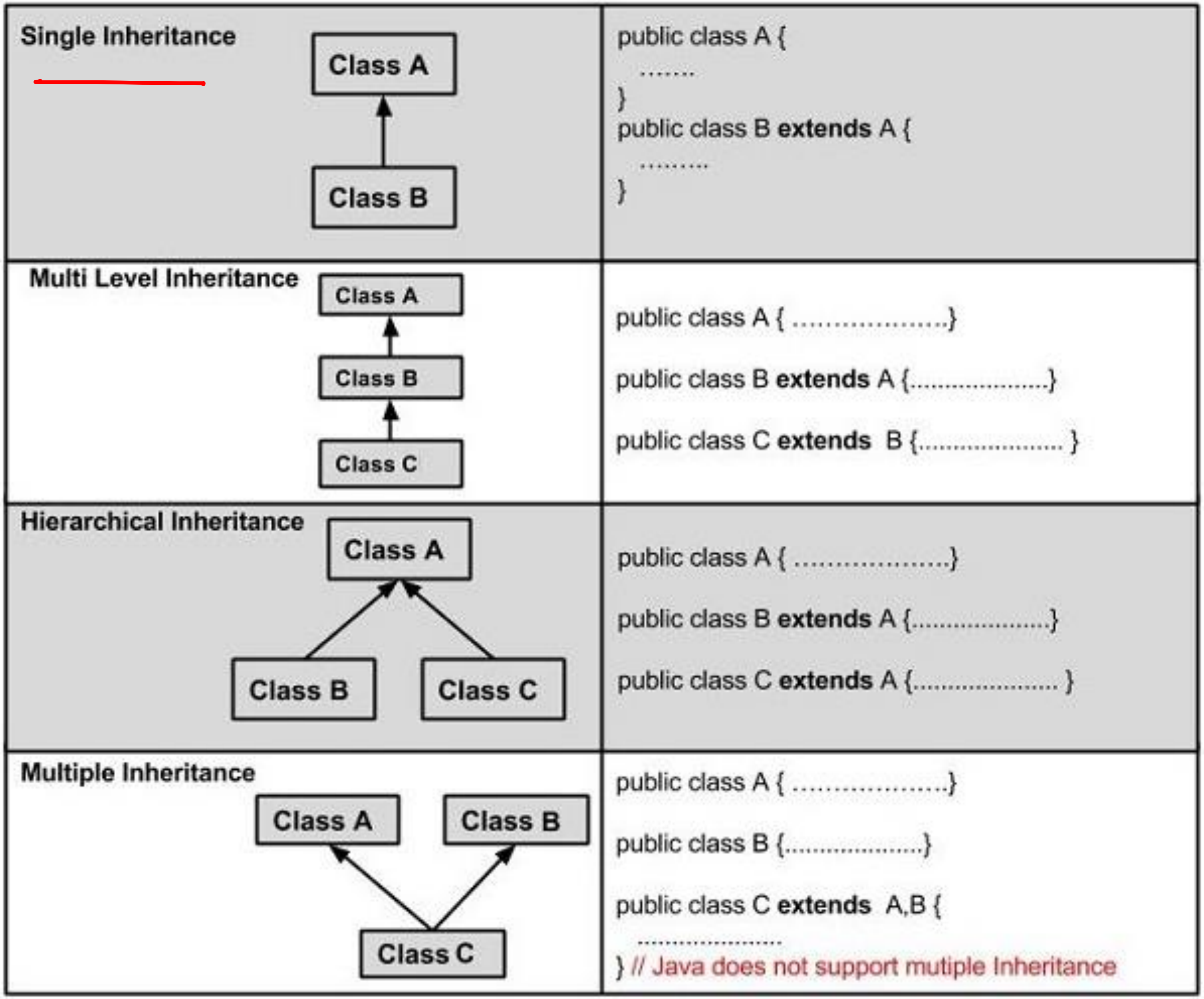
Inheritance

Inheritance

- Same inheritance concept of C++ in Java with some modifications
  - One class inherits the other using ***extends*** keyword
  - The classes involved in inheritance are known as ***superclass*** and ***subclass***
  - ***Multilevel*** inheritance but no ***multiple*** inheritance
  - There is a special way to call the superclass's ***constructor***
  - There is automatic ***dynamic method dispatch***
- Inheritance provides code reusability (code of any class can be used by extending that class)



Single Inheritance	public class A {
	……..
	}
	public class B **extends** A {
	………
	}

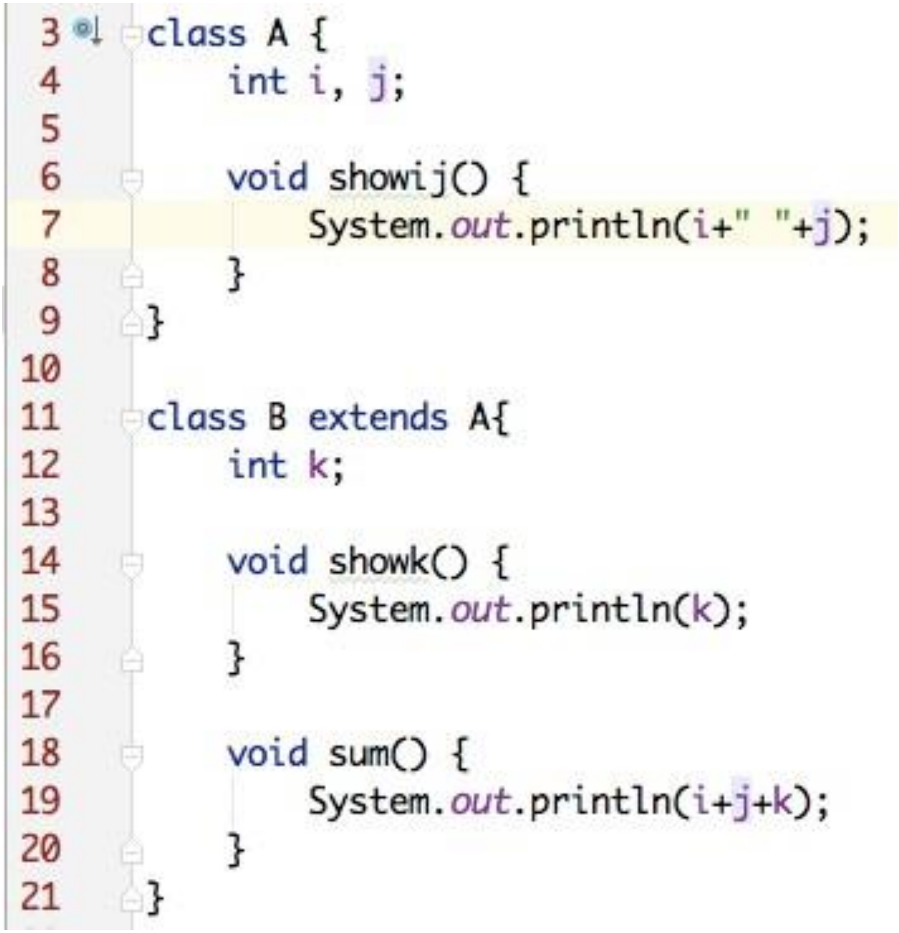
Multi Level Inheritance	public class A { ………………..}
	public class B **extends** A {……………….}
	public class C **extends** B {………………… }

Hierarchical Inheritance	public class A { ………………..}
	public class B **extends** A {……………….}
	public class C **extends** A {………………… }

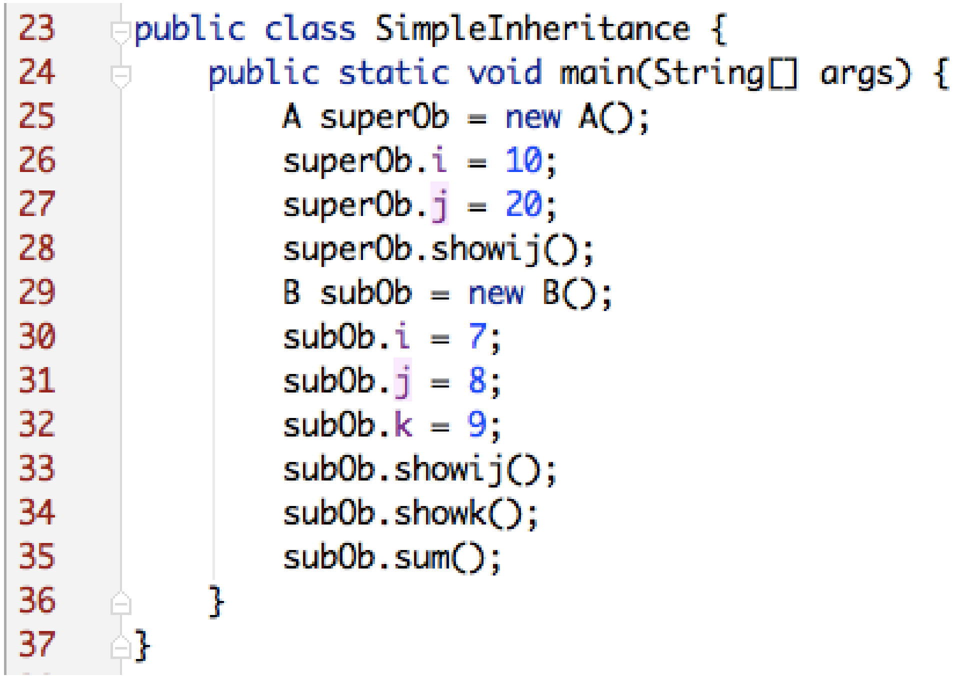
Multiple Inheritance	public class A { ………………..}
	public class B {……………….}
	public class C **extends** A,B {
	…………………
	} // Java does not support mutiple Inheritance

Class A
Class B
Class A
Class B
Class C
Class A
Class B
Class C
Class A
Class B
Class C

Simple Inheritance



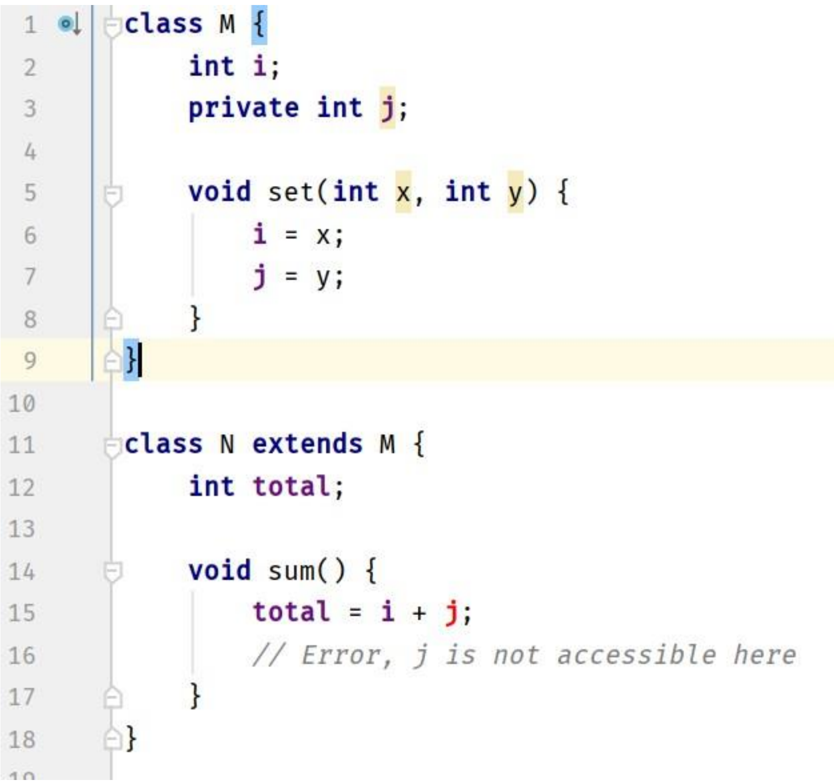
```
class A {
    int i, j;

    void showij() {
        System.out.println(i+" "+j);
    }
}

class B extends A{
    int k;

    void showk() {
        System.out.println(k);
    }

    void sum() {
        System.out.println(i+j+k);
    }
}
```



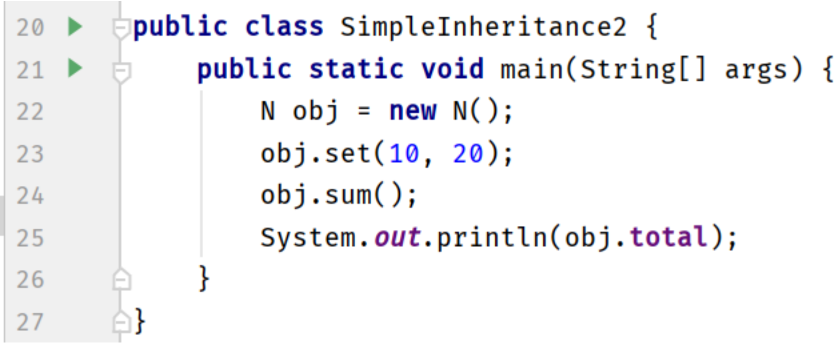
```
public class SimpleInheritance {
    public static void main(String[] args) {
        A superOb = new A();
        superOb.i = 10;
        superOb.j = 20;
        superOb.showij();
        B subOb = new B();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        subOb.showij();
        subOb.showk();
        subOb.sum();
    }
}
```

P : SimpleInheritance.java

Inheritance and Member Access



```java
class M {
    int i;
    private int j;

    void set(int x, int y) {
        i = x;
        j = y;
    }
}

class N extends M {
    int total;

    void sum() {
        total = i + j;
        // Error, j is not accessible here
    }
}
```

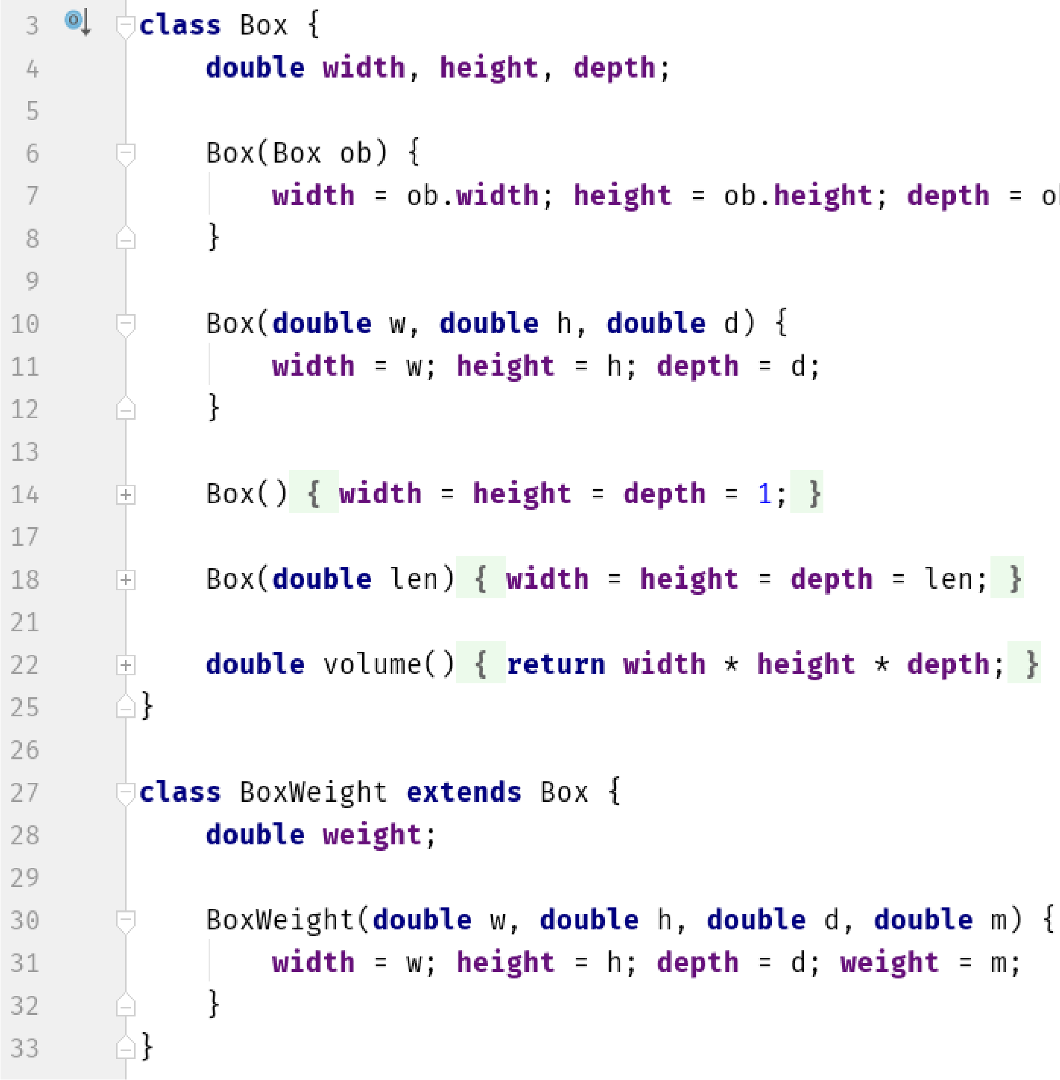


```java
public class SimpleInheritance2 {
    public static void main(String[] args) {
        N obj = new N();
        obj.set(10, 20);
        obj.sum();
        System.out.println(obj.total);
    }
}
```

- A class member that has been declared as private will remain private to its class
- It is not accessible by any code outside its class, including subclasses

P : SimpleInheritance2.java

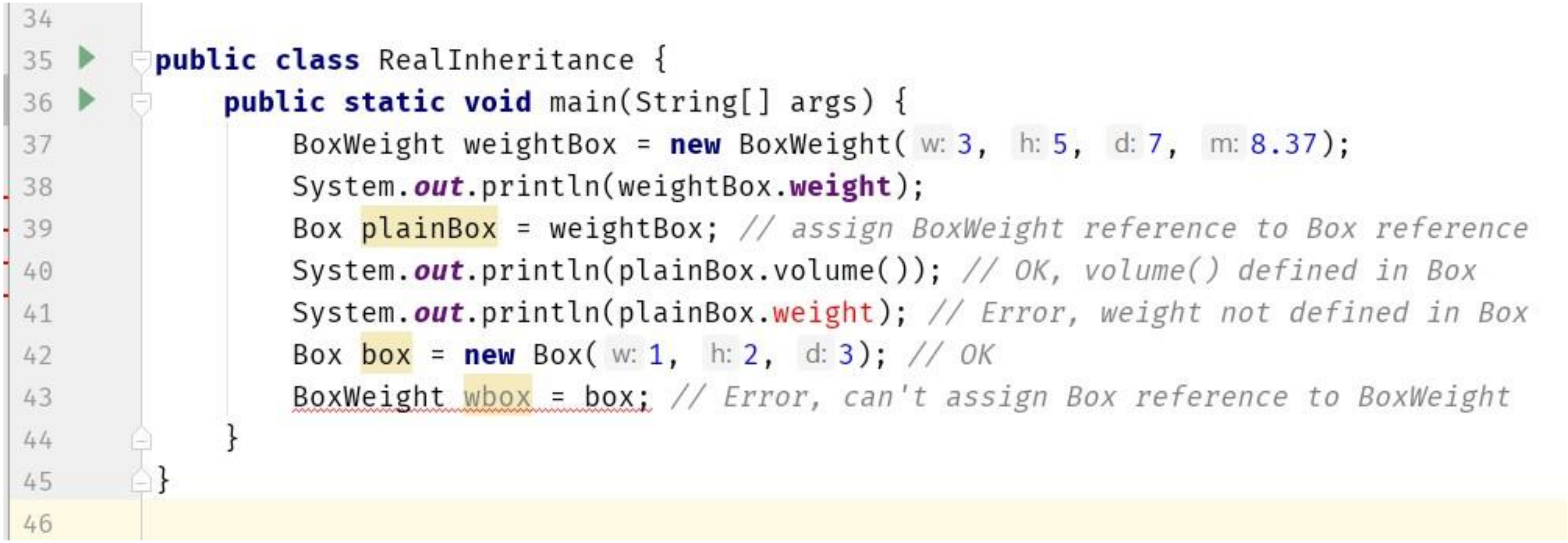
Practical Example



```java
class Box {
    double width, height, depth;

    Box(Box ob) {
        width = ob.width; height = ob.height; depth = ob.depth;
    }

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

    Box() { width = height = depth = 1; }

    Box(double len) { width = height = depth = len; }

    double volume() { return width * height * depth; }
}

class BoxWeight extends Box {
    double weight;

    BoxWeight(double w, double h, double d, double m) {
        width = w; height = h; depth = d; weight = m;
    }
}
```

P : RealInheritance.java

Superclass variable reference to Subclass object



```
public class RealInheritance {
    public static void main(String[] args) {
        BoxWeight weightBox = new BoxWeight( w: 3,  h: 5,  d: 7,  m: 8.37);
        System.out.println(weightBox.weight);
        Box plainBox = weightBox; // assign BoxWeight reference to Box reference
        System.out.println(plainBox.volume()); // OK, volume() defined in Box
        System.out.println(plainBox.weight); // Error, weight not defined in Box
        Box box = new Box( w: 1,  h: 2,  d: 3); // OK
        BoxWeight wbox = box; // Error, can't assign Box reference to BoxWeight
    }
}
```

P : RealInheritance.java

Using super to call Superclass Constructors

There are three cases to use super() in Java.

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

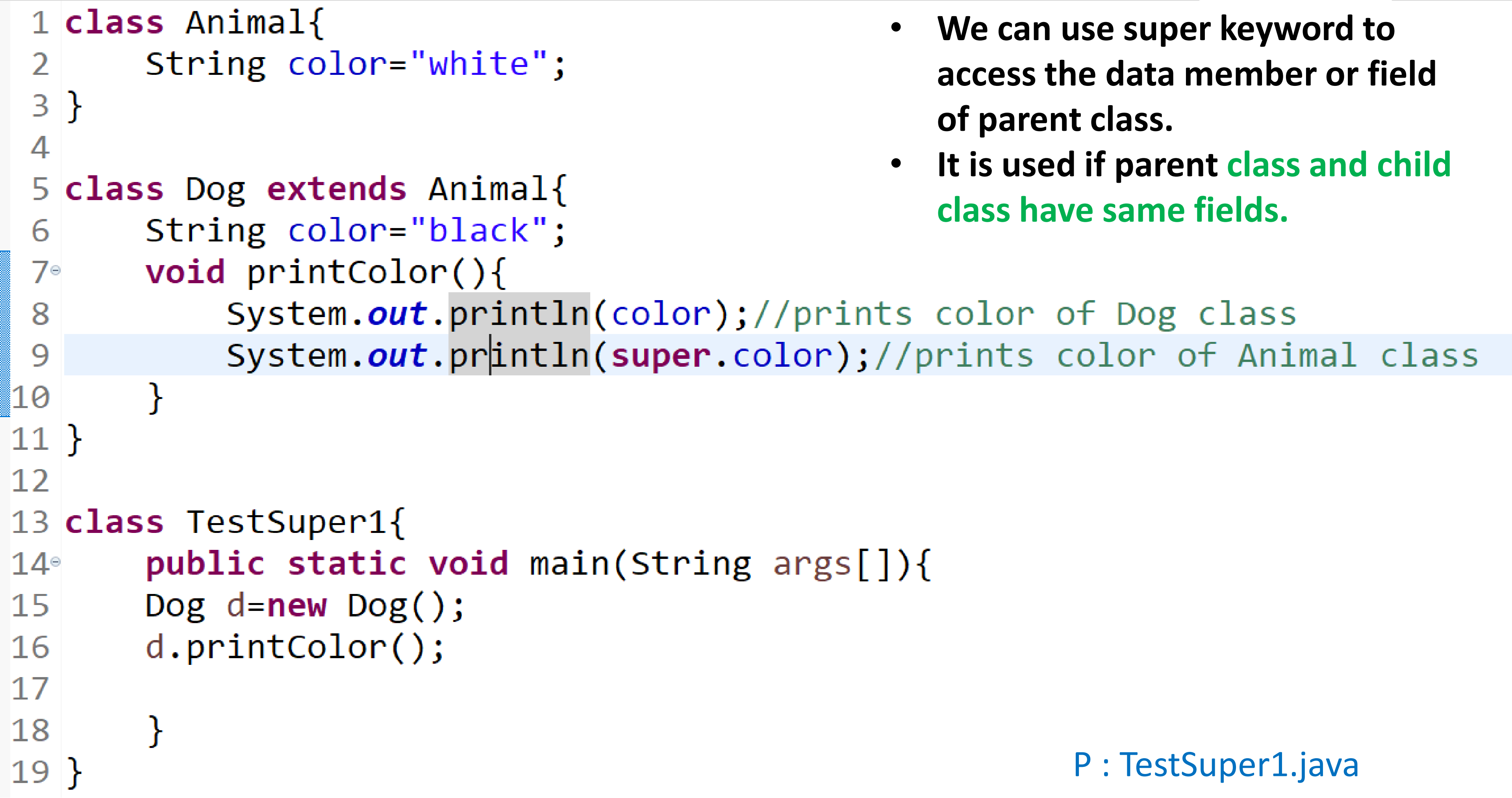
Note:

super( ) must always be the first executable statement inside a subclass’ constructor

P : SuperTest.java

Using super to call Superclass Constructors

Case 1: super can be used to refer to immediate parent class instance variable.



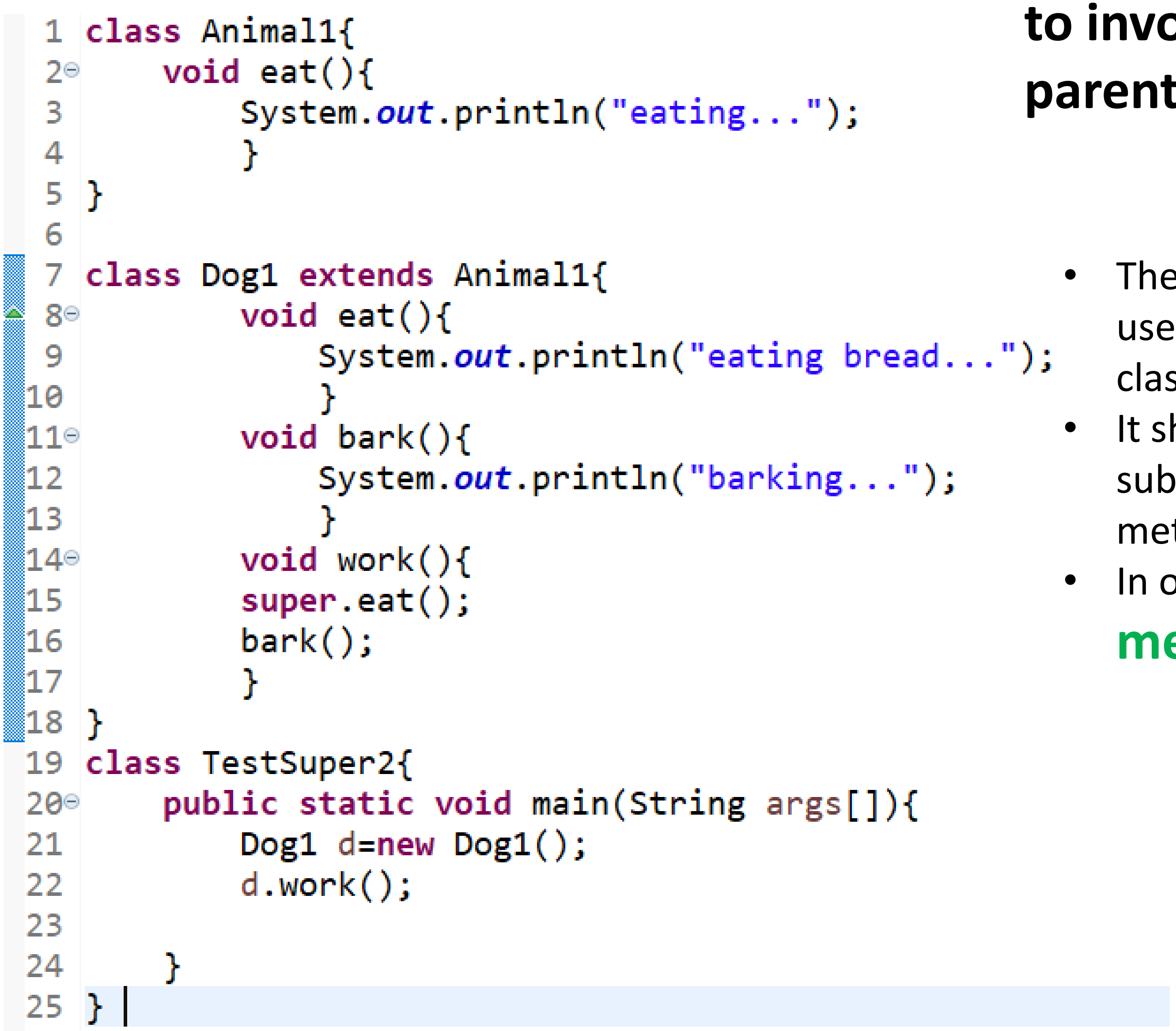
```java
class Animal{
    String color="white";
}

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();

    }
}
```

- **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.**

P : TestSuper1.java

Using super to call Superclass Constructors

Case 2: super can be used to invoke the immediate parent class method.



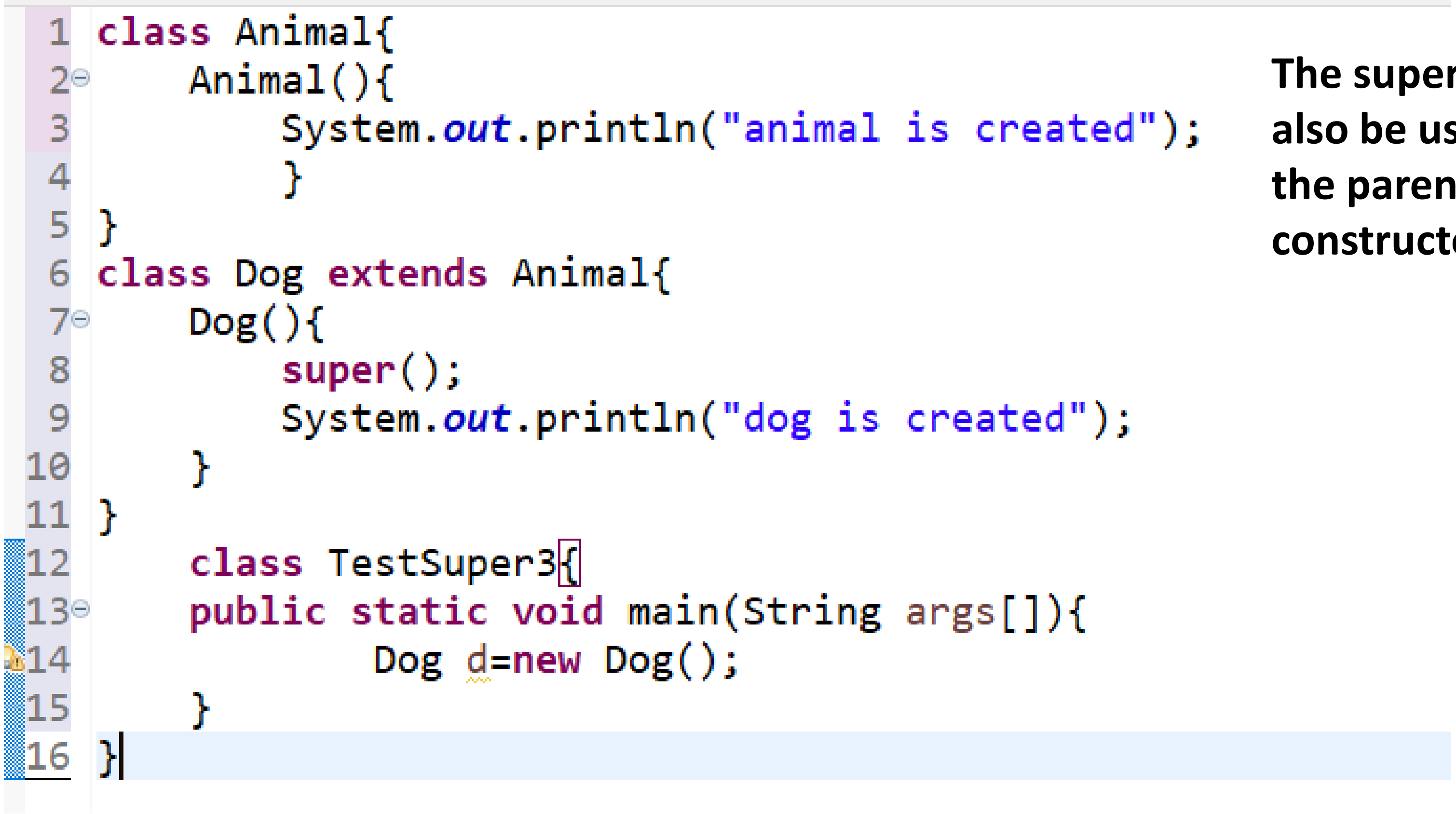
```java
class Animal1{
    void eat(){
        System.out.println("eating...");
        }
}

class Dog1 extends Animal1{
        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[]){
        Dog1 d=new Dog1();
        d.work();

    }
}
```

- The super keyword can also be used to invoke the parent class method.
- It should be used if the subclass contains the same method as the parent class.
- In other words, it is used if the **method is overridden**.

P : TestSuper2.java

Using super to call Superclass Constructors

Case 3: super() can be used to invoke immediate parent class constructor

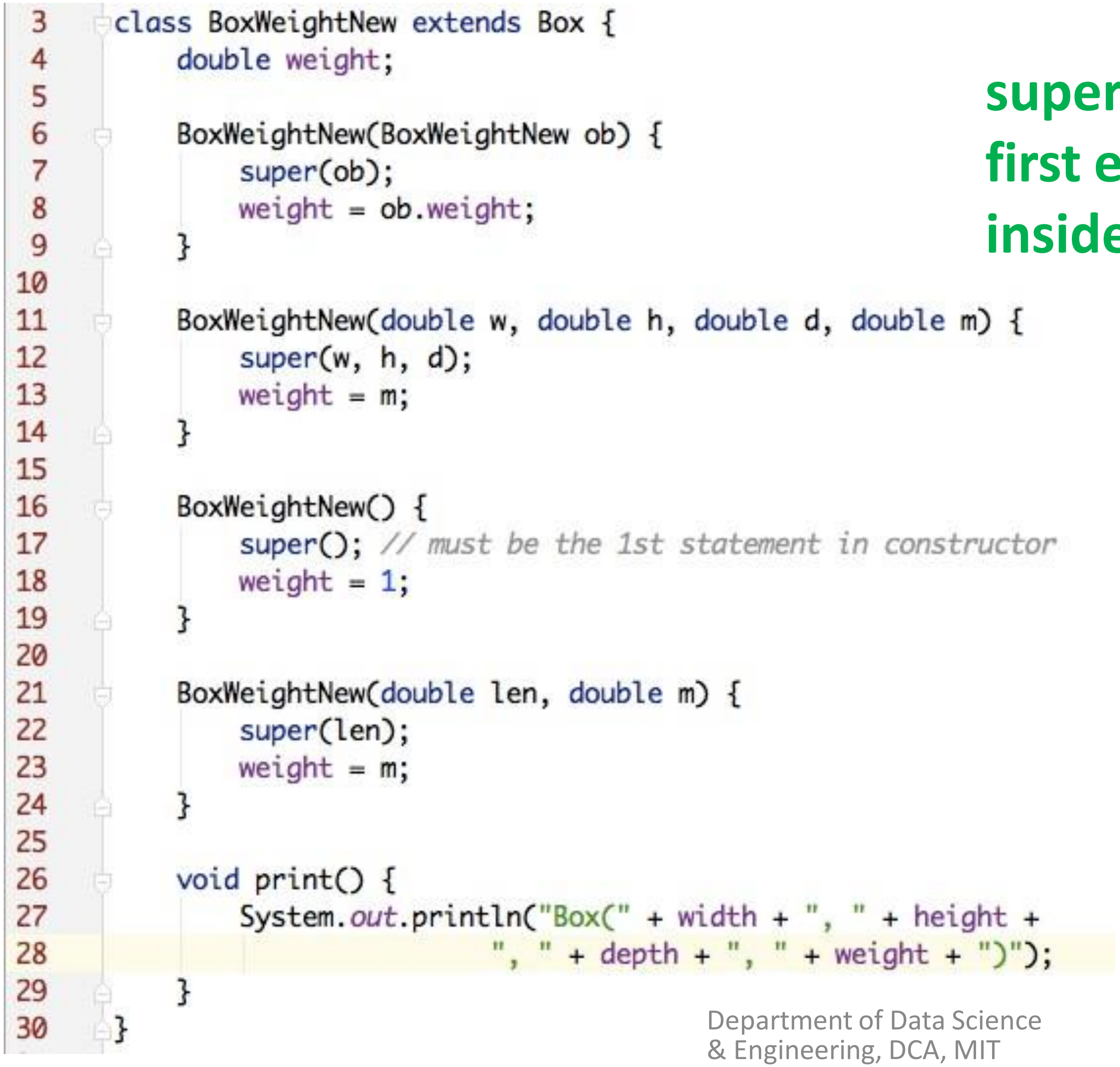


```
class Animal{
    Animal(){
        System.out.println("animal is created");
        }
}
class Dog extends Animal{
    Dog(){
        super();
        System.out.println("dog is created");
    }
}
    class TestSuper3{
    public static void main(String args[]){
            Dog d=new Dog();
    }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

The super keyword can also be used to invoke the parent class constructor

P : TestSuper3.java

Using super to call Superclass Constructors

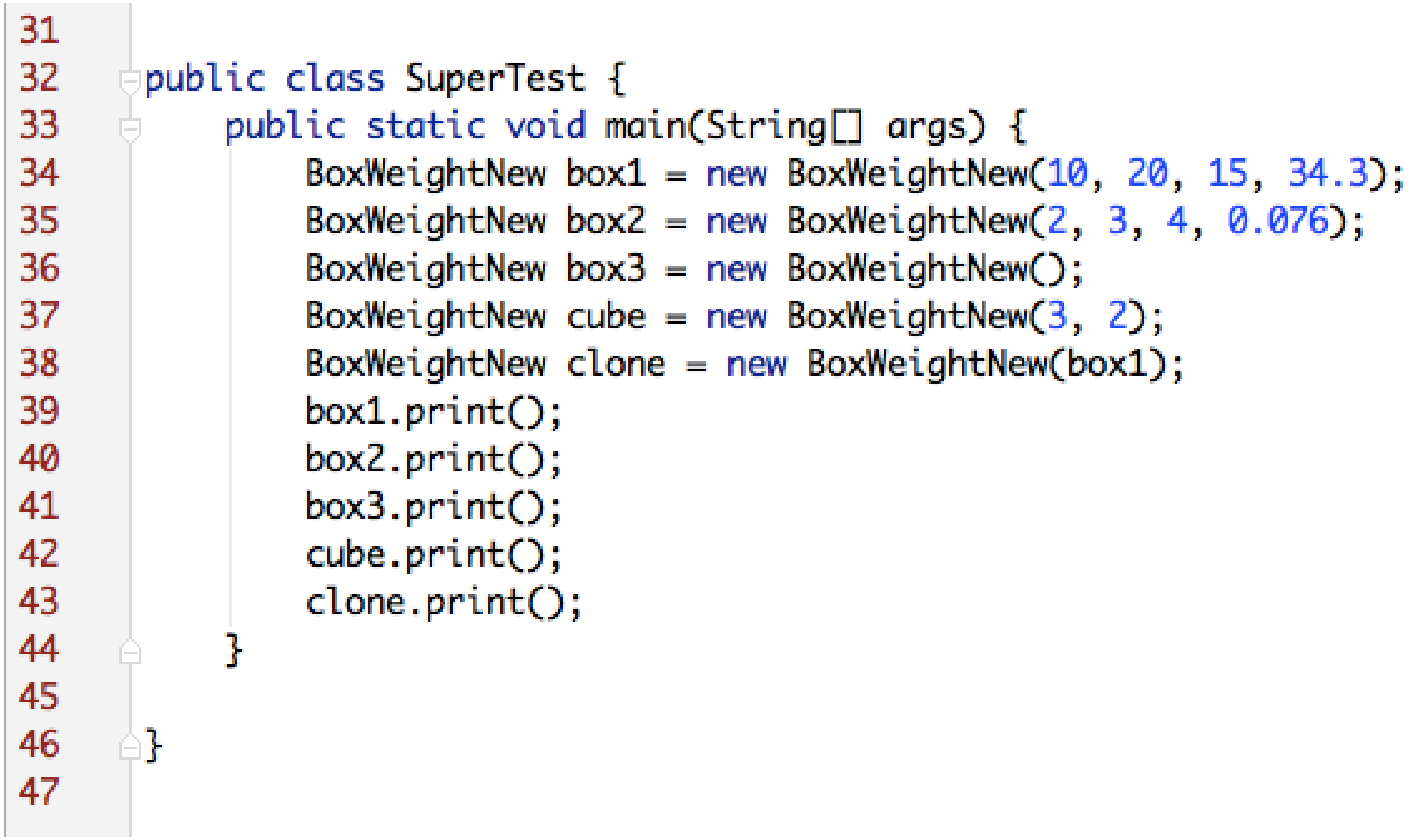


```
class BoxWeightNew extends Box {
    double weight;

    BoxWeightNew(BoxWeightNew ob) {
        super(ob);
        weight = ob.weight;
    }

    BoxWeightNew(double w, double h, double d, double m) {
        super(w, h, d);
        weight = m;
    }

    BoxWeightNew() {
        super(); // must be the 1st statement in constructor
        weight = 1;
    }

    BoxWeightNew(double len, double m) {
        super(len);
        weight = m;
    }

    void print() {
        System.out.println("Box(" + width + ", " + height +
                                ", " + depth + ", " + weight + ")");
    }
}
```

super( ) must always be the first executable statement inside a subclass' constructor

P : SuperTest.java

Using super to call Superclass Constructors

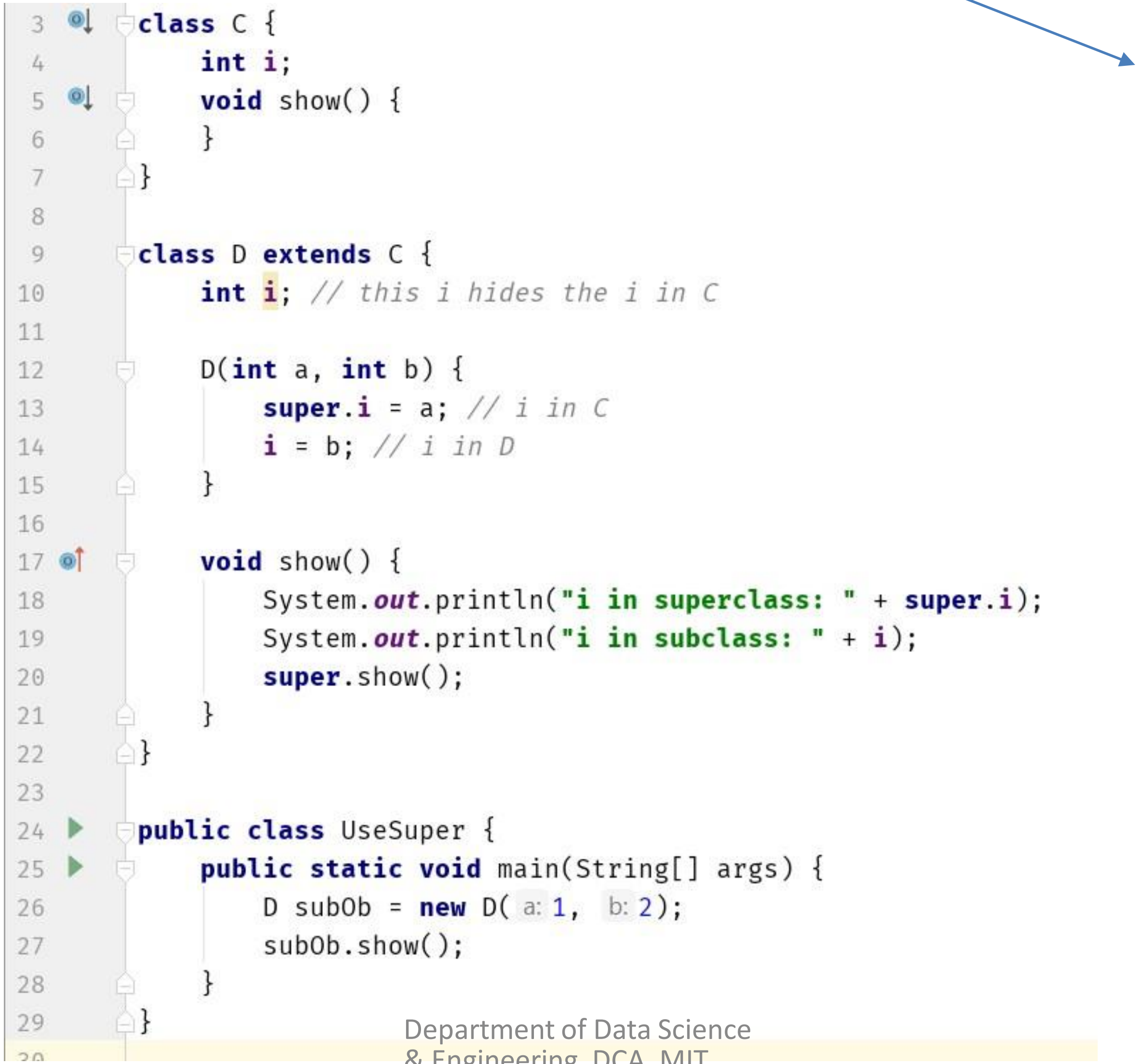


```java
public class SuperTest {
    public static void main(String[] args) {
        BoxWeightNew box1 = new BoxWeightNew(10, 20, 15, 34.3);
        BoxWeightNew box2 = new BoxWeightNew(2, 3, 4, 0.076);
        BoxWeightNew box3 = new BoxWeightNew();
        BoxWeightNew cube = new BoxWeightNew(3, 2);
        BoxWeightNew clone = new BoxWeightNew(box1);
        box1.print();
        box2.print();
        box3.print();
        cube.print();
        clone.print();
    }

}
```

P : SuperTest.java

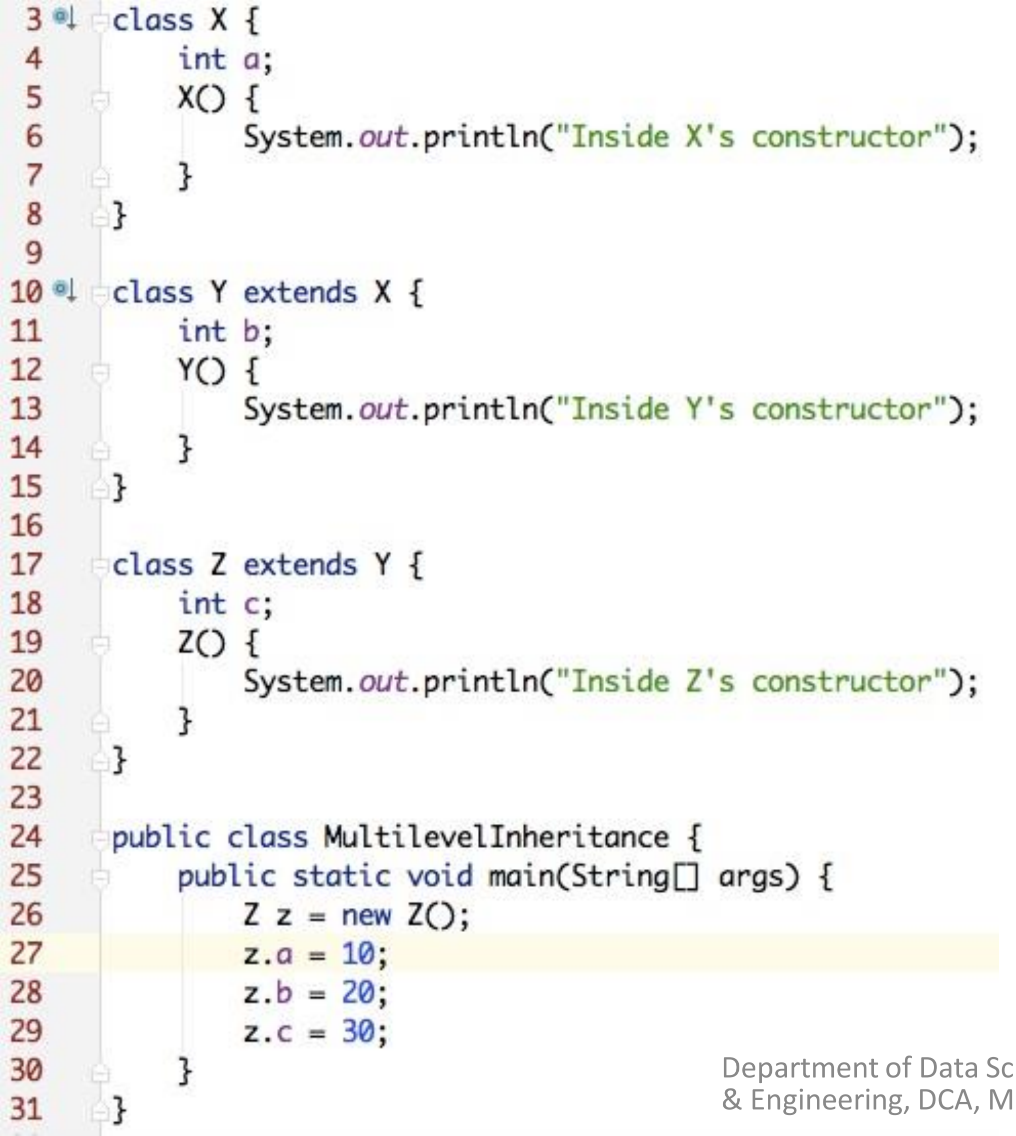
Using super to access Superclass hidden members

In general



```java
class C {
    int i;
    void show() {
    }
}

class D extends C {
    int i; // this i hides the i in C

    D(int a, int b) {
        super.i = a; // i in C
        i = b; // i in D
    }

    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
        super.show();
    }
}

public class UseSuper {
    public static void main(String[] args) {
        D subOb = new D( a: 1,  b: 2);
        subOb.show();
    }
}
```

Multilevel Inheritance

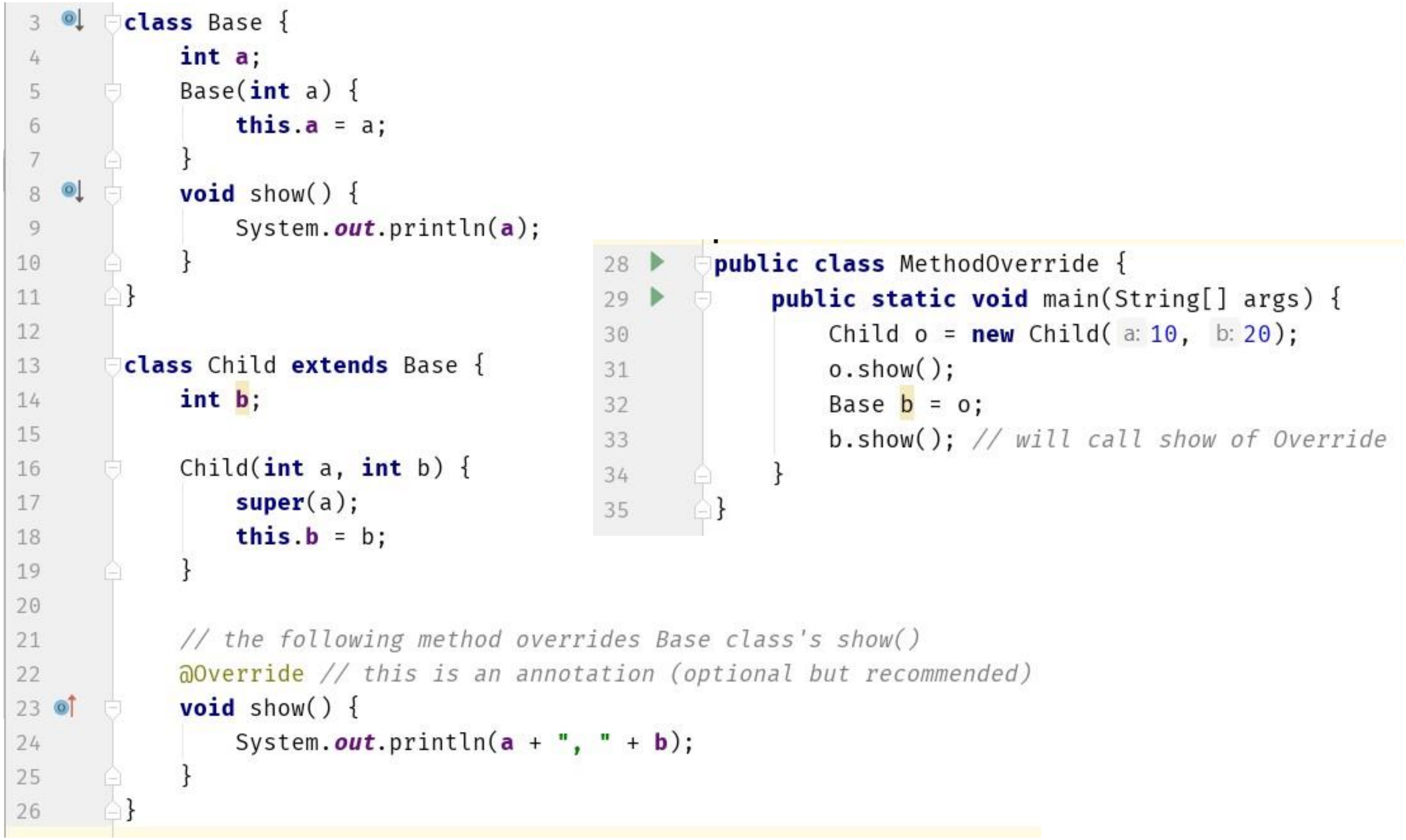


```java
class X {
    int a;
    X() {
        System.out.println("Inside X's constructor");
    }
}

class Y extends X {
    int b;
    Y() {
        System.out.println("Inside Y's constructor");
    }
}

class Z extends Y {
    int c;
    Z() {
        System.out.println("Inside Z's constructor");
    }
}

public class MultilevelInheritance {
    public static void main(String[] args) {
        Z z = new Z();
        z.a = 10;
        z.b = 20;
        z.c = 30;
    }
}
```

```
Inside X's constructor
Inside Y's constructor
Inside Z's constructor
```

P : MultilevelInheritance.java

Method Overriding

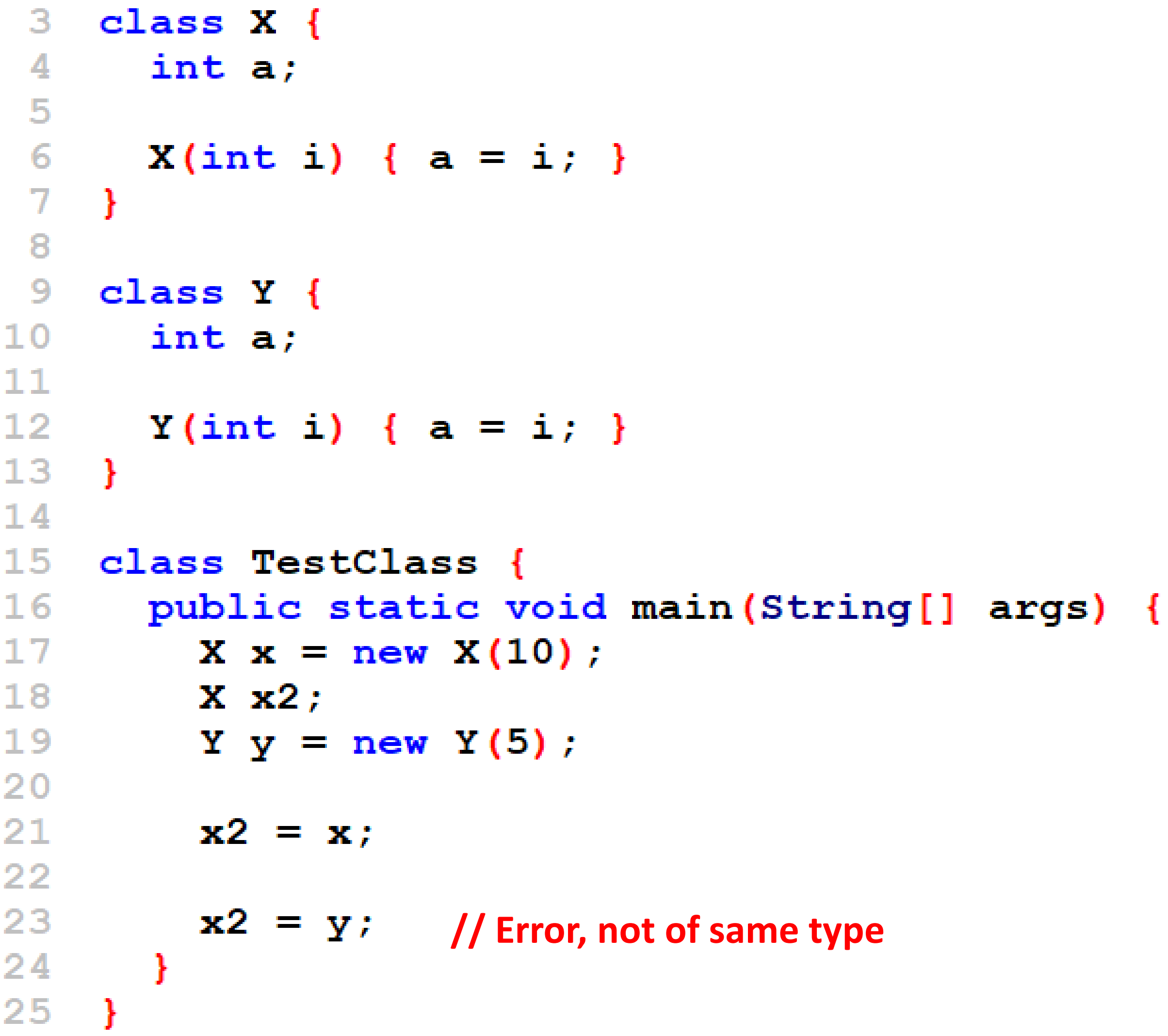


```java
class Base {
    int a;
    Base(int a) {
        this.a = a;
    }
    void show() {
        System.out.println(a);
    }
}

class Child extends Base {
    int b;

    Child(int a, int b) {
        super(a);
        this.b = b;
    }

    // the following method overrides Base class's show()
    @Override // this is an annotation (optional but recommended)
    void show() {
        System.out.println(a + ", " + b);
    }
}
```

```java
public class MethodOverride {
    public static void main(String[] args) {
        Child o = new Child( a: 10,  b: 20);
        o.show();
        Base b = o;
        b.show(); // will call show of Override
    }
}
```

P : MethodOverride.java

Question-1:

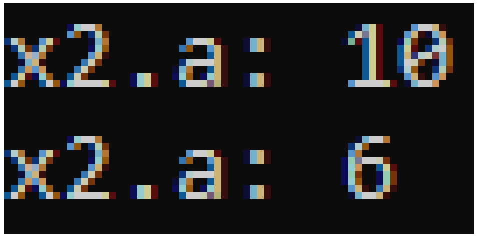


```
class X {
  int a;

  X(int i) { a = i; }
}

class Y {
  int a;

  Y(int i) { a = i; }
}

class TestClass {
  public static void main(String[] args) {
    X x = new X(10);
    X x2;
    Y y = new Y(5);

    x2 = x;

    x2 = y;      // Error, not of same type
  }
}
```

Question-2

```
class X
{
  int a;

  X(int i) { a = i; }
}

class Y extends X
{
  int b;

  Y(int i, int j)
  {
    super(j);
    b = i;
  }
}
```

```
class SupSubRef2 {
  public static void main(String[] args)
  {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // OK, both of same type
    System.out.println("x2.a: " + x2.a);

    x2 = y;
    System.out.println("x2.a: " + x2.a);

    x2.a = 19;
  }
}
```



```
x2.a: 10
x2.a: 6
```

Dynamic Method Dispatch

- Mechanism by which a call to overridden method is resolved at run time, rather than at compile time
  - Basis for run-time polymorphism
- Principle used: A superclass reference variable can refer to a subclass object
  - When a overridden method is called through a superclass reference, Java determines which version of that method to execute at that time
  - Decision is made based on the type of the object being referred to and not on the type the reference variable

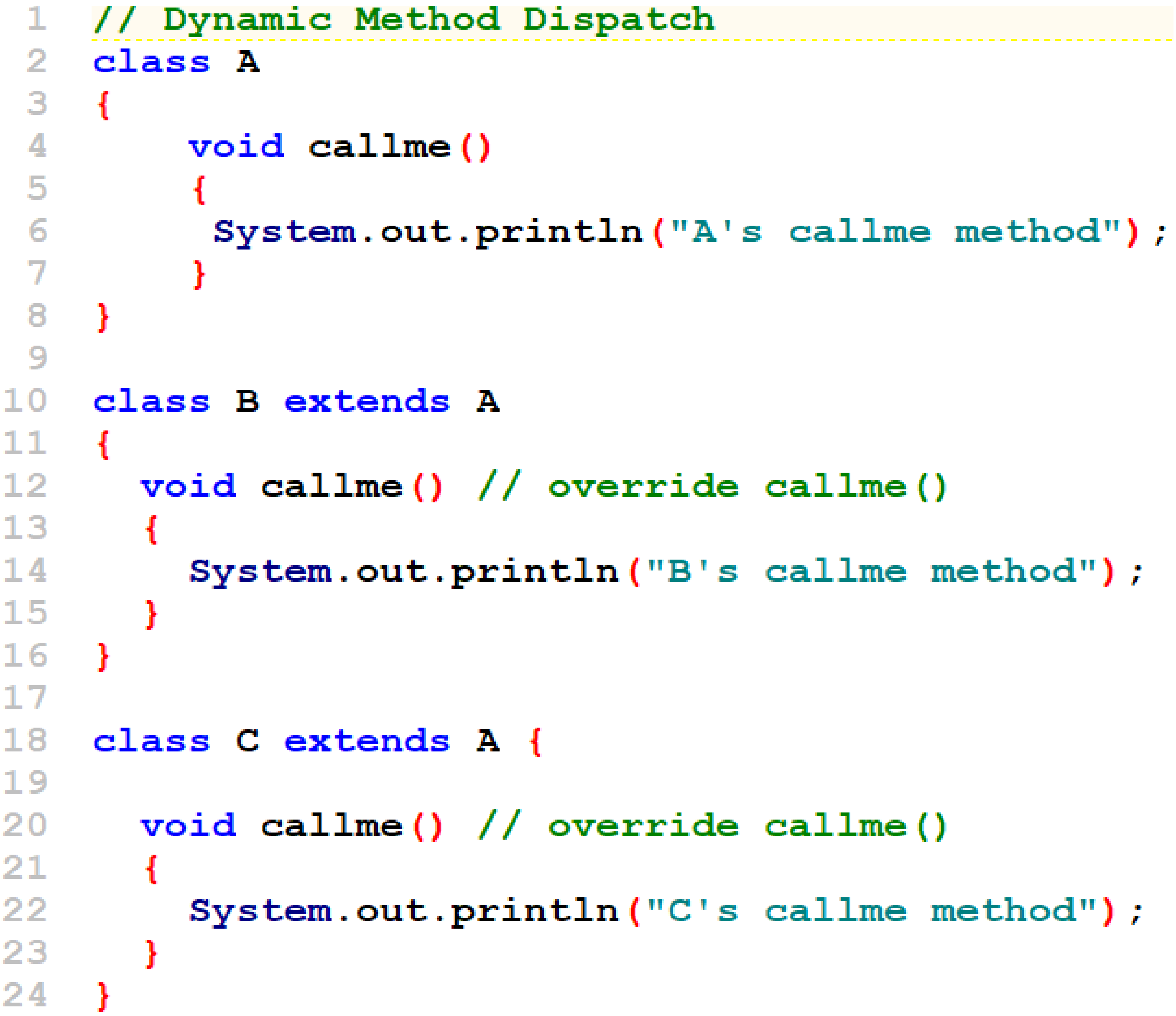
Dynamic Method Dispatch

- Upcasting: Reference variable of superclass referring to object of subclass
  - Example:

    class A { }

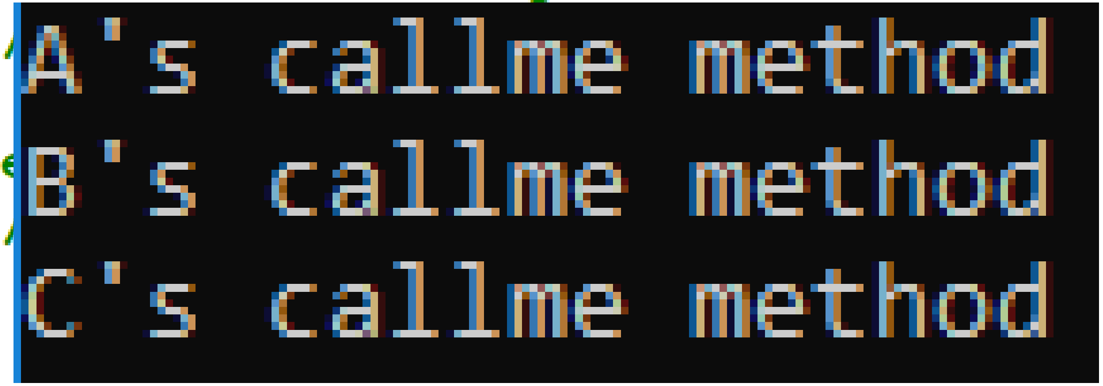
    class B extends A { }

    In main(): A obj = new B(); // upcasting



```java
// Dynamic Method Dispatch
class A
{
    void callme()
    {
     System.out.println("A's callme method");
    }
}

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

class C extends A {

  void callme() // override callme()
  {
    System.out.println("C's callme method");
  }
}
```

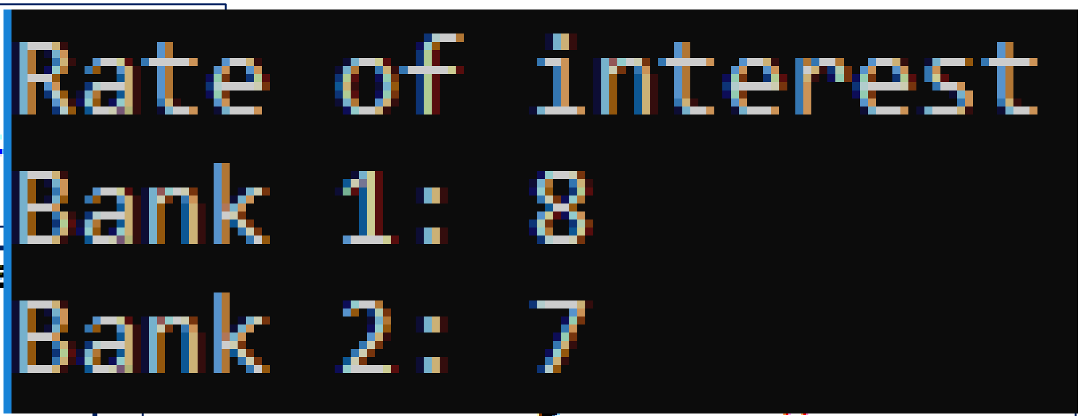
```
class Dispatch
{
  public static void main(String args[])
  {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    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(); //

    r = c; // r re
    r.callme(); //
  }
}
```



```
A's callme method
B's callme method
C's callme method
```

```java
class Bank
{
      int getRate() {ret
}
```

```java
class Bank1 extends Ba
{
      int getRate()
      { return 8; }
}
```

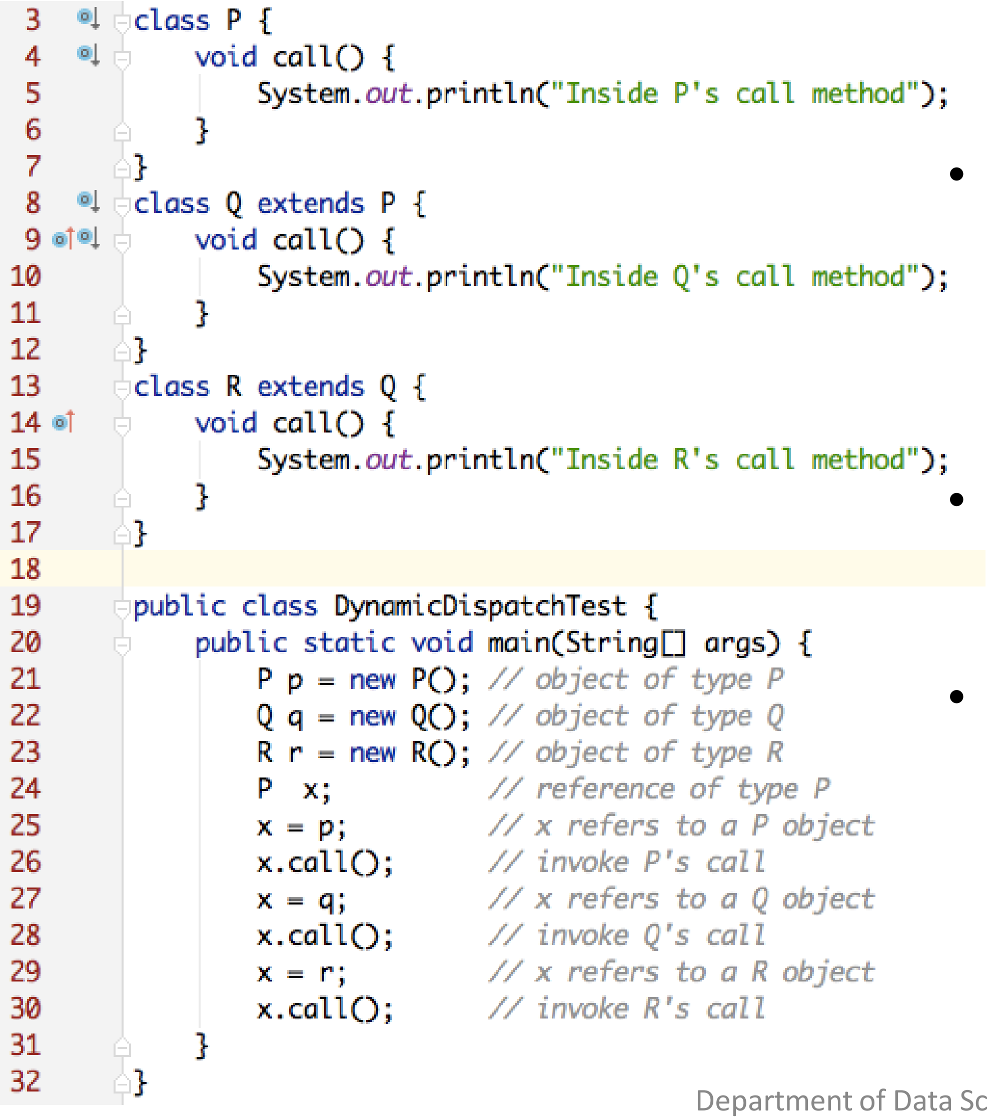


```
Rate of interest
Bank 1: 8
Bank 2: 7
```

```java
      { return 7; }
}
```

```java
class BankTest
{
      public static void main(String args[])
      {
            Bank b;
            b = new Bank1();
            System.out.println ("Rate of interest");
            System.out.println ("Bank 1: "+ b.getRate());
            b = new Bank2();
            System.out.println ("Bank 2: "+ b.getRate());
      }
}
```

Dynamic Method Dispatch



```java
class P {
    void call() {
        System.out.println("Inside P's call method");
    }
}
class Q extends P {
    void call() {
        System.out.println("Inside Q's call method");
    }
}
class R extends Q {
    void call() {
        System.out.println("Inside R's call method");
    }
}

public class DynamicDispatchTest {
    public static void main(String[] args) {
        P p = new P(); // object of type P
        Q q = new Q(); // object of type Q
        R r = new R(); // object of type R
        P  x;          // reference of type P
        x = p;         // x refers to a P object
        x.call();      // invoke P's call
        x = q;         // x refers to a Q object
        x.call();      // invoke Q's call
        x = r;         // x refers to a R object
        x.call();      // invoke R's call
    }
}
```

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

- **DMD is a way Java implement run time polymorphism.**

- **When an overridden method is called with super class reference. Java creates different versions of an overridden method.**

P : DynamicDispatchTest.java
FindAreas.java

Use of overridden methods

- Polymorphism (one interface, multiple methods)
  - Allows a general class to specify methods that will be common to all its subclasses
  - Allows subclasses to define specific implementations of some or all these methods
- 3 methods to implement polymorphism
  - Method overloading
  - Method overriding
  - Interfaces

Abstract Class

- Abstract class is a class that cannot be instantiated
- Superclass declares the structure of a given abstraction without providing a complete implementation of every method
  - Defines a generalized form that will be shared by all its subclasses, leaving it to each subclass to fill in the details
- To specify that certain methods must be overridden by subclasses, specify *abstract* type modifier with superclass
  - No implementation in superclass
  - Subclass' responsibility to implement them
  - Syntax of declaring an abstract method:

abstract type_name (parameter-list);

Abstract Class

- Any class that contains one or more abstract methods must be declared abstract
  - use ***abstract*** keyword before the keyword class
  - Cannot create objects of abstract class because such objects are of no use
  - Cannot declare abstract constructors
  - Cannot have abstract static methods
- Any subclass of an abstract class must either implement all abstract methods specified in the superclass or be itself an abstract class

Abstract Class

Note:

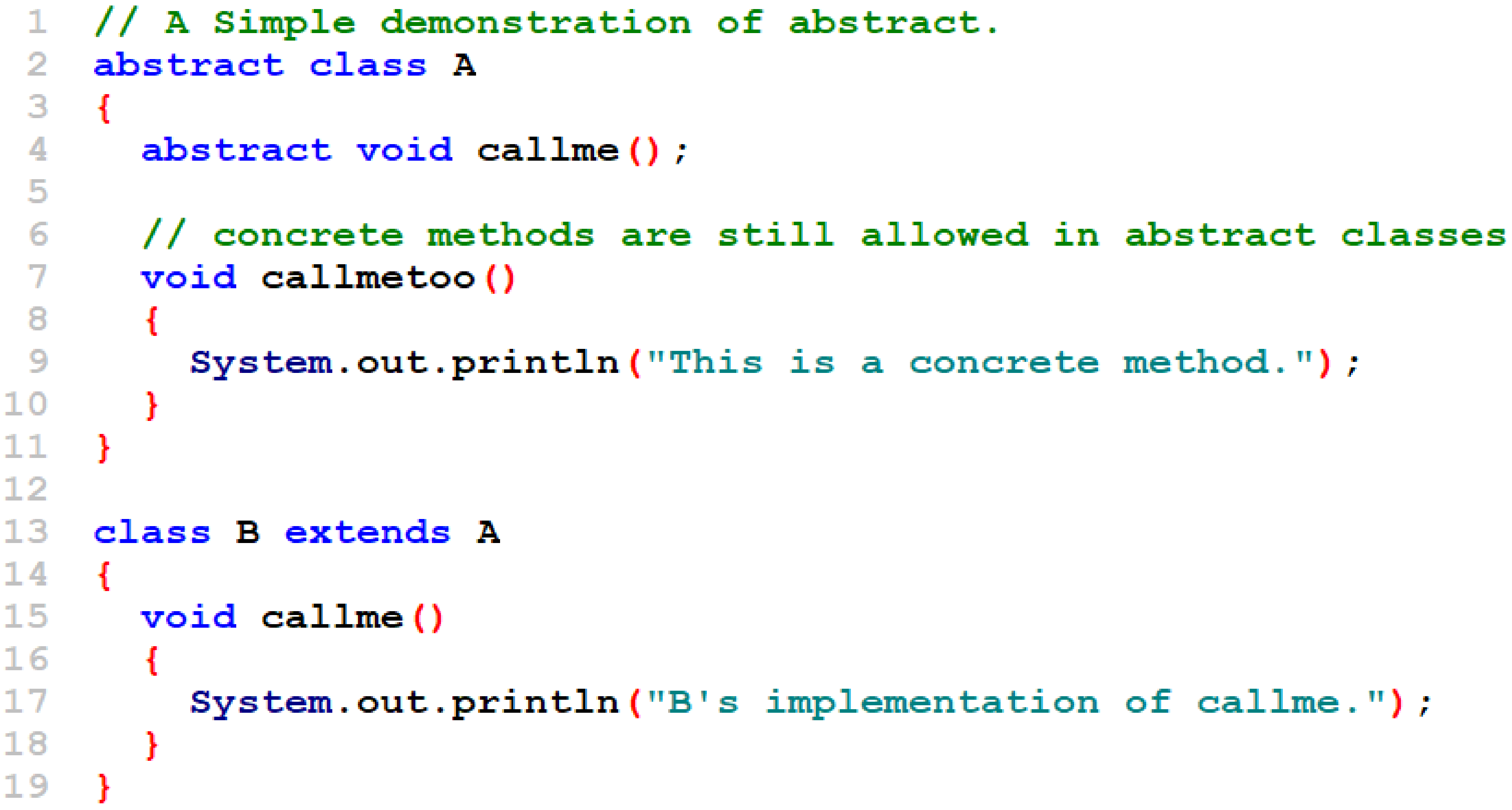
- ***A non-abstract class is called a concrete class***

  abstract class A

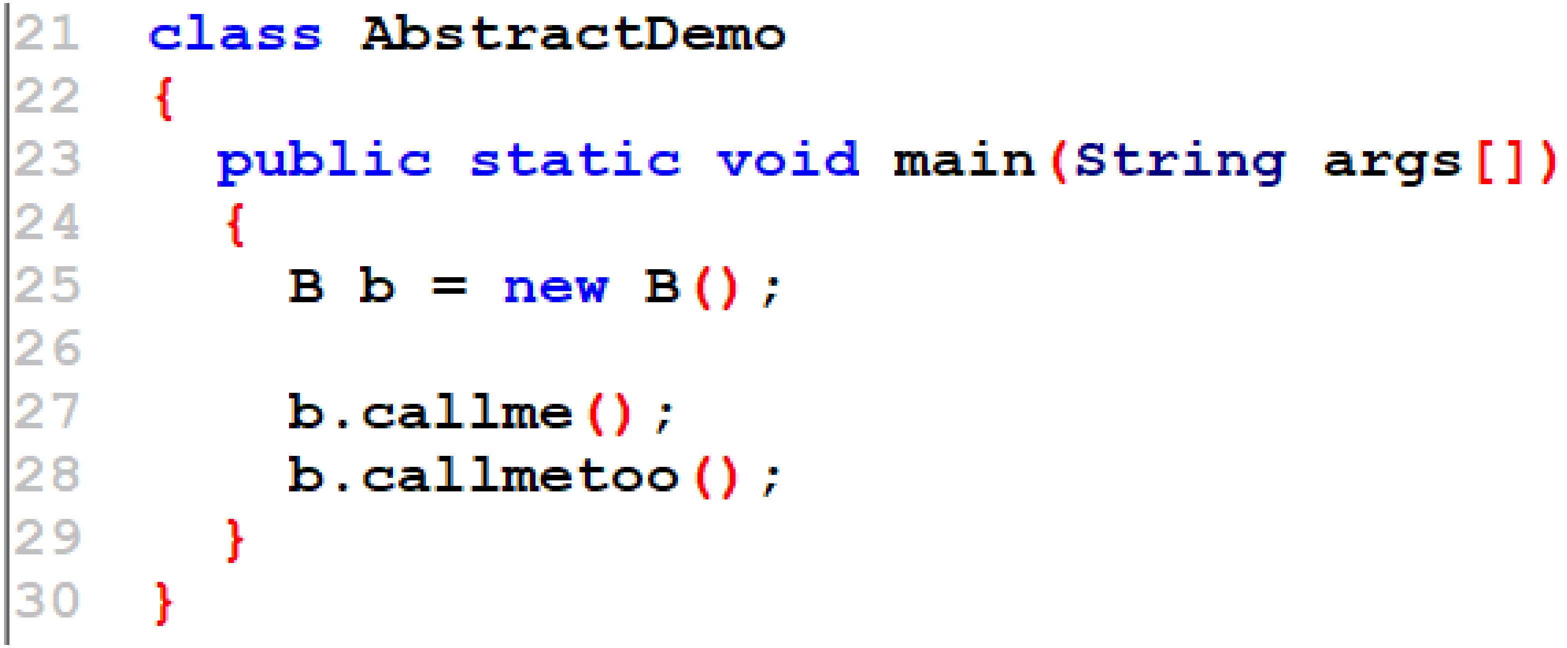
- Abstract class contains abstract method

  abstract method f()

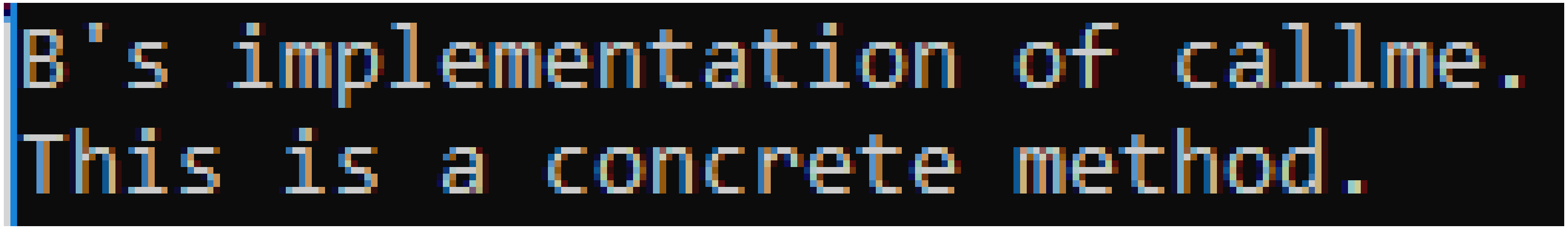
- No instance can be created of an abstract class
- The subclass must implement the abstract method
- Otherwise the subclass will be an abstract class too



```java
// A Simple demonstration of abstract.
abstract class A
{
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo()
  {
    System.out.println("This is a concrete method.");
  }
}

class B extends A
{
  void callme()
  {
    System.out.println("B's implementation of callme.");
  }
}
```

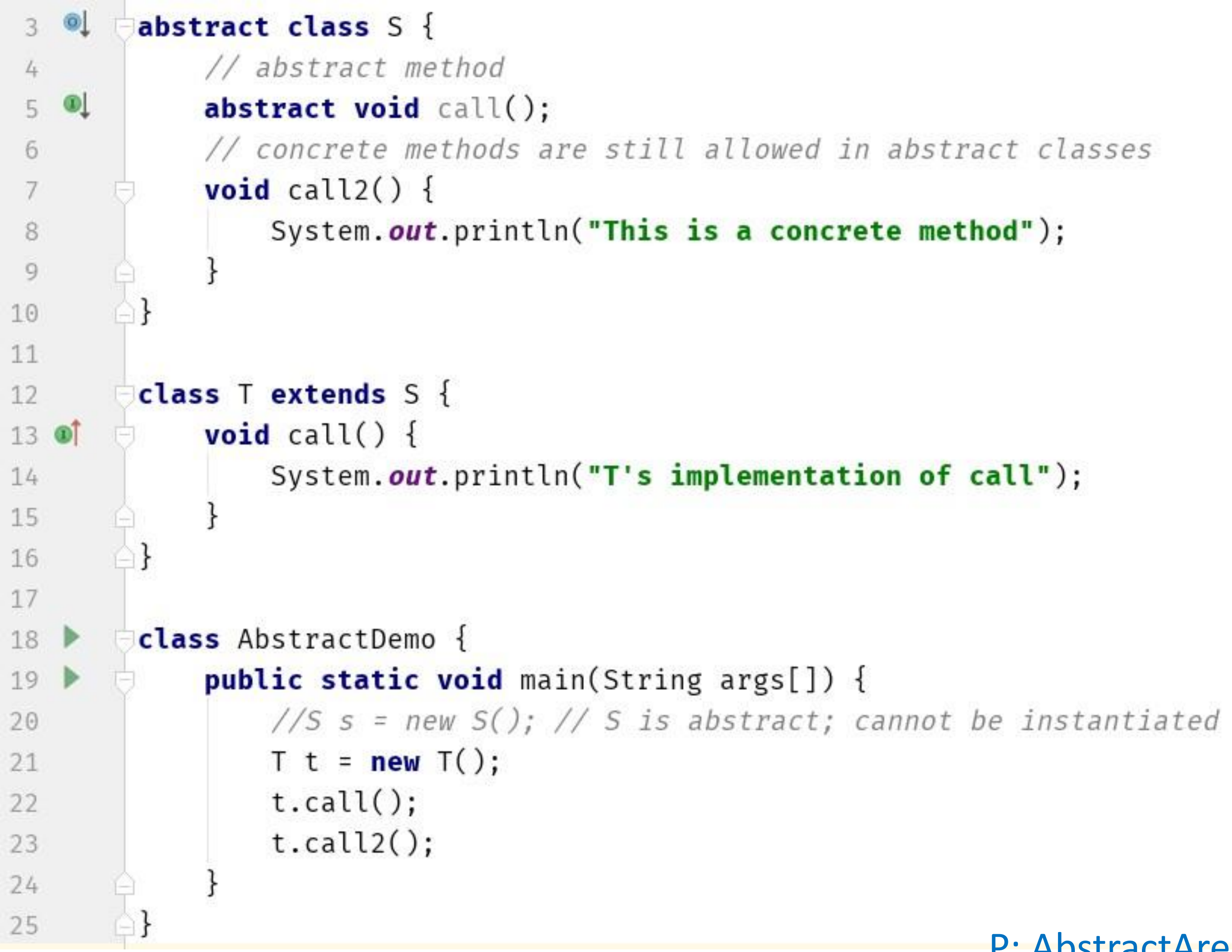


```java
class AbstractDemo
{
  public static void main(String args[])
  {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```



```
B's implementation of callme.
This is a concrete method.
```

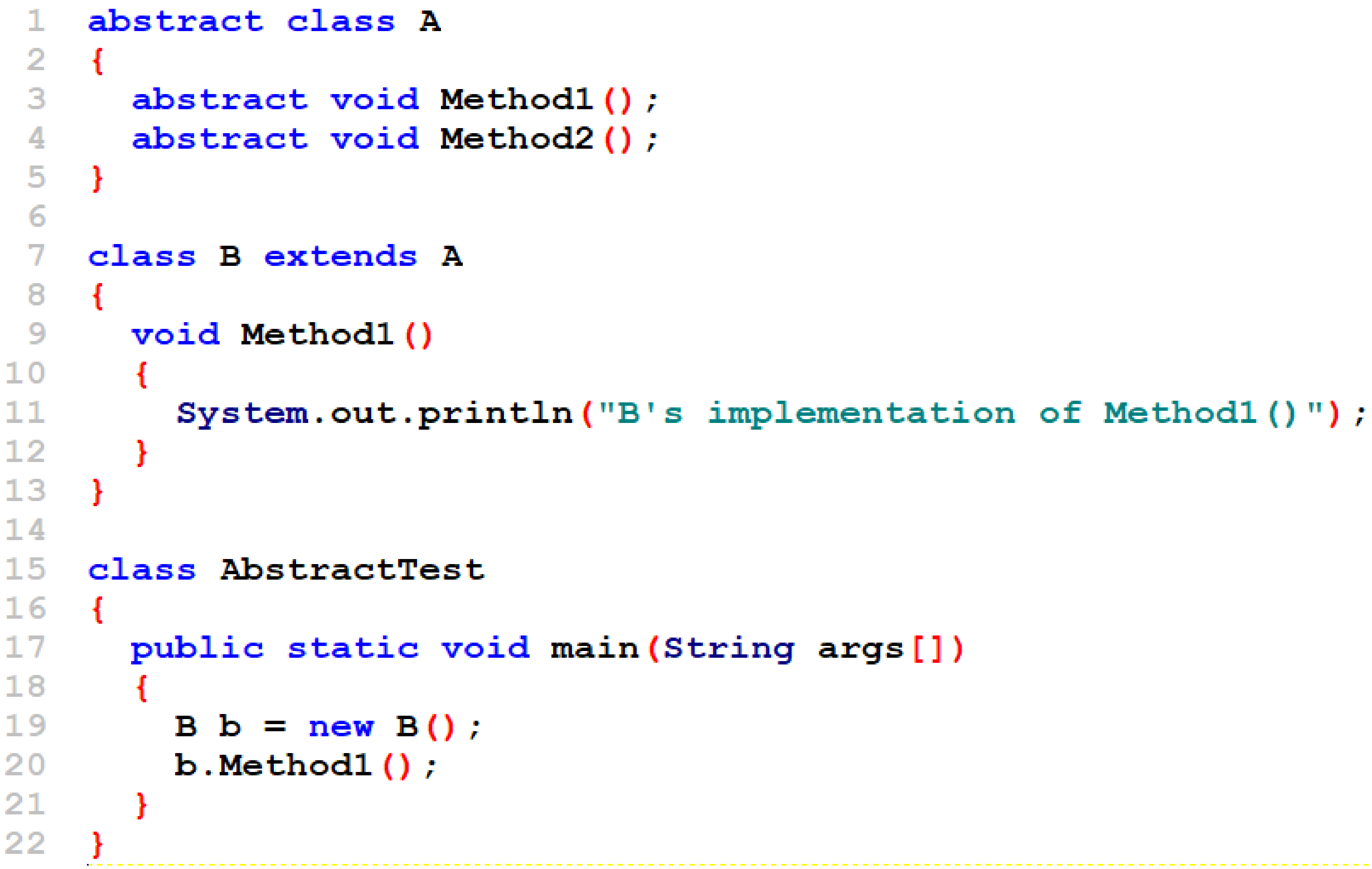
Abstract Class



```java
abstract class S {
    // abstract method
    abstract void call();
    // concrete methods are still allowed in abstract classes
    void call2() {
        System.out.println("This is a concrete method");
    }
}

class T extends S {
    void call() {
        System.out.println("T's implementation of call");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        //S s = new S(); // S is abstract; cannot be instantiated
        T t = new T();
        t.call();
        t.call2();
    }
}
```

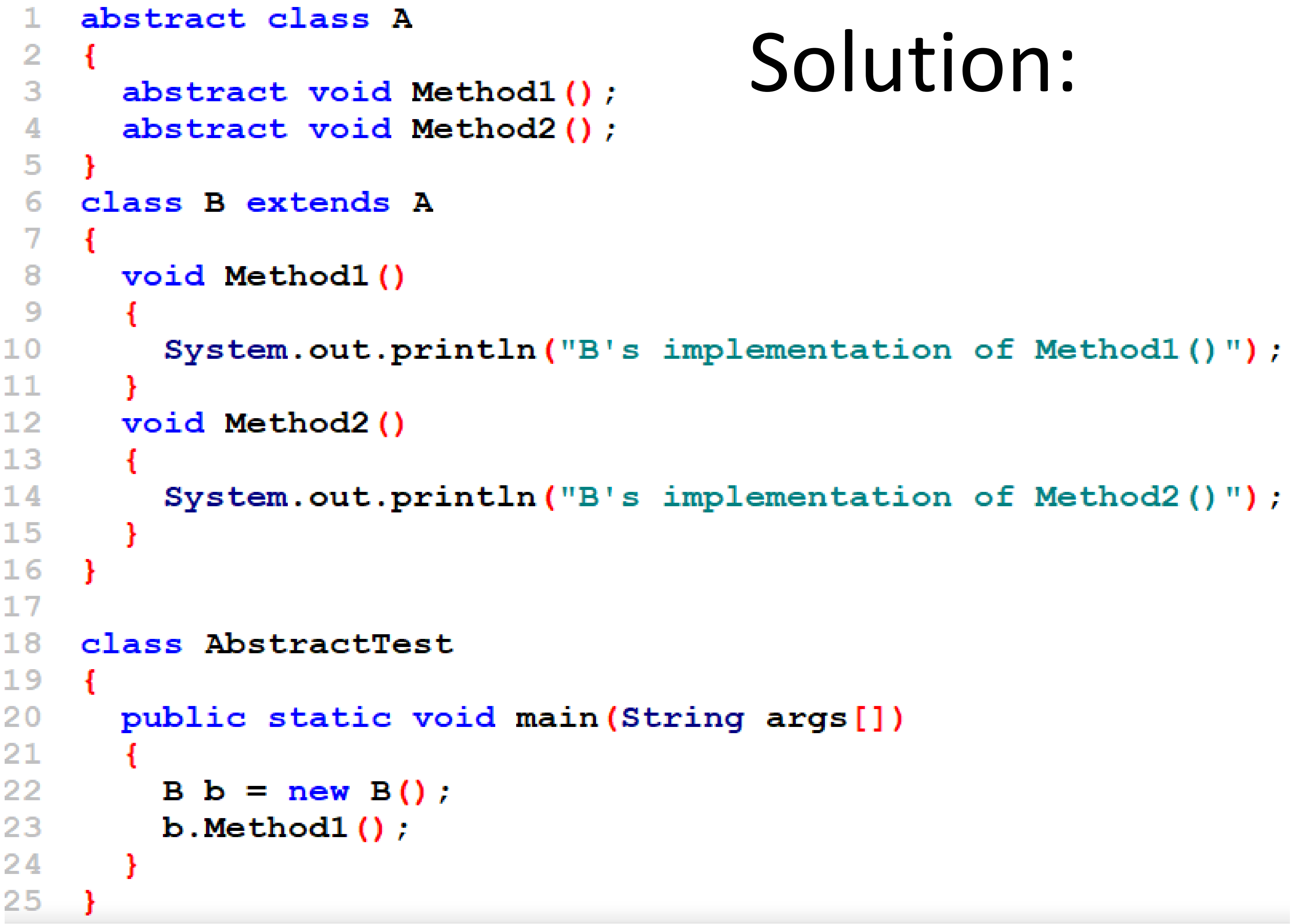
P: AbstractAreas.java
`AbstractDemo.java`

Question 1



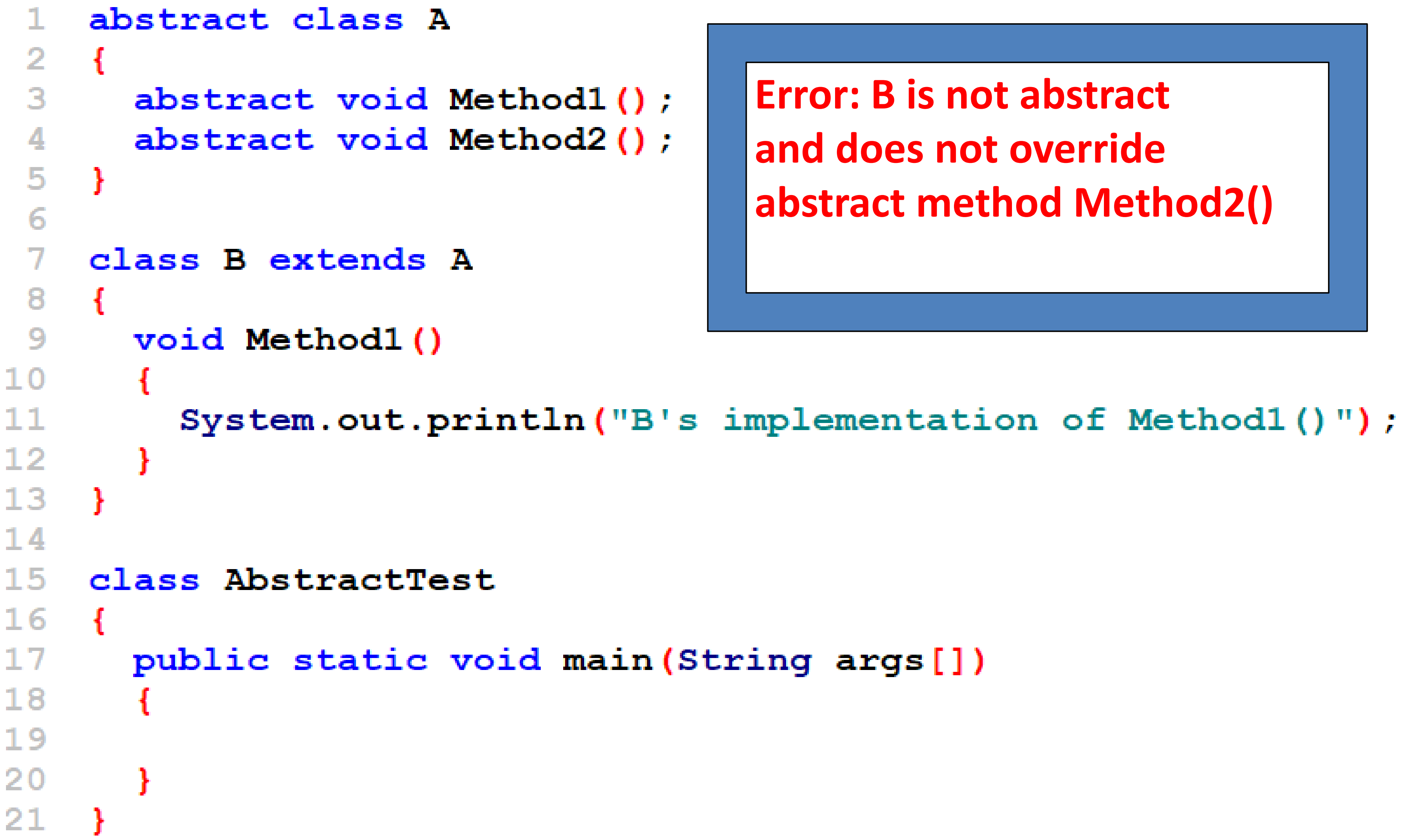
```java
abstract class A
{
  abstract void Method1();
  abstract void Method2();
}

class B extends A
{
  void Method1()
  {
    System.out.println("B's implementation of Method1()");
  }
}

class AbstractTest
{
  public static void main(String args[])
  {
    B b = new B();
    b.Method1();
  }
}
```

Solution:



```java
abstract class A
{
  abstract void Method1();
  abstract void Method2();
}
class B extends A
{
  void Method1()
  {
    System.out.println("B's implementation of Method1()");
  }
  void Method2()
  {
    System.out.println("B's implementation of Method2()");
  }
}

class AbstractTest
{
  public static void main(String args[])
  {
    B b = new B();
    b.Method1();
  }
}
```

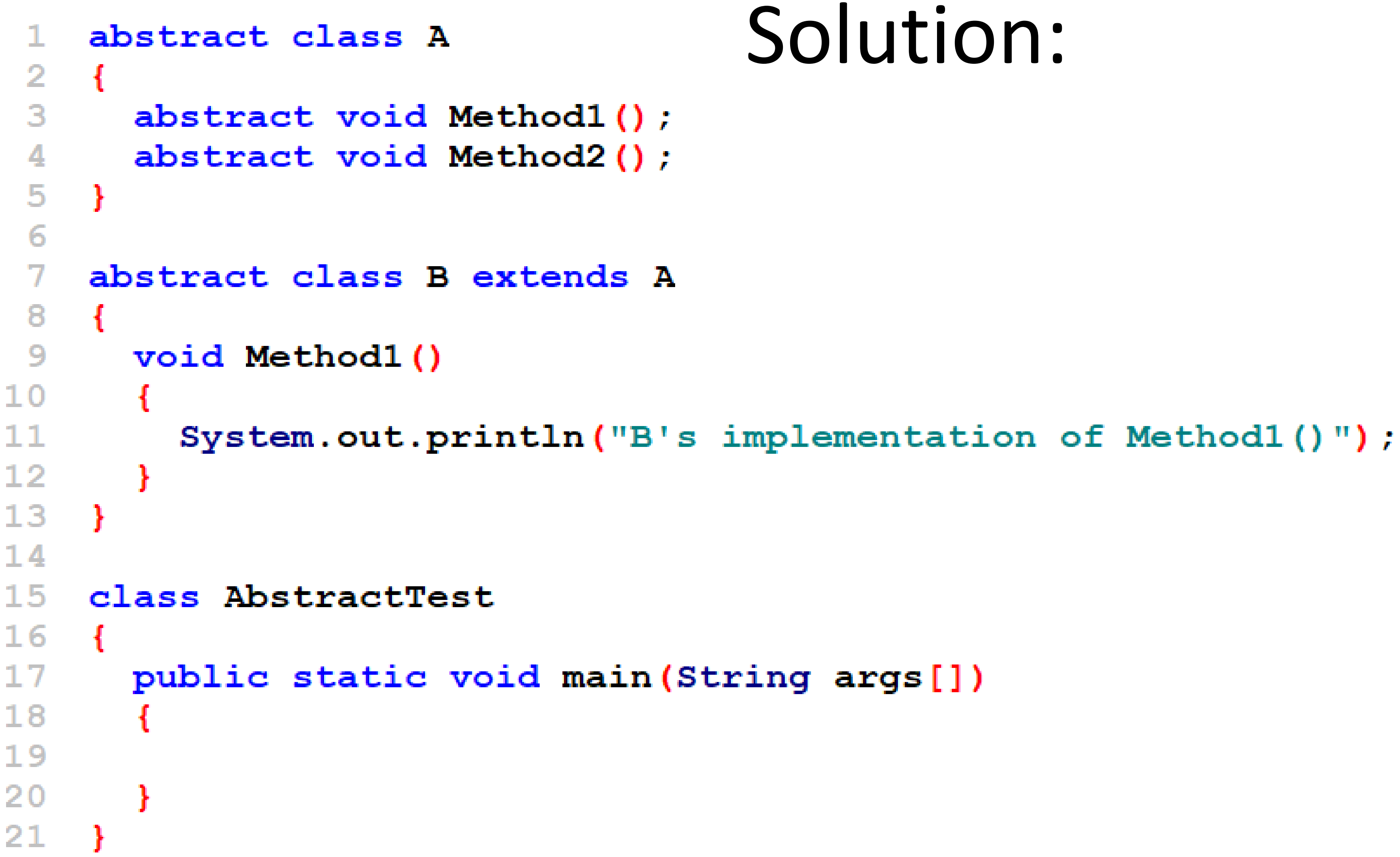
Question 2:



```
abstract class A
{
  abstract void Method1();
  abstract void Method2();
}

class B extends A
{
  void Method1()
  {
    System.out.println("B's implementation of Method1()");
  }
}

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

  }
}
```

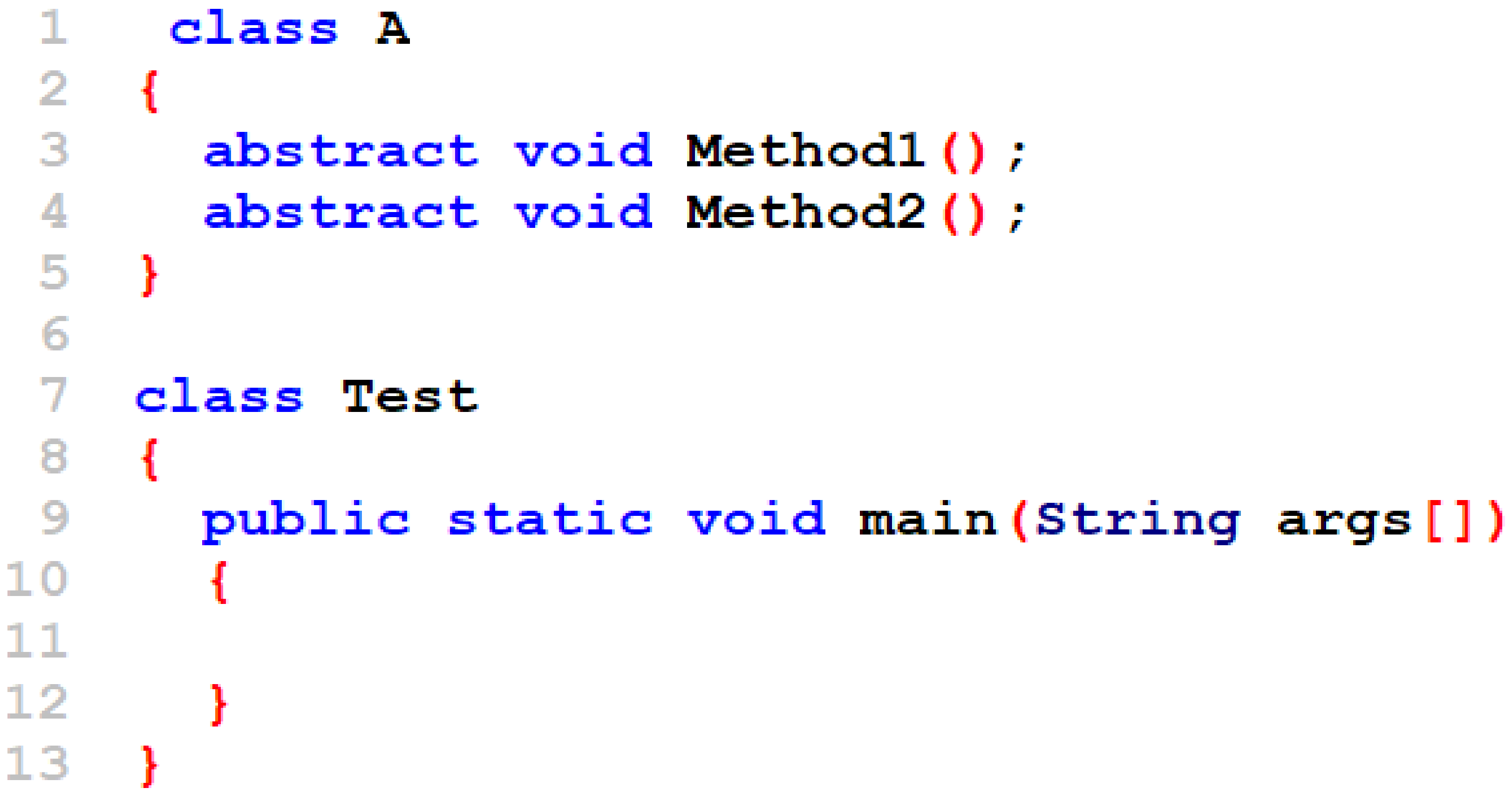
Error: B is not abstract and does not override abstract method Method2()

Solution:



```
abstract class A
{
   abstract void Method1();
   abstract void Method2();
}

abstract class B extends A
{
   void Method1()
   {
     System.out.println("B's implementation of Method1()");
   }
}

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

   }
}
```

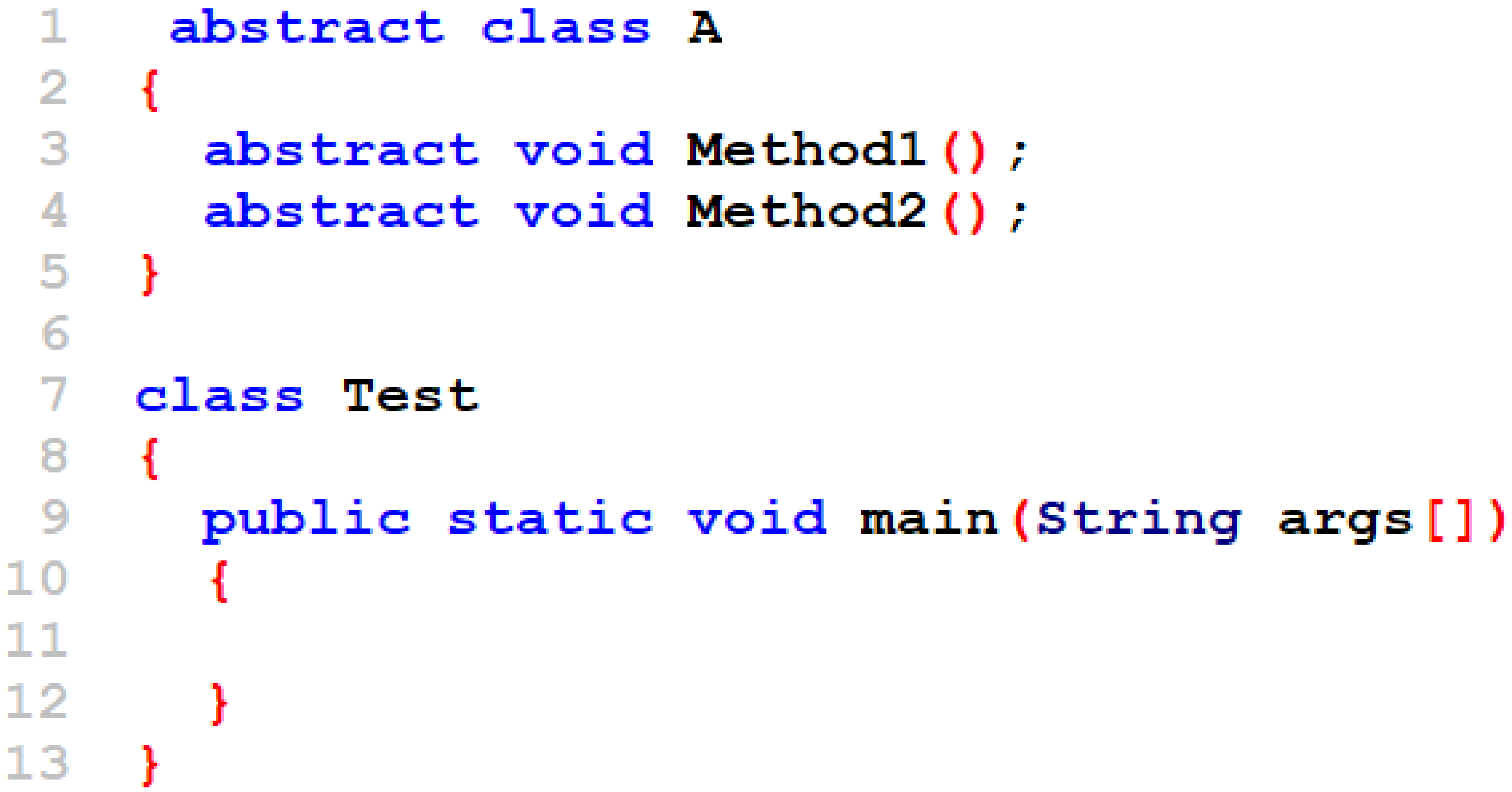
Question-3:



```
 class A
{
  abstract void Method1();
  abstract void Method2();
}

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

  }
}
```

Error: A is not abstract and does not override abstract method Method2() in A

Solution:



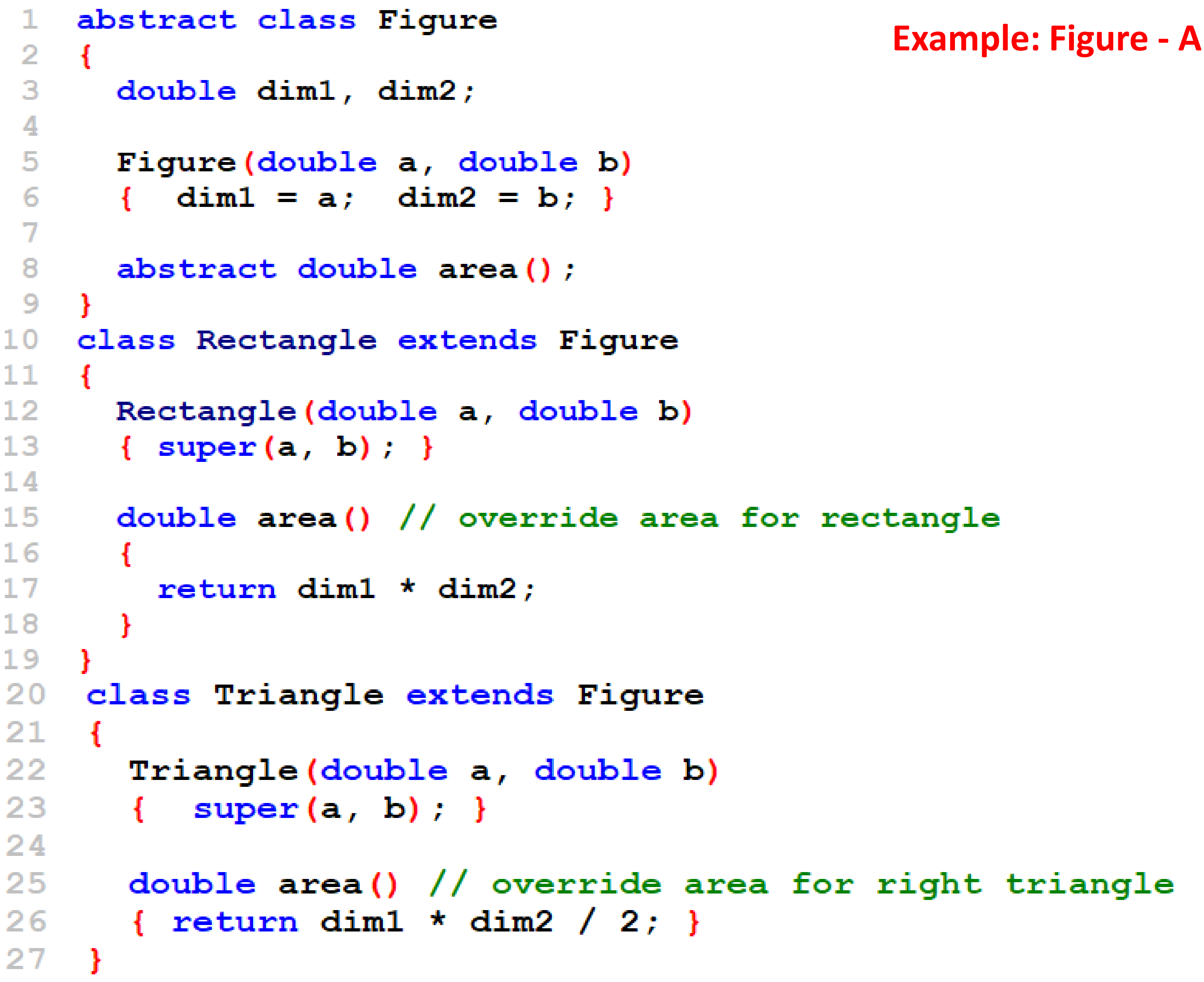
```
 abstract class A
{
  abstract void Method1();
  abstract void Method2();
}

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

  }
}
```

Run-time polymorphism

- Not possible to instantiate objects of Abstract classes; but, object references can be created
- Run-time polymorphism implemented through the use of superclass references
- Possible to create a reference to an abstract class so that it can be used to point to a subclass object

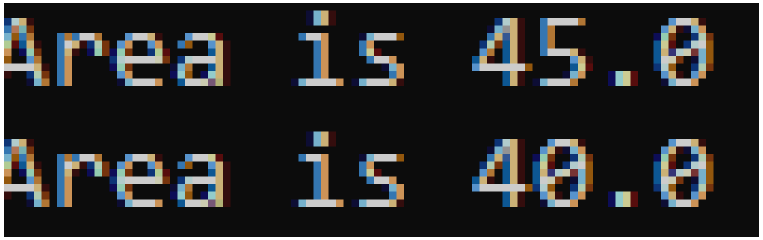
Example: Polymorphism in Figure class

Example: Figure - Abstract class



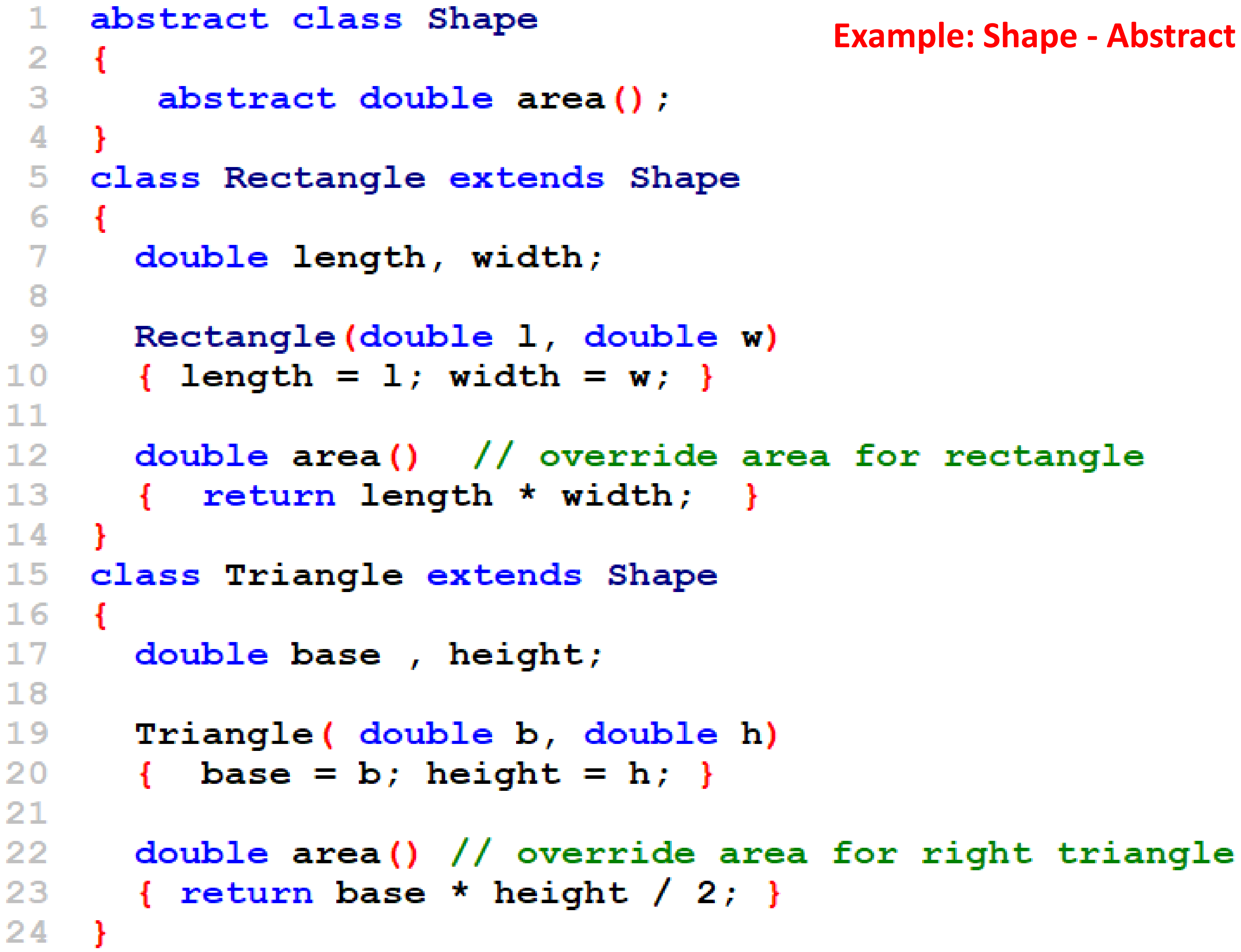
```java
abstract class Figure
{
  double dim1, dim2;

  Figure(double a, double b)
  {   dim1 = a;   dim2 = b; }

  abstract double area();
}
class Rectangle extends Figure
{
  Rectangle(double a, double b)
  { super(a, b); }

  double area() // override area for rectangle
  {
    return dim1 * dim2;
  }
}
class Triangle extends Figure
{
  Triangle(double a, double b)
  {  super(a, b); }

  double area() // override area for right triangle
  { return dim1 * dim2 / 2; }
}
```

```java
class AbstractAreas
{
  public static void main(String args[])
  {
   // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);

    Figure figref; // this is OK, no object is created

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());
  }
}
```

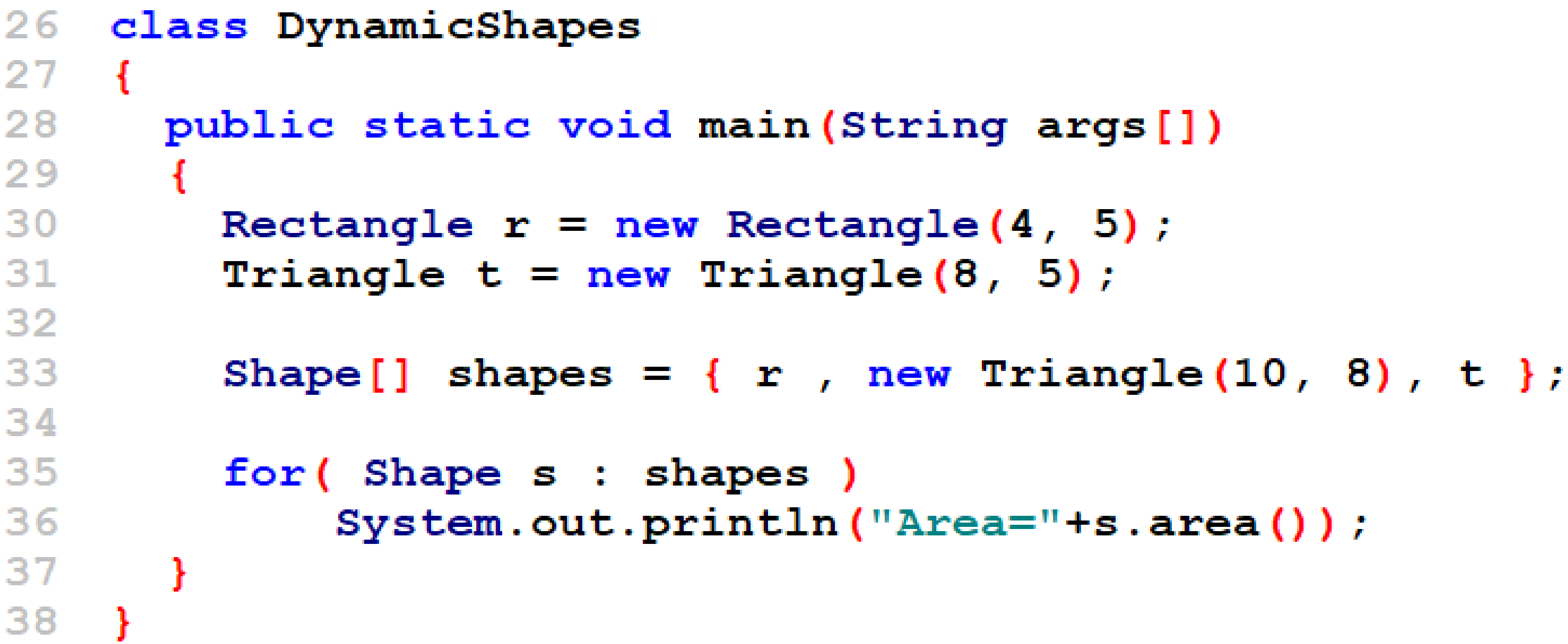


```
Area is 45.0
Area is 40.0
```

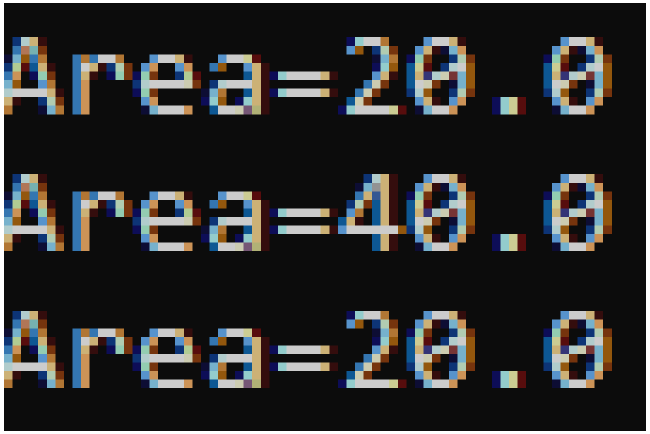
Example: Shape - Abstract class



```
abstract class Shape
{
   abstract double area();
}
class Rectangle extends Shape
{
  double length, width;

  Rectangle(double l, double w)
  { length = l; width = w; }

  double area()  // override area for rectangle
  {  return length * width;  }
}
class Triangle extends Shape
{
  double base , height;

  Triangle( double b, double h)
  {  base = b; height = h; }

  double area() // override area for right triangle
  { return base * height / 2; }
}
```



```java
class DynamicShapes
{
  public static void main(String args[])
  {
    Rectangle r = new Rectangle(4, 5);
    Triangle t = new Triangle(8, 5);

    Shape[] shapes = { r , new Triangle(10, 8), t };

    for( Shape s : shapes )
        System.out.println("Area="+s.area());
  }
}
```



```
Area=20.0
Area=40.0
Area=20.0
```

Till here for Sessional-1