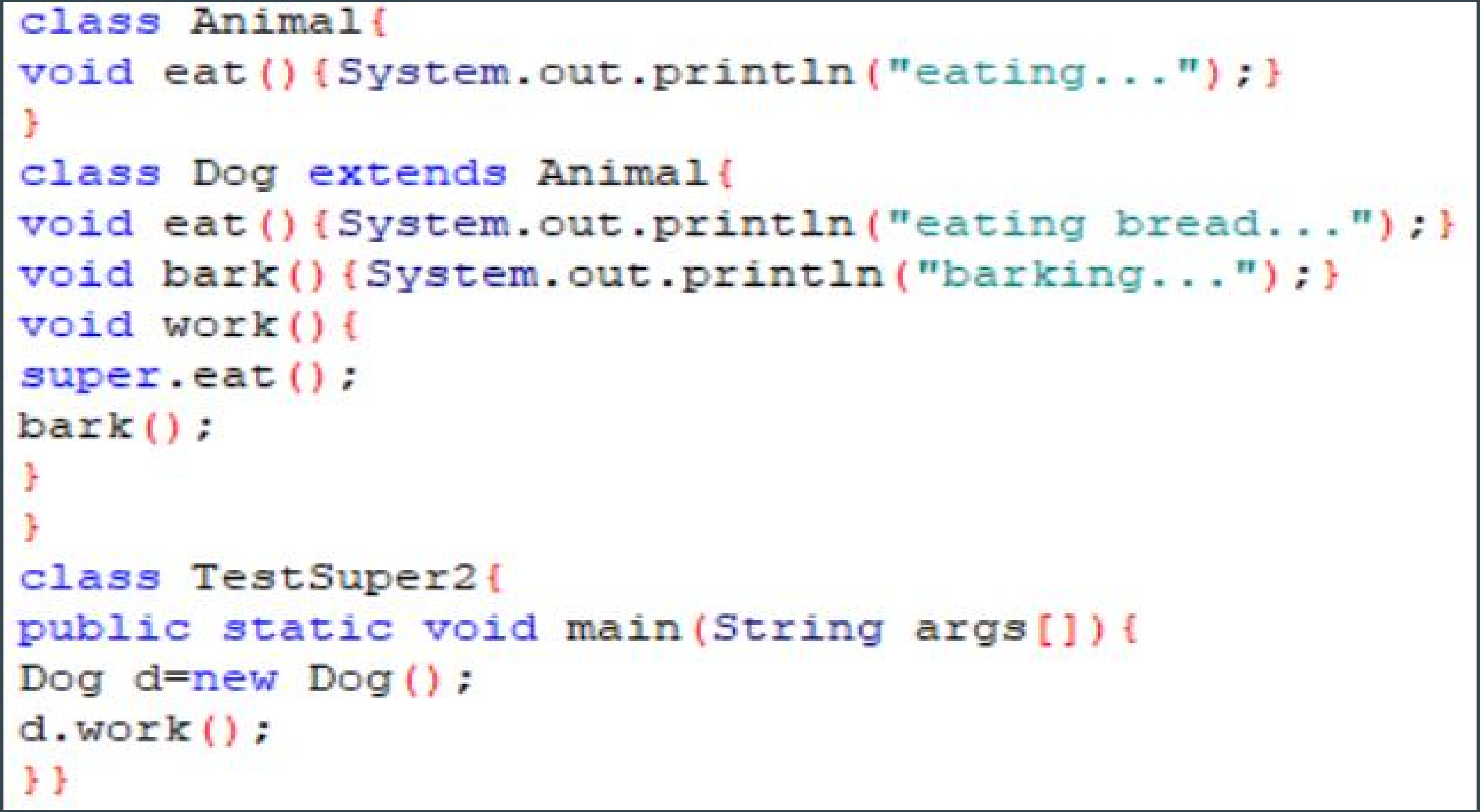
```
class Parent
{
 String name;
}
class Child extends Parent {

 String name;

 void detail()
 {
  super.name = "Parent";
  name = "Child";
 }
}
```

Inheritance - super Keyword

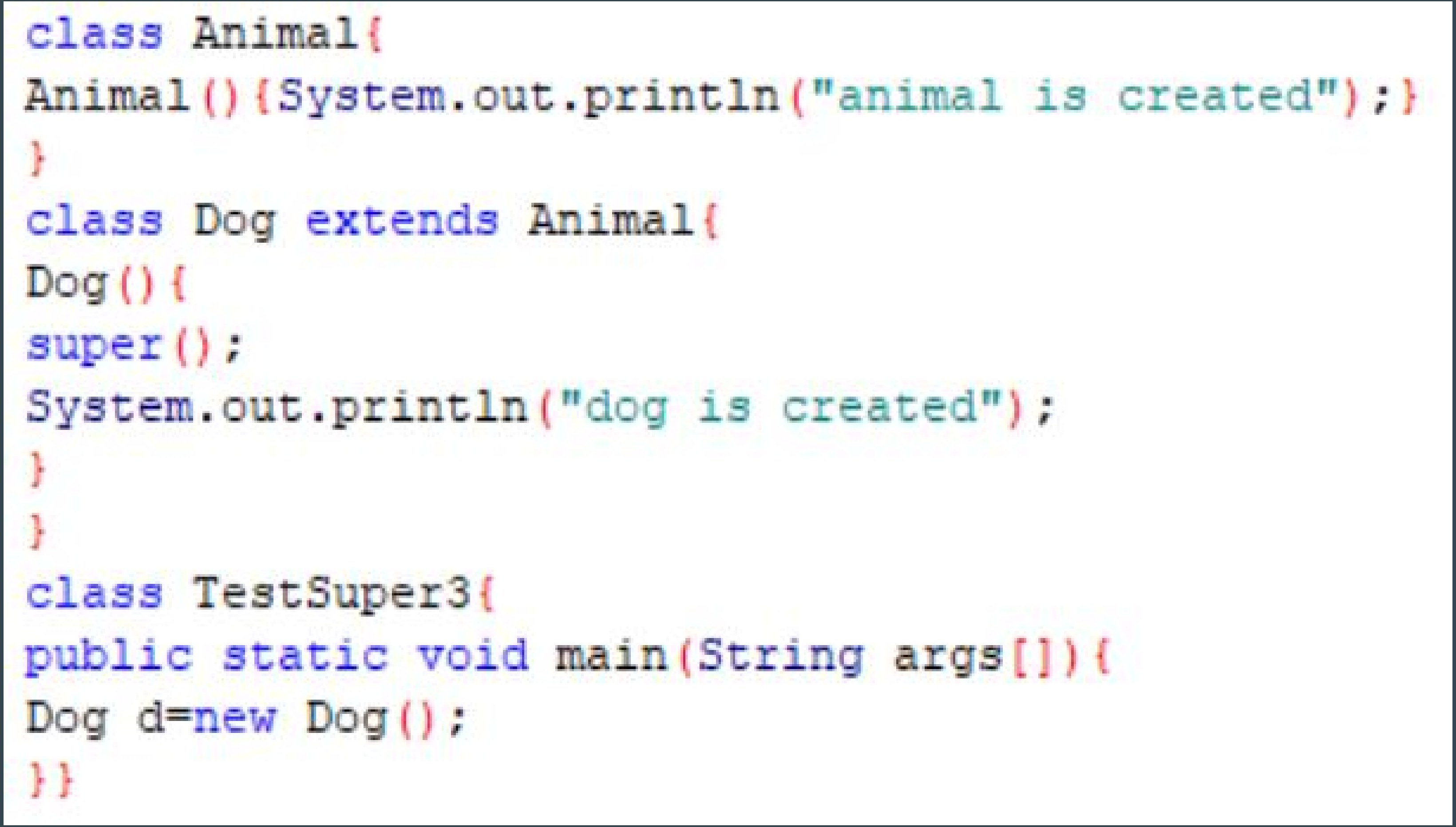
1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```



```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Inheritance - Example

```
class Animal {

  private String type;
  public Animal(String aType)
    {  type = aType;
  }
  public String toString()
    {   return "This is a " +
    type;
  }
}
```

```
class Dog extends Animal
 {  private String name;
 private String breed;
 public Dog(String aName)
  {  super("Dog");
  name =  aName;
  breed = "Unknown";
 }
 public Dog(String aName, String aBreed)
  {  super("Dog");
  name = aName;
  breed = aBreed;
 }}
```

Superdemo.java

Inheritance - Example

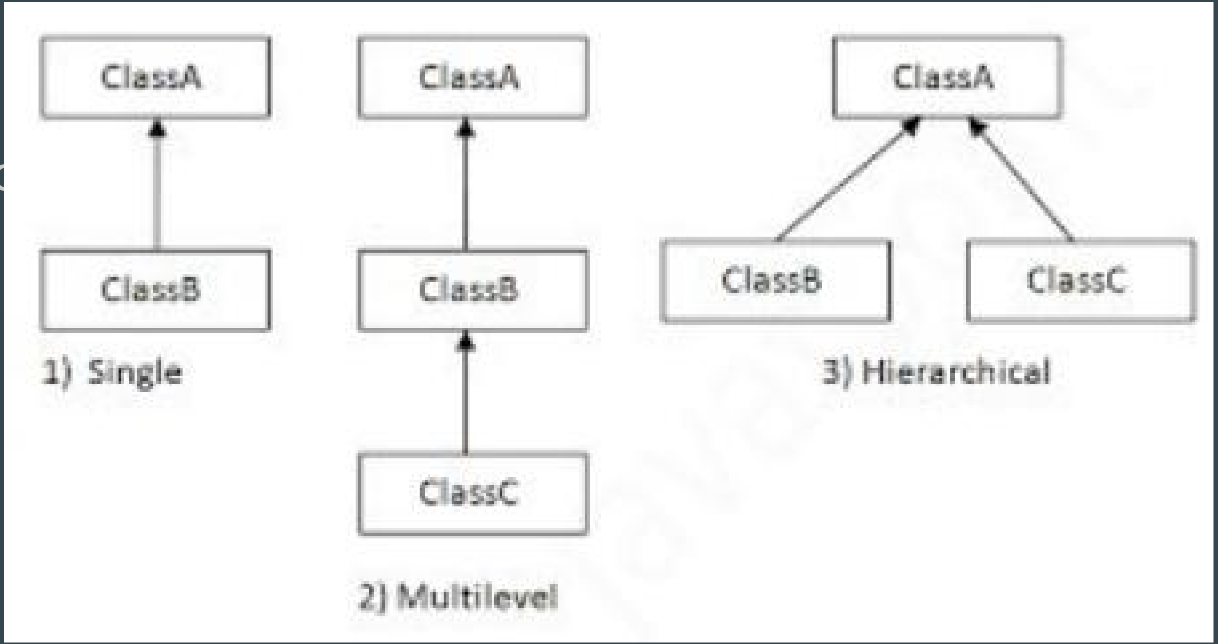
```
class
    Parent
    {  String
    name;
}
public class Child extends Parent
    {  String name;
    public void details(){
    super.name =  "Parent"; //refers to parent class
member
    name  =  "Child";
    System.out.println(super.name+" and "+name);
    }
    public static void main(String[]
    args){ Child cobj =  new Child();
    cobj.details();
    }
}
```

```
class Parent{
        String name;
        public void
                details(){   name
                = "Parent";
                System.out.println(name);
        }
}
public class Child extends Parent
        {  String name;
        public void details(){
                super.details();     //calling Parent class
details() method
                name  =  "Child";
                System.out.println(name);
        }
        public static void main(String[] args){
                Child cobj = new Child();
                cobj.details();
        }
```

Inheritance - Types

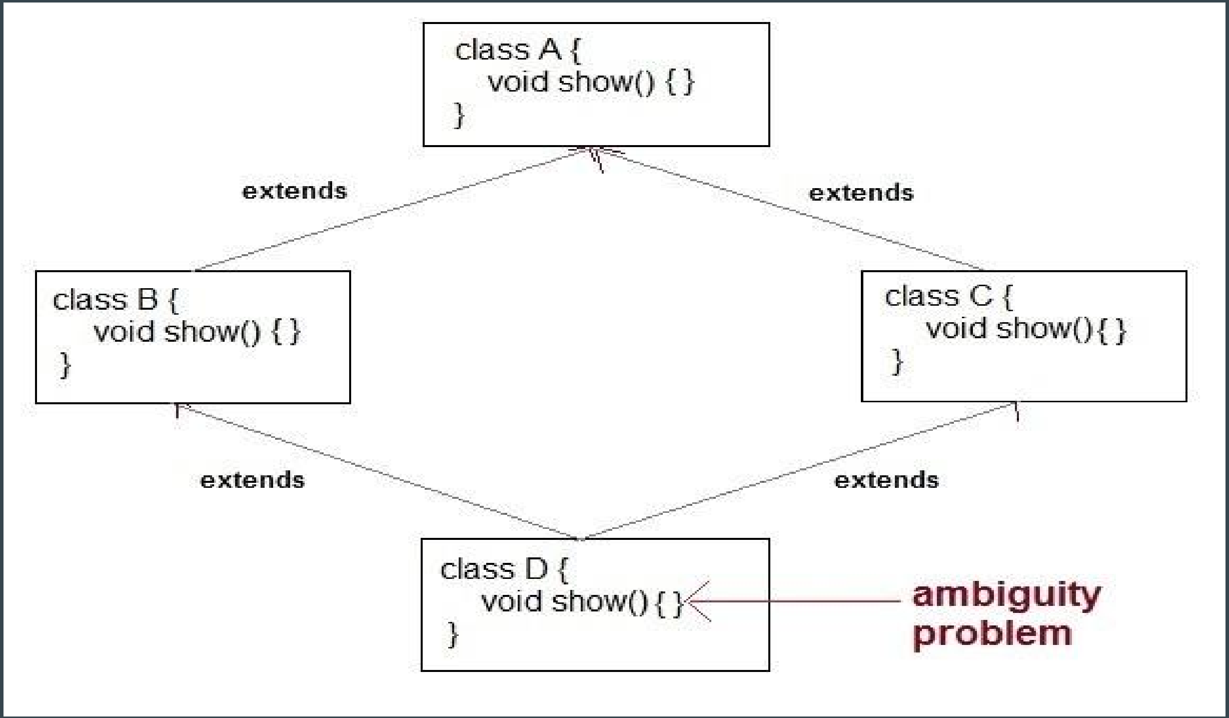
Java mainly supports only three types of inheritance that are listed below.

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritanc



multiple inheritance in Java?

- To remove ambiguity.
- To provide more maintainable and clear design.



Multilevel Inheritance

- hierarchies that contain as many layers of inheritance
- As mentioned, subclass become 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 properties found in all of its superclasses.
- In this case, C inherits all aspects of B and A.

- multilevelInheritance.java

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

Box() {
     width = -1;
      height = -1;
     depth = -1; // box
   }
```

```
 double volume() {
    return width * height *
          depth;
    }
} // end of class box
```

```
class BoxWeight extends Box {
 private double weight;
  // constructor when all parameters
are specified
BoxWeight(double w, double h, double d,
double m) {

  super(w, h, d); // call superclass
                      constructor
      weight = m;
  }

 BoxWeight() {
    super(); // call superclass
                      constructor
    weight = -1;
  }
```

```
 void dispweight(){

  System.out.println
("Weight of box is :"+weight);
}

 // method
 double getWeight(){
       return weight;
   }

}
```

```
class Shipment extends BoxWeight {
 double cost;
  // constructor when all
parameters are specified
  Shipment(double w, double h,
double d, double m, double c) {
     super(w, h, d, m); // call
superclass constructor
      cost = c;
     }
 // default constructor
     Shipment() {
       super();
       cost = -1;
     }
```

```
 void dispcost()  {
    double vol = volume();
    System.out.println
       ("volume is :" + vol);
   dispweight();
   double wgh = getWeight();
   System.out.println
     ("Cost is :" + wgh*cost);
     }

}
```

```
class MultiDemo2 {
    public static void main(String arg[]){
      Shipment shipment1 =  new Shipment(10, 20, 15, 10, 31.41);
      Shipment shipment2 =  new Shipment(2, 3, 4, 0.76, 11.28);
        shipment1.dispcost();
        shipment2.dispcost();
    }
}
```

Hierarchy of Constructors

- class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.
- If super( ) is not used, then the default or **parameterless constructor of each superclass will be executed**

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

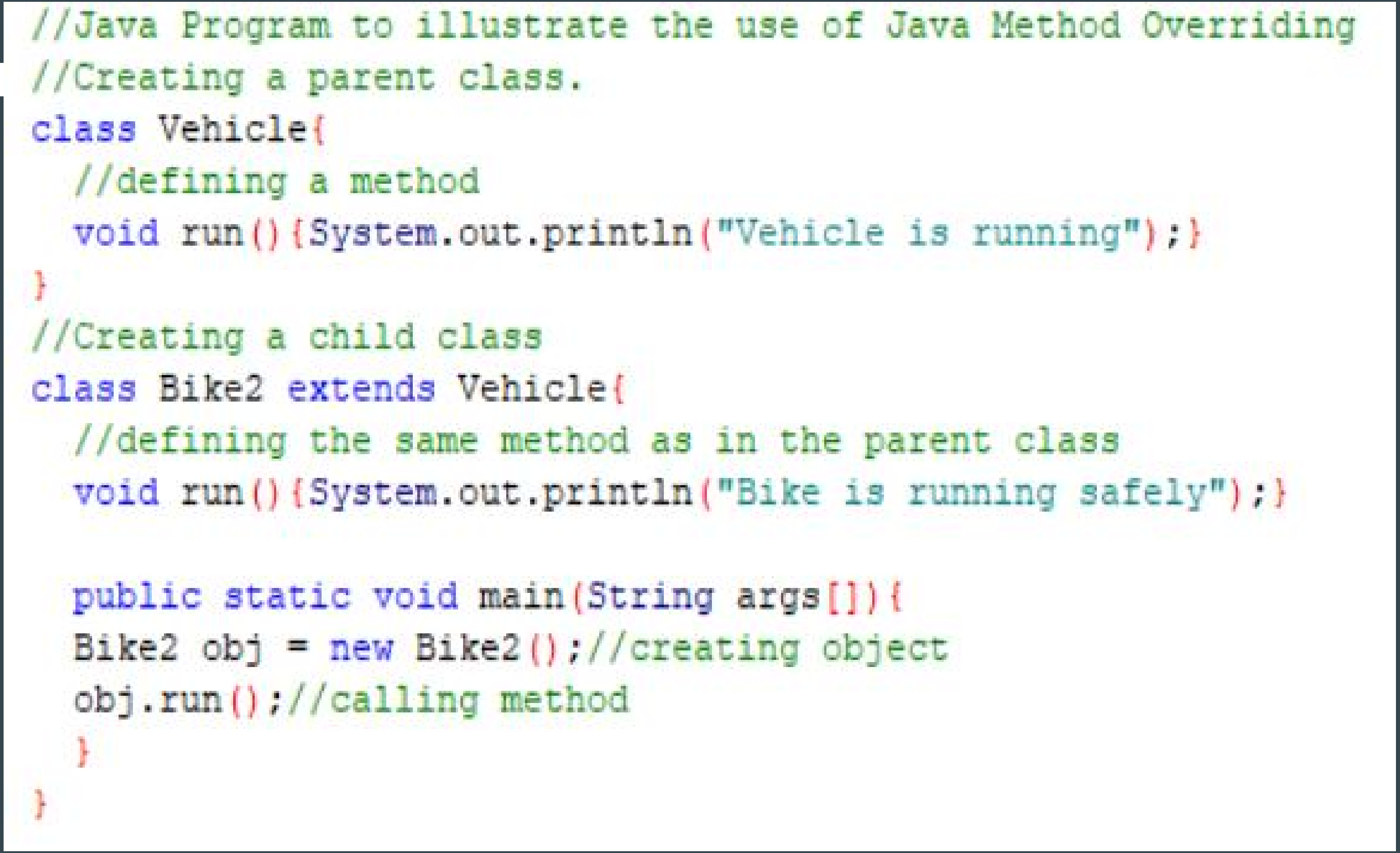
```
class C extends B {
C() {
System.out.println("Inside C's
constructor.");
    }
}
class CallingCons {
public static void main(String args[]) {
      C c = new C();
   }
}
```

```
Output:
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Inheritance - Method Overriding

- When you extends a class, you can change the behavior of a method in the parent class.
- This is called method overriding.
- This happens when you write in a subclass a method that has the same signature as a method in the parent class.
- If only the name is the same but the list of arguments is not, then it is method overloading.

Inher

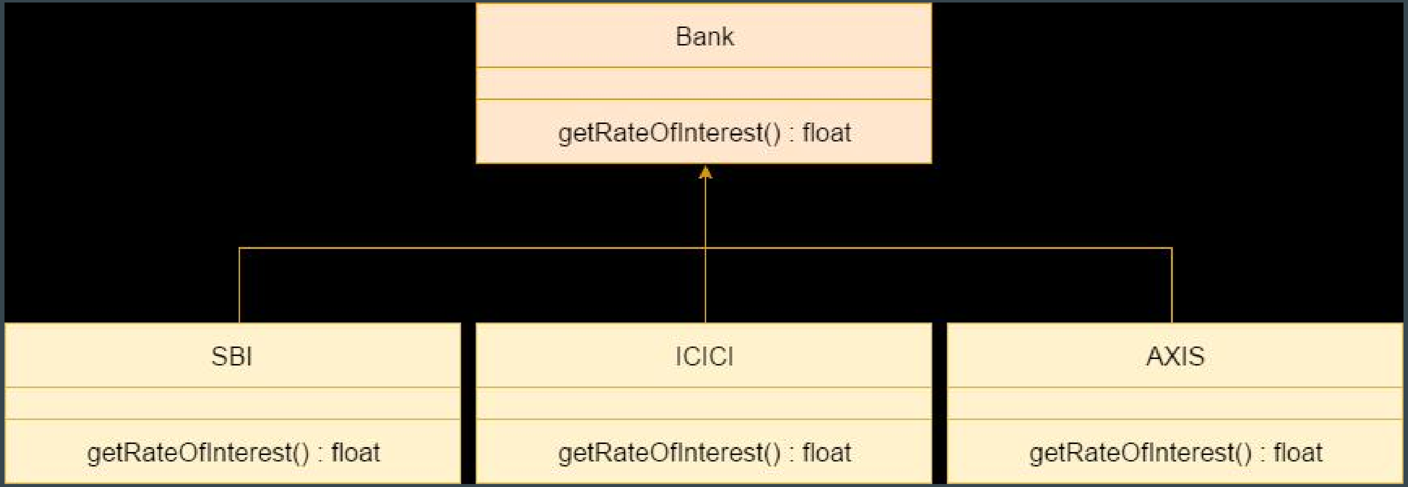


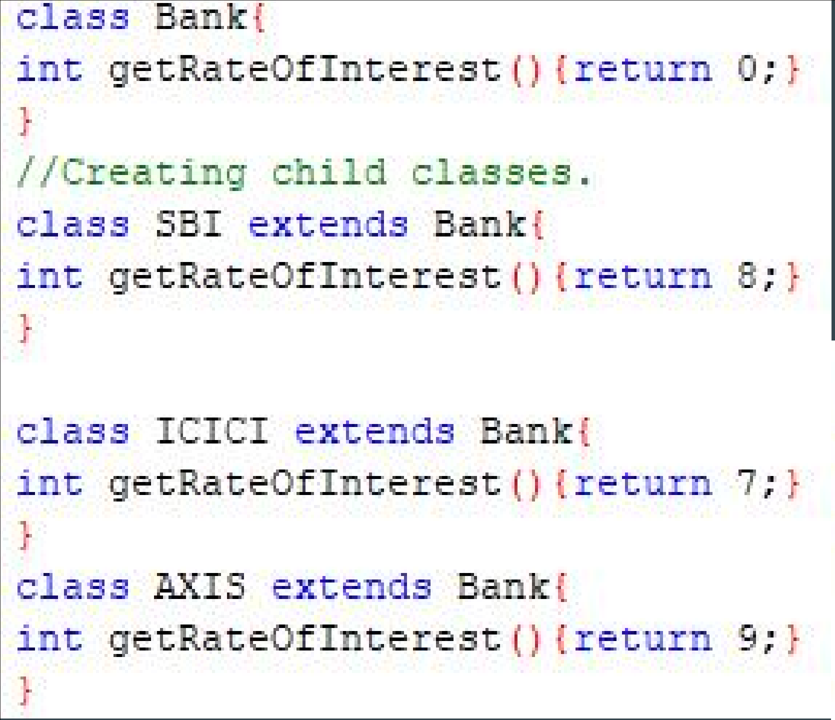
```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Inheritance - Method Overriding

A real example of Java Method Overriding

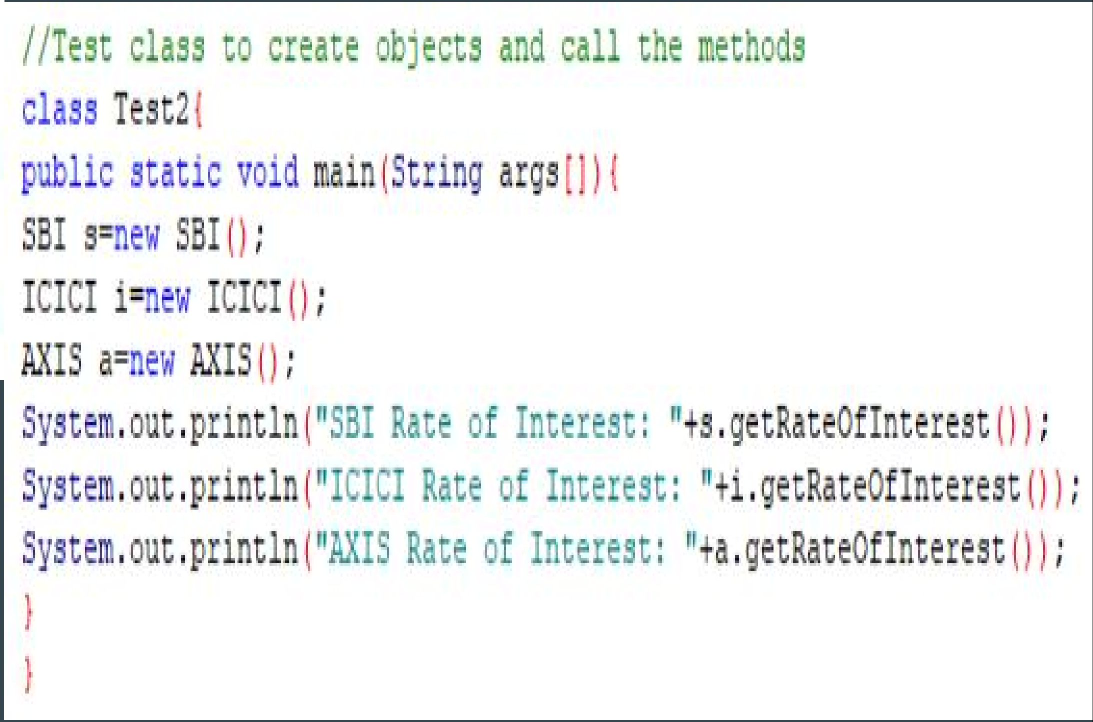
Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.


Bank
getRateOfInterest() : float
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float



```java
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
```

OverrideDemo.java



```java
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

Output :

```
SBI Rate of Interest : 8
ICICI Rate of Interest : 7
Axis Rate of Interest : 9
```

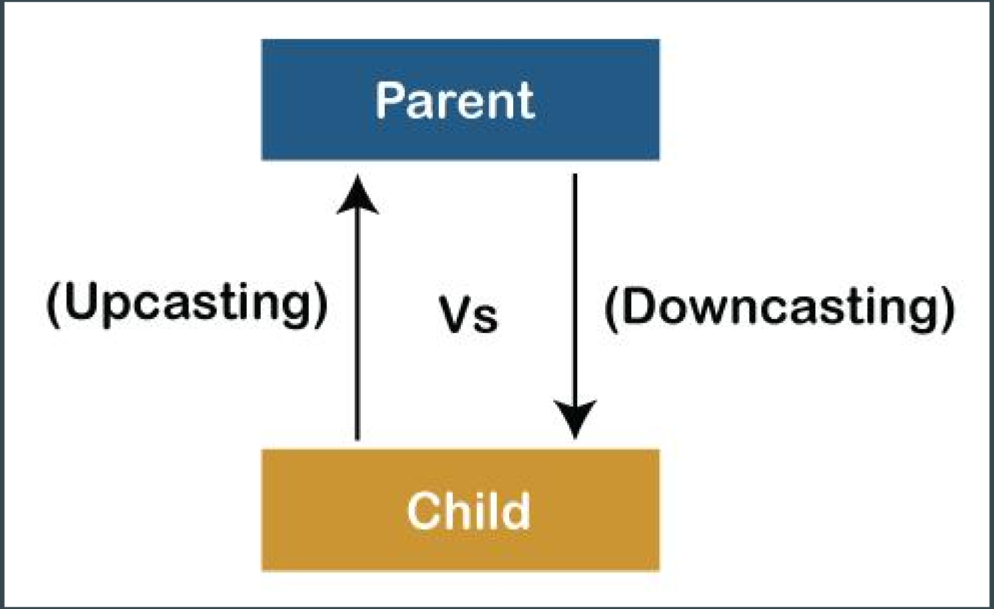
Inheritance - Type Casting

A process of converting one data type to another is known as Typecasting
Upcasting and Downcasting is the type of object typecasting.

In Java, the object can also be typecasted like the datatypes.
Parent and Child objects are two types of objects.
So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting.

Inheritance - Type Casting

In Upcasting and Downcasting, we typecast a child object to a parent object and a parent object to a child object.

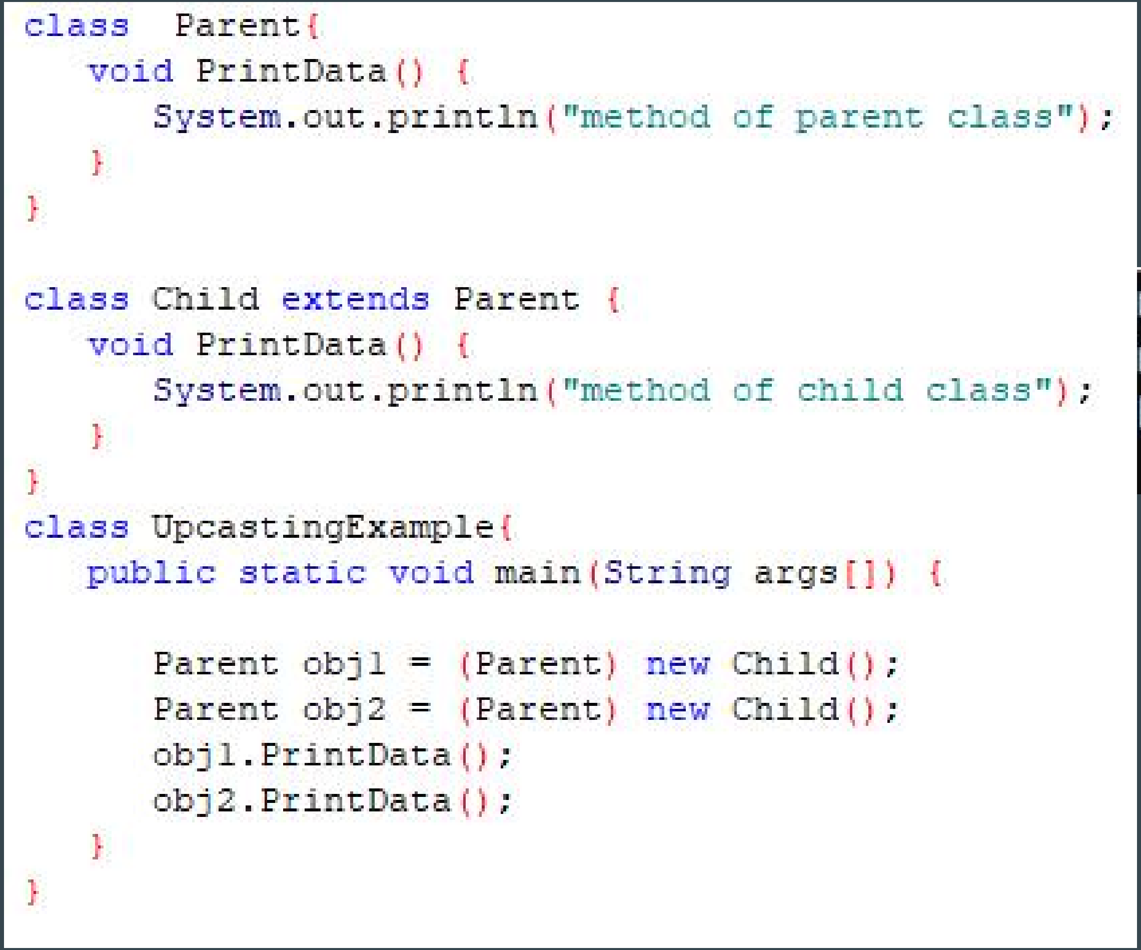


Inheritance - Type Casting

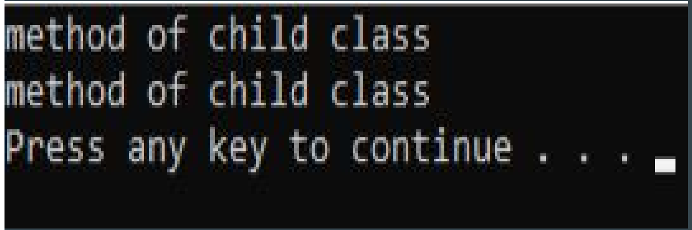
1) Upcasting

Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object.

By using the Upcasting, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method.
We access only some specified variables and methods of the child class.
Upcasting is also known as Generalization and Widening.



```
class  Parent{
   void PrintData() {
      System.out.println("method of parent class");
   }
}

class Child extends Parent {
   void PrintData() {
      System.out.println("method of child class");
   }
}
class UpcastingExample{
   public static void main(String args[]) {

      Parent obj1 = (Parent) new Child();
      Parent obj2 = (Parent) new Child();
      obj1.PrintData();
      obj2.PrintData();
   }
}
```



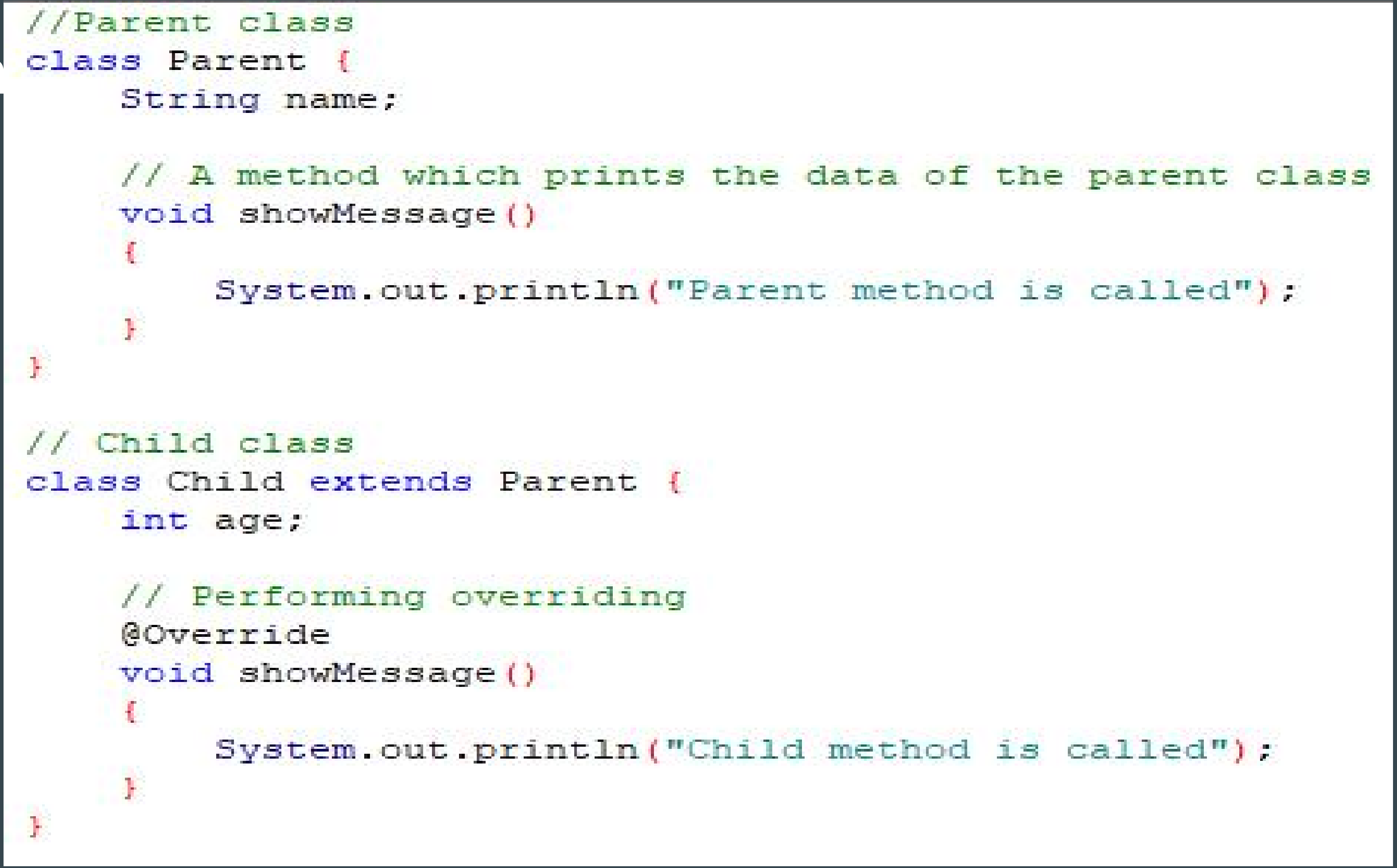
```
method of child class
method of child class
Press any key to continue . . .
```

Inheritance - Type Casting

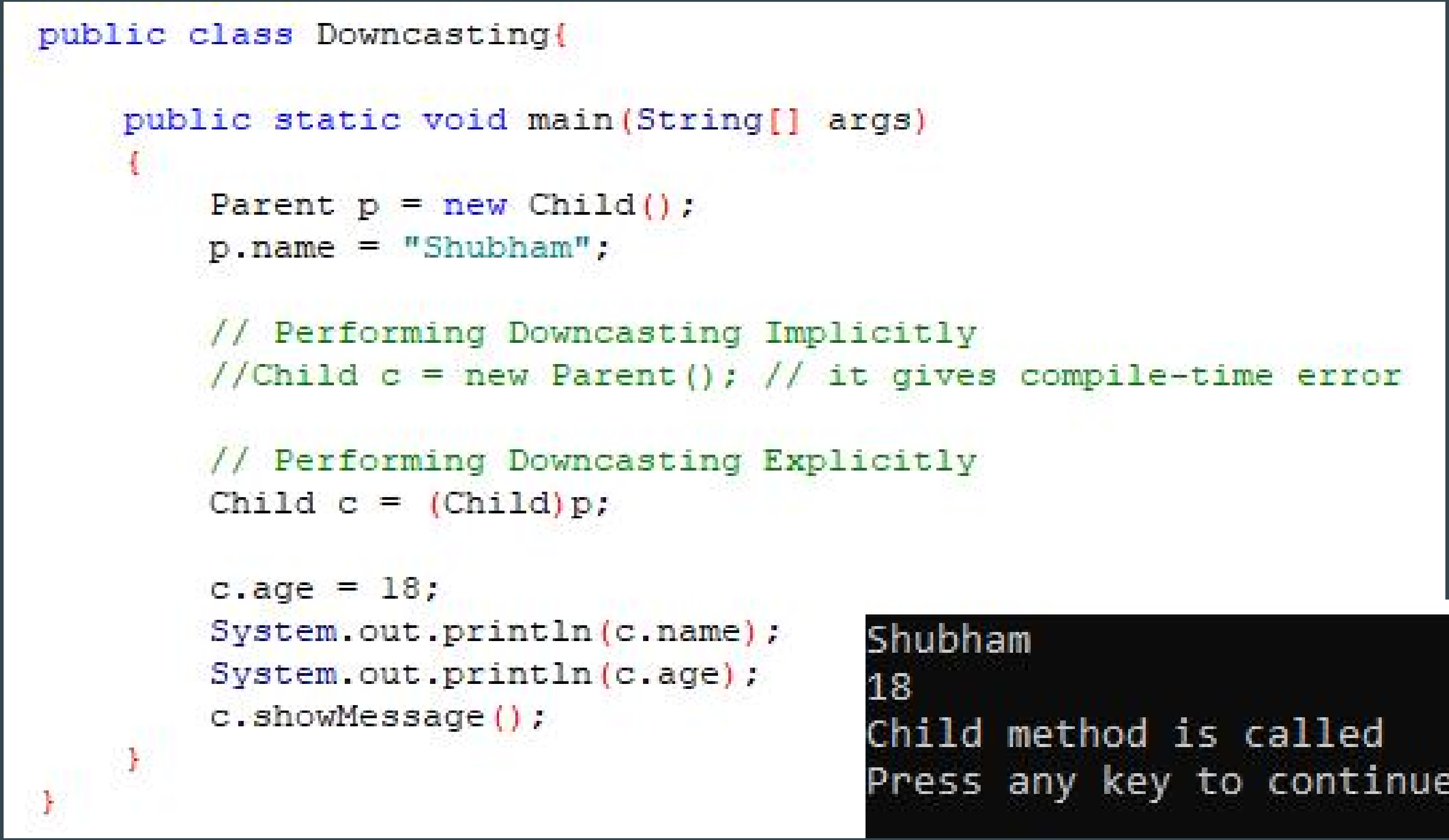
2) Downcasting

Upcasting is another type of object typecasting.
In Upcasting, we assign a parent class reference object to the child class.
In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the "ClassCastException".

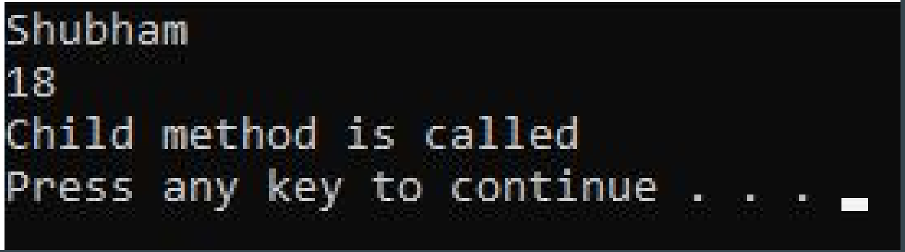
In



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

    // A method which prints the data of the parent class
    void showMessage()
    {
        System.out.println("Parent method is called");
    }
}

// Child class
class Child extends Parent {
    int age;

    // Performing overriding
    @Override
    void showMessage()
    {
        System.out.println("Child method is called");
    }
}
```



```java
public class Downcasting{

    public static void main(String[] args)
    {
        Parent p = new Child();
        p.name = "Shubham";

        // Performing Downcasting Implicitly
        //Child c = new Parent(); // it gives compile-time error

        // Performing Downcasting Explicitly
        Child c = (Child)p;

        c.age = 18;
        System.out.println(c.name);
        System.out.println(c.age);
        c.showMessage();
    }
}
```

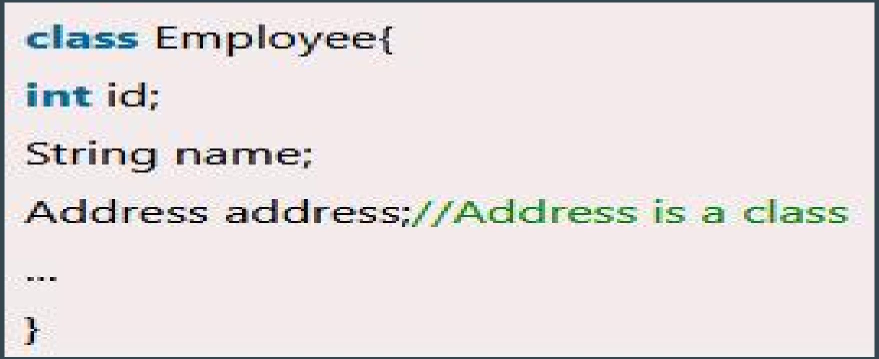


```
Shubham
18
Child method is called
Press any key to continue . . .
```

Aggregation (HAS-A relationship) in Java

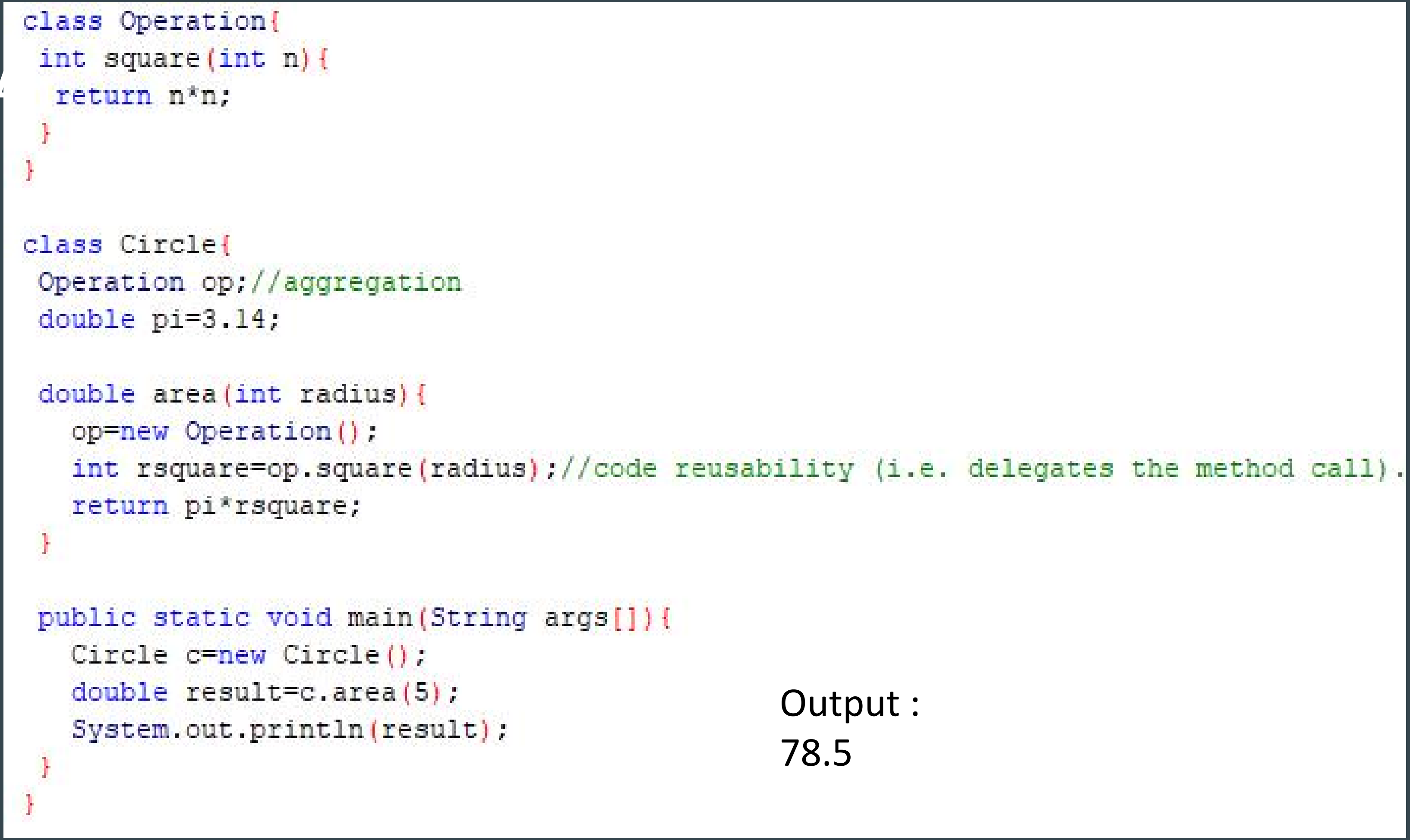
If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.



```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

A



```java
class Operation{
 int square(int n){
  return n*n;
 }
}

class Circle{
 Operation op;//aggregation
 double pi=3.14;

 double area(int radius){
   op=new Operation();
   int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
   return pi*rsquare;
 }

 public static void main(String args[]){
   Circle c=new Circle();
   double result=c.area(5);
   System.out.println(result);
 }
}
```

Output :
78.5

Java – Abstract Classes

What is an Abstract class?

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract and non-abstract methods (method with the body).
- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Abstract class

- A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented. It cannot be instantiated.
- Points to Remember
  -
  - An abstract class must be declared with an abstract keyword. It can have abstract and non-abstract
  - methods.
  - It cannot be instantiated.

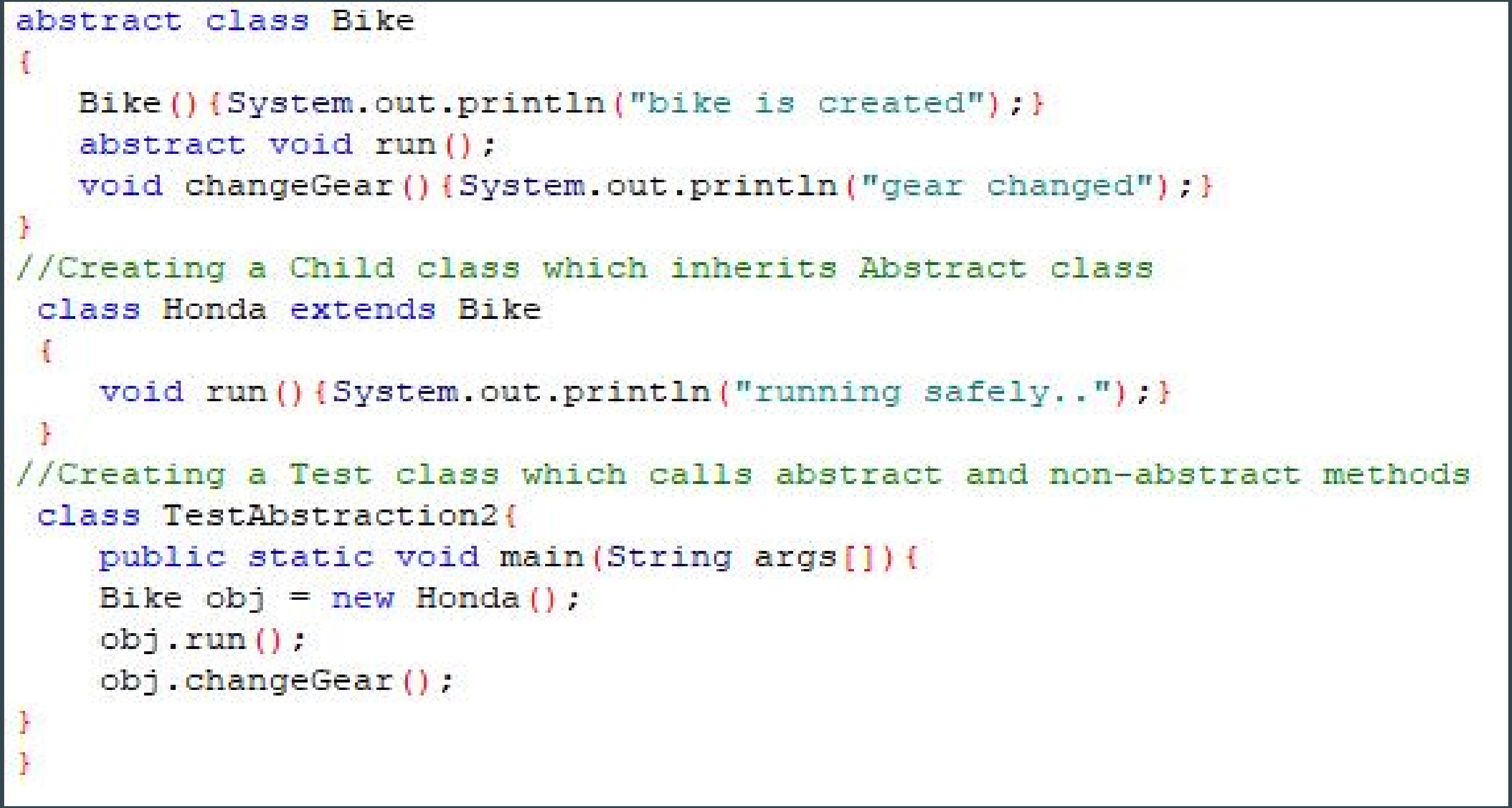
    It can have constructors and static methods also.

    It can have final methods which will force the subclass not to change the body of the method.

Abstract method

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- abstract void printStatus();//no method body and abstract

Abstract class having constructor, data member and methods



```
abstract class Bike
{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
}
//Creating a Child class which inherits Abstract class
 class Honda extends Bike
 {
    void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2{
    public static void main(String args[]){
    Bike obj = new Honda();
    obj.run();
    obj.changeGear();
}
}
```

Abstract Method

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.
- The abstract keyword is also used to declare a method as abstract. An abstract methods consist of a method signature, but no method body.
- The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- Any child class must either override the abstract method or declare itself abstract.
- A child class that inherits an abstract method must override it. If they do not, they must be abstract,and any of their children must override it.

Abstract Class

- An abstract class cannot be instantiated.
- All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner.
- You just cannot create an instance of the abstract class.
- If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclassed.
- Use the abstract keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the class keyword.
- AbstractDemo.java

Final Method

When final is applied to the method , it provides the restriction of method overriding in subclass.
Method declared as final cannot be overridden

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

Final Class

Declaring class as a final implicitly declares all of its method as final too.

Final class can not be inherited

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

Java - Interface

What is an Interface?

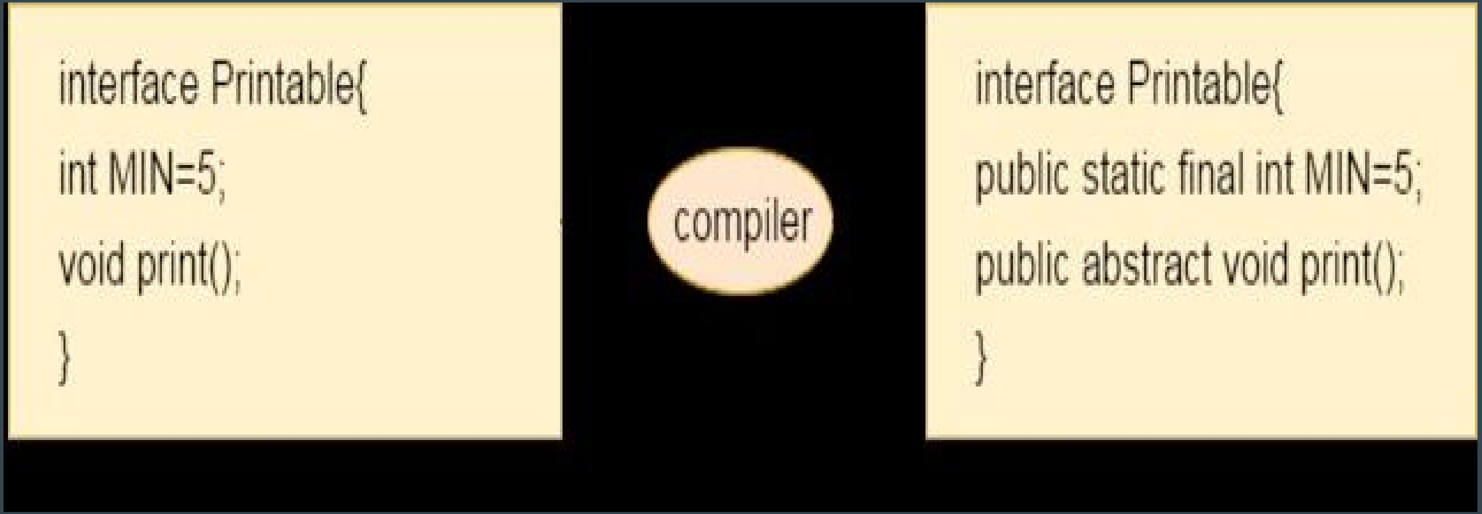
- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- nterfaces can have abstract methods and variables. It cannot have a method body.
- It cannot be instantiated just like the abstract class.
-

How to declare an interface?

- 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.

```
interface <interface_name>{
   return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
 type final-varname1 = value;
 type final-varname2 = value;
 //...
}
```

Internal addition by the compiler

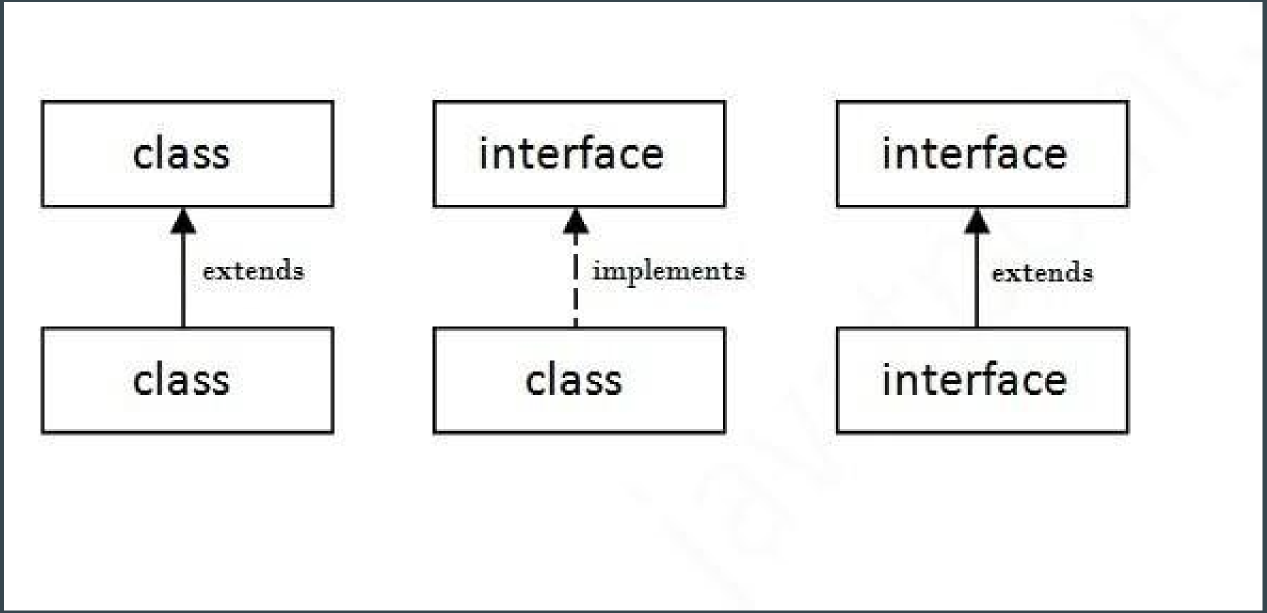


```
interface Printable{
int MIN=5;
void print();
}
```

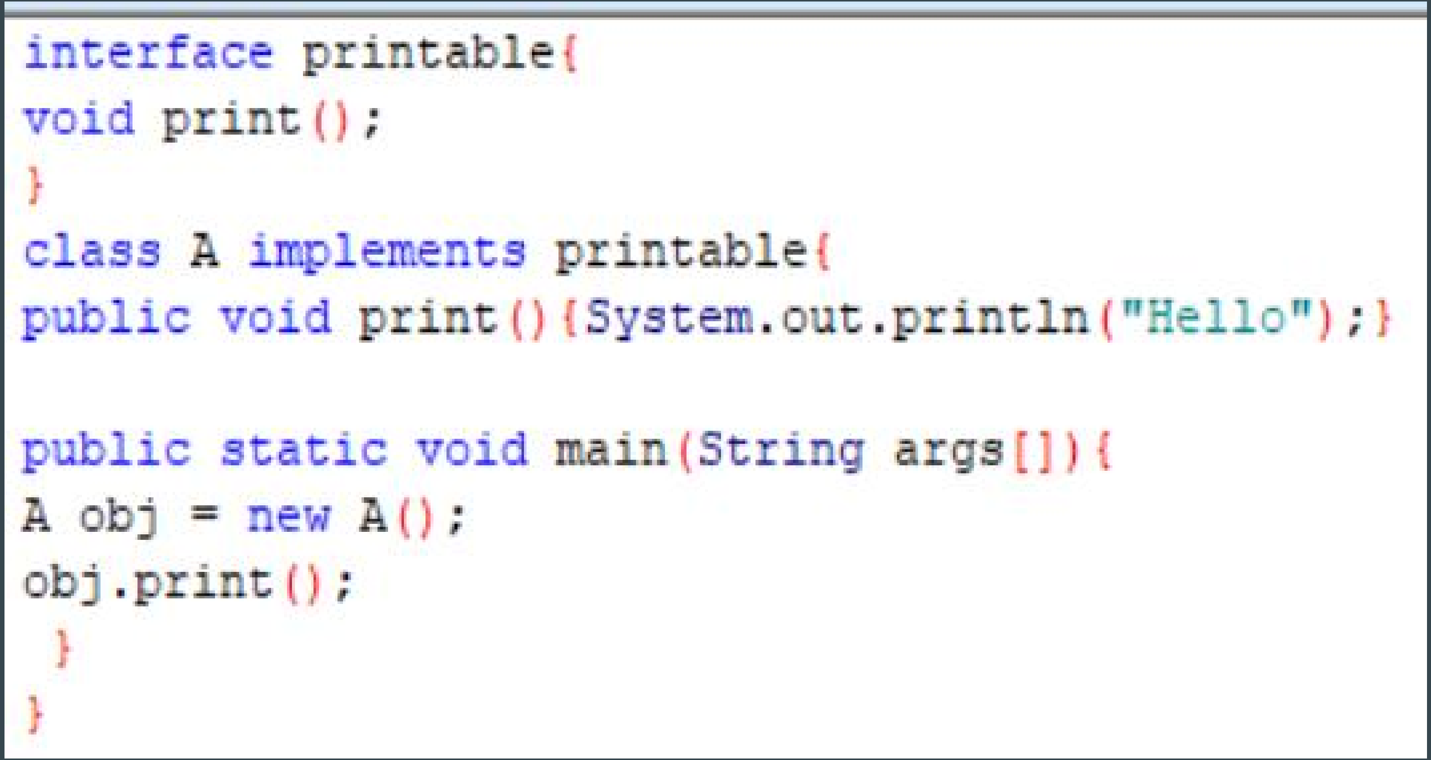
```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



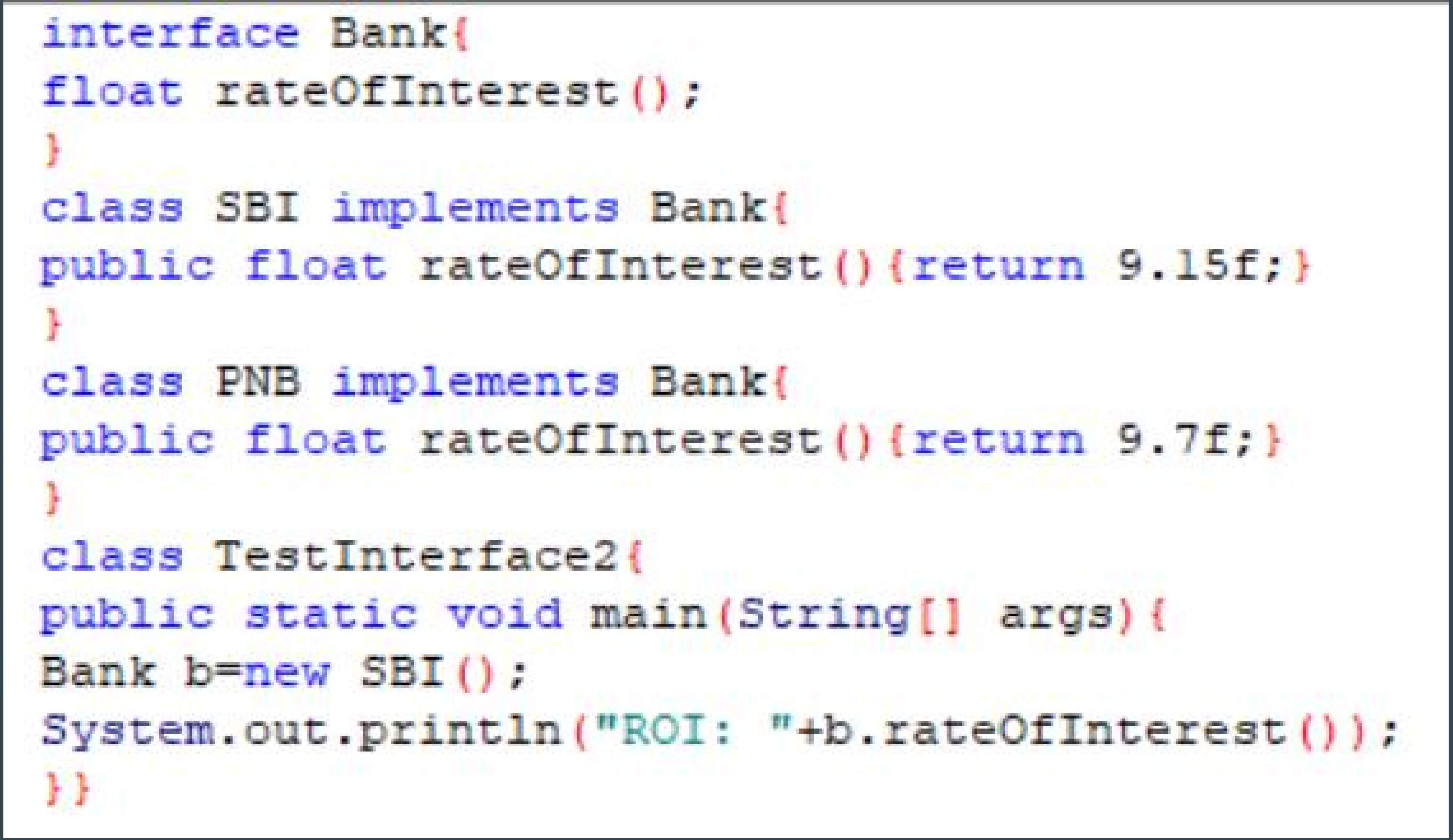
Java Interface Example



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

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

Java Interface Example

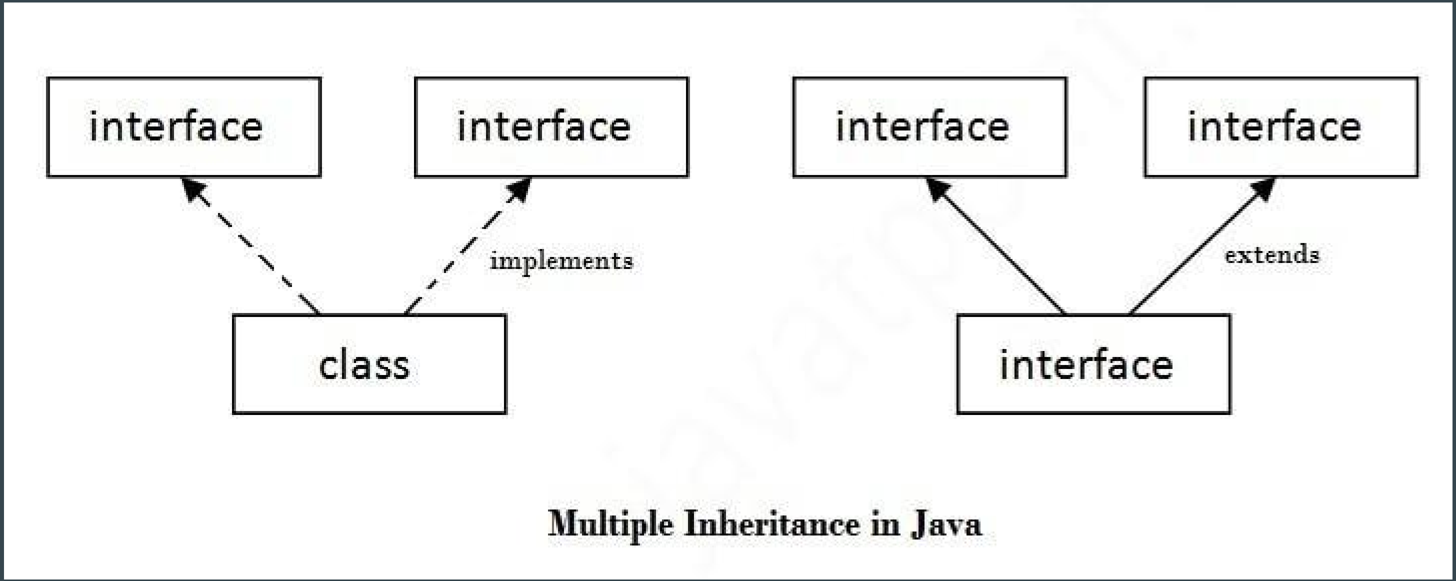
InterfaceDemo.java



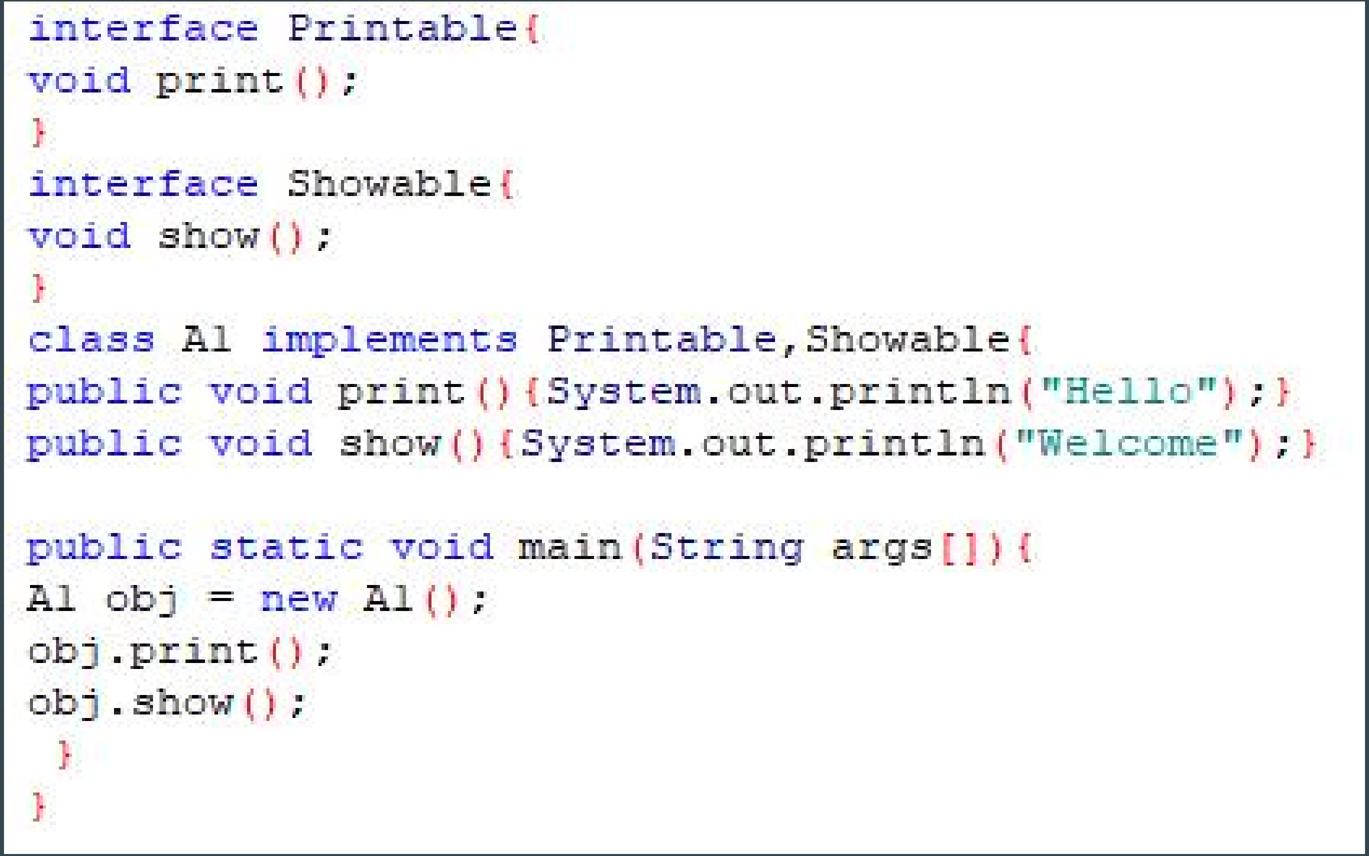
```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Multiple inheritance in Java using interface

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



Multiple inheritance Example



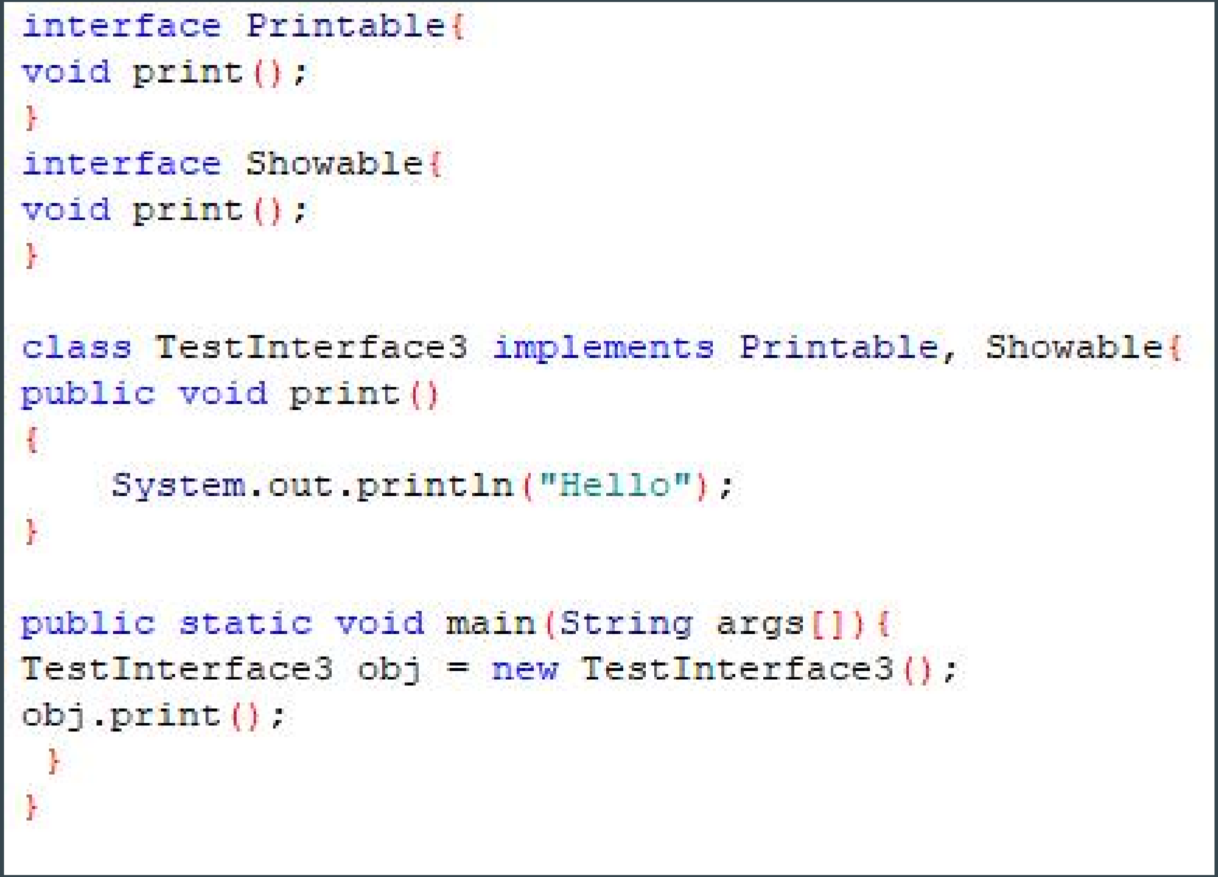
```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A1 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A1 obj = new A1();
obj.print();
obj.show();
 }
}
```

Multiple inheritance Example

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

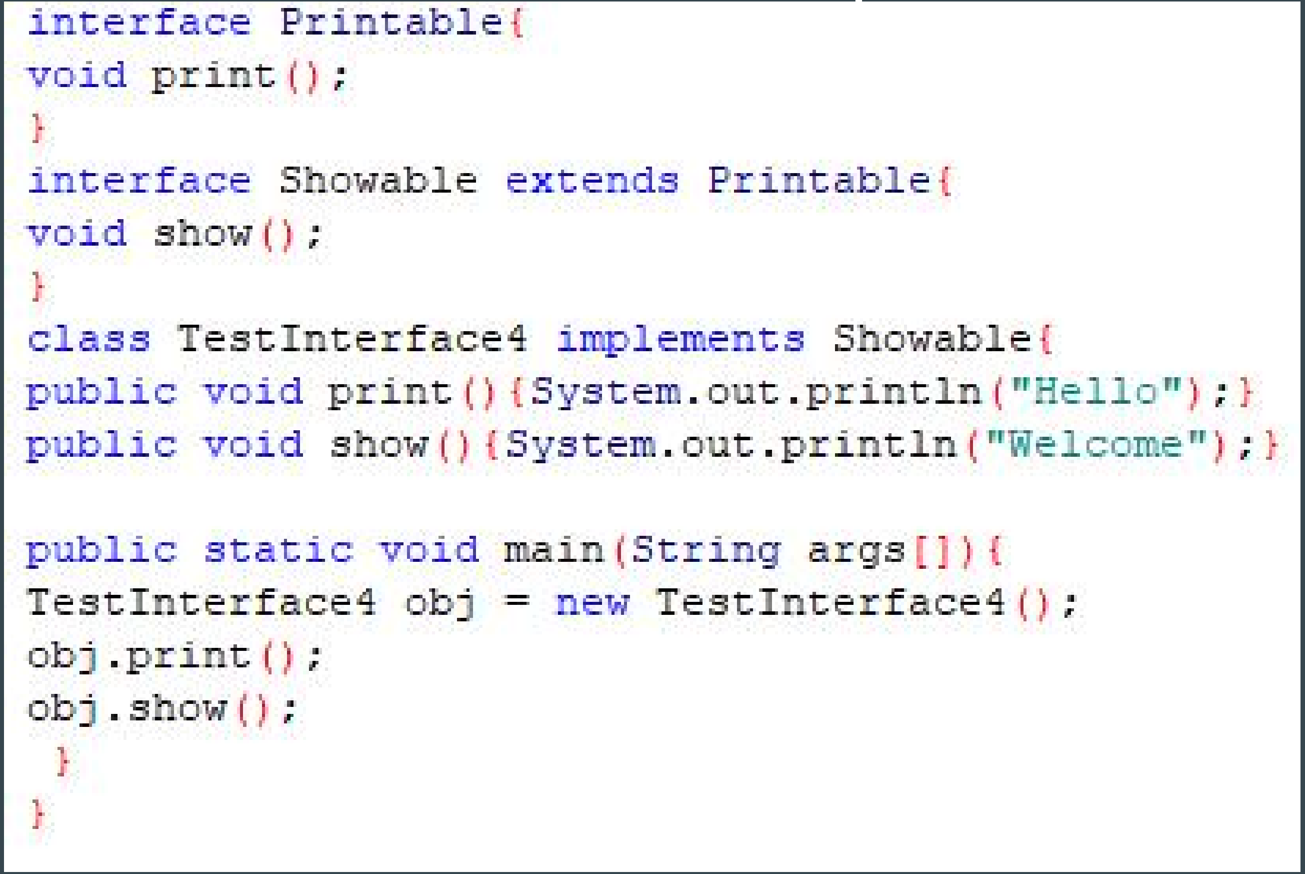
As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Multiple inheritance Example



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

class TestInterface3 implements Printable, Showable{
public void print()
{
    System.out.println("Hello");
}

public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Interface inheritance Example



```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Multiple Inheritance?

- Some people (and textbooks) have said that allowing classes to implement multiple interfaces is the same thing as multiple inheritance

- This is NOT true. When you implement an interface:
  - The implementing class does not inherit instance variables
  - The implementing class does not inherit methods (none are defined)
  - The Implementing class does not inherit associations

- Implementation of interfaces is not inheritance. An interface defines a list of methods which must be implemented.

Abstract Classes Versus Interfaces

- When should one use an Abstract class instead of an interface?
  - If the subclass-superclass relationship is genuinely an "is a" relationship.
  - If the abstract class can provide an implementation at the appropriate level of abstraction

- When should one use an interface in place of an Abstract Class?
  - When the methods defined represent a small portion of a class
  - When the subclass needs to inherit from another class
  - When you cannot reasonably implement any of the methods