Inheritance

- Inheritance is the process by which objects of one class can acquire the properties of objects of another class.
- It provides the idea of reusability.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Terms used in inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Syntax of Java inheritance

class Subclass-name **extends** Superclass-name

{

//methods and fields

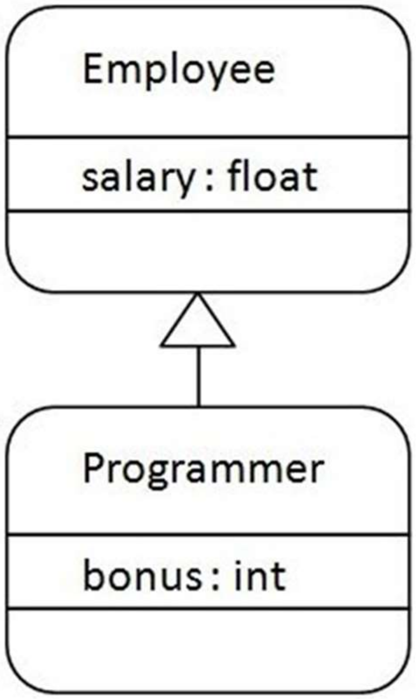
}

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.
- extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.
- Syntax

class Super { .... }

class Sub extends Super { .... }

Inheritance Example



- It displayed in the above figure, Programmer is the subclass and Employee is the superclass.

- The relationship between the two classes is **Programmer IS-A Employee.** It means Programmer is a type of Employee

```
class Employee{

float salary=4000;

}
class Programmer extends Employee{

int bonus=10000;

public static void main(String args[]){

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

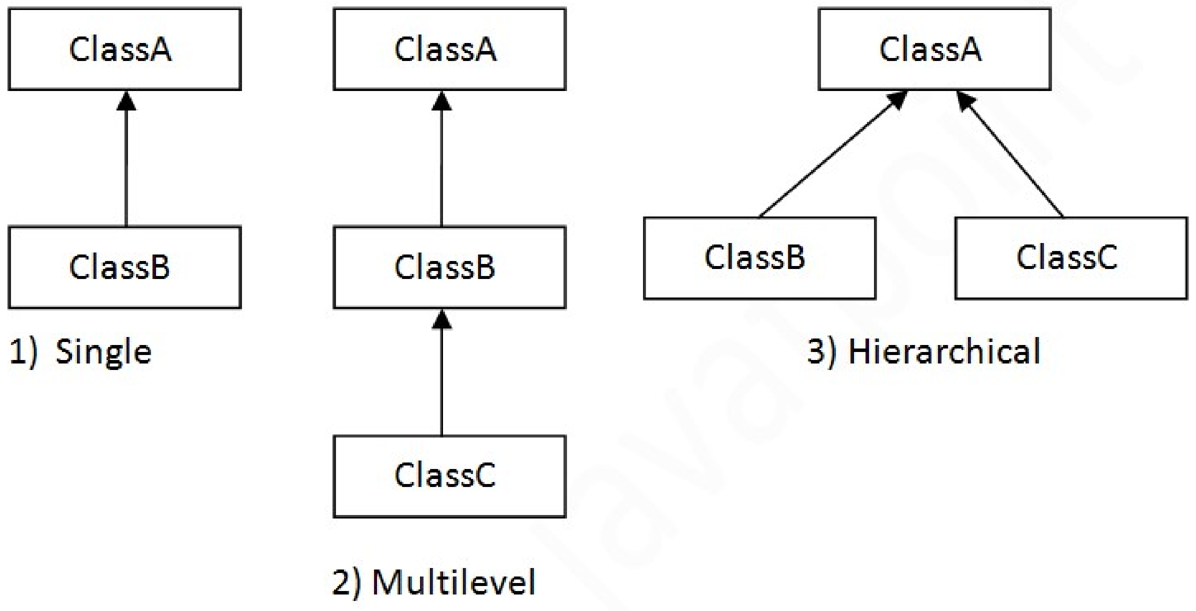
Output:

```
Programmer salary is: 4000.0
Bonus of Programmer is : 10000.0
```

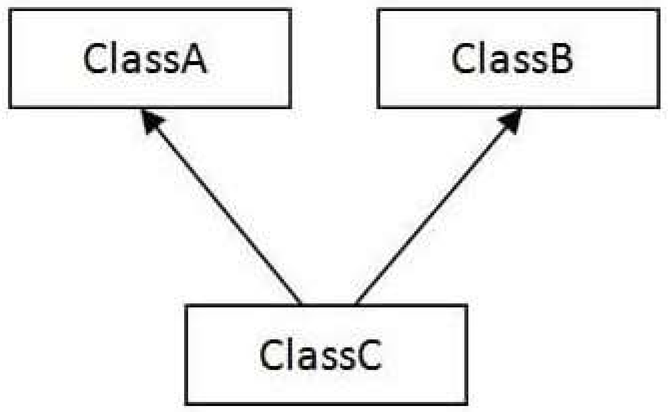
Types of Inheritance

- There are three types of inheritance in java:
    - single,
    - multilevel and
    - hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.

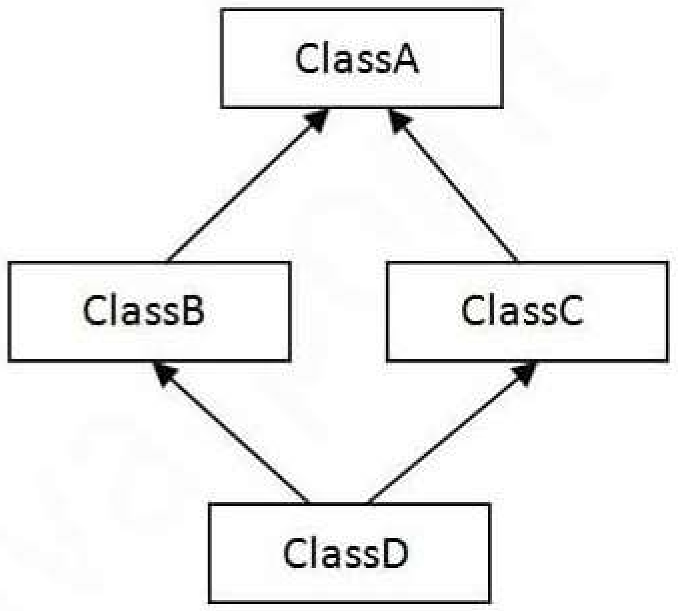
Types of inheritance in java



Inheritance not supported in java



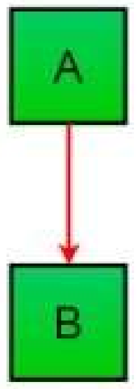
4) Multiple



5) Hybrid

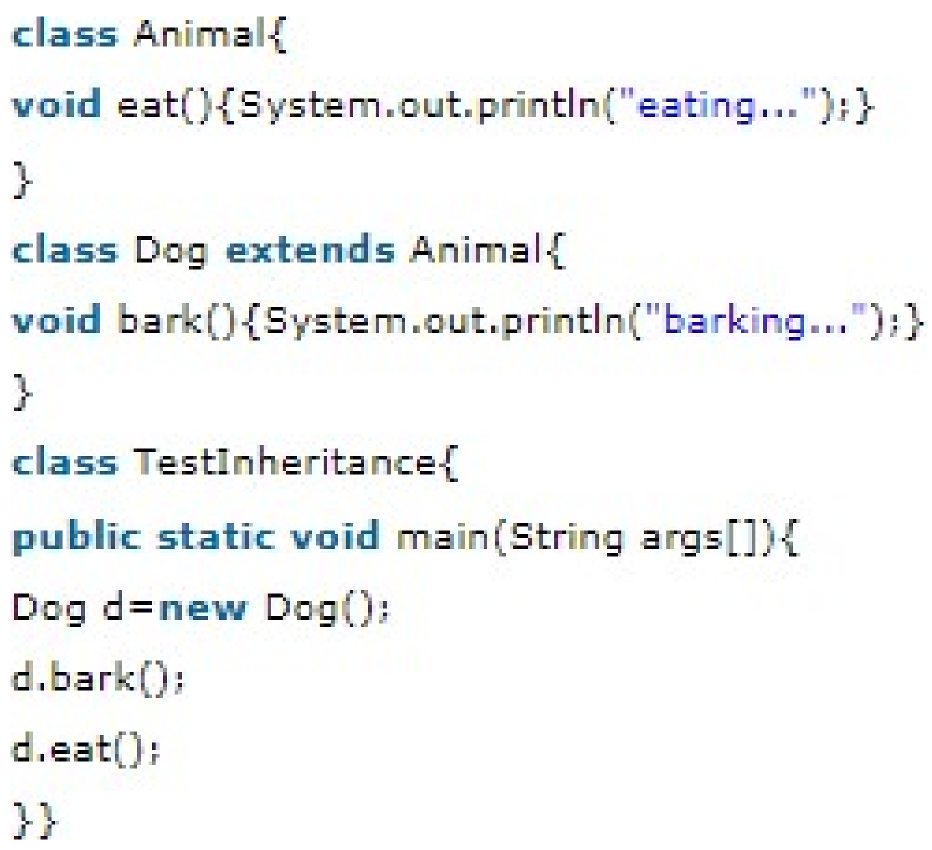
Single Inheritance

- In single inheritance, subclasses inherit the features of one superclass.
- In image below, the class A serves as a base class for the derived class B

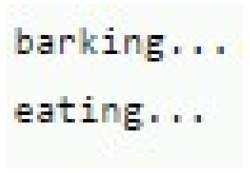


Single Inheritance

Single Inheritance Example



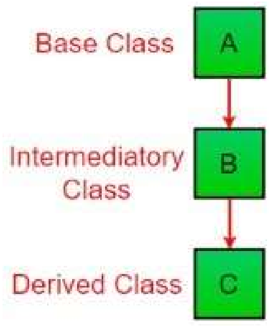
```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```



```
barking...
eating...
```

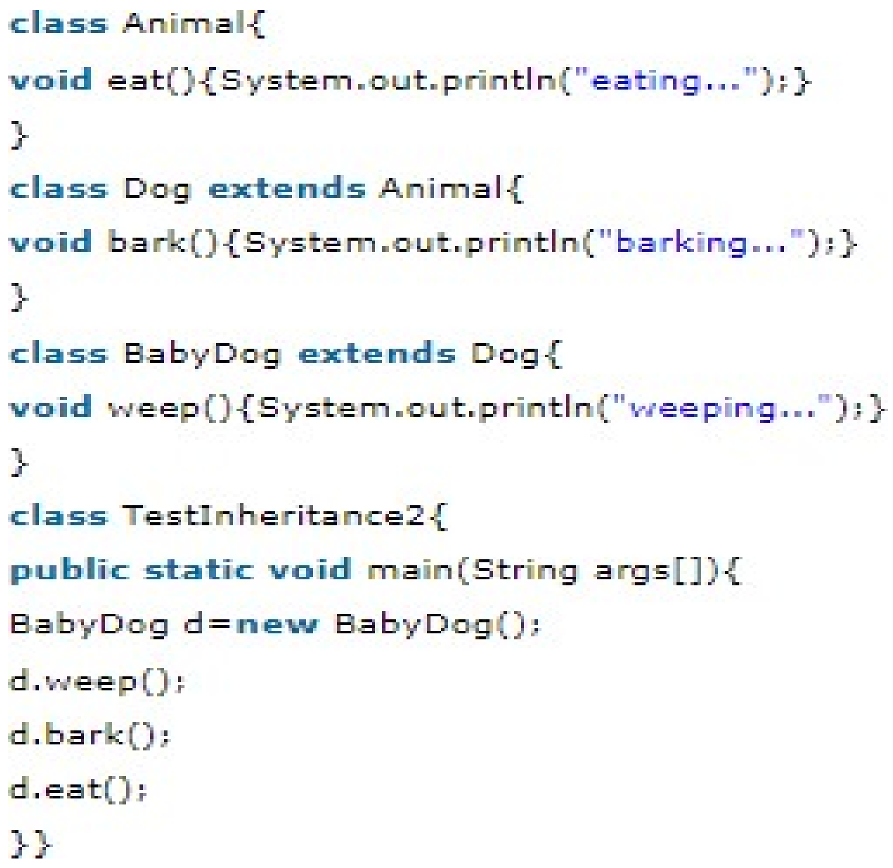
Multilevel Inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
- In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

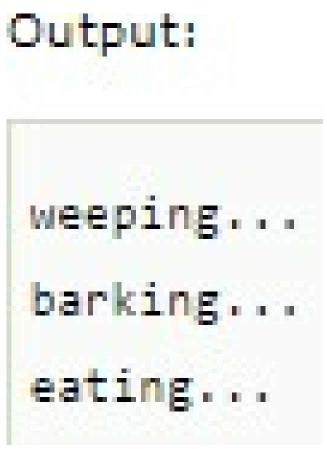


Multilevel Inheritance

Multilevel Inheritance Example



```java
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

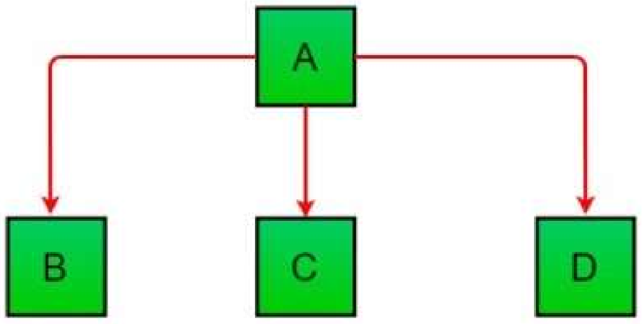


Output:

```
weeping...
barking...
eating...
```

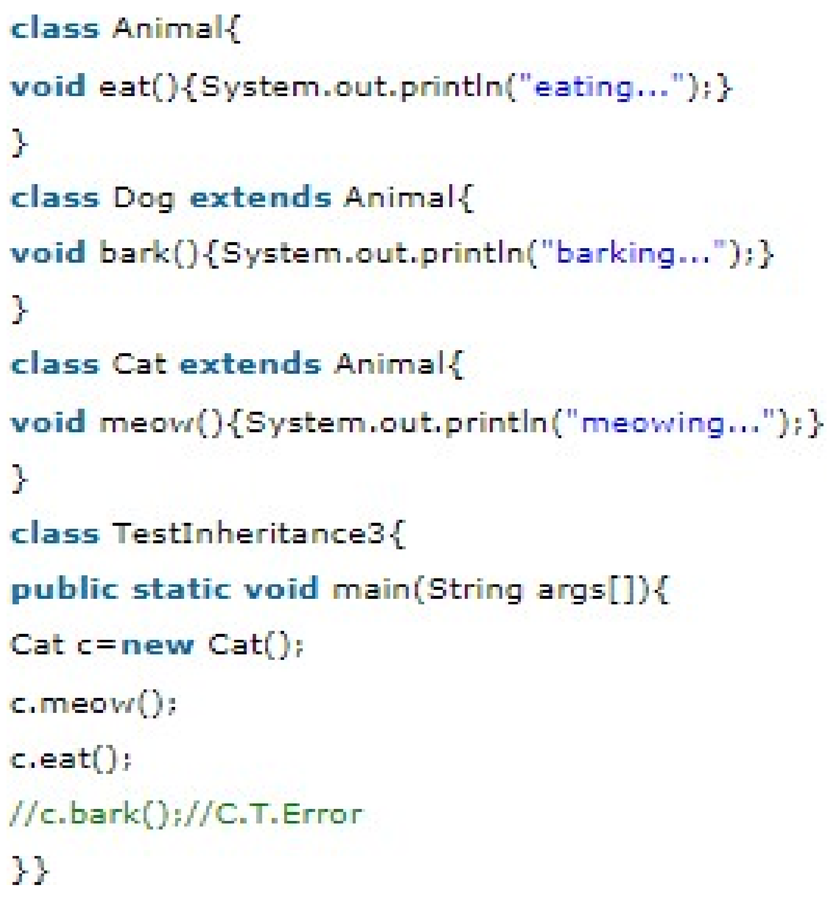
Hierarchical Inheritance

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.
- In below image, the class A serves as a base class for the derived class B,C and D.

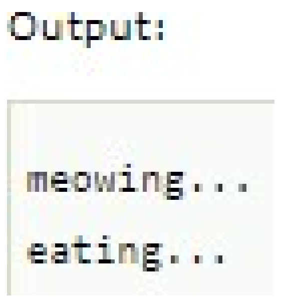


Hierarchical Inheritance

Hierarchical Inheritance Example



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

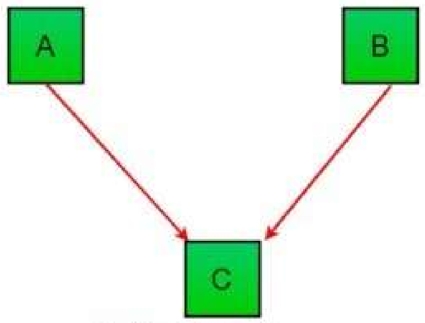


Output:

```
meowing...
eating...
```

Multiple Inheritance (Through Interfaces)

- In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes.
- Please note that Java does **not** support multiple inheritance with classes.
- In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.

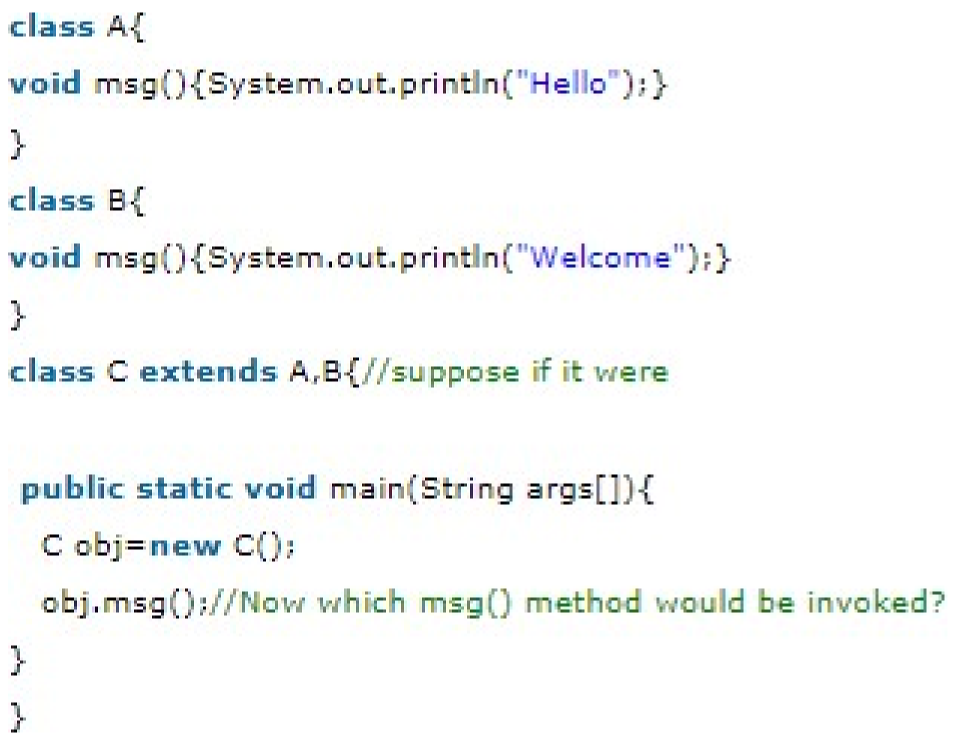


Multiple Inheritance

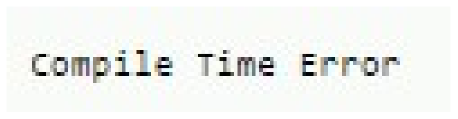
Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes.
- If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Multiple inheritance is not supported in java



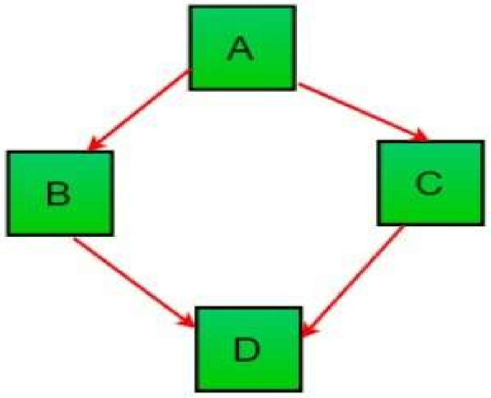
```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```



```
Compile Time Error
```

Hybrid Inheritance (Through Interfaces)

- It is a mix of two or more of the above types of inheritance.
- Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes.
- In java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass

super keyword

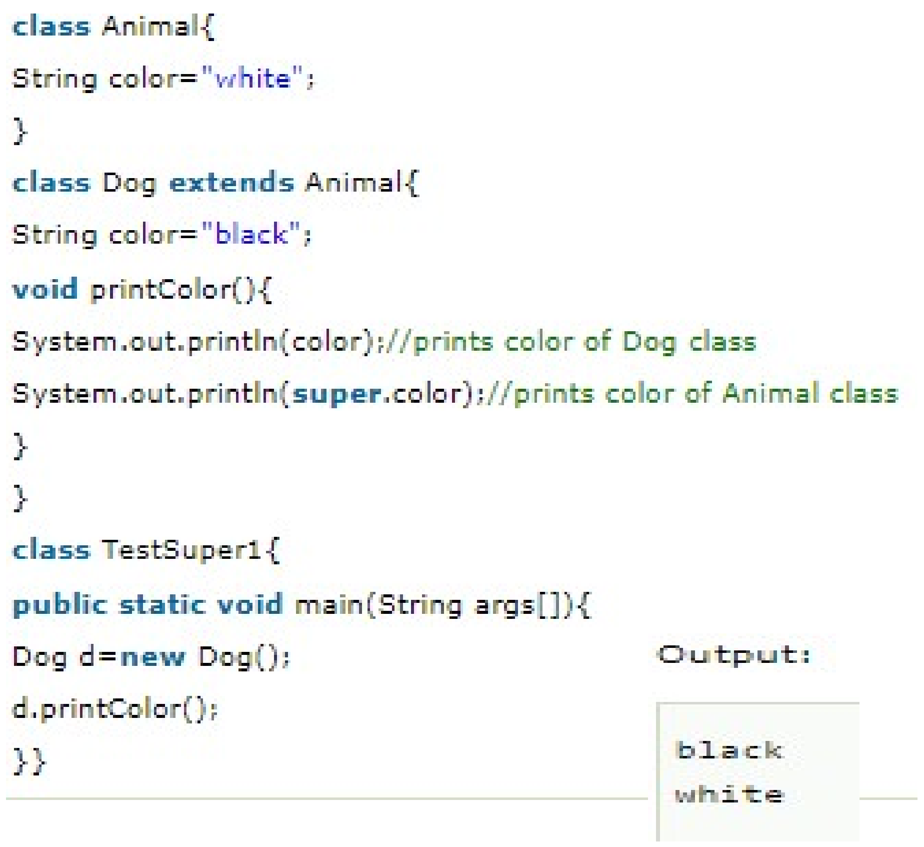
- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

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

super is used to refer immediate parent class instance variable.



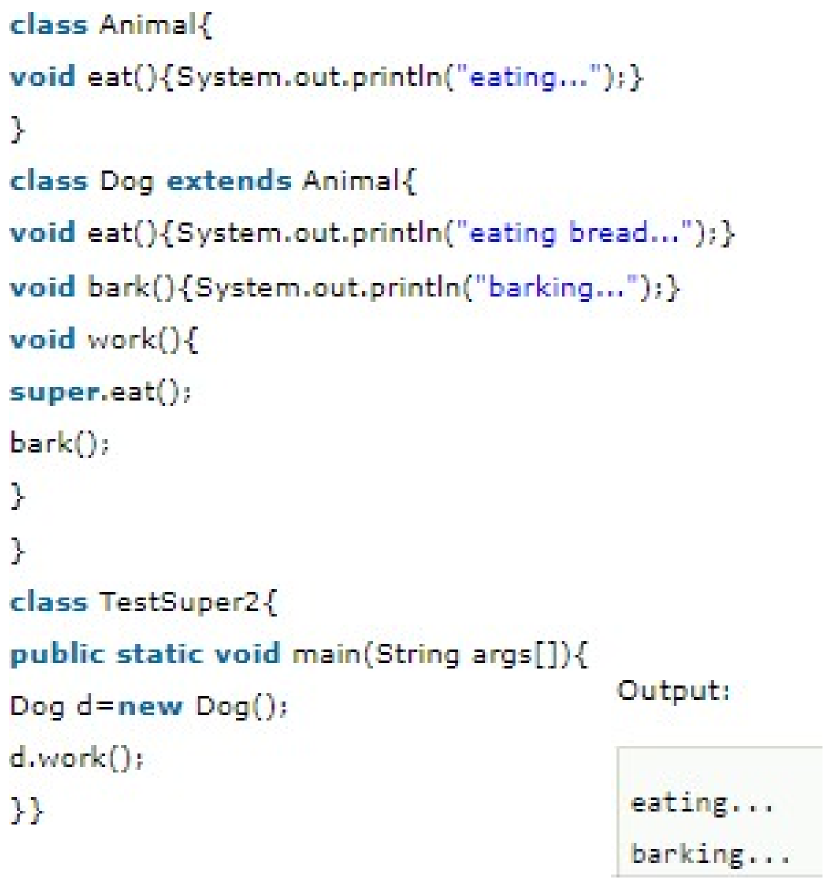
```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();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

Output:

```
black
white
```

- We can use super keyword to access the data member or field of parent class.
- It is used if parent class and child class have same fields.
- In this example, Animal and Dog both classes have a common property color.
- If we print color property, it will print the color of current class by default.
- To access the parent property, we need to use Super keyword.
- .

super can be used to invoke parent class method



```
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();
}}
```

Output:

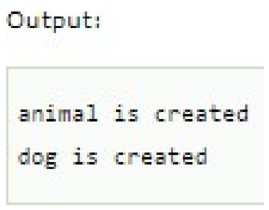
```
eating...
barking...
```

- The super keyword can also be used to invoke parent class method.
- It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.
- In this example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.
- To call the parent class method, we need to use super keyword.

super is used to invoke parent class constructor

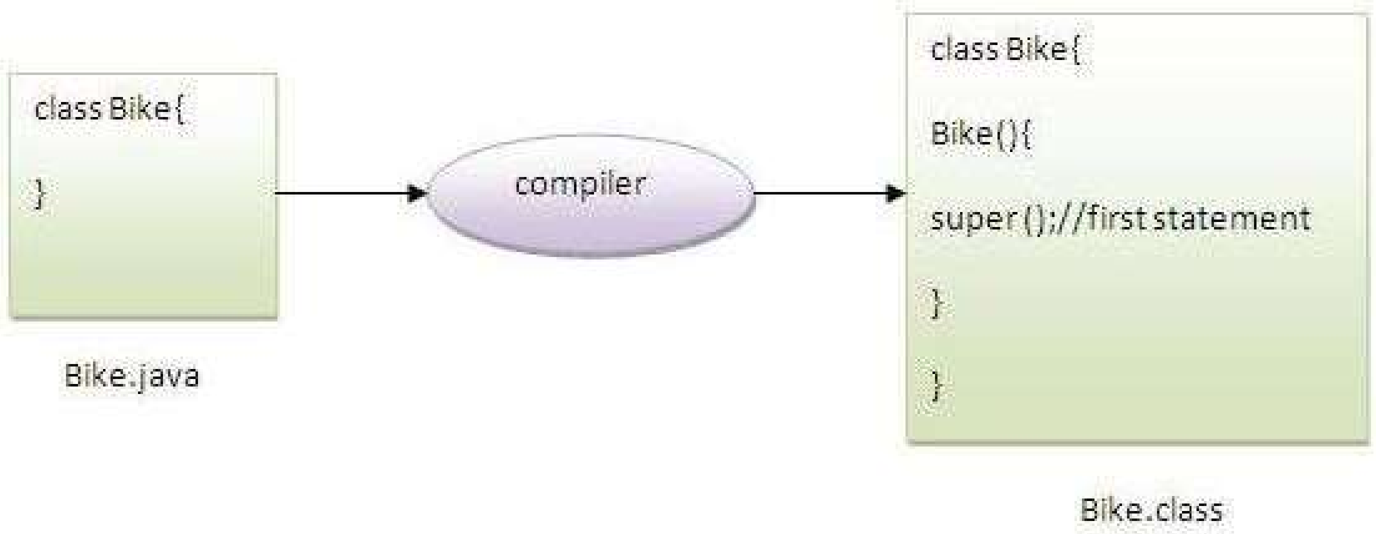
```
class Animal{
Animal(){
System.out.println(“animal is created”);
}
}
class Dog extends Animal{
Dog(){
System.out.println(“dog is created”);
}
}
class TestSuper3{
Public static void main (String args[]){
Dog d = new Dog();
}}
```

- The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:


Output:

```
animal is created
dog is created
```

super() is added in each class constructor automatically by compiler if there is no super() or this()

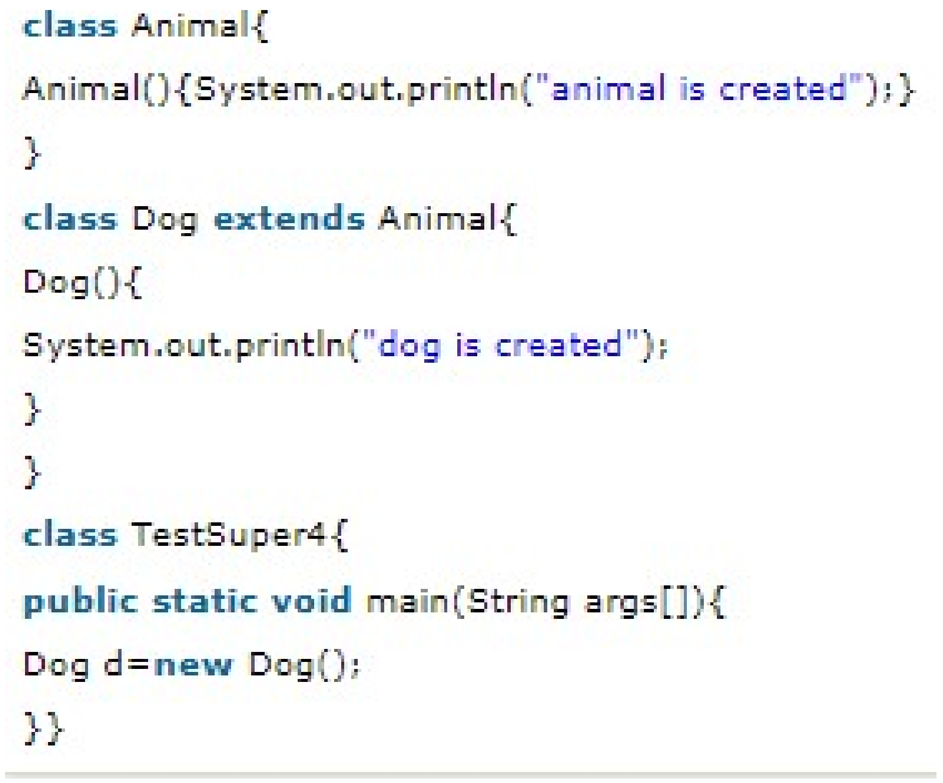


```
class Bike{

}
```

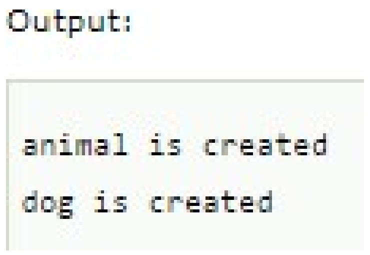
```
class Bike{

Bike(){

super();//first statement

}

}
```

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

super() is provided by the compiler implicitly



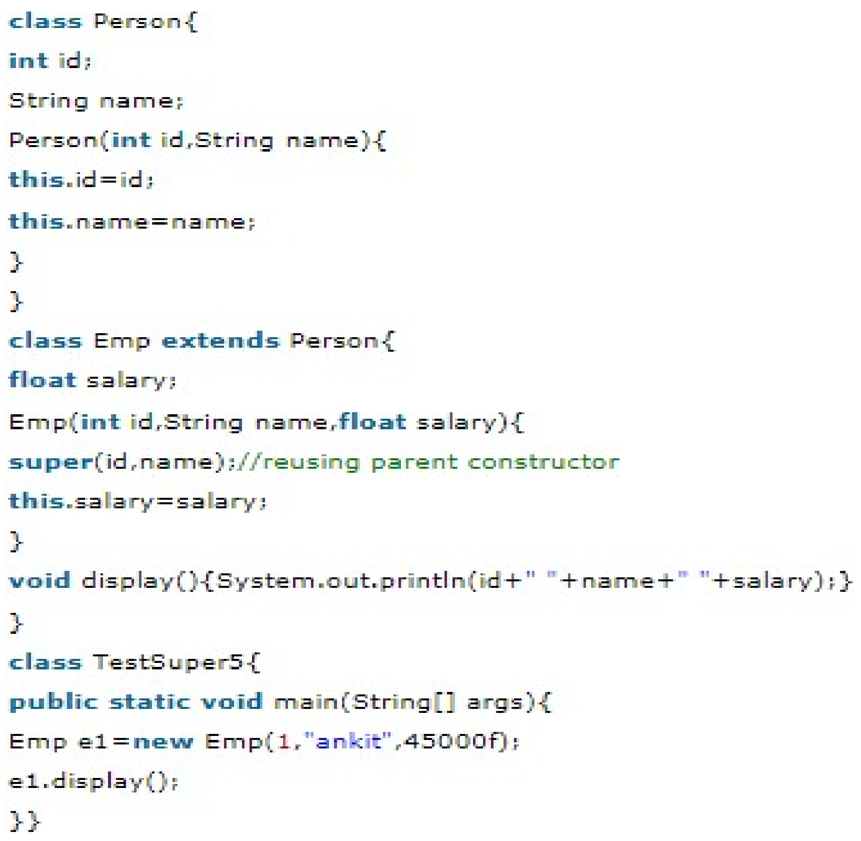
```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
```



Output:

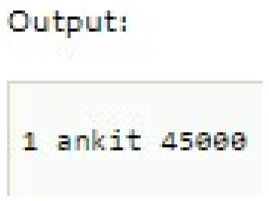
```
animal is created
dog is created
```

super example: real use



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

- Let's see the real use of super keyword.
- Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default.
- To initialize all the property, we are using parent class constructor from child class.
- In such way, we are reusing the parent class constructor.



Output:

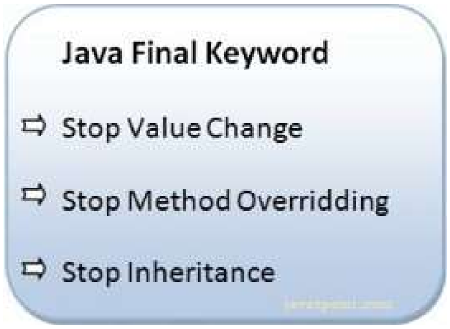
```
1 ankit 45000
```

final keyword

- The **final keyword** in java is used to restrict the user.

- The java final keyword can be used in many context.

- Final can be:
  - Variable
  - method
  - Class



Java Final Keyword

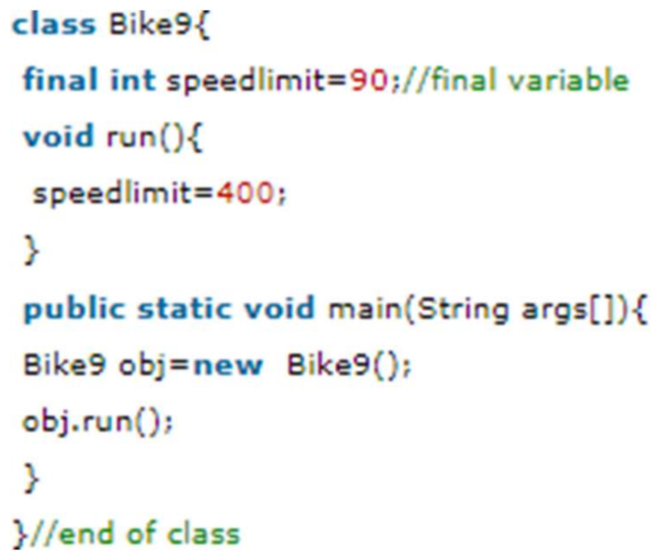
⇨ Stop Value Change

⇨ Stop Method Overridding

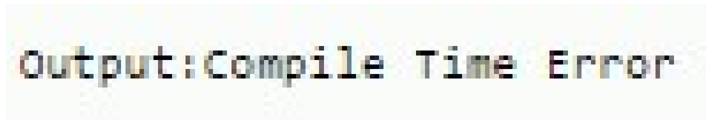
⇨ Stop Inheritance

Java final variable

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



```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

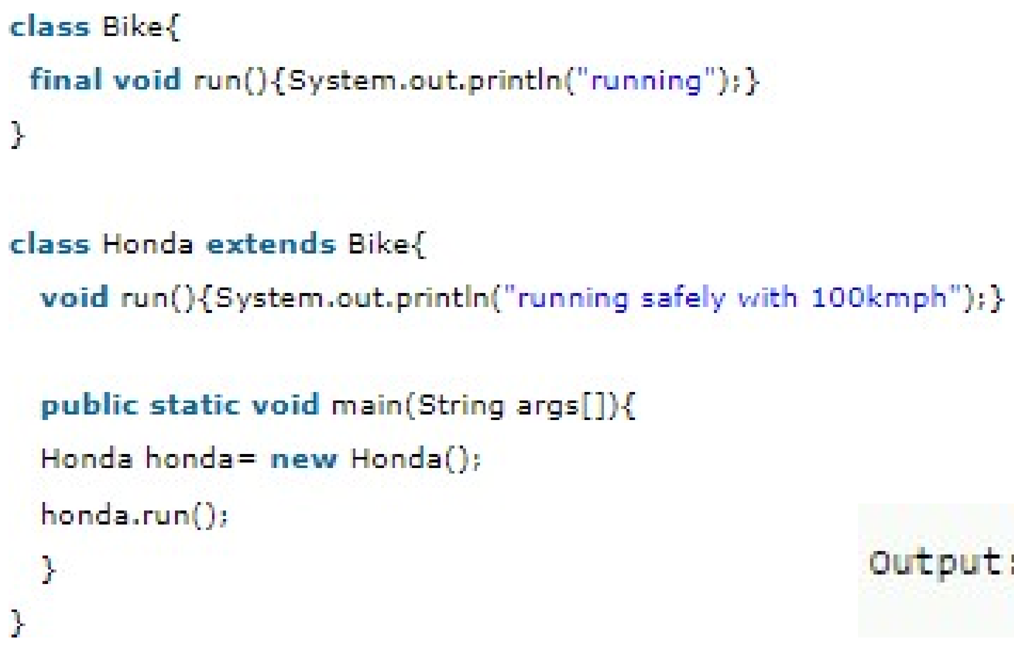


```
Output:Compile Time Error
```

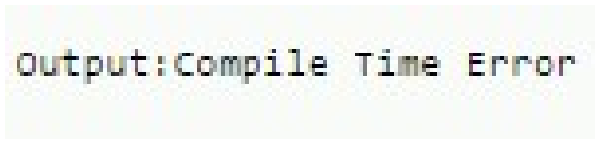
- There is a final variable speedlimit, we are going to change the value of this variable,
- But It can't be changed because final variable once assigned a value can never be changed.

Java final method

- If you make any method as final, you cannot override it.



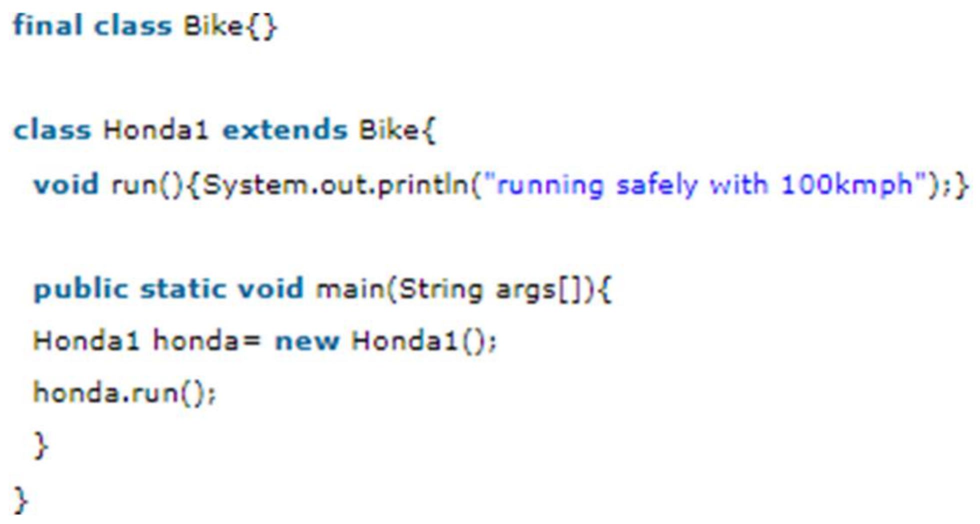
```
class Bike{
 final void run(){System.out.println("running");}
}

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

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



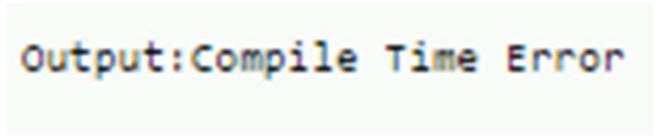
```
Output:Compile Time Error
```

Java final class

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



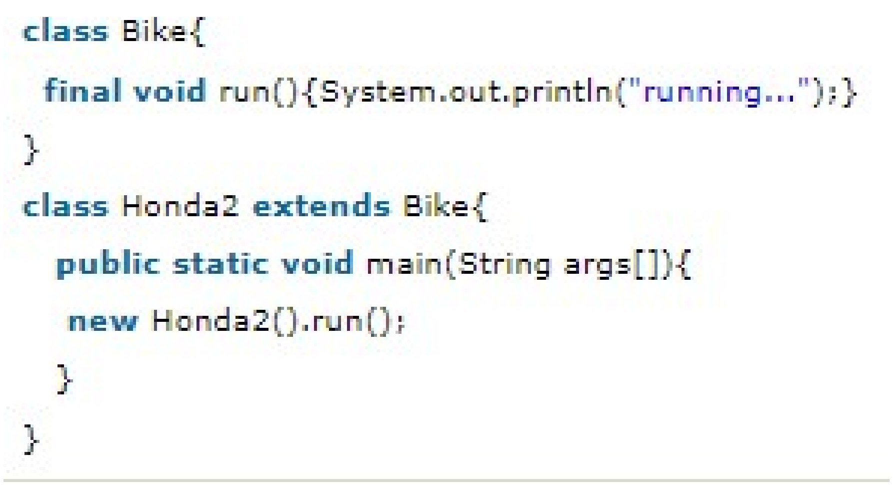
```
final class Bike{}

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

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



```
Output:Compile Time Error
```

Is final method inherited?

- Yes, final method is inherited but you cannot override it. For Example:

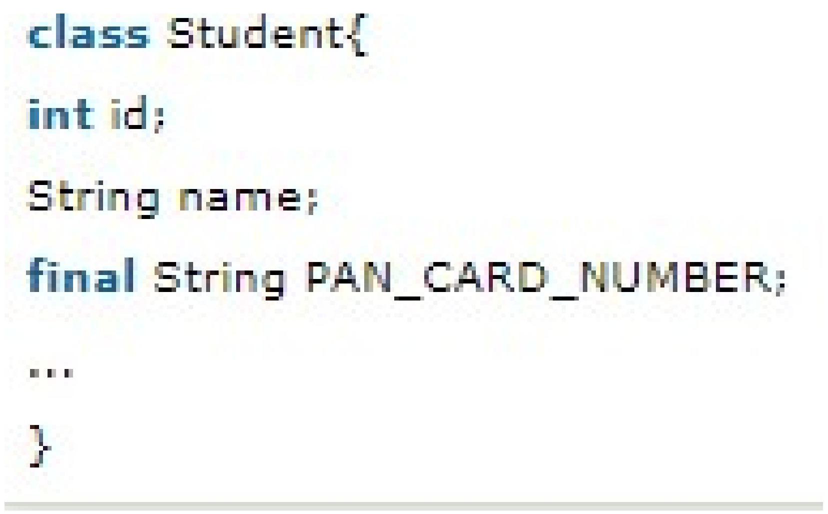


```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

What is blank or uninitialized final variable?

- A final variable that is not initialized at the time of declaration is known as blank final variable.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.
- It can be initialized only in constructor.

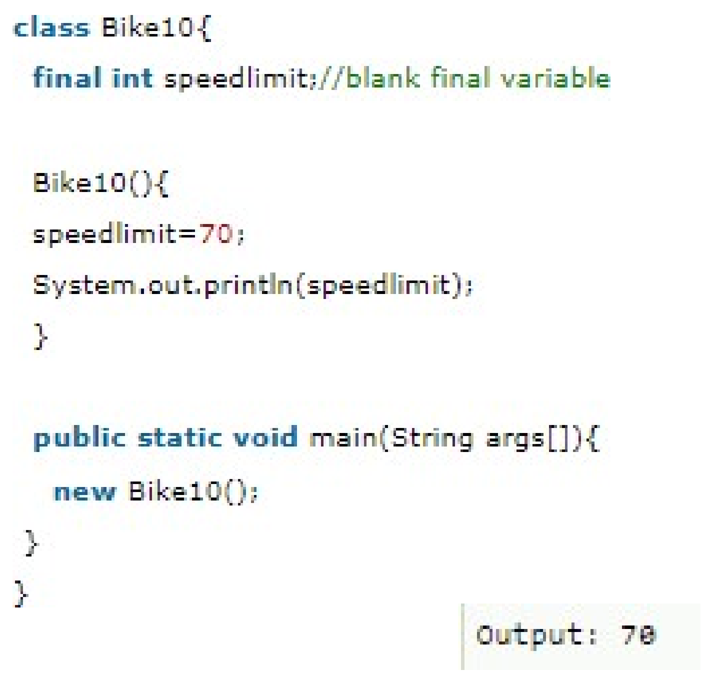
Example of blank final variable



```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Can we initialize blank final variable?

Yes, but only in constructor. For example:

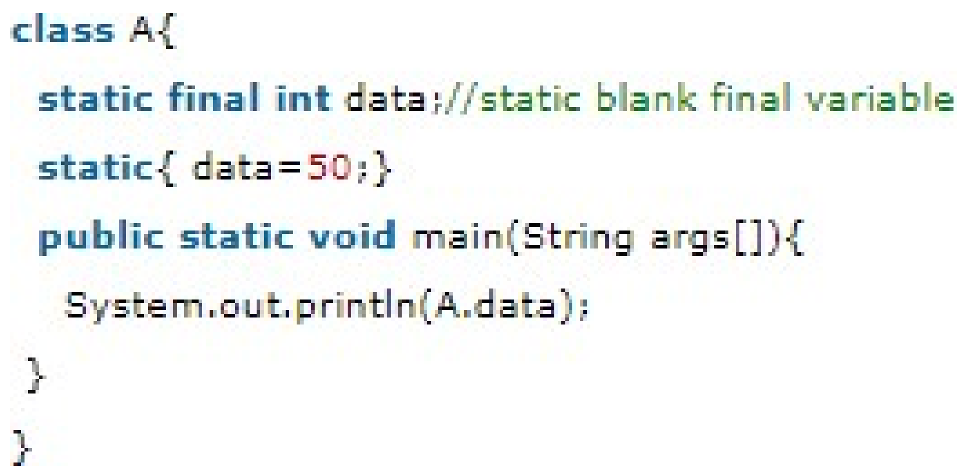


```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

```
Output: 70
```

Example of static blank final variable

- A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.



```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

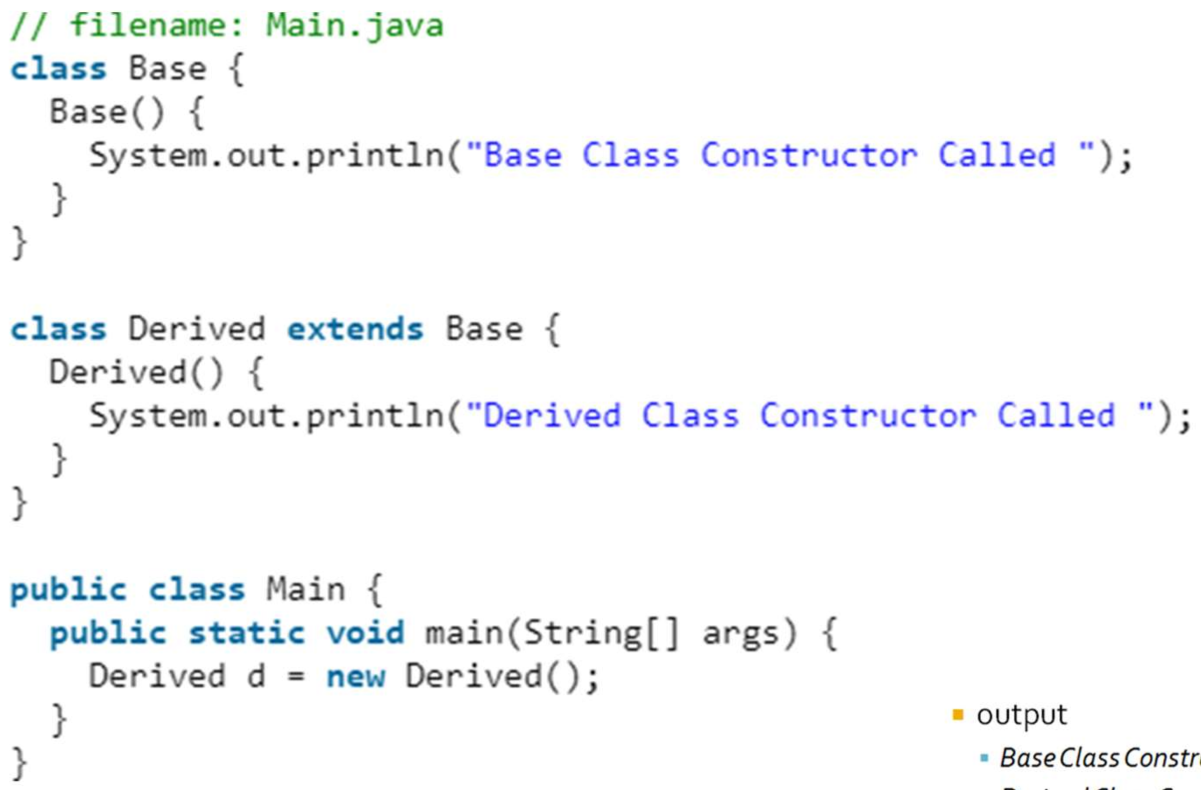
What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11
{
        int cube(final int n)
                {
                        n=n+2; // can not change as n is final
                        return n*n*n;
                }
public static void main(String args[])
{
        Bike11 b = new Bike11();
        int c = b.cube(5);
        System.out.println("Result is" +c);
}
}
```

Constructors and Inheritance

- In Java, constructor of base class with no argument gets automatically called in derived class constructor.



```
// filename: Main.java
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}

class Derived extends Base {
  Derived() {
    System.out.println("Derived Class Constructor Called ");
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }
}
```

- output
  - *Base Class Constructor Called*
  - *Derived Class Constructor Called*

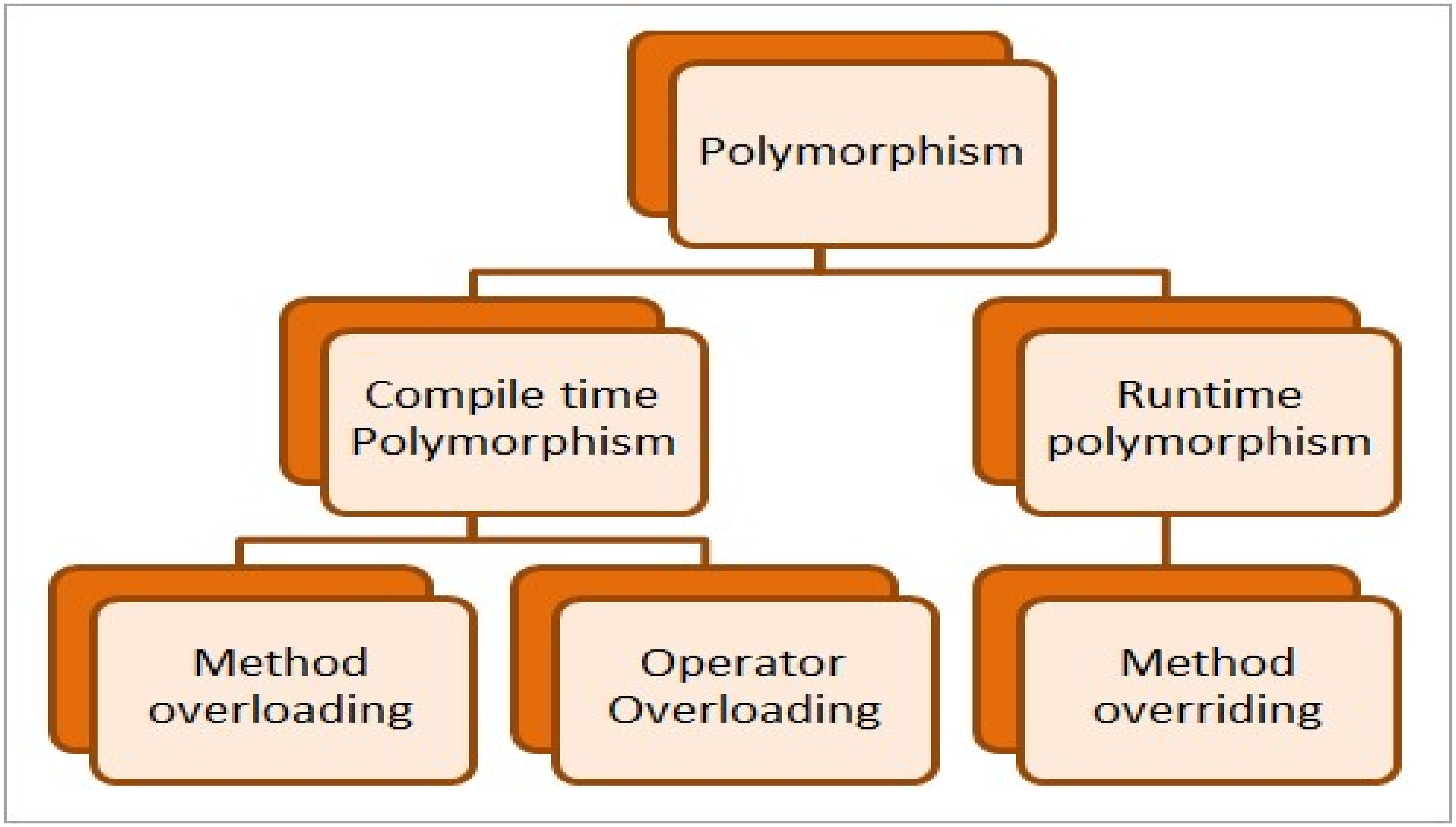
Polymorphism

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

- Polymorphism is considered as one of the important features of Object Oriented Programming.

Types of Polymorphism



- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs.

- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

Compile time polymorphism

- This type of polymorphism is achieve by function overloading and operator overloading
- **Function Overloading:** When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.
- **Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Runtime polymorphism

- This type of polymorphism is achieved by Function Overriding.
- **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Dynamic method dispatch mechanism

- Runtime Polymorphism in Java is achieved by Method overriding in which a child class overrides a method in its parent.
- An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.
- This method call resolution happens at runtime and is termed as Dynamic method dispatch mechanism.

Dynamic method dispatch mechanism example

```java
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog {

   public static void main(String args[]) {

      Animal a = new Animal(); // Animal reference and object
      Animal b = new Dog(); // Animal reference but Dog object

      a.move(); // runs the method in Animal class
      b.move(); // runs the method in Dog class
   }
}
```

Output

```
Animals can move
Dogs can walk and run
```

Overriding

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

- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

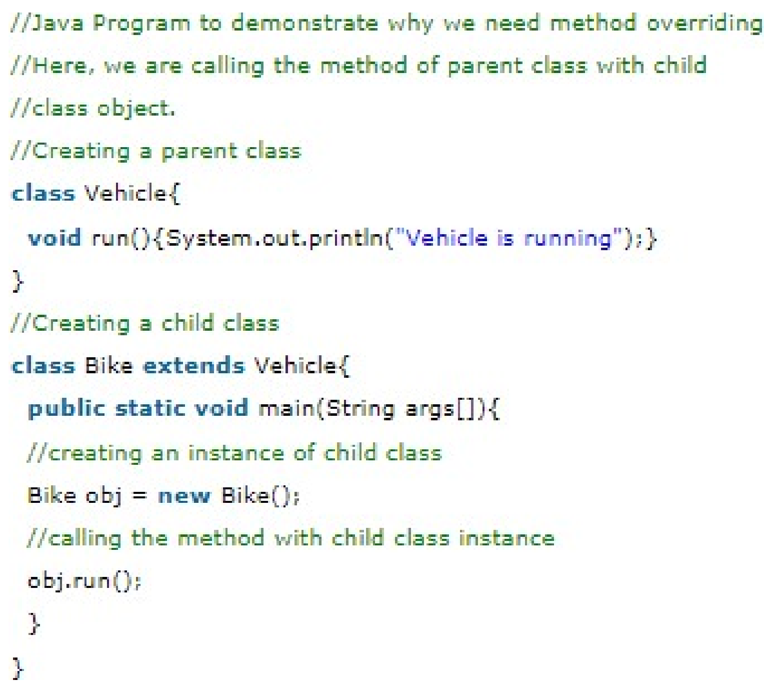
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

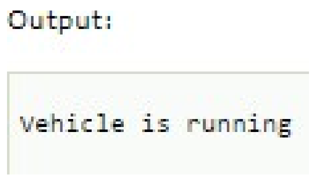
- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.



```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

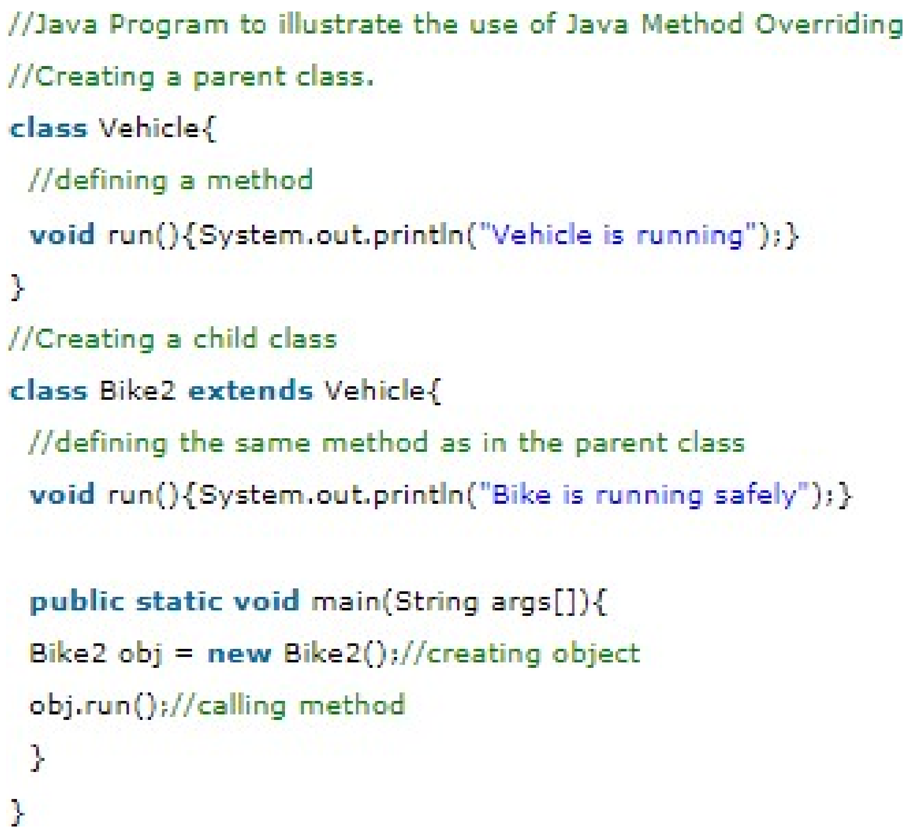


Output:

```
Vehicle is running
```

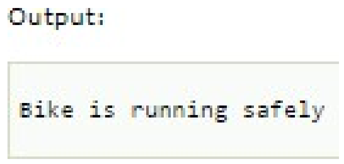
Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding



```
//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
  }
}
```

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

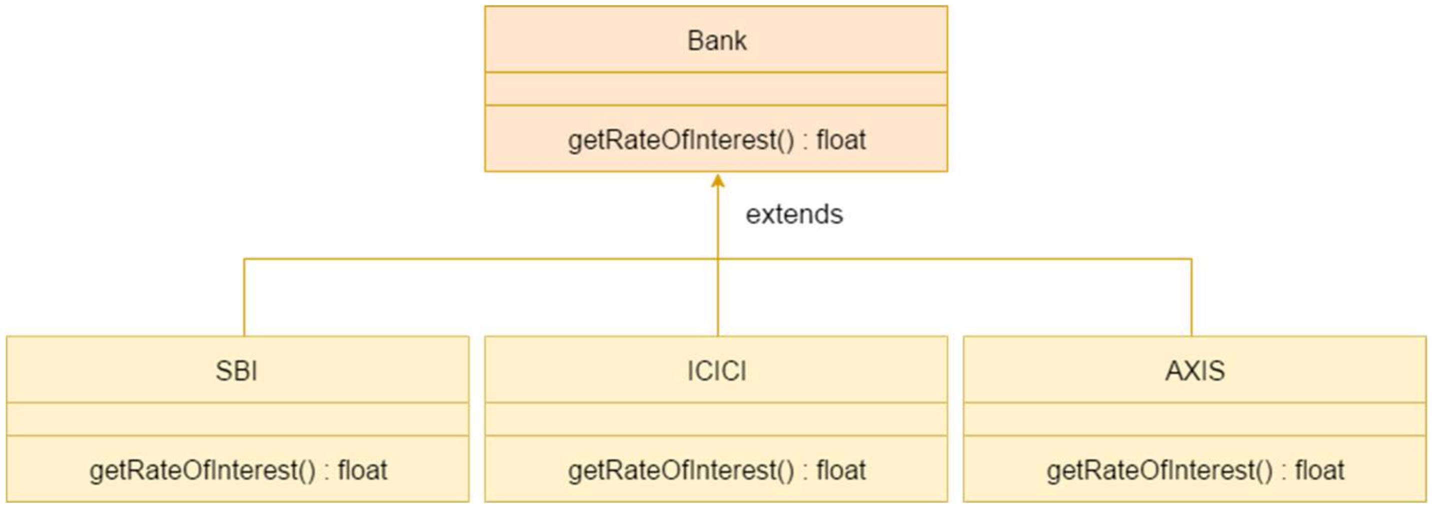


Output:

```
Bike is running safely
```

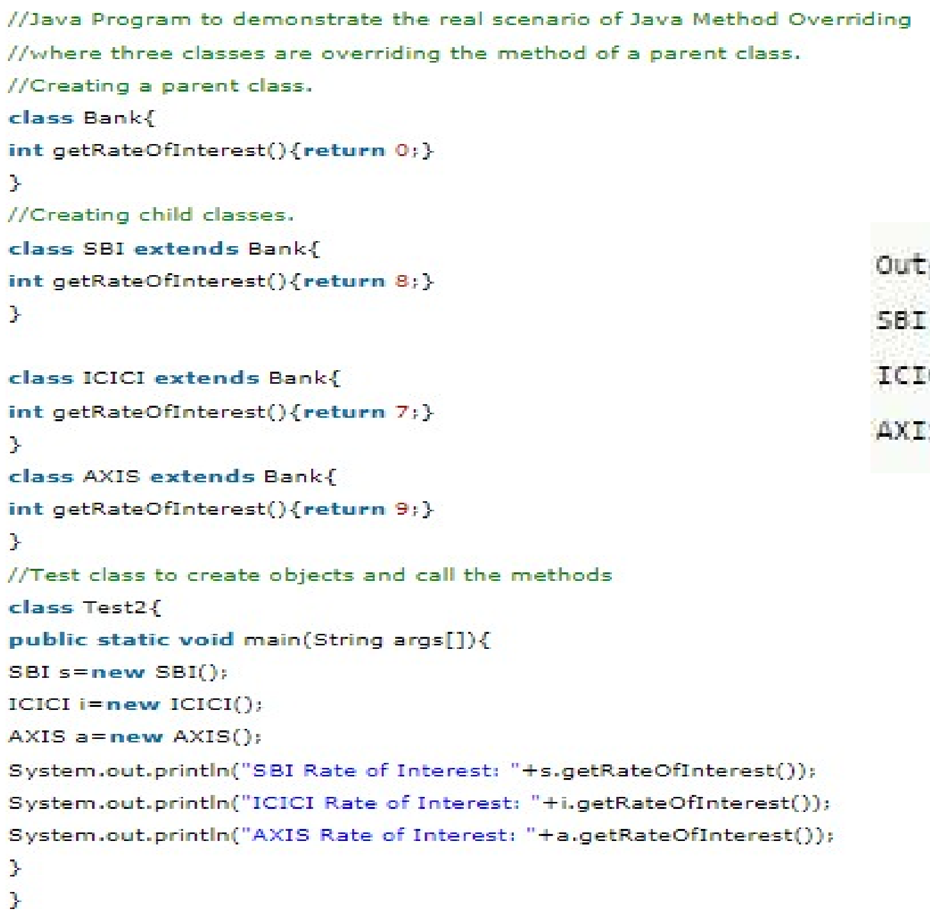
A real example of Java Method Overriding

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.

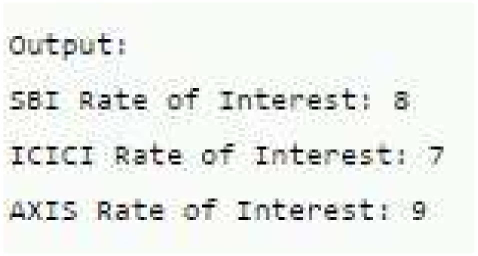

Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Java method overriding is mostly used in Runtime Polymorphism

A real example of Java Method Overriding



```
//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
//Creating a parent class.
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;}
}
//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
```

Can we override static method?

- No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

- It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

- No, because the main is a static method.