# Modern Programming Tools And Techniques-I

## Lecture 8: Inheritance

## Introduction

- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.

- In the Java programming language, each class is allowed to have one direct super-class, and each super-class has the potential for an unlimited number of *subclasses*.

- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes.

## Inheritance

- Inheritance defines an is-a relationship between a super-class and its subclasses. It means that the subclass (child) is a more specific version of the super-class (parent).

- An object of a subclass can be used wherever an object of the super-class can be used.

- Inheritance is used to build new classes from existing classes.

- The inheritance relationship is transitive: if class y extends class x, and a class z extends class y, then z will also inherit from class x.

## Inheritance

- Using inheritance, we can create a general class that defines traits(state and behaviors) common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In Java, a class that is inherited is called a **super-class**.
- The class that does the inheriting is called a **subclass**.
- Therefore, a subclass is a specialized version of a super-class. It inherits all of the instance variables and methods defined by the super-class and adds its own, unique elements.

## Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

## Syntax of Java Inheritance

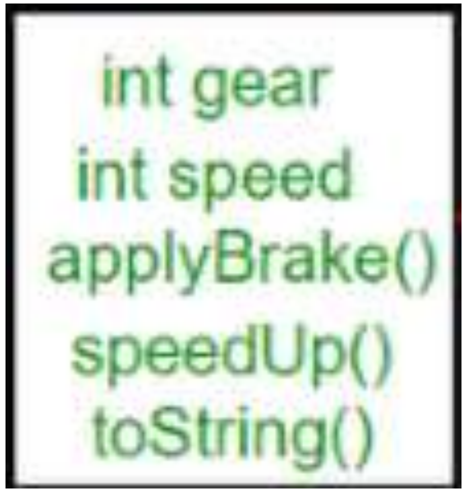
```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

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 parent or super class and the new class is called child or subclass.

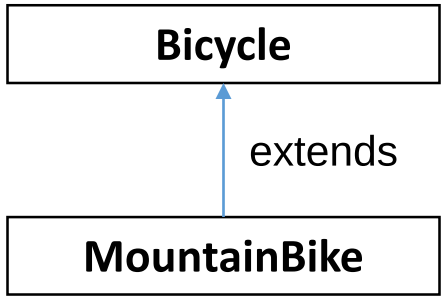
## Understanding inheritance

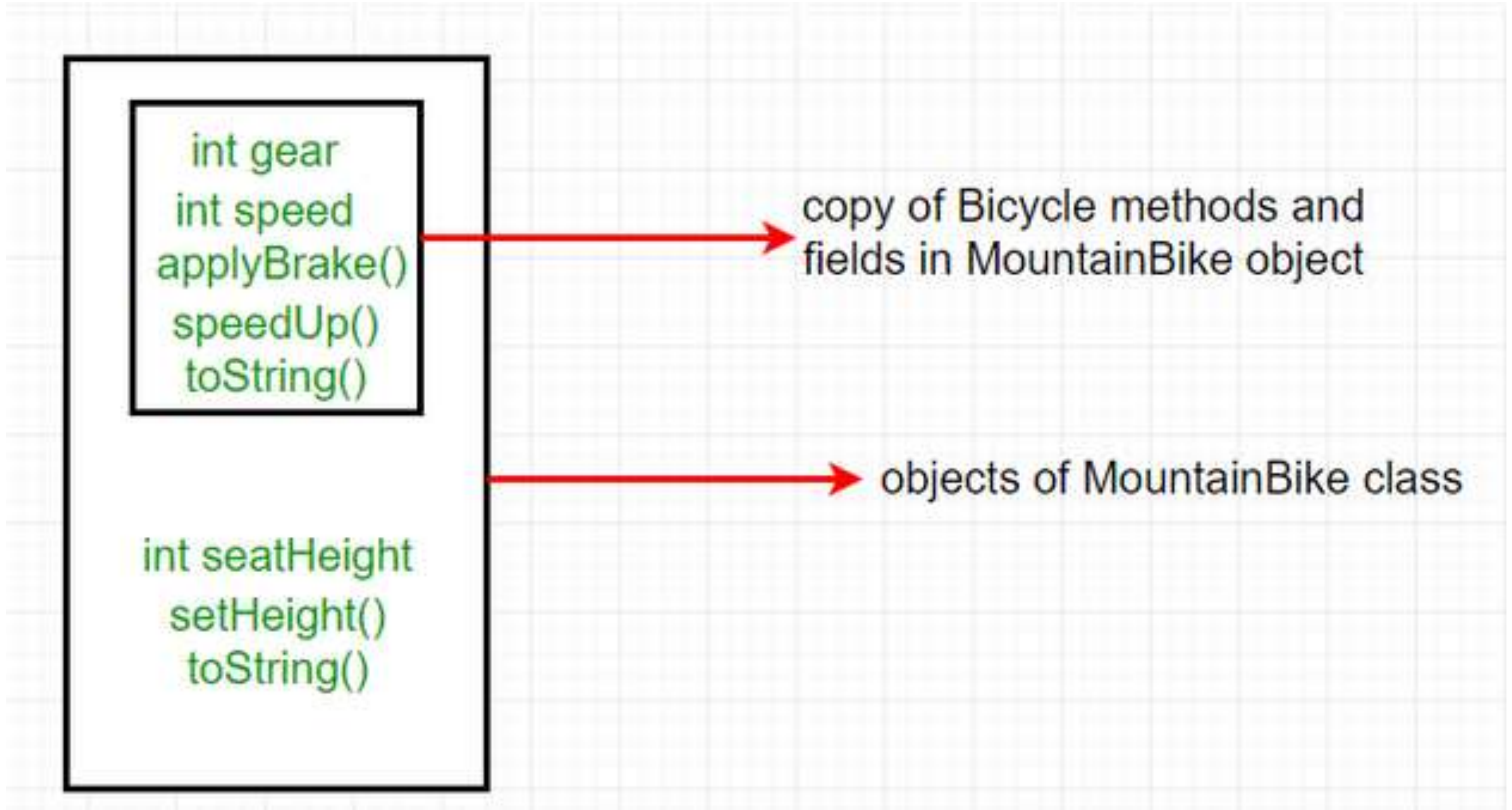
**Bicycle**



int gear
int speed
applyBrake()
speedUp()
toString()

# Inheritance

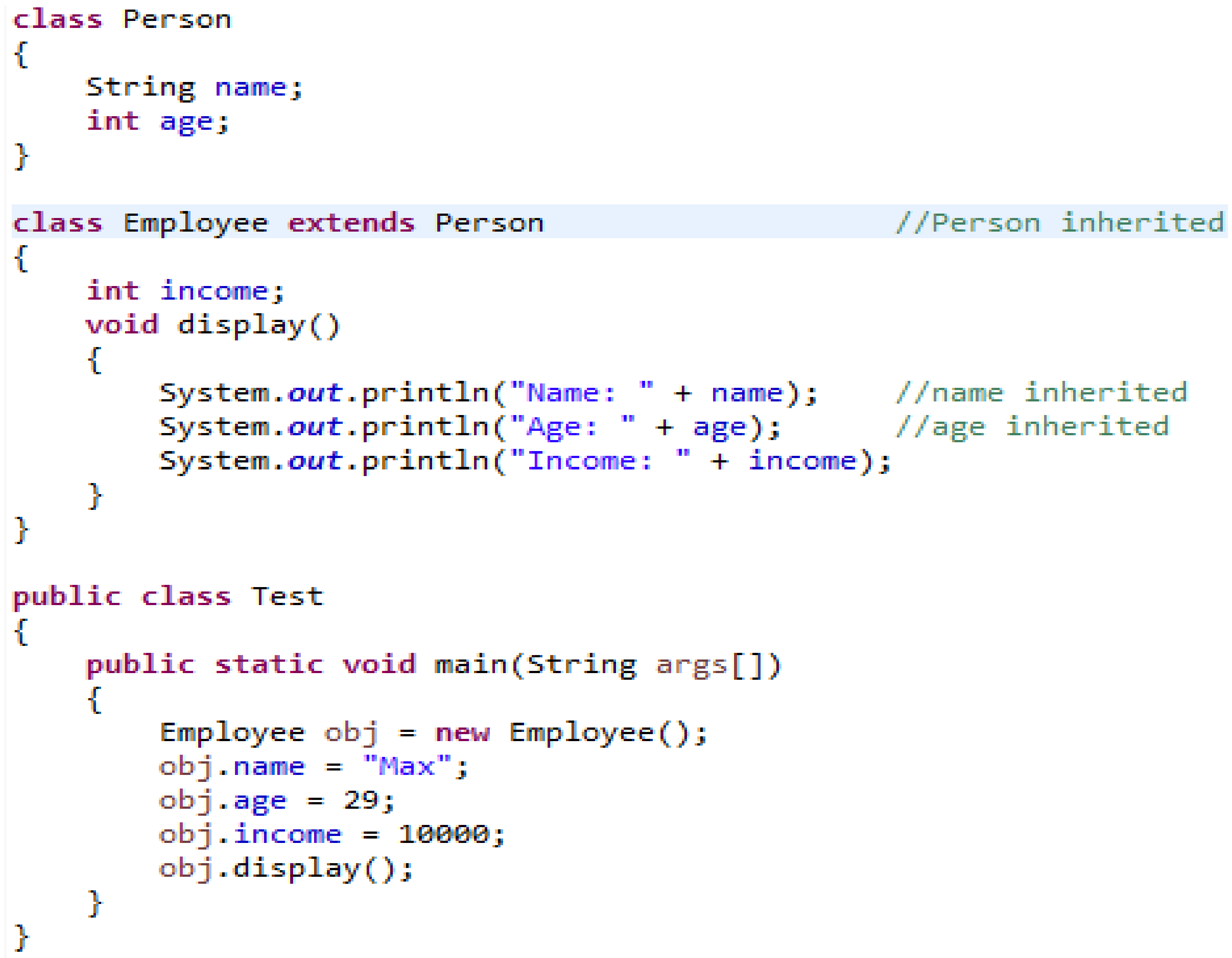



int gear
int speed
applyBrake()
speedUp()
toString()
int seatHeight
setHeight()
toString()
copy of Bicycle methods and fields in MountainBike object
objects of MountainBike class

In above scenario, when an object of MountainBike class is created, a copy of the all methods and fields of the superclass acquire memory in this object. That is why, by using the object of the subclass we can also access the members of a superclass.

**Please note that during inheritance only object of subclass is created, not the superclass.**

# Example (Simple inheritance)



```
class Person
{
    String name;
    int age;
}

class Employee extends Person                         //Person inherited
{
    int income;
    void display()
    {
        System.out.println("Name: " + name);    //name inherited
        System.out.println("Age: " + age);      //age inherited
        System.out.println("Income: " + income);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Employee obj = new Employee();
        obj.name = "Max";
        obj.age = 29;
        obj.income = 10000;
        obj.display();
    }
}
```

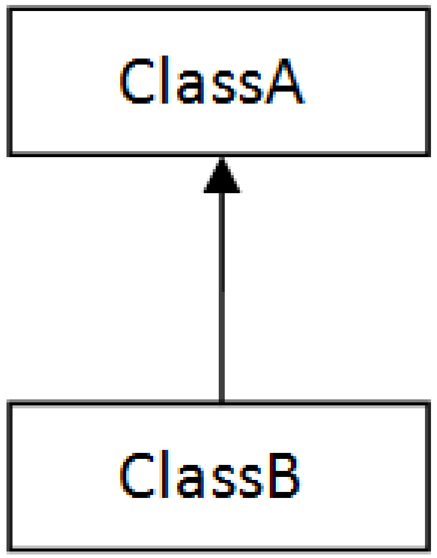
## Exercise

Write a program to create base-class Person having **private** field: name

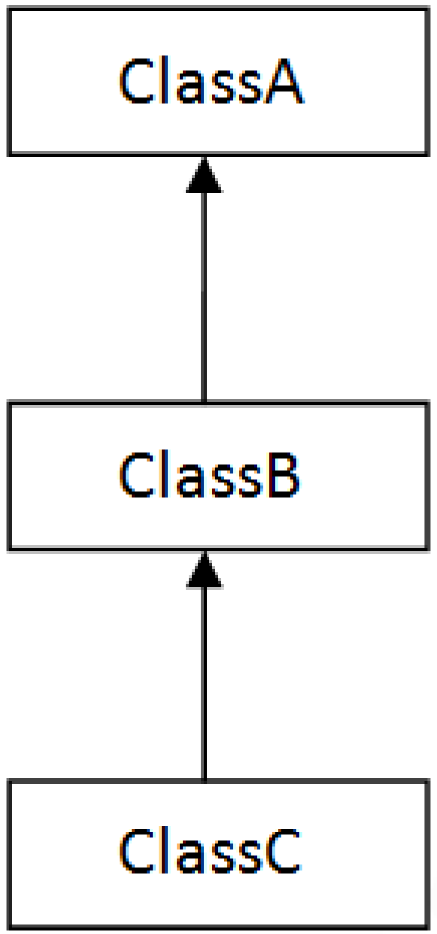
Now create a subclass Student having field: regNo and a display method to print all details.

Now create a class Test in which main() method resides. Create a student class object and display his details.

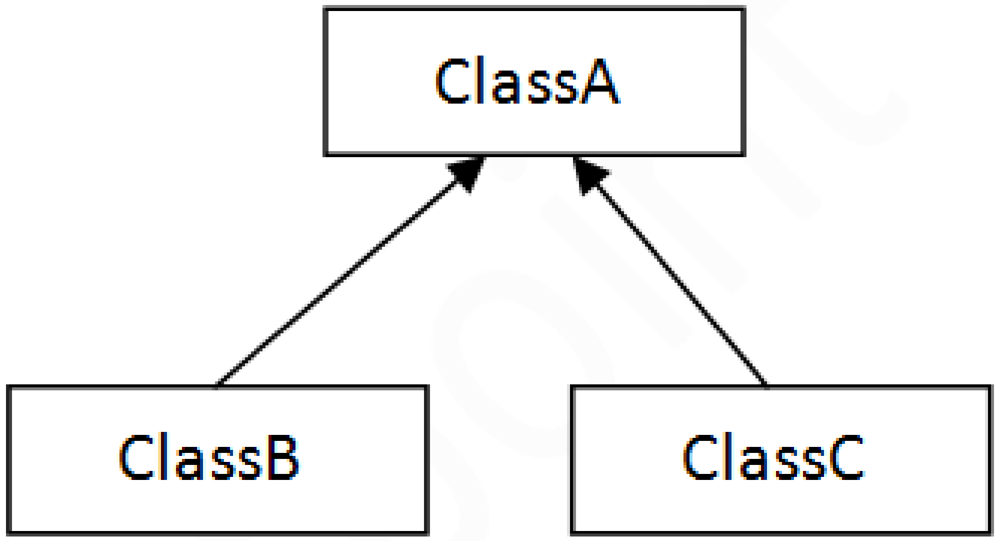
# Types of Inheritance



1) Single



2) Multilevel



3) Hierarchical

## Multiple Inheritance

- The mechanism of inheriting the features of more than one base class into a single class is known as **multiple inheritance**.

- Java does not support multiple inheritance using classes but the multiple inheritance can be achieved by using the **interface**.

- In Java Multiple Inheritance can be achieved through use of **interfaces** by implementing more than one **interfaces** in a class.

## Exercise 1

Create a class named **Horse** that contains data fields for the name, color, and birth year. Include get and set methods for these fields.

Next, create a subclass named **RaceHorse**, which contains an additional field that holds the number of races in which the horse has competed and additional methods to get and set the new field.

Write an application that demonstrates using objects of each class. Save the files as Horse.java, RaceHorse.java, and DemoHorses.java.

## Exercise 2

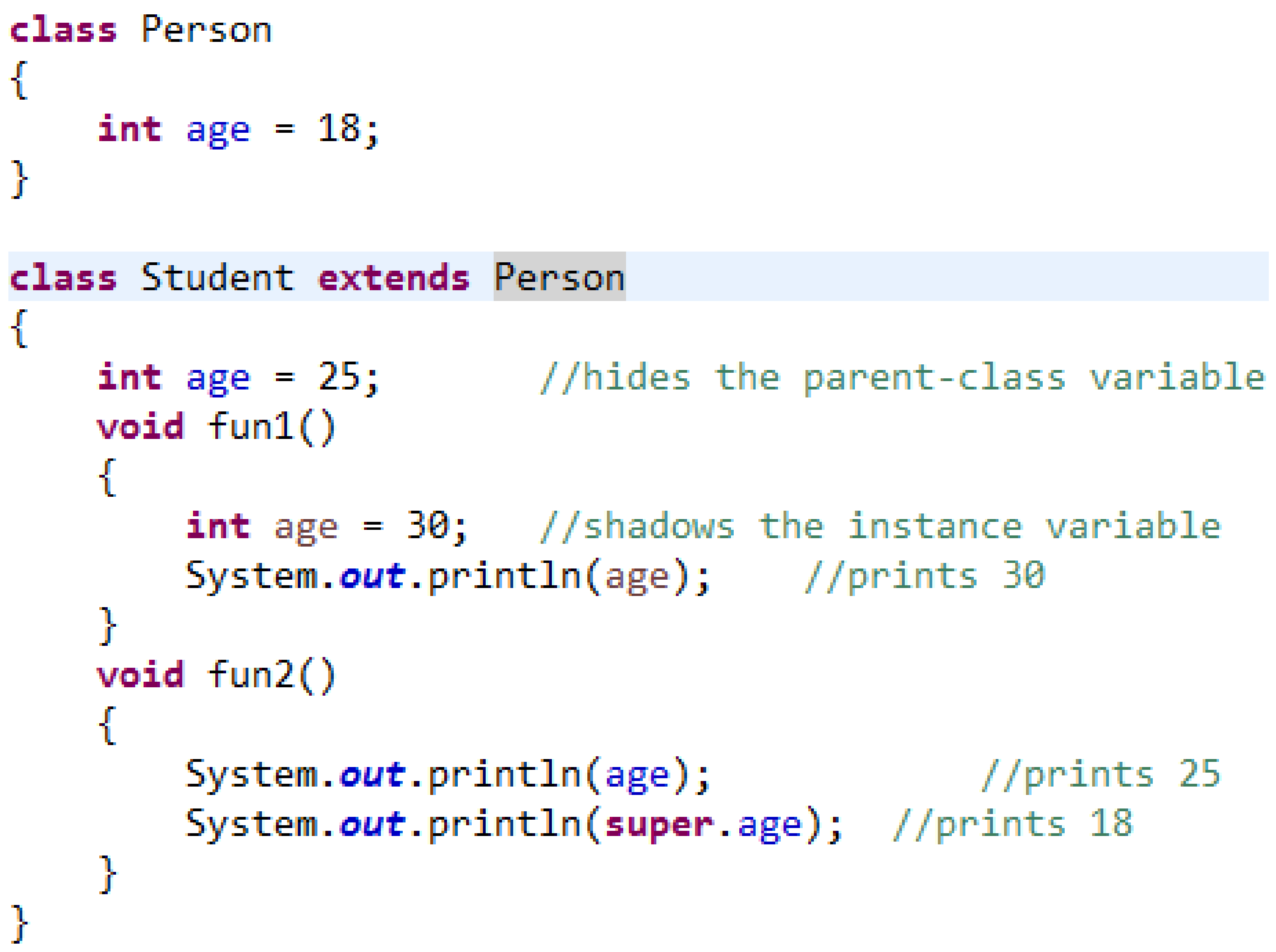
Create a class named **Poem** that contains fields for the name of the poem and the number of lines in it. Include a constructor that requires values for both fields. Also include get methods to retrieve field values.

Create three subclasses: **Couplet**, **Limerick**, and **Haiku**. The constructor for each subclass requires only a title; the lines field is set using a constant value. A couplet has two lines, a limerick has five lines, and a haiku has three lines.

Create an application that demonstrates usage of an object of each type. Save the files as **Poem.java, Couplet.java, Limerick.java, Haiku.java**, and **DemoPoems.java**.

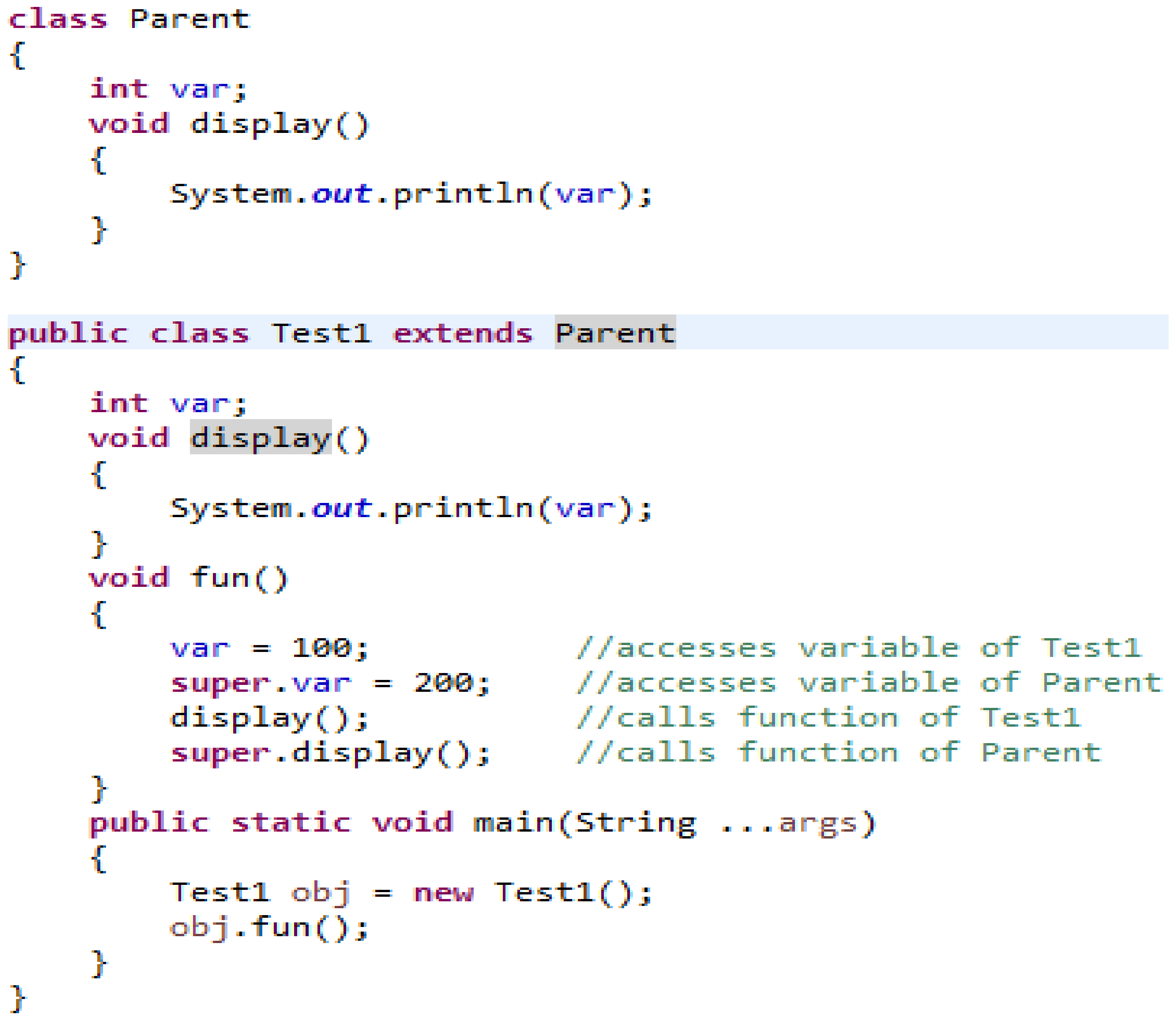
## Accessing parent class members

## Example (access parent member)



```java
class Person
{
    int age = 18;
}

class Student extends Person
{
    int age = 25;       //hides the parent-class variable
    void fun1()
    {
        int age = 30;   //shadows the instance variable
        System.out.println(age);    //prints 30
    }
    void fun2()
    {
        System.out.println(age);            //prints 25
        System.out.println(super.age);  //prints 18
    }
}
```

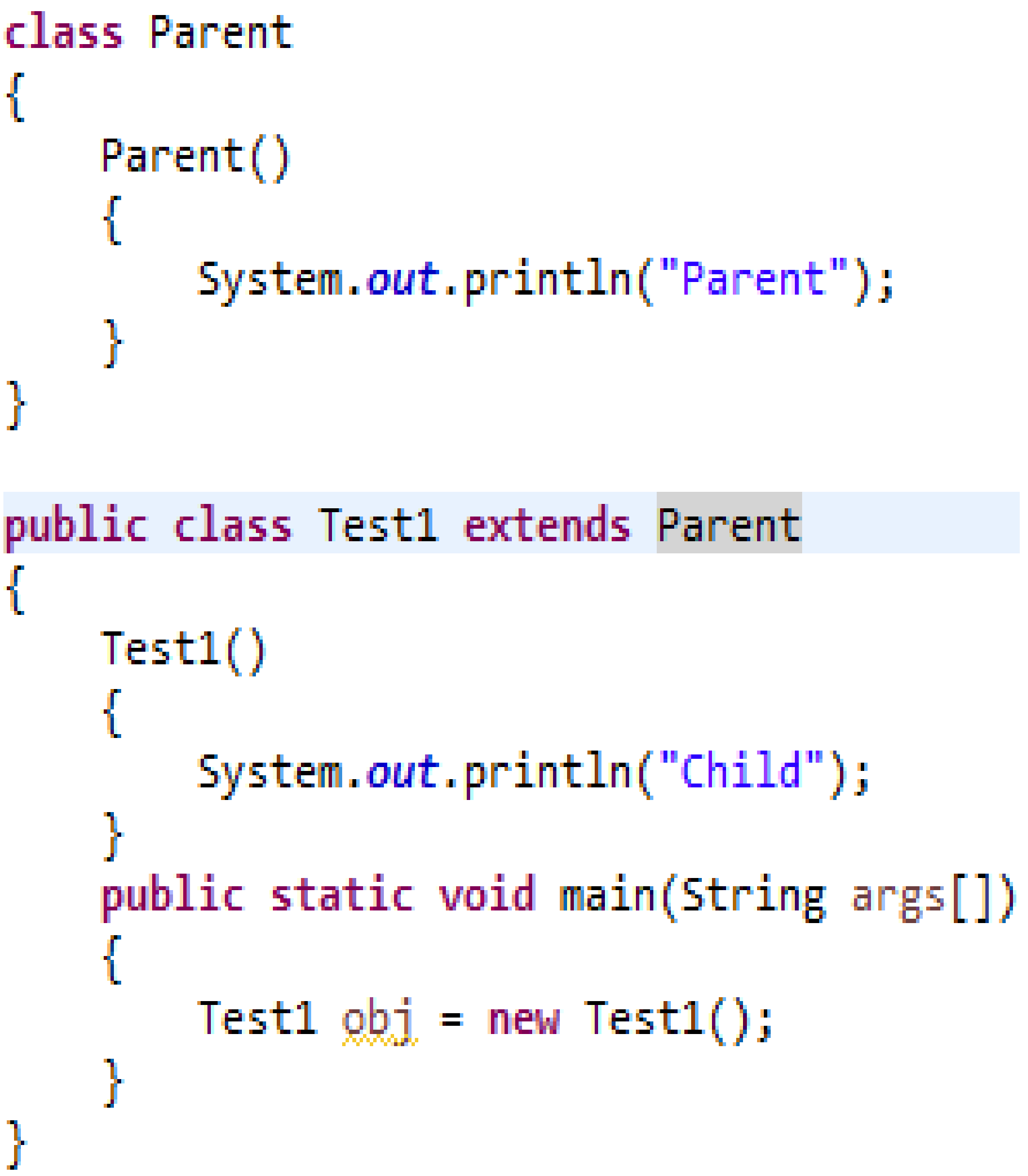
## Example (access parent member)



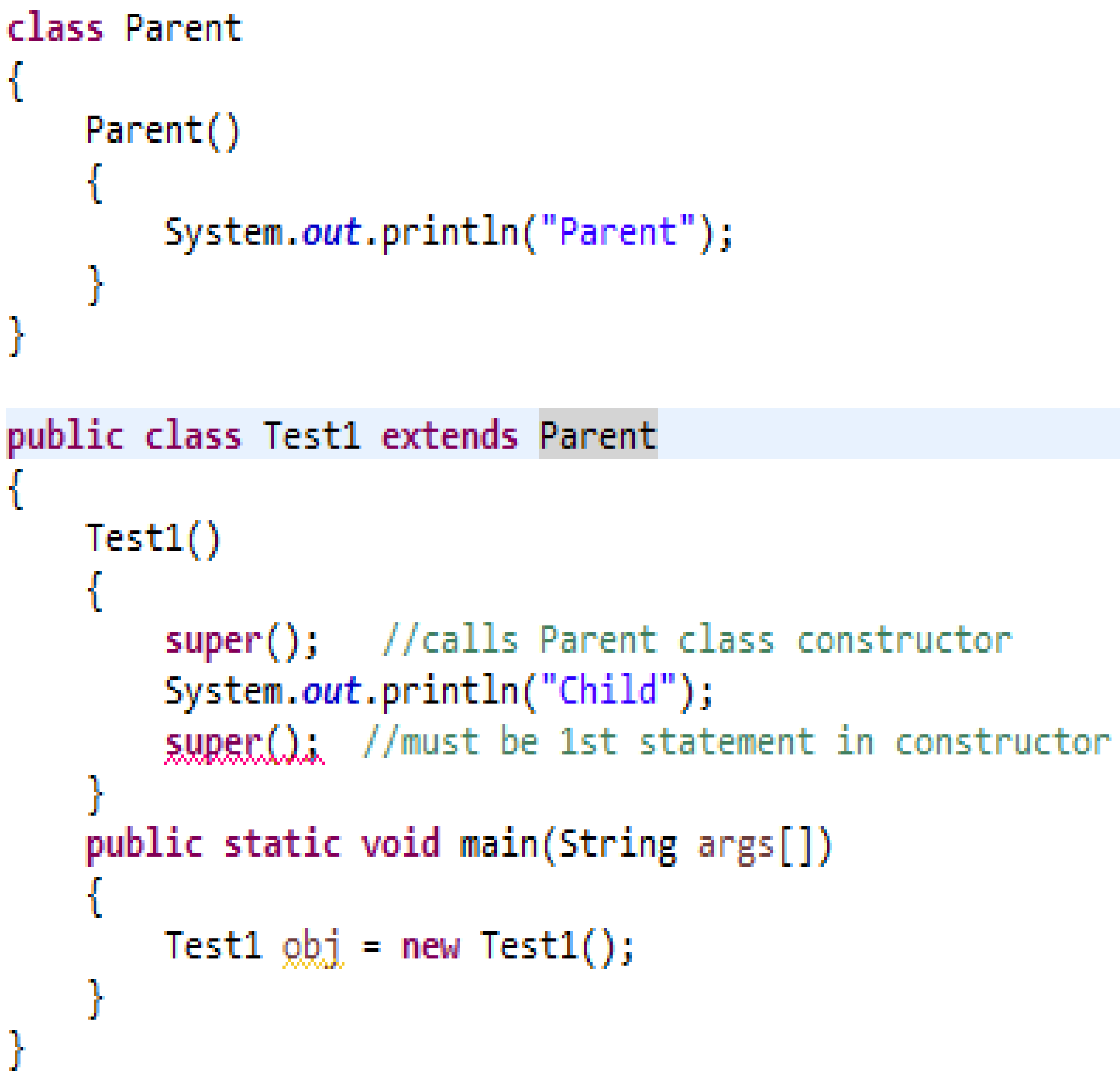
```java
class Parent
{
    int var;
    void display()
    {
        System.out.println(var);
    }
}

public class Test1 extends Parent
{
    int var;
    void display()
    {
        System.out.println(var);
    }
    void fun()
    {
        var = 100;          //accesses variable of Test1
        super.var = 200;    //accesses variable of Parent
        display();          //calls function of Test1
        super.display();    //calls function of Parent
    }
    public static void main(String ...args)
    {
        Test1 obj = new Test1();
        obj.fun();
    }
}
```

## Functionality of constructors in Inheritance

# Example (constructor calling in inheritance)

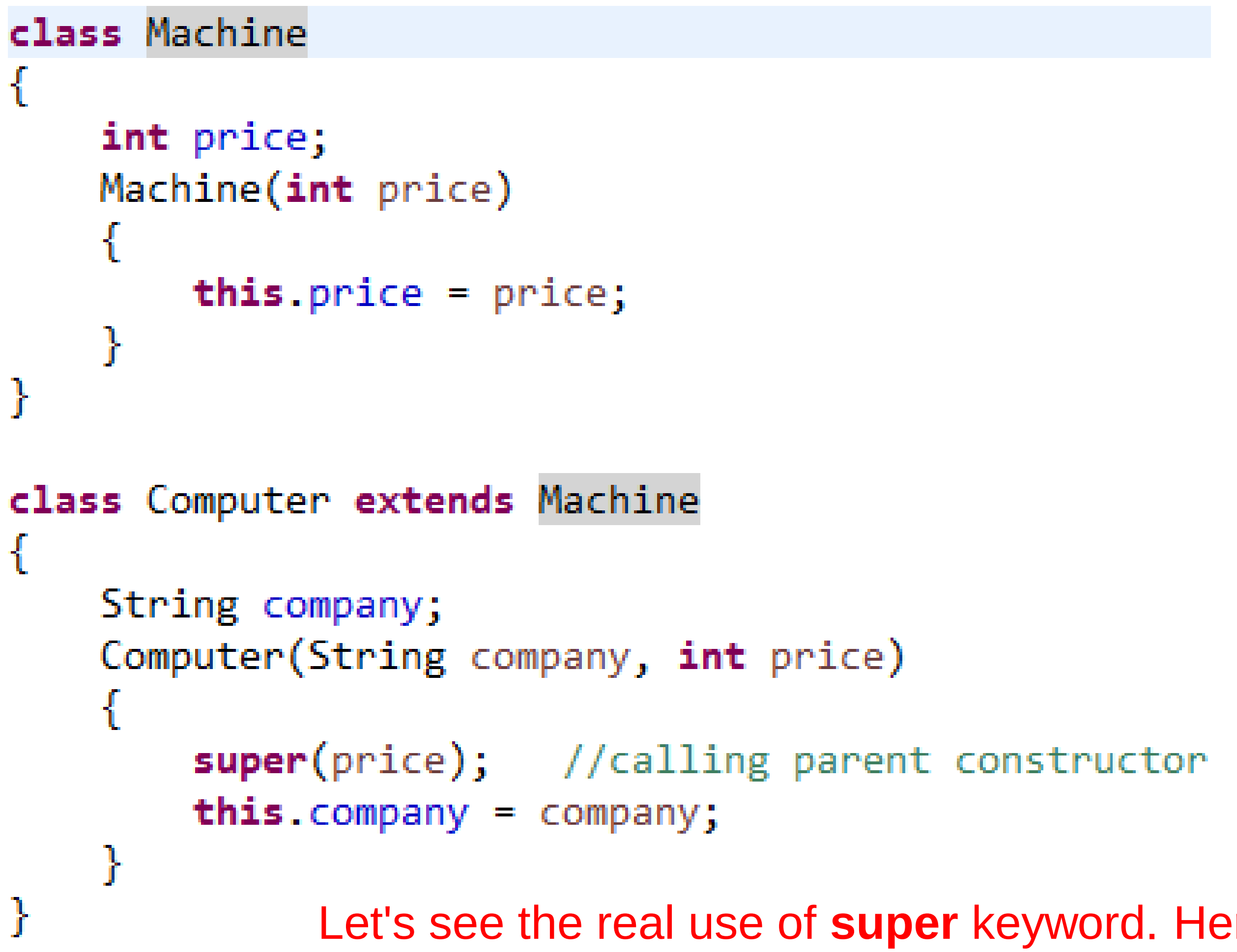


```java
class Parent
{
    Parent()
    {
        System.out.println("Parent");
    }
}

public class Test1 extends Parent
{
    Test1()
    {
        System.out.println("Child");
    }
    public static void main(String args[])
    {
        Test1 obj = new Test1();
    }
}
```

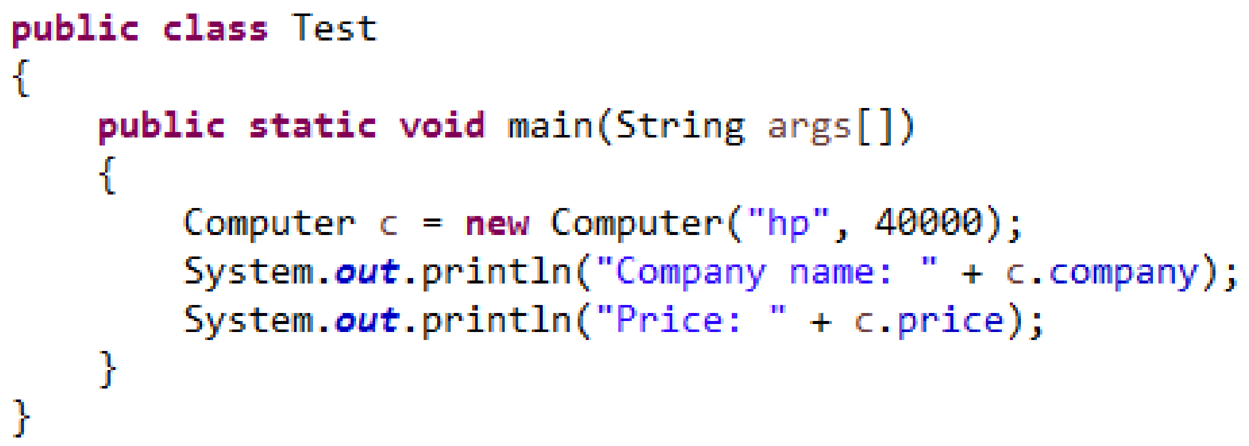


```java
class Parent
{
    Parent()
    {
        System.out.println("Parent");
    }
}

public class Test1 extends Parent
{
    Test1()
    {
        super();   //calls Parent class constructor
        System.out.println("Child");
        super();  //must be 1st statement in constructor
    }
    public static void main(String args[])
    {
        Test1 obj = new Test1();
    }
}
```

# Example (constructor calling in inheritance)



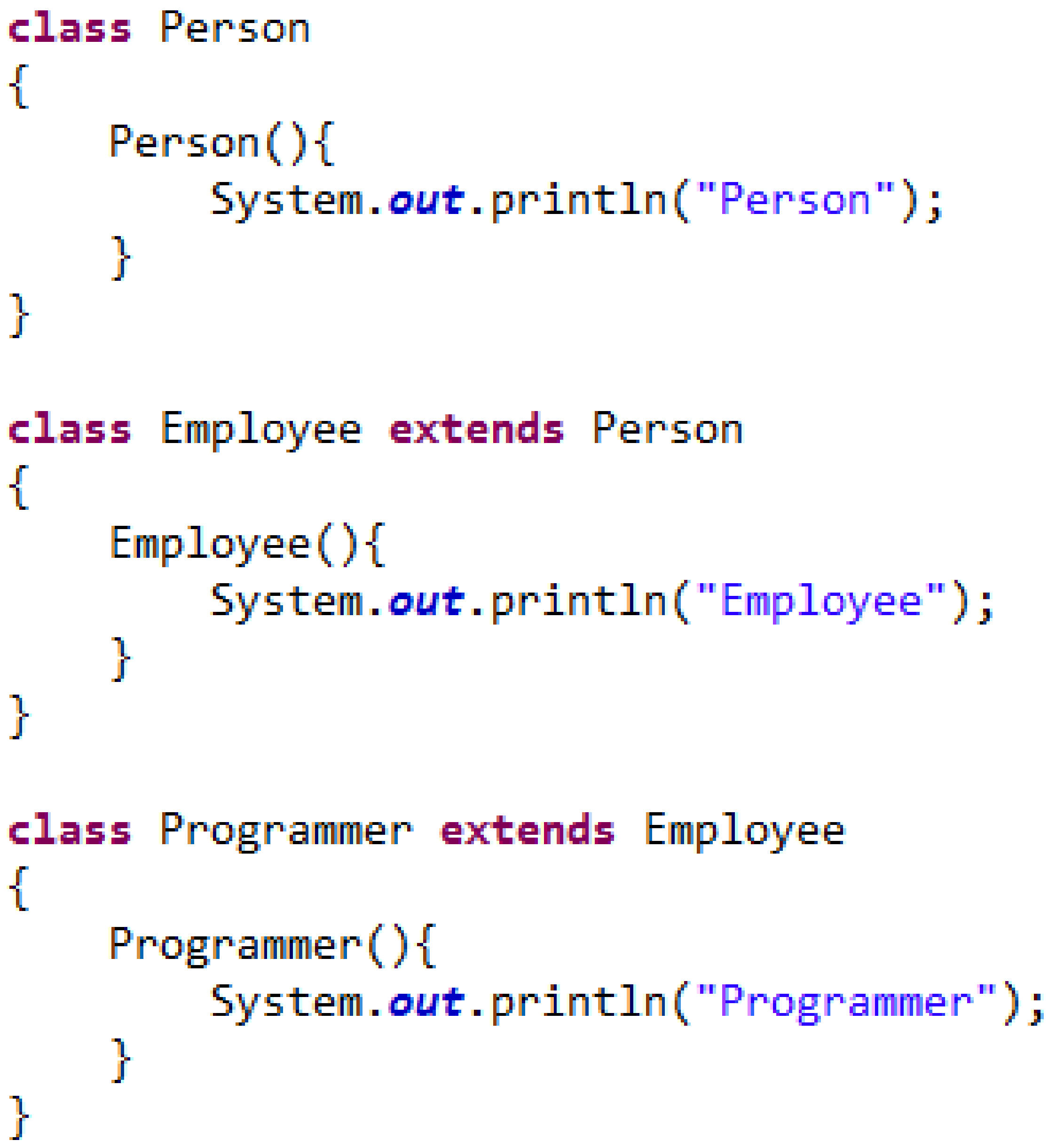
```
class Machine
{
    int price;
    Machine(int price)
    {
        this.price = price;
    }
}

class Computer extends Machine
{
    String company;
    Computer(String company, int price)
    {
        super(price);   //calling parent constructor
        this.company = company;
    }
}
```



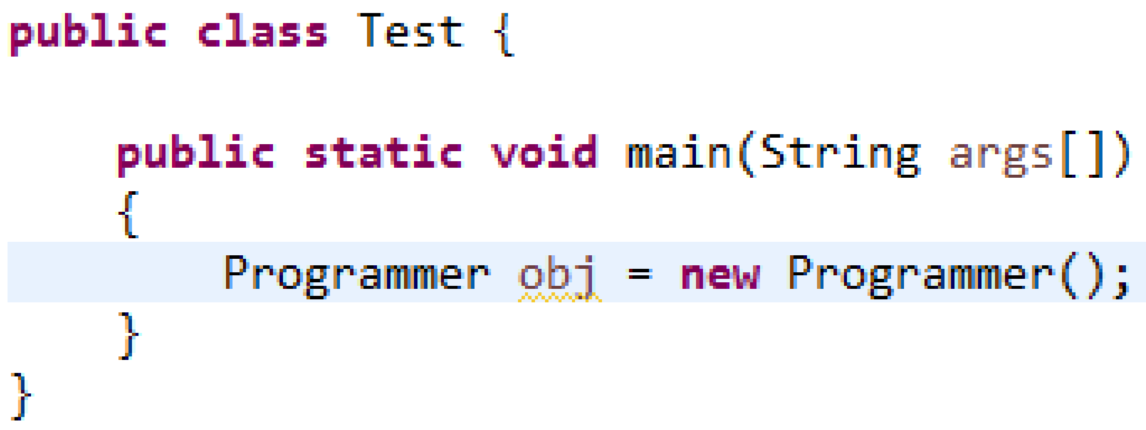
```
public class Test
{
    public static void main(String args[])
    {
        Computer c = new Computer("hp", 40000);
        System.out.println("Company name: " + c.company);
        System.out.println("Price: " + c.price);
    }
}
```

Let's see the real use of **super** keyword. Here, **Computer** class inherits **Machine** class, so all the properties of **Machine** will be inherited to **Computer** 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.

# Constructor chaining using **super** keyword



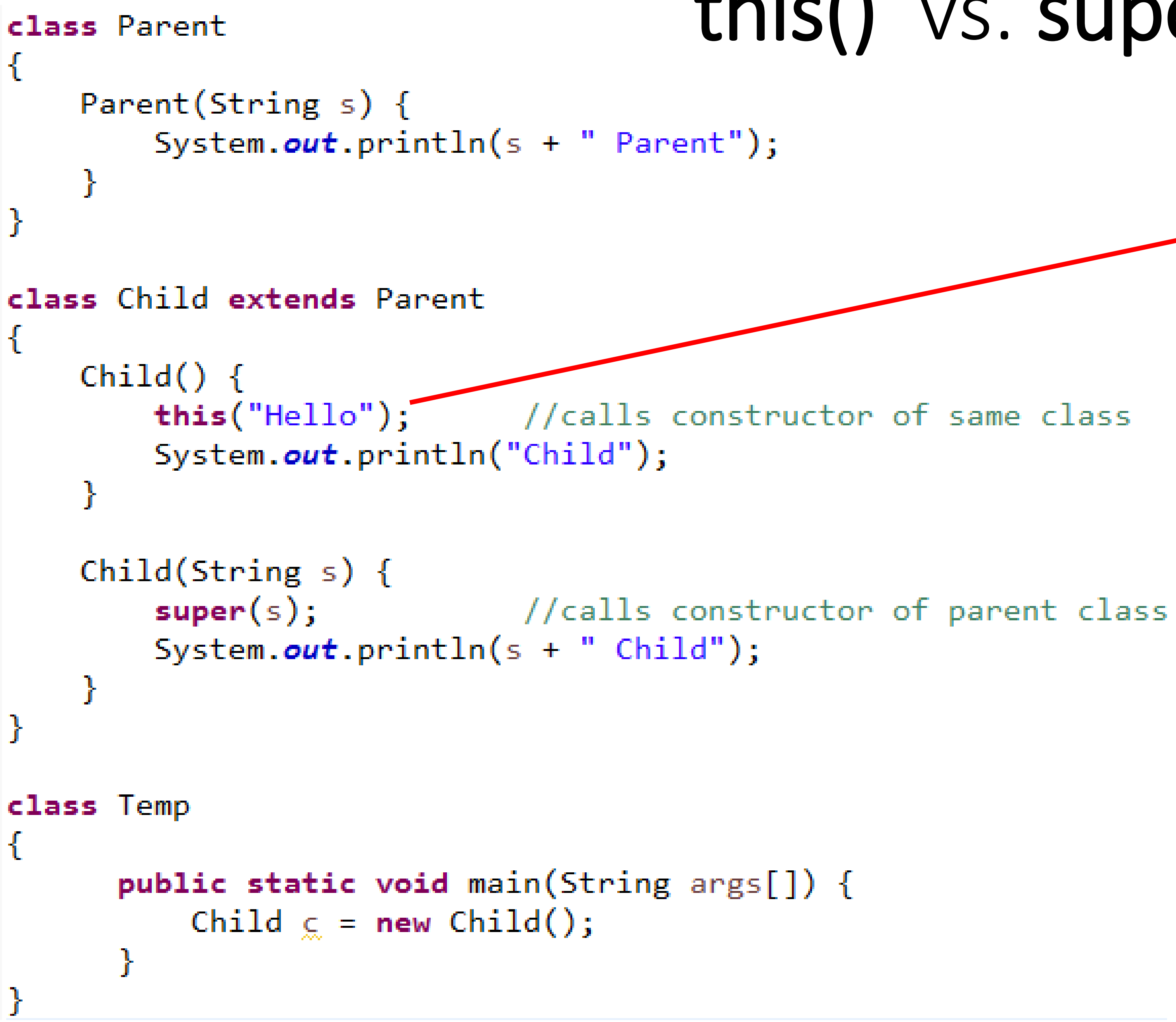
```java
class Person
{
    Person(){
        System.out.println("Person");
    }
}

class Employee extends Person
{
    Employee(){
        System.out.println("Employee");
    }
}

class Programmer extends Employee
{
    Programmer(){
        System.out.println("Programmer");
    }
}
```



```java
public class Test {

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

By default, statement **super()** is inserted at beginning of every constructor by the compiler, which calls the parent class constructor.

# this()  vs. super()



```
class Parent
{
    Parent(String s) {
        System.out.println(s + " Parent");
    }
}

class Child extends Parent
{
    Child() {
        this("Hello");      //calls constructor of same class
        System.out.println("Child");
    }

    Child(String s) {
        super(s);           //calls constructor of parent class
        System.out.println(s + " Child");
    }
}

class Temp
{
      public static void main(String args[]) {
          Child c = new Child();
      }
}
```

Here, **this()** supercedes the **super()** i.e. as the complier sees that **this()** has been used in the constructor, then it won’t insert **super()** as 1st line of constructor.

## Role of access-specifiers in Inheritance

## Key Points

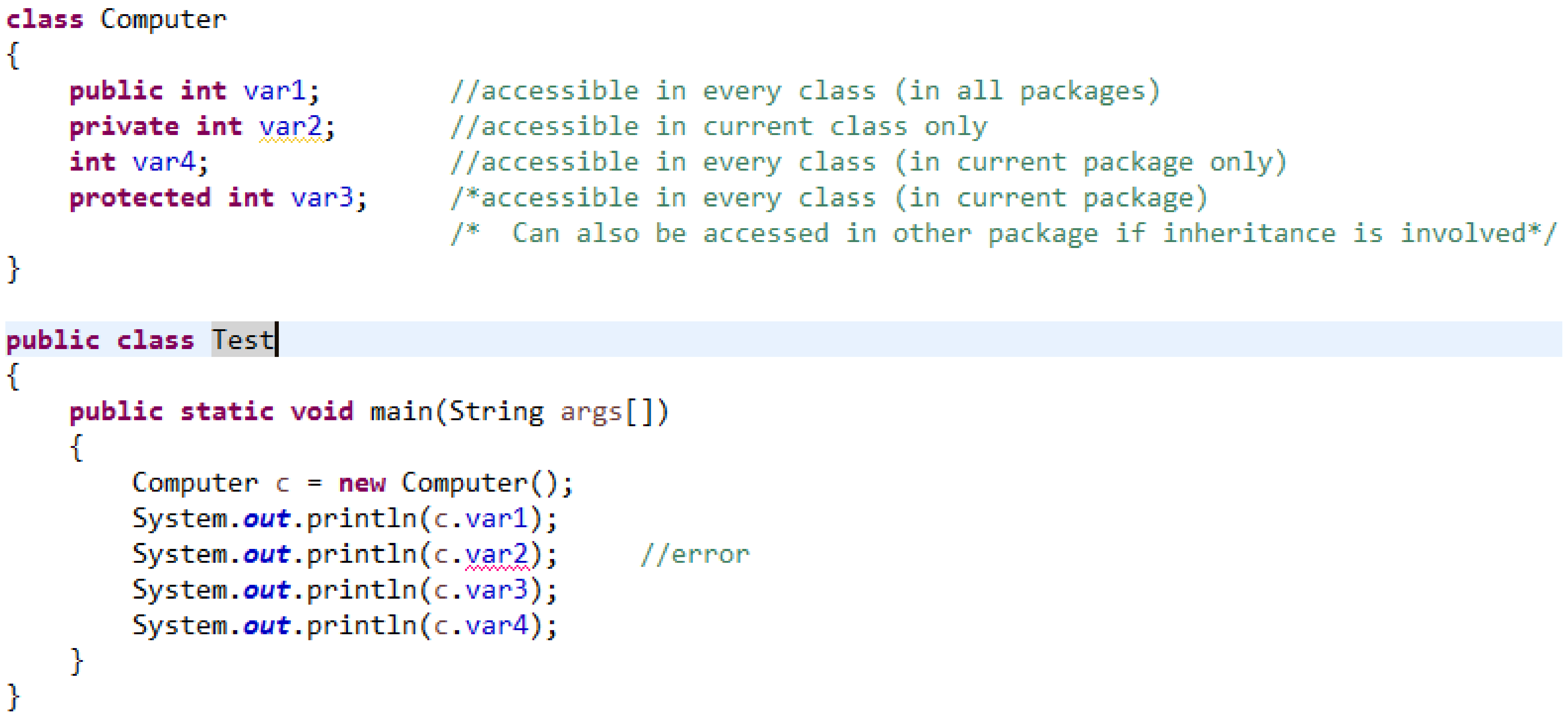
- **Private members** of the super-class are not inherited by the subclass.

- Members that have default accessibility in the super-class are also not inherited by subclasses in other packages.

- Constructors and initializer blocks are not inherited by a subclass.

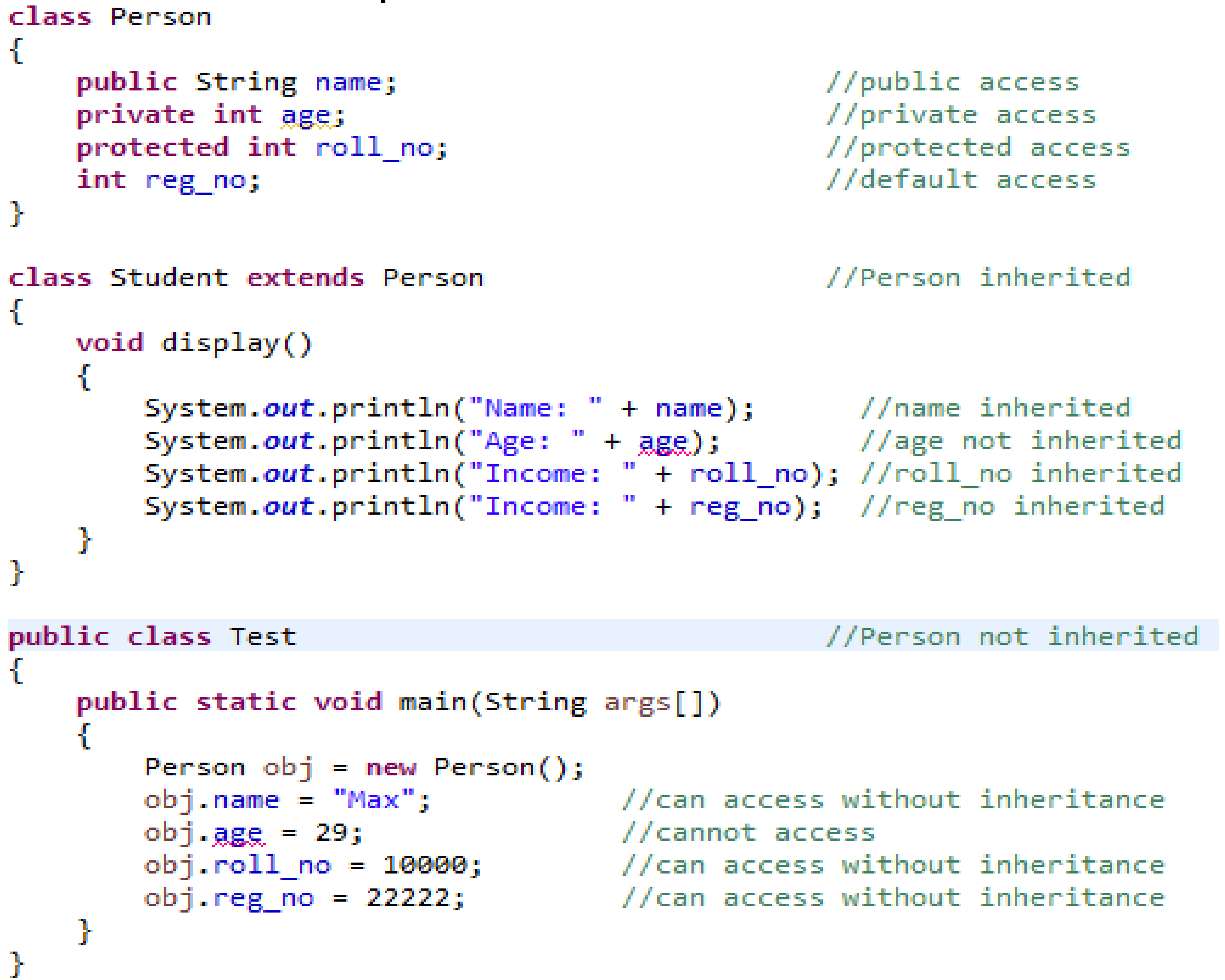
- A subclass can extend only one super-class.

# Example (access specifiers in same package)



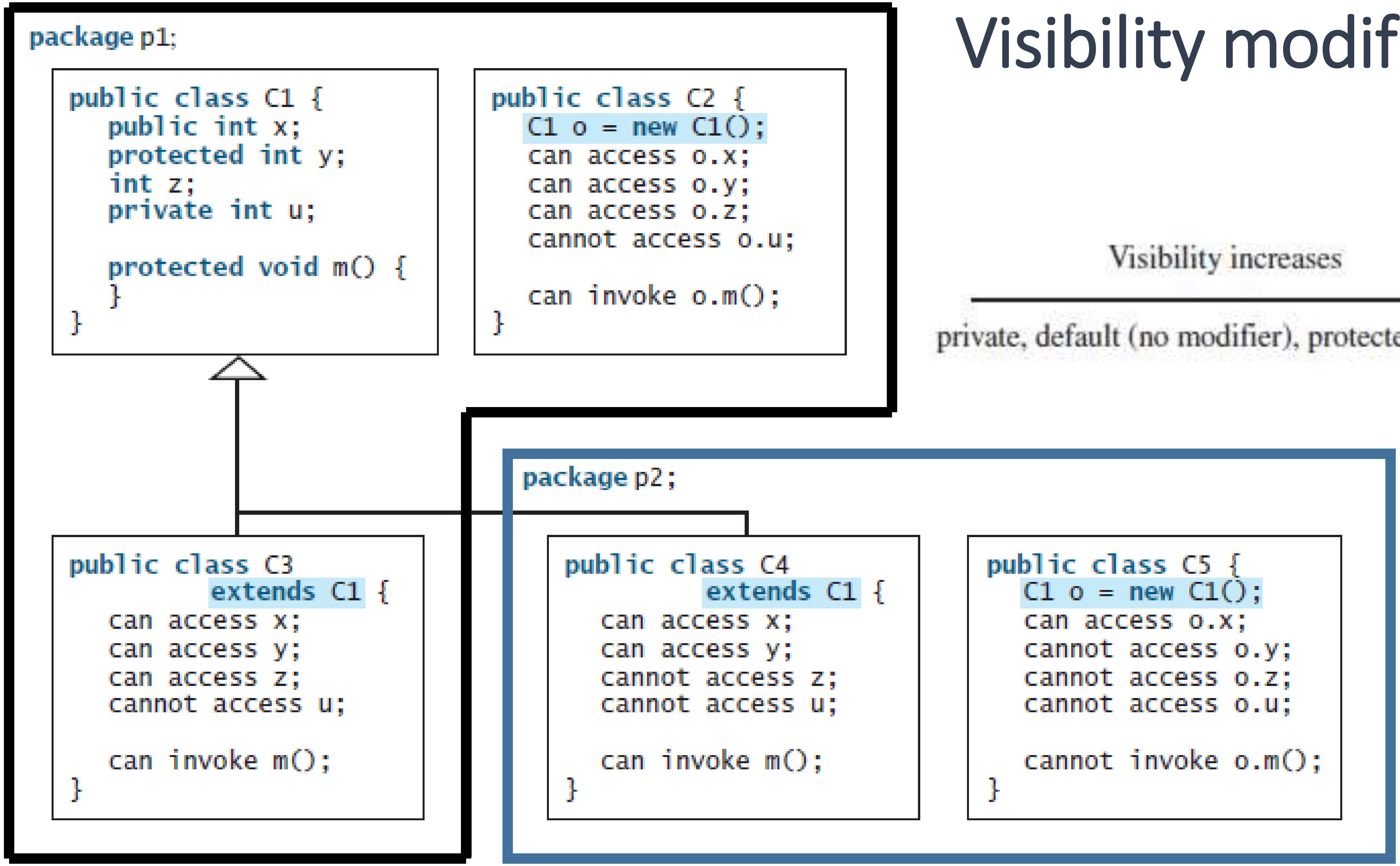
```java
class Computer
{
    public int var1;        //accessible in every class (in all packages)
    private int var2;       //accessible in current class only
    int var4;               //accessible in every class (in current package only)
    protected int var3;     /*accessible in every class (in current package)
                            /*  Can also be accessed in other package if inheritance is involved*/
}

public class Test
{
    public static void main(String args[])
    {
        Computer c = new Computer();
        System.out.println(c.var1);
        System.out.println(c.var2);     //error
        System.out.println(c.var3);
        System.out.println(c.var4);
    }
}
```

# Example (access specifiers in same package)



```
class Person
{
    public String name;                               //public access
    private int age;                                  //private access
    protected int roll_no;                            //protected access
    int reg_no;                                       //default access
}

class Student extends Person                          //Person inherited
{
    void display()
    {
        System.out.println("Name: " + name);        //name inherited
        System.out.println("Age: " + age);          //age not inherited
        System.out.println("Income: " + roll_no);   //roll_no inherited
        System.out.println("Income: " + reg_no);    //reg_no inherited
    }
}

public class Test                                     //Person not inherited
{
    public static void main(String args[])
    {
        Person obj = new Person();
        obj.name = "Max";            //can access without inheritance
        obj.age = 29;                //cannot access
        obj.roll_no = 10000;         //can access without inheritance
        obj.reg_no = 22222;          //can access without inheritance
    }
}
```

# Visibility modifiers



```
public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

```
public class C2 {
   C1 o = new C1();
   can access o.x;
   can access o.y;
   can access o.z;
   cannot access o.u;

   can invoke o.m();
}
```

```
public class C3
         extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

```
public class C4
         extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

```
public class C5 {
   C1 o = new C1();
   can access o.x;
   cannot access o.y;
   cannot access o.z;
   cannot access o.u;

   cannot invoke o.m();
}
```

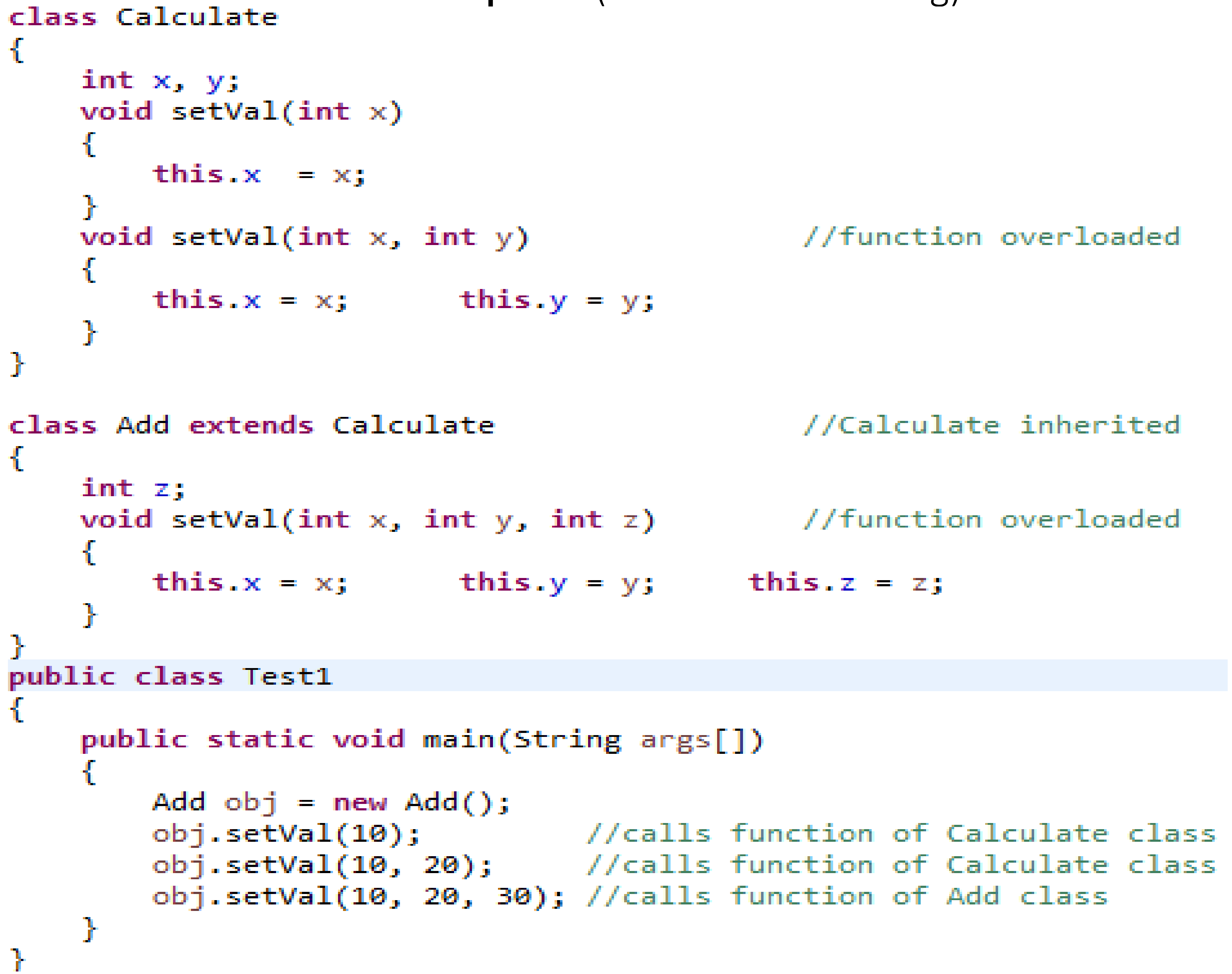
Visibility increases

$$\xrightarrow{\hspace{6cm}}$$

private, default (no modifier), protected, public

## Method overloading in Inheritance

# Example (method overloading)



```java
class Calculate
{
    int x, y;
    void setVal(int x)
    {
        this.x  = x;
    }
    void setVal(int x, int y)               //function overloaded
    {
        this.x = x;     this.y = y;
    }
}

class Add extends Calculate                 //Calculate inherited
{
    int z;
    void setVal(int x, int y, int z)        //function overloaded
    {
        this.x = x;     this.y = y;     this.z = z;
    }
}
public class Test1
{
    public static void main(String args[])
    {
        Add obj = new Add();
        obj.setVal(10);         //calls function of Calculate class
        obj.setVal(10, 20);     //calls function of Calculate class
        obj.setVal(10, 20, 30); //calls function of Add class
    }
}
```

## Method overriding

# Method Overriding

- Method overriding means having a **different implementation of the same method** in the **inherited class**.

- These two methods would have the **same signature, but different implementation**.

- One of these would exist in the **base class** and another in the **derived class**. These cannot exist in the same class.

Note:- A **class declared as final** cannot be inherited and a **method declared final** cannot be overridden in sub-class.

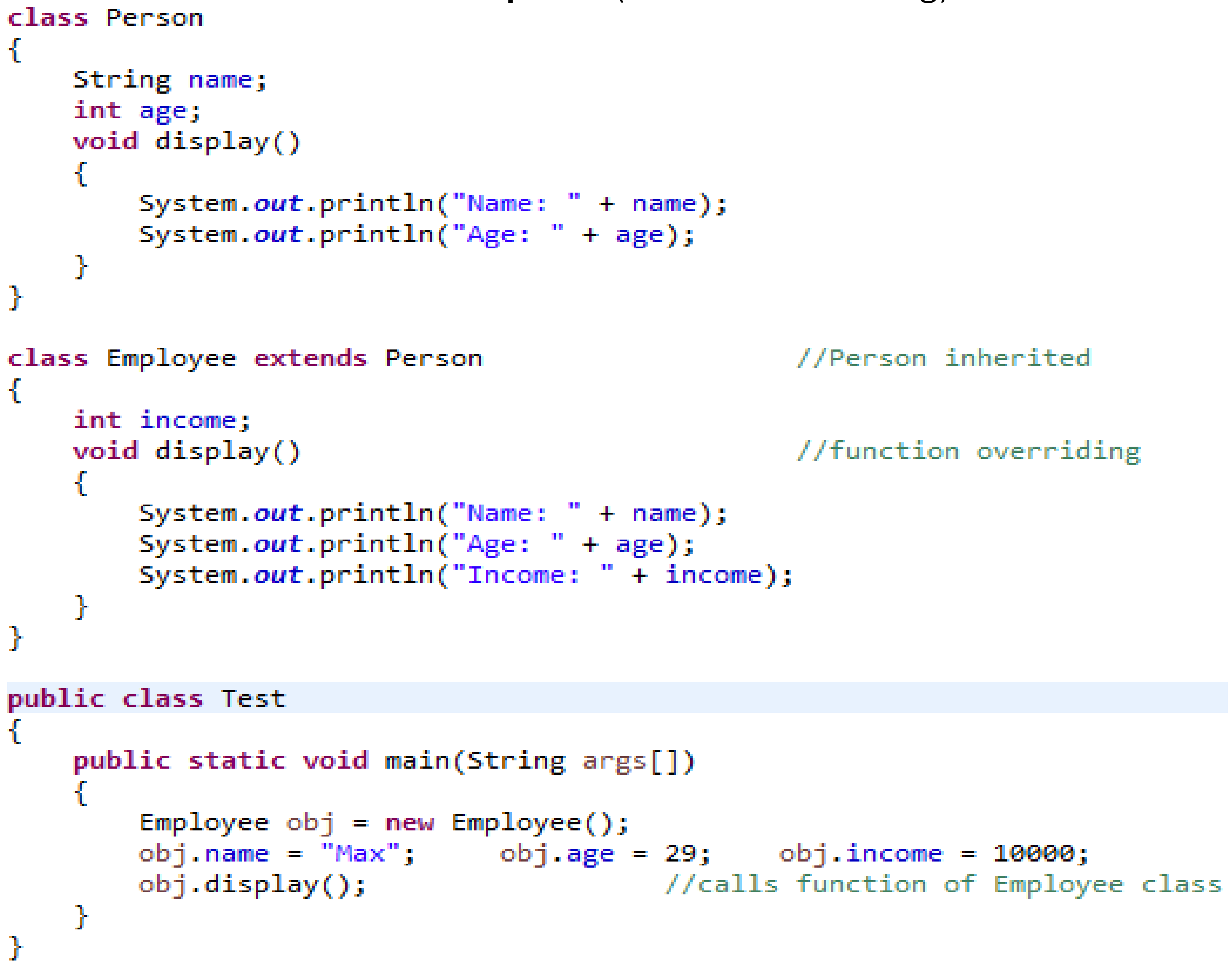
## Key Points

- The version of a method that is executed will be determined by the object that is used to invoke it.

- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed.

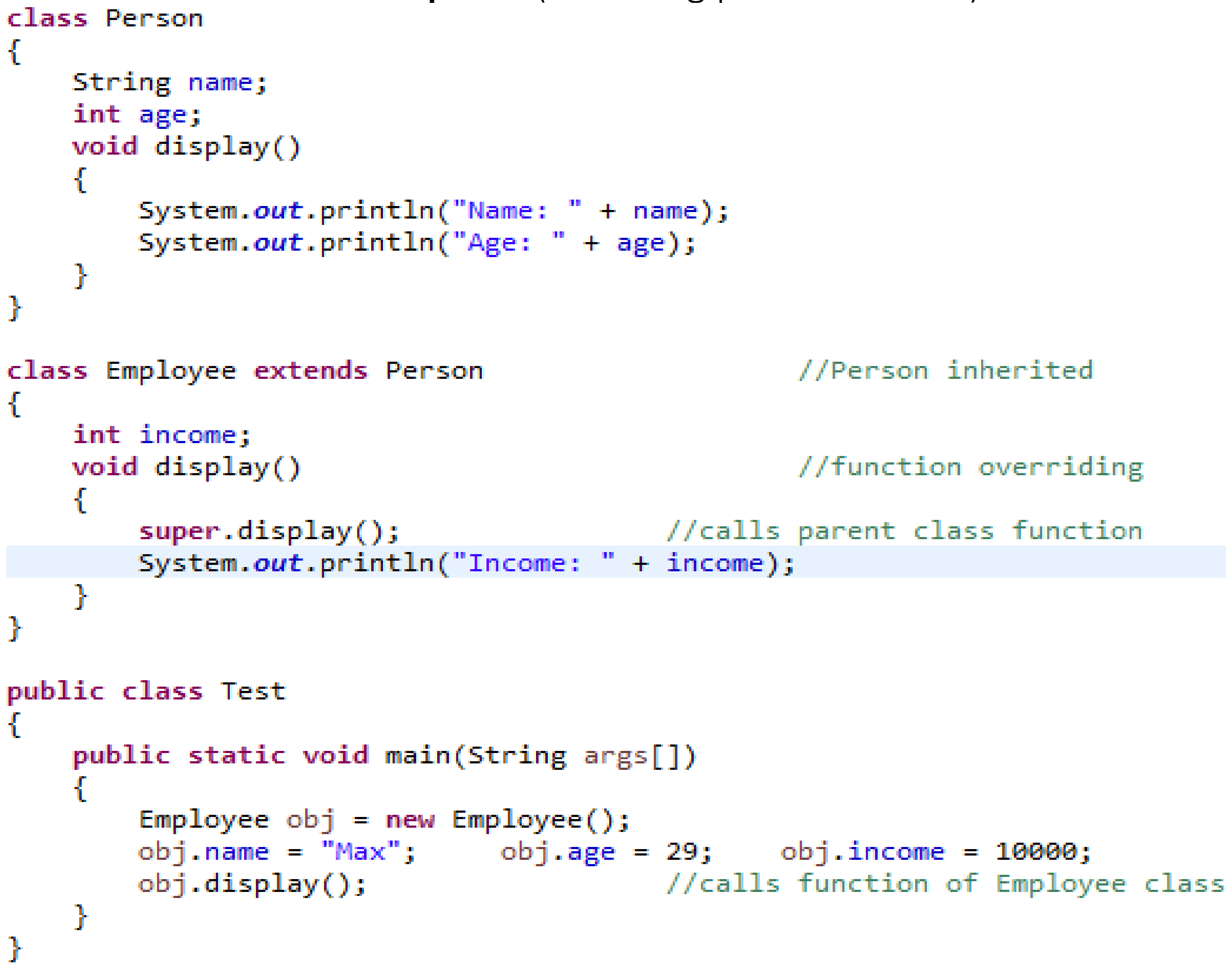
- If an object of the subclass is used to invoke the method, then the version in the child class will be executed.

# Example (method overriding)



```java
class Person
{
    String name;
    int age;
    void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

class Employee extends Person                   //Person inherited
{
    int income;
    void display()                              //function overriding
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Income: " + income);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Employee obj = new Employee();
        obj.name = "Max";    obj.age = 29;    obj.income = 10000;
        obj.display();                  //calls function of Employee class
    }
}
```

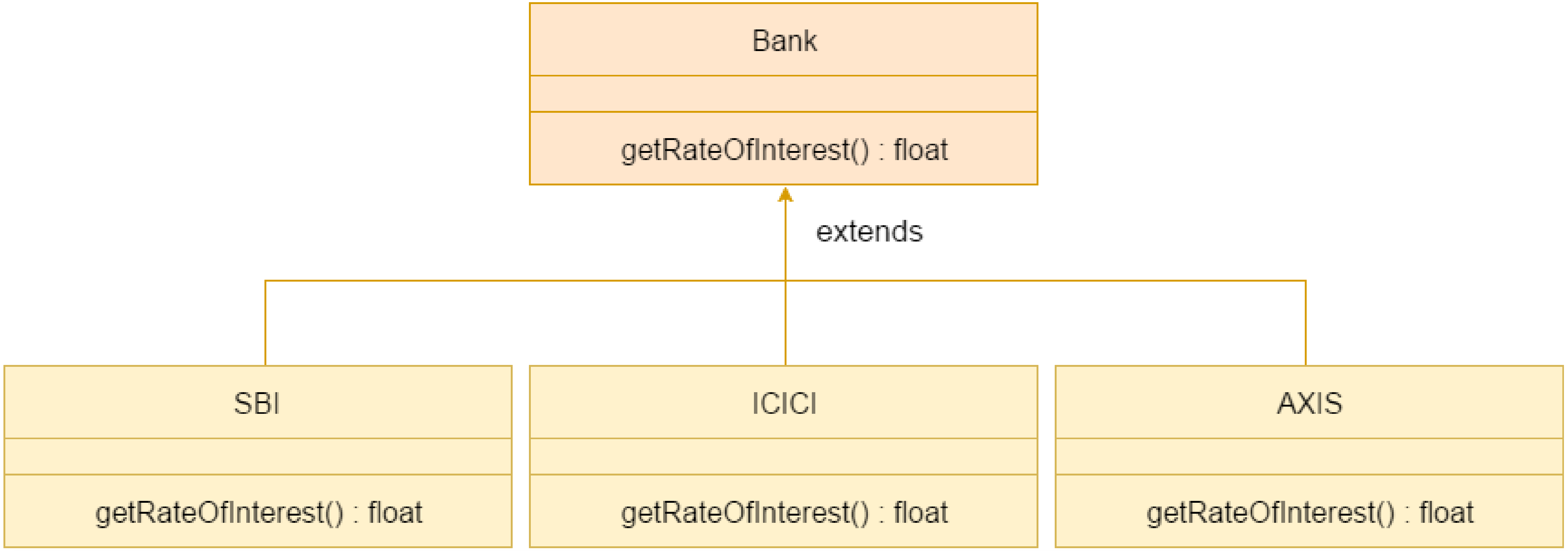
# Example (accessing parent method)



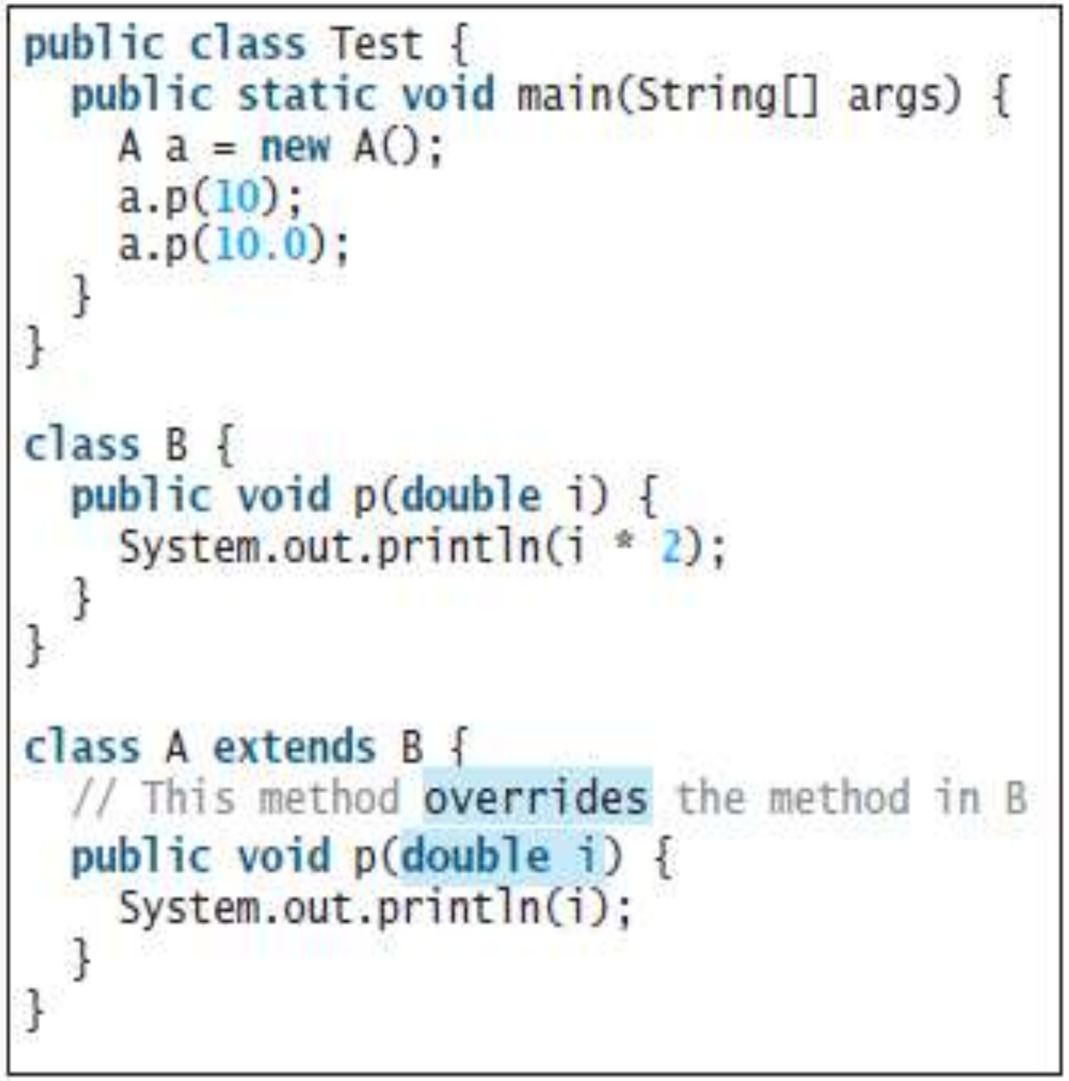
```java
class Person
{
    String name;
    int age;
    void display()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

class Employee extends Person                   //Person inherited
{
    int income;
    void display()                              //function overriding
    {
        super.display();                //calls parent class function
        System.out.println("Income: " + income);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Employee obj = new Employee();
        obj.name = "Max";       obj.age = 29;    obj.income = 10000;
        obj.display();                  //calls function of Employee class
    }
}
```

## Real example of Java Method Overriding

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

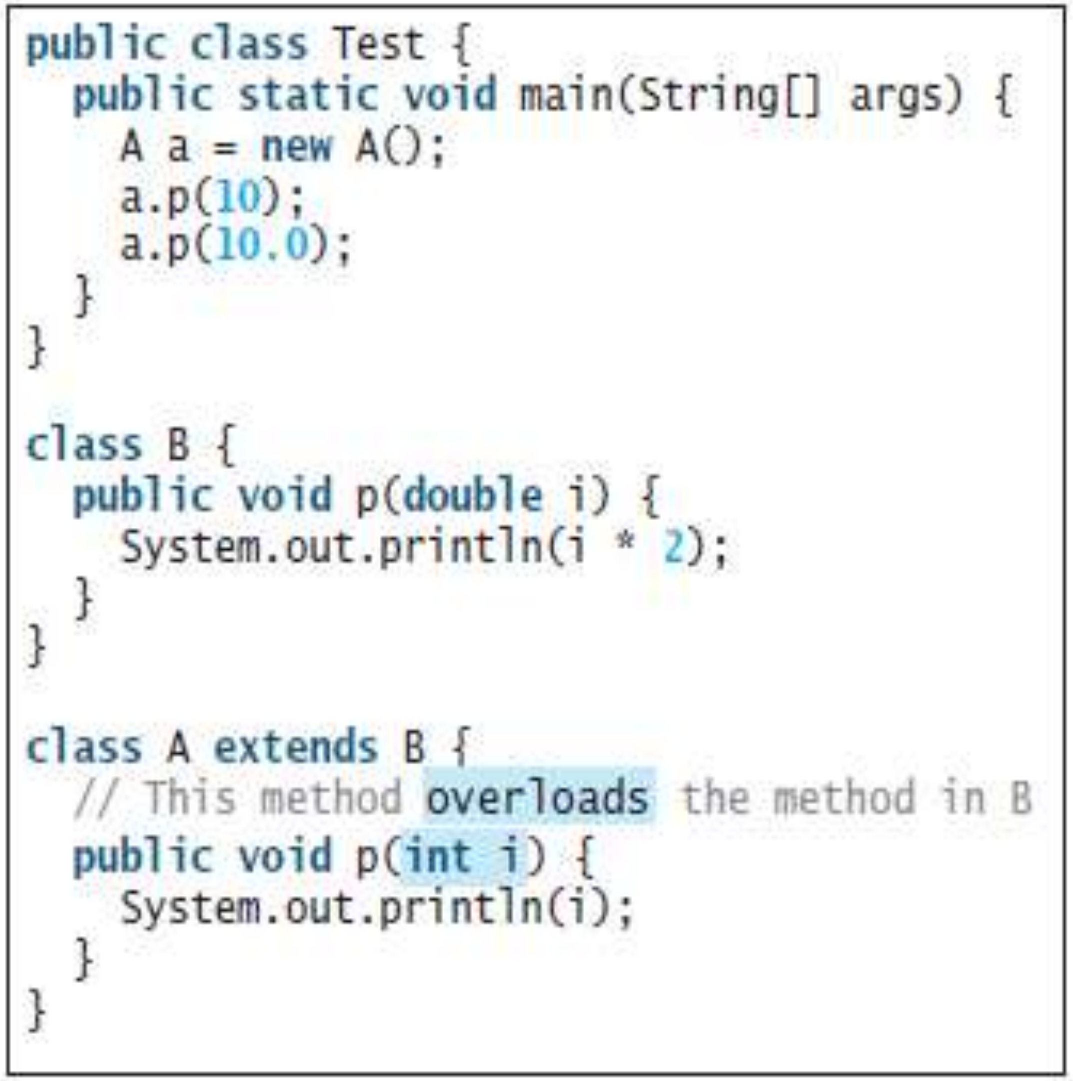

Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

# Difference b/w Overloading & Overriding



```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

(a)



```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

# Exercise

- Write a program to create **Person** class having attribues: name & age and method: **void display()** which will display the details of Person object.
- Create 2 sub-classes of Person class, namely: **Student** and **Employee** such that:

  Student class has attributes: regNo and method: void display()

  Employee class has attributes: id and salary and method: void display()

- Now create a **Test** class having main() method in which you have to create objects of all 3 classes. Initialize their attributes using parameterized constructor and display the details of all 3 objects.

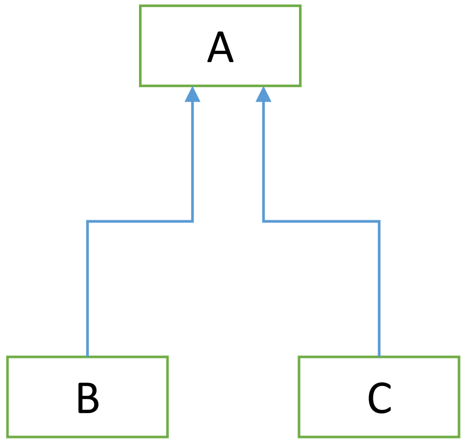
## Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a **call to an overridden method is resolved at run time**, rather than compile time.

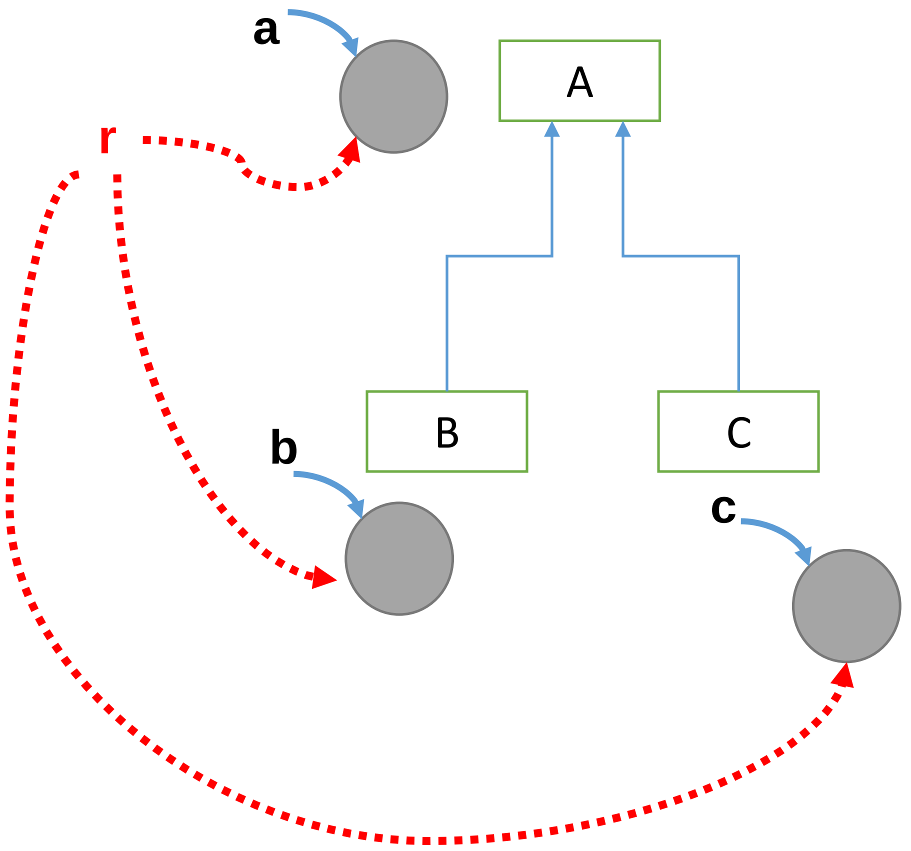
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

# Example

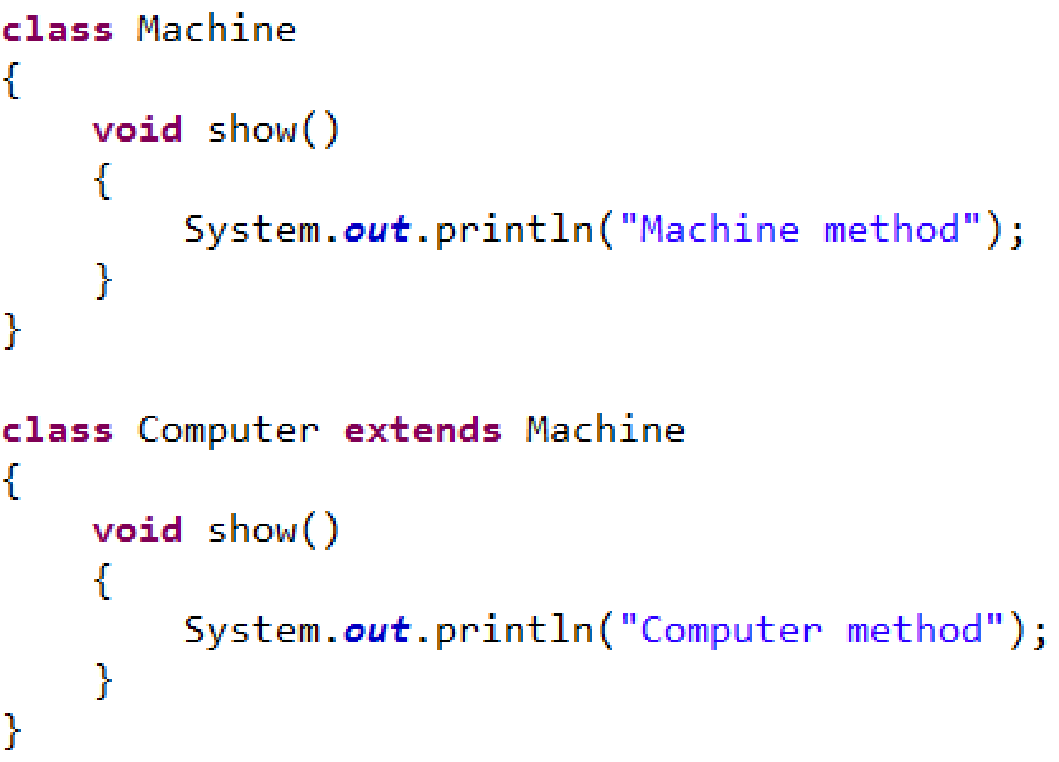
```
class A
{
        void callme() {
                System.out.println("Inside A's callme method");
        }
}

class B extends A
{
        void callme() {
                System.out.println("Inside B's callme method");
        }
}

class C extends A
{
        void callme() {
                System.out.println("Inside C's callme method");
        }
}
```



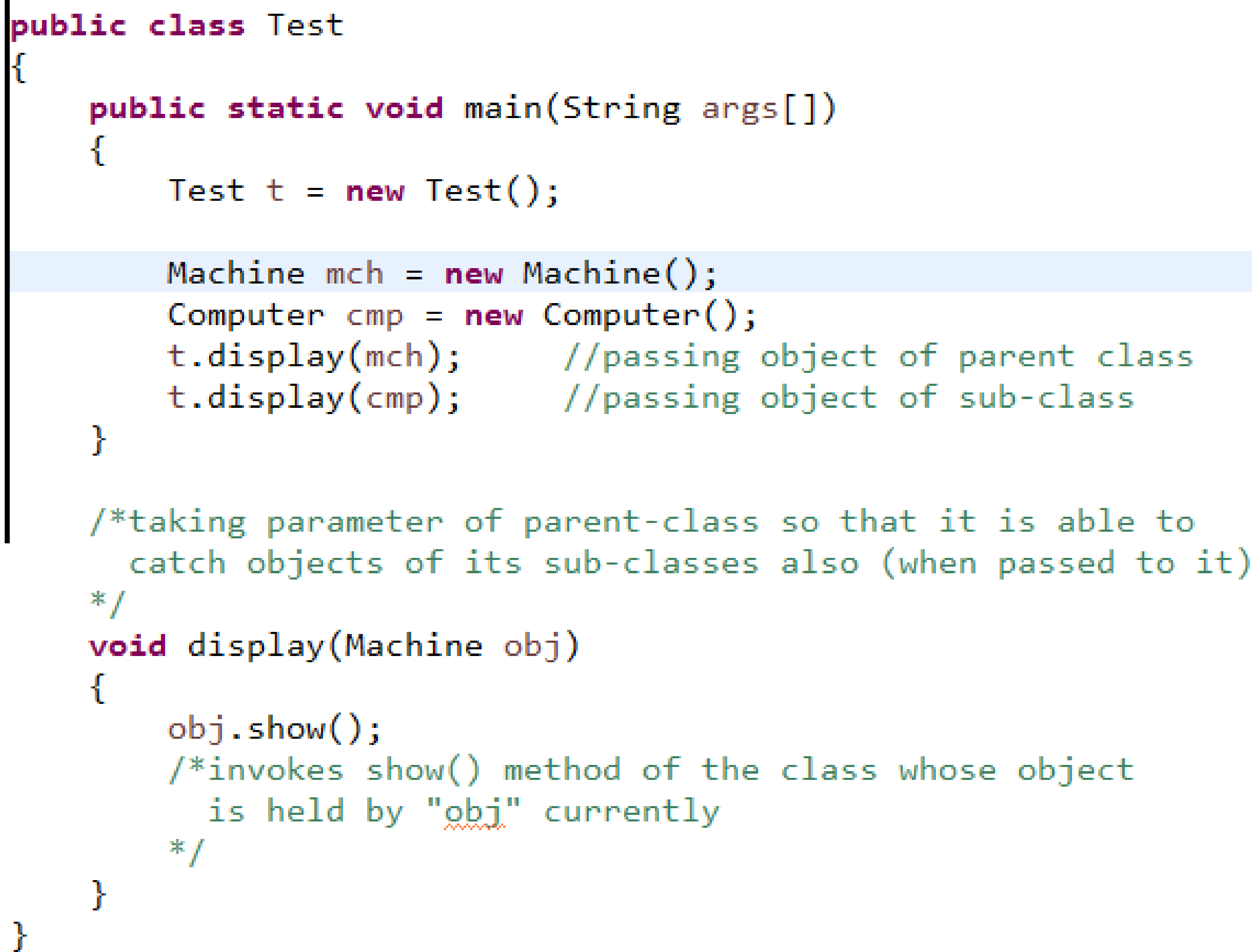
```
class Dispatch
  {
          public static void main(String args[])
          {
                    A a = new A();                    // object of type A
                    B b = new B();                    // object of type B
                    C c = new C();                    // object of type C

                    A r;              // obtain a reference of type A

                    r = a;            // r refers to an A object
                    r.callme();       // calls A's version of callme()

                    r = b;            // r refers to a B object
                    r.callme();       // calls B's version of callme()

                    r = c;            // r refers to a C object
                    r.callme();       // calls C's version of callme()
          }
  }
```



# Example



```
class Machine
{
    void show()
    {
        System.out.println("Machine method");
    }
}

class Computer extends Machine
{
    void show()
    {
        System.out.println("Computer method");
    }
}
```



```
public class Test
{
    public static void main(String args[])
    {
        Test t = new Test();

        Machine mch = new Machine();
        Computer cmp = new Computer();
        t.display(mch);     //passing object of parent class
        t.display(cmp);     //passing object of sub-class
    }

    /*taking parameter of parent-class so that it is able to
      catch objects of its sub-classes also (when passed to it)
    */
    void display(Machine obj)
    {
        obj.show();
        /*invokes show() method of the class whose object
          is held by "obj" currently
        */
    }
}
```

## Points to remember

- **Parent class reference variable can point(or refer) to child class object(or reference), but reverse is not true; i.e. child class reference variable cannot point(or refer) to parent class object (or reference).**

- **Parent class reference can access only those members of child class which where inherited from parent class to child class; i.e. new members of child class cannot be accessed using parent class reference variable.**

- **Parent-object cannot be downcasted to child class, but child-object can be upcasted to parent.**

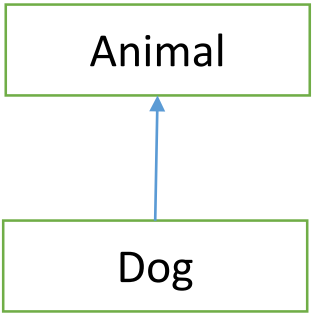
- **If you want to access child class members using parent class reference variable, then down-casting of parent class reference to child class needs to be done.**

- **"instanceof" operator is used to test whether the object is an instance of specified type or not. All child class objects are types of themselves as well as their parent.**

## Upcasting & Downcasting

# Upcasting & Downcasting

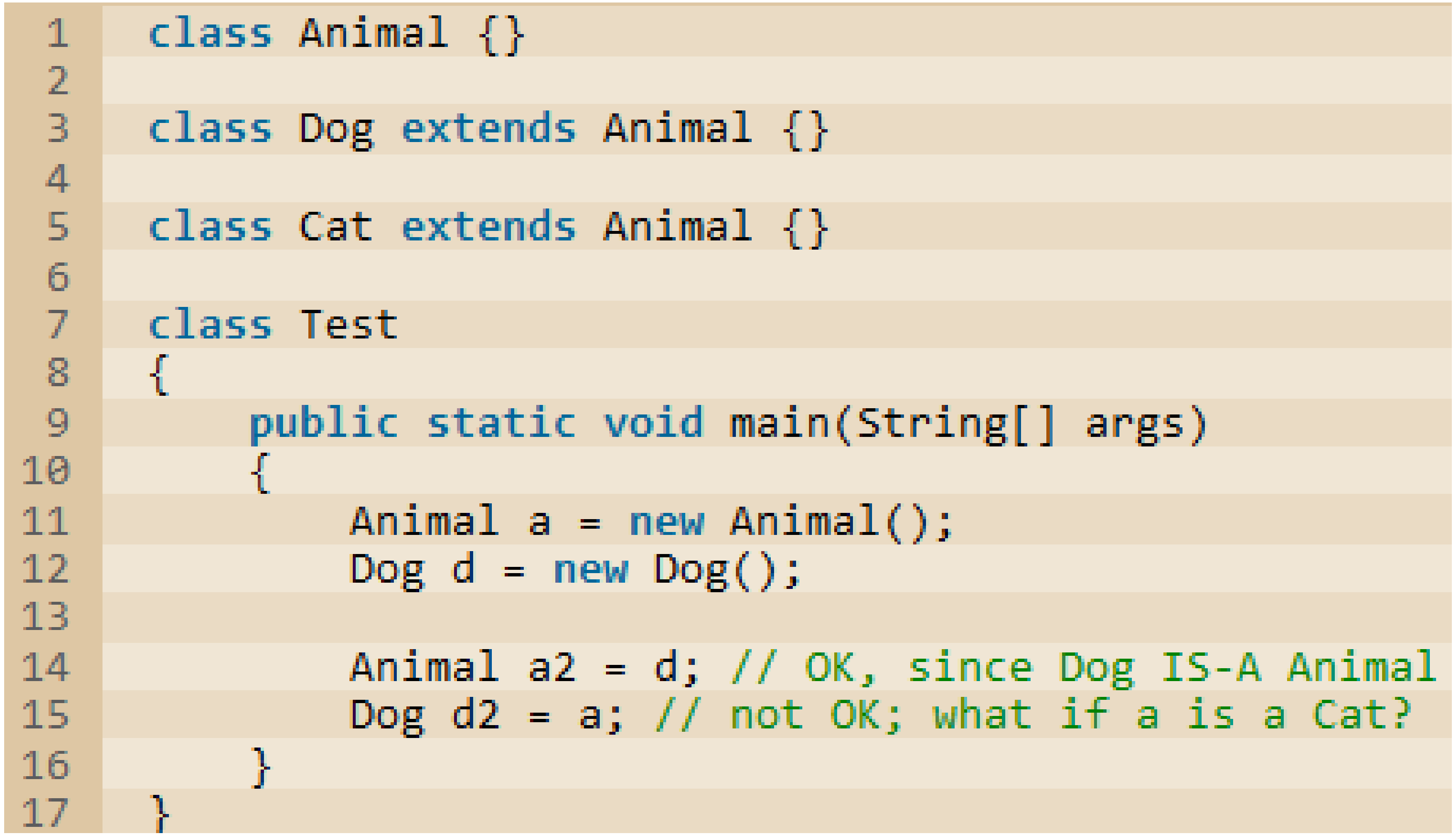
- **Upcasting** is assigning the sub class reference object to the parent class.



```
Dog d = new Dog();
Animal a = d;          //assigning child's object to parent (i.e. upcasting)
Dog dd = (Dog)a;       //assigning parent's reference to child (i.e downcasting)
```

- **Downcasting** is assigning parent class reference object to the sub class. It has to be explicitly done, since downcasting is not automatically done by java.
- Downcasting is legal in some scenarios only where the actual object referred by the parent class is of sub class.

## Point to understand



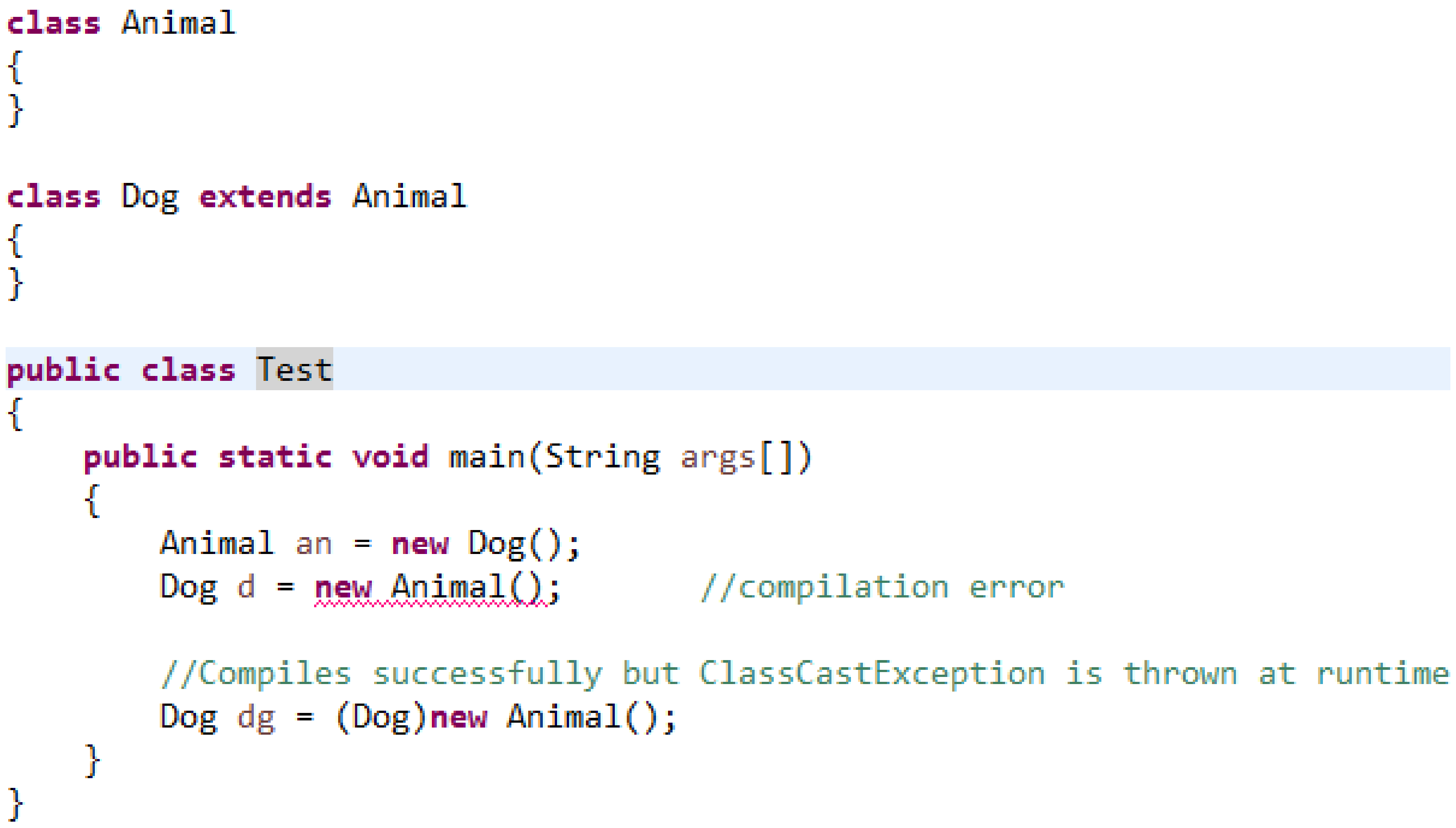
```
class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}

class Test
{
    public static void main(String[] args)
    {
        Animal a = new Animal();
        Dog d = new Dog();

        Animal a2 = d; // OK, since Dog IS-A Animal
        Dog d2 = a; // not OK; what if a is a Cat?
    }
}
```

# Example 1

```java
class Parent
{
    void method1()
    {
        System.out.println("Parent method");
    }
}

class Child extends Parent
{
    void method1()
    {
        System.out.println("Child method");
    }
}
```

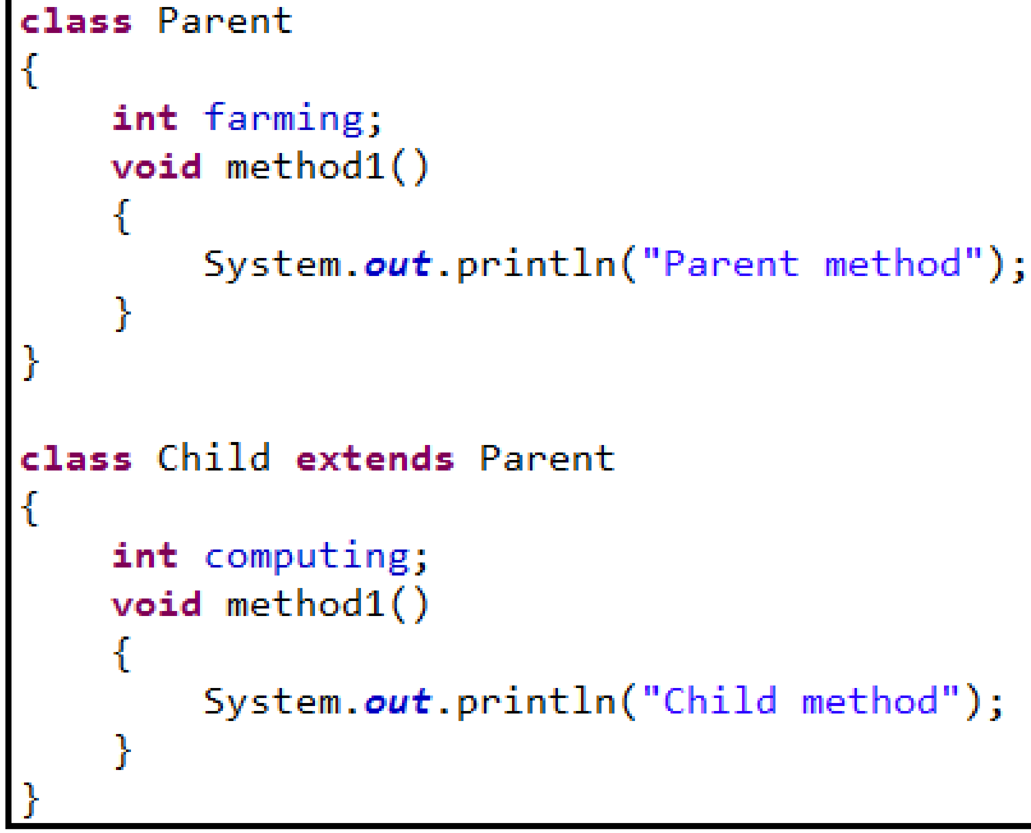
```java
class Temp
{
        public static void main(String args[])
        {
            Parent p = new Parent();
            Child c = new Child();

            p = c;              //auto Upcasting
            p = (Parent)c;   //explicit Upcasting

            c = p;           //Downcasting (not automatic, hence error)
            c = (Child)p; //explicit Downcasting
        }
}
```

# Example 2

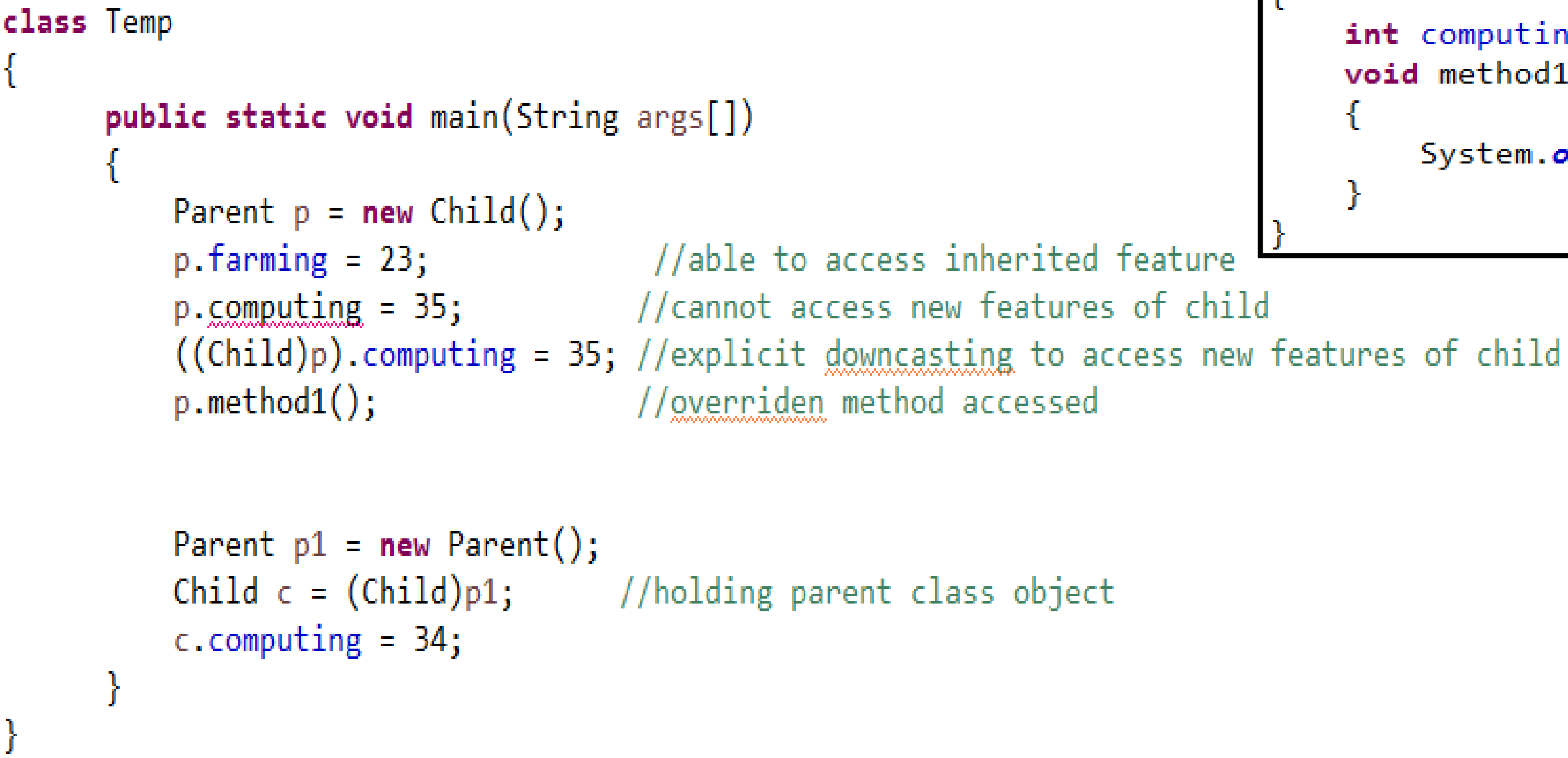


```java
class Animal
{
}

class Dog extends Animal
{
}

public class Test
{
    public static void main(String args[])
    {
        Animal an = new Dog();
        Dog d = new Animal();       //compilation error

        //Compiles successfully but ClassCastException is thrown at runtime
        Dog dg = (Dog)new Animal();
    }
}
```

# Example 3



```
class Parent
{
    int farming;
    void method1()
    {
        System.out.println("Parent method");
    }
}

class Child extends Parent
{
    int computing;
    void method1()
    {
        System.out.println("Child method");
    }
}
```

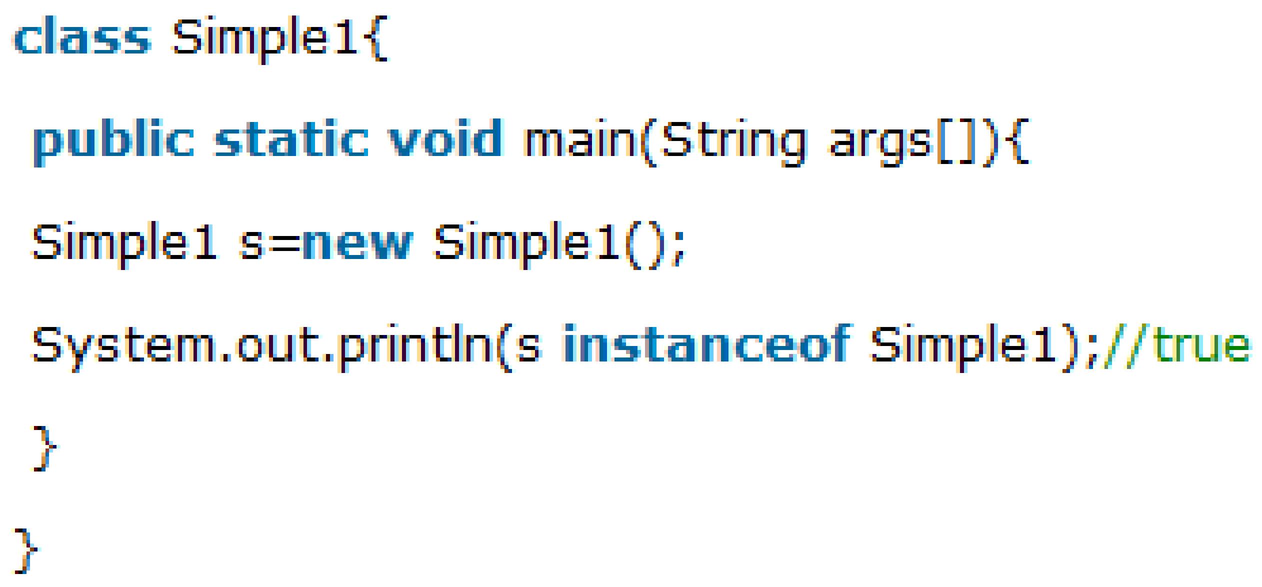


```
class Temp
{
    public static void main(String args[])
    {
        Parent p = new Child();
        p.farming = 23;              //able to access inherited feature
        p.computing = 35;            //cannot access new features of child
        ((Child)p).computing = 35;   //explicit downcasting to access new features of child
        p.method1();                 //overriden method accessed


        Parent p1 = new Parent();
        Child c = (Child)p1;         //holding parent class object
        c.computing = 34;
    }
}
```

## Role of instanceof operator

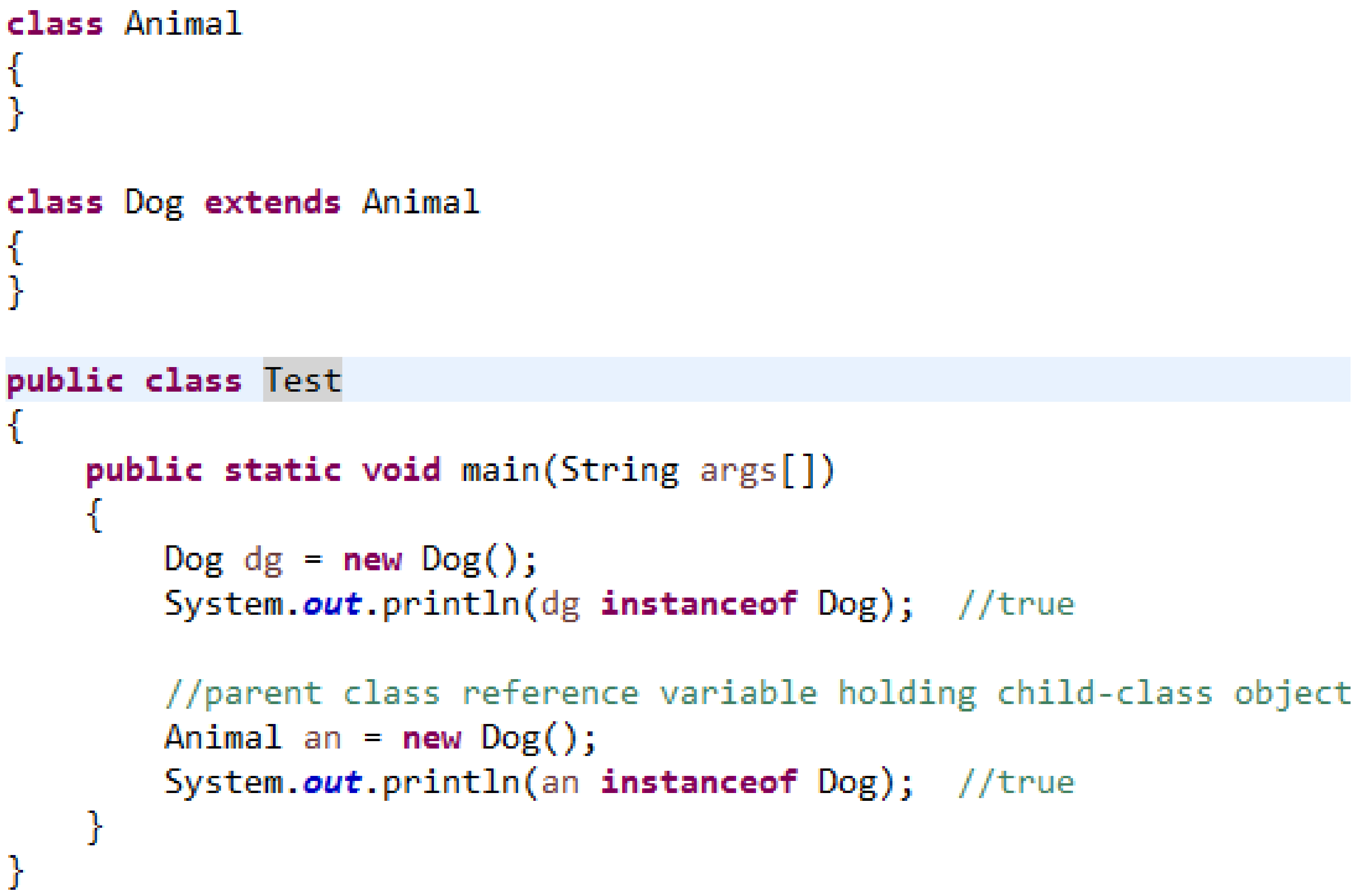
# instanceof operator

- The java **instanceof** operator is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The **instanceof** in java is also known as type ***comparison operator*** because it compares the instance with type. It returns either true or false. If we apply the **instanceof** operator with any variable that has null value, it returns false.



```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}
```

- An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.



```
class Animal
{
}

class Dog extends Animal
{
}

public class Test
{
    public static void main(String args[])
    {
        Dog dg = new Dog();
        System.out.println(dg instanceof Dog);  //true

        //parent class reference variable holding child-class object
        Animal an = new Dog();
        System.out.println(an instanceof Dog);  //true
    }
}
```

# Use of **instanceof** operator

**instanceof** operator is used to check if the reference variable is pointing to a particular type of inherited object or not.

```
class Shape
{
    void showColor()    {
        System.out.println("Red");
    }
}
```

```
class Circle extends Shape
{
    double radius;
    void displayRadius() {
        System.out.println(radius);
    }
}
```

```
class Rectangle extends Shape
{
    int length, width;
    void displayLength() {
        System.out.println(length);
    }
}
```

```
class Temp{
    public static void main(String args[])  {
        Shape obj = new Circle();
        if(obj instanceof Circle) {
            //obj.displayRadius();  //error
            ( (Circle)obj ).displayRadius();
        }
        else if (obj instanceof Rectangle) {
            ( (Rectangle)obj ).displayLength();
        }
    }
}
```

## instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

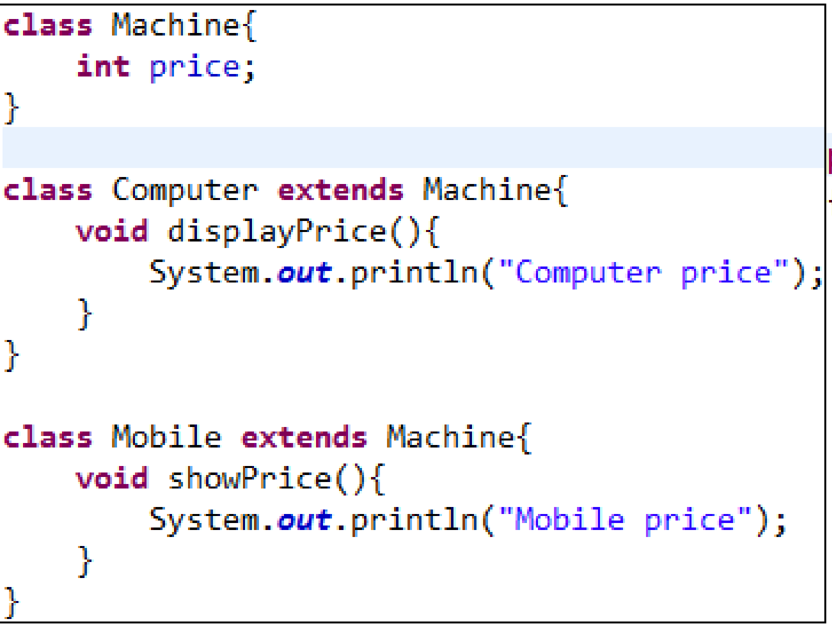
```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 }
}
```

# Possibility of downcasting with instanceof

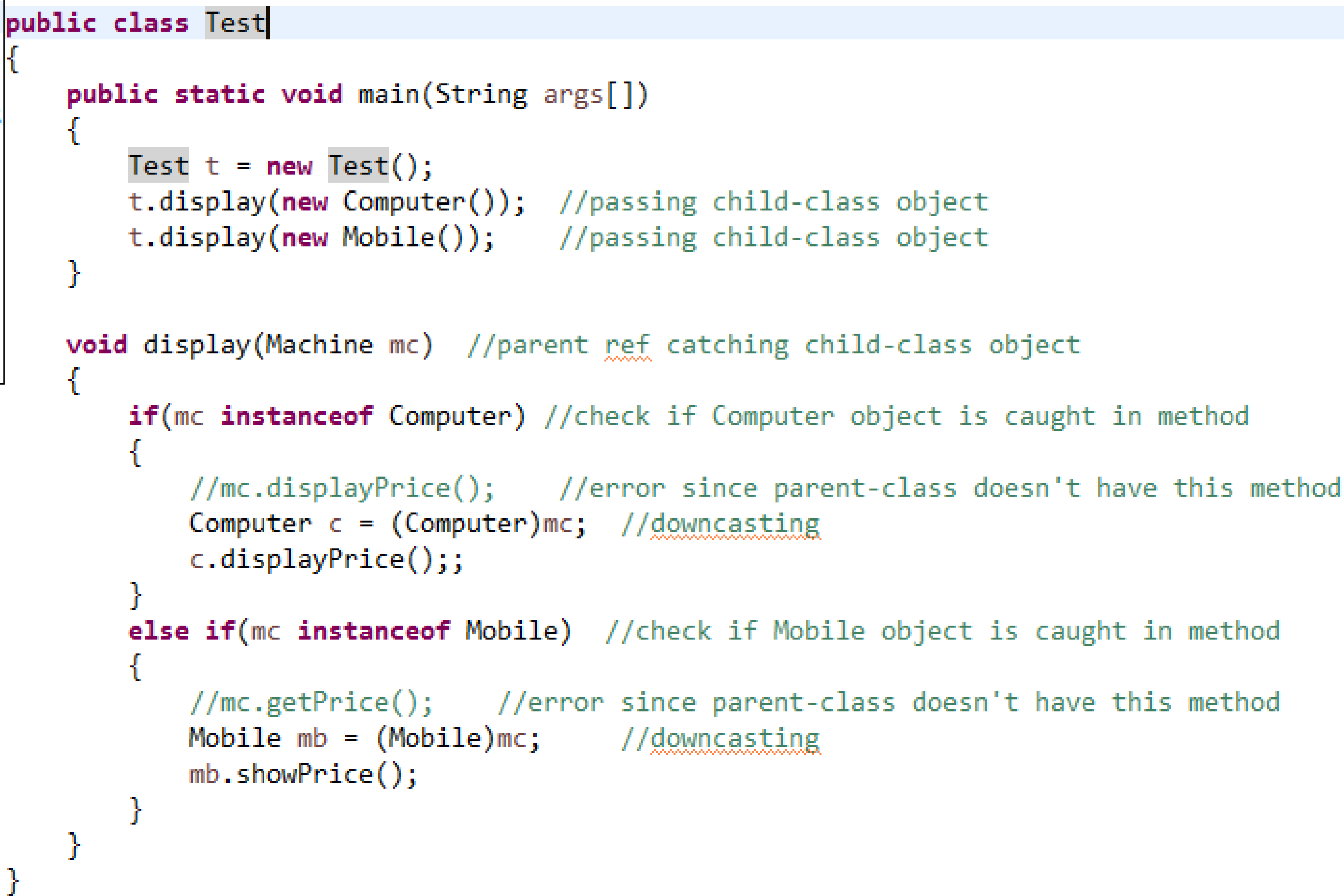
```
class Machine
{
    int price;
}

class Computer extends Machine
{
    void getPrice()
    {
        System.out.println(price);
    }
}
```

```
public class Test
{
    public static void main(String args[])
    {
        Test t = new Test();
        Machine mch = new Machine();
        Computer comp = new Computer();

        t.display(comp);    //passing child-class object
        t.display(mch);     //passing parent-class object
    }

    void display(Machine mc)  //parent ref catching object
    {
        //checking if child-class object is passed to the method
        if(mc instanceof Computer)
        {
            //mc.getPrice();    //error since parent-class doesn't have this method
            Computer c = (Computer)mc;  //downcasting
            c.getPrice();
        }
        else
            System.out.println("downcasting not possible");
    }
}
```

# Downcasting to different classes with instanceof



```java
class Machine{
    int price;
}

class Computer extends Machine{
    void displayPrice(){
        System.out.println("Computer price");
    }
}

class Mobile extends Machine{
    void showPrice(){
        System.out.println("Mobile price");
    }
}
```



```java
public class Test
{
    public static void main(String args[])
    {
        Test t = new Test();
        t.display(new Computer());  //passing child-class object
        t.display(new Mobile());    //passing child-class object
    }

    void display(Machine mc)  //parent ref catching child-class object
    {
        if(mc instanceof Computer) //check if Computer object is caught in method
        {
            //mc.displayPrice();    //error since parent-class doesn't have this method
            Computer c = (Computer)mc;  //downcasting
            c.displayPrice();;
        }
        else if(mc instanceof Mobile)  //check if Mobile object is caught in method
        {
            //mc.getPrice();    //error since parent-class doesn't have this method
            Mobile mb = (Mobile)mc;     //downcasting
            mb.showPrice();
        }
    }
}
```

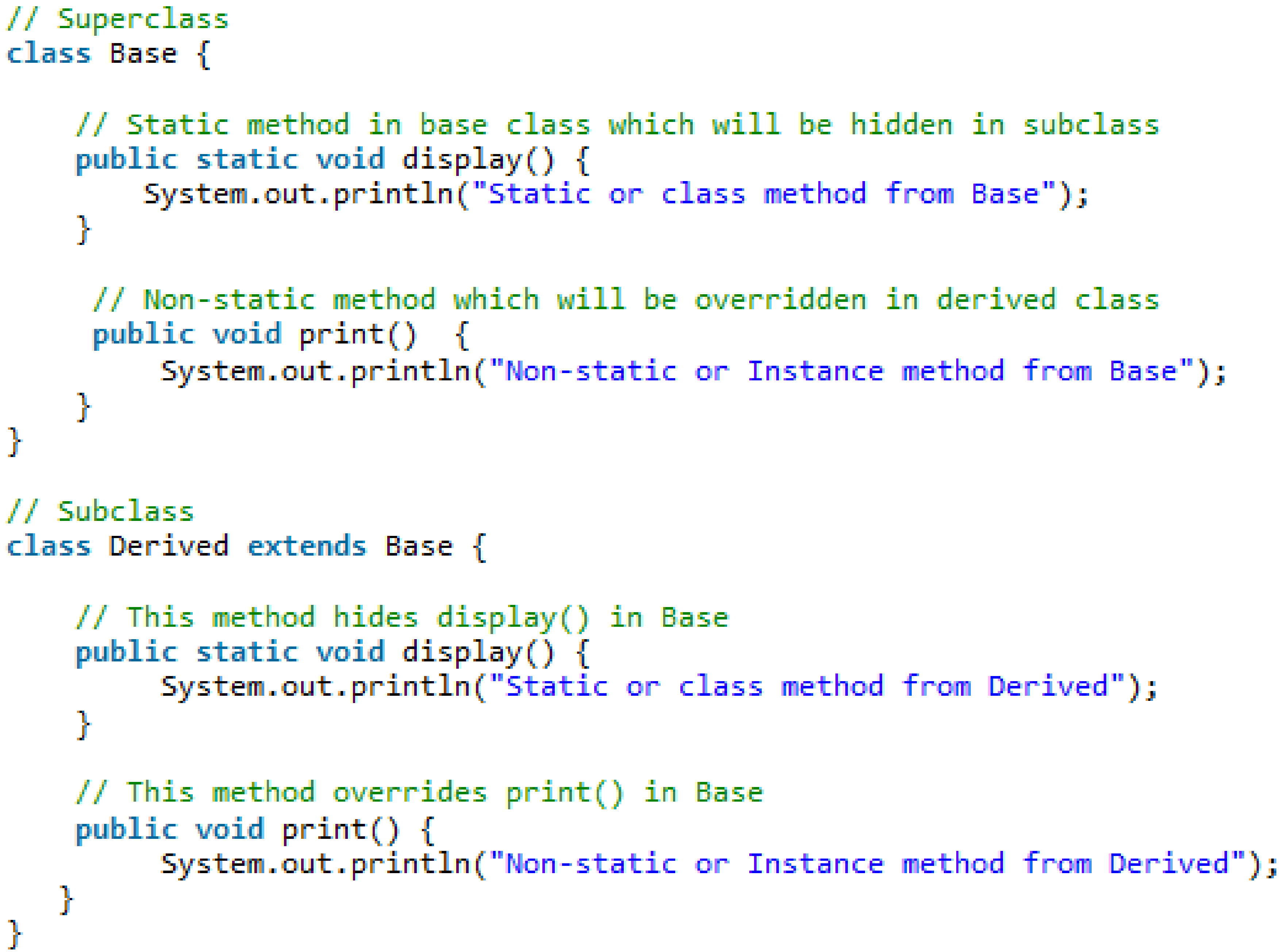
# Exercise

- Write a program to create a class **Employee** which contains attributes: **name** and **date_of_birth** and an abstract method: **public int calculateSalary()**
- Create two subclasses of Employee i.e. **FulltimeEmployee** and **PartTimeEmployee** such that the salary of FulltimeEmployee is printed from the attribute directly whereas the salary of part time employee must be calculated after taking the input: **number_of_hours_per_day**. ParttimeEmployee having age greater than 60 must be given an incentive of 15% along with the salary.
- PartTimeEmployee must have a fixed **per_hour_amount** attribute and FullTimeEmployee must have an instance variable **salary**.

- Store all the objects in an arraylist and implement a method which shows all the employees which are above 60 years in age and having daily income less than 250.

## Role of static methods in Inheritance

## **Example** (static method in inheritance)



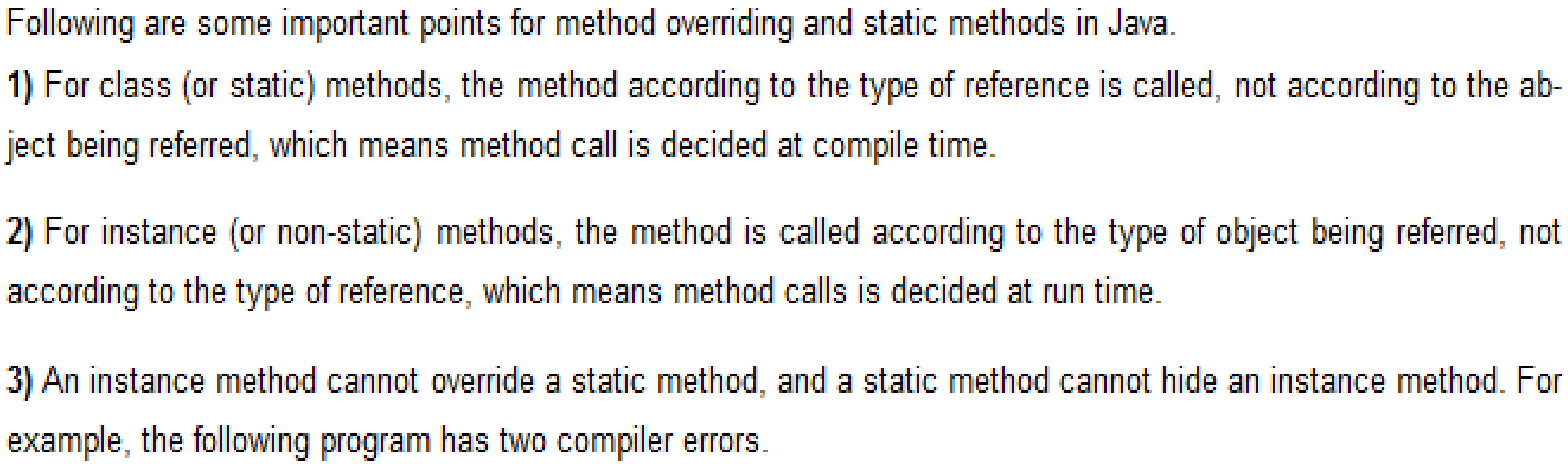
```java
// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // This method hides display() in Base
    public static void display() {
         System.out.println("Static or class method from Derived");
    }

    // This method overrides print() in Base
    public void print() {
         System.out.println("Non-static or Instance method from Derived");
   }
}
```

```
// Driver class
public class Test {
    public static void main(String args[ ])  {
       Base obj1 = new Derived();

       // As per overriding rules this should call to class Derive's static
       // overridden method. Since static method can not be overridden, it
       // calls Base's display()
       obj1.display();

       // Here overriding works and Derive's print() is called
       obj1.print();
    }
}
```

Output:

```
Static or class method from Base
Non-static or Instance method from Derived
```

# Key points



Following are some important points for method overriding and static methods in Java.

**1)** For class (or static) methods, the method according to the type of reference is called, not according to the object being referred, which means method call is decided at compile time.

**2)** For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.

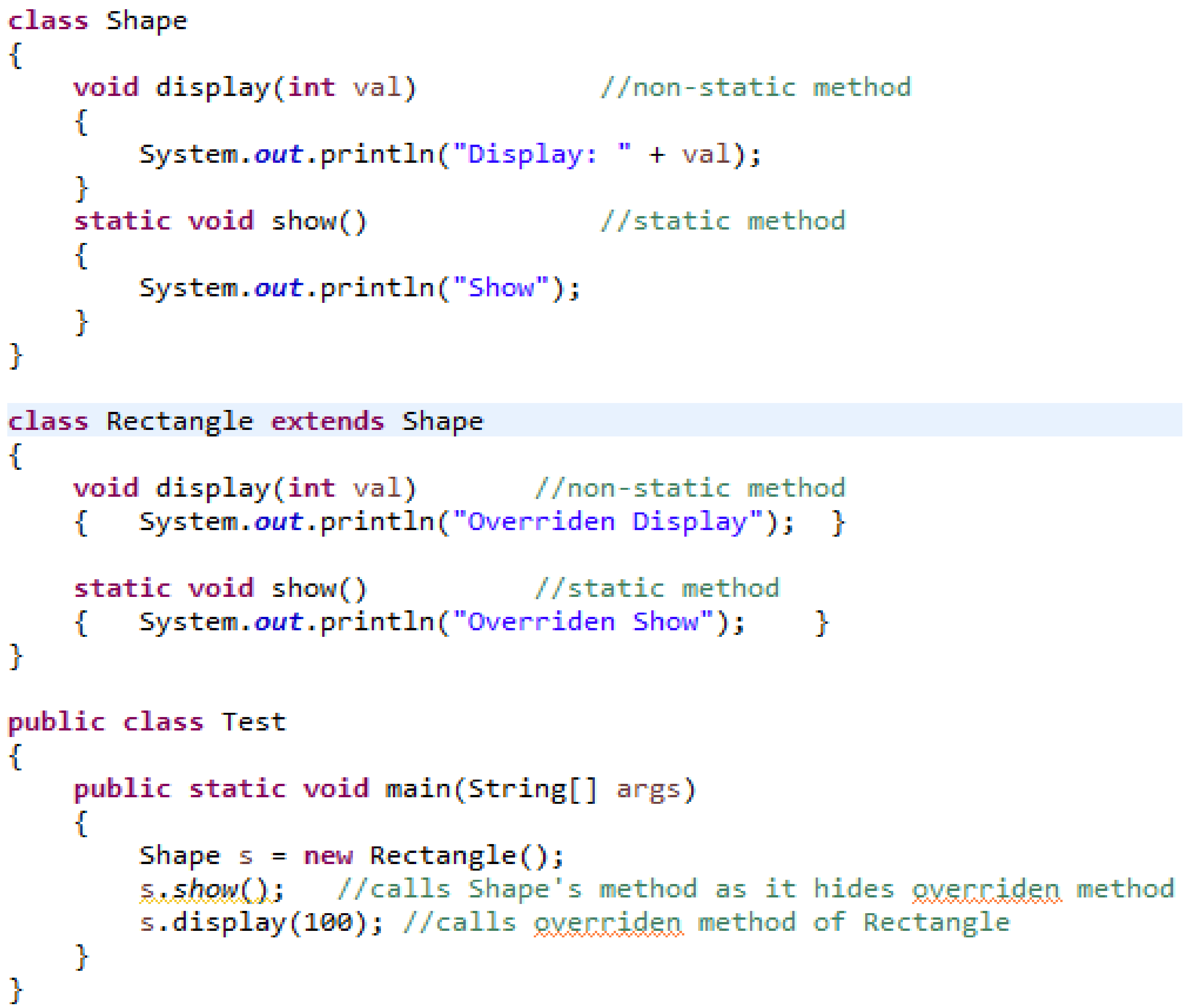
**3)** An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

```
// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // Static is removed here (Causes Compiler Error)
    public void display() {
        System.out.println("Non-static method from Derived");
    }

    // Static is added here (Causes Compiler Error)
    public static void print() {
        System.out.println("Static method from Derived");
   }
}
```

Error

**4)** In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

# Example (accessibility)



```java
class Shape
{
    void display(int val)           //non-static method
    {
        System.out.println("Display: " + val);
    }
    static void show()              //static method
    {
        System.out.println("Show");
    }
}

class Rectangle extends Shape
{
    void display(int val)       //non-static method
    {   System.out.println("Overriden Display");  }

    static void show()          //static method
    {   System.out.println("Overriden Show");    }
}

public class Test
{
    public static void main(String[] args)
    {
        Shape s = new Rectangle();
        s.show();   //calls Shape's method as it hides overriden method
        s.display(100); //calls overriden method of Rectangle
    }
}
```