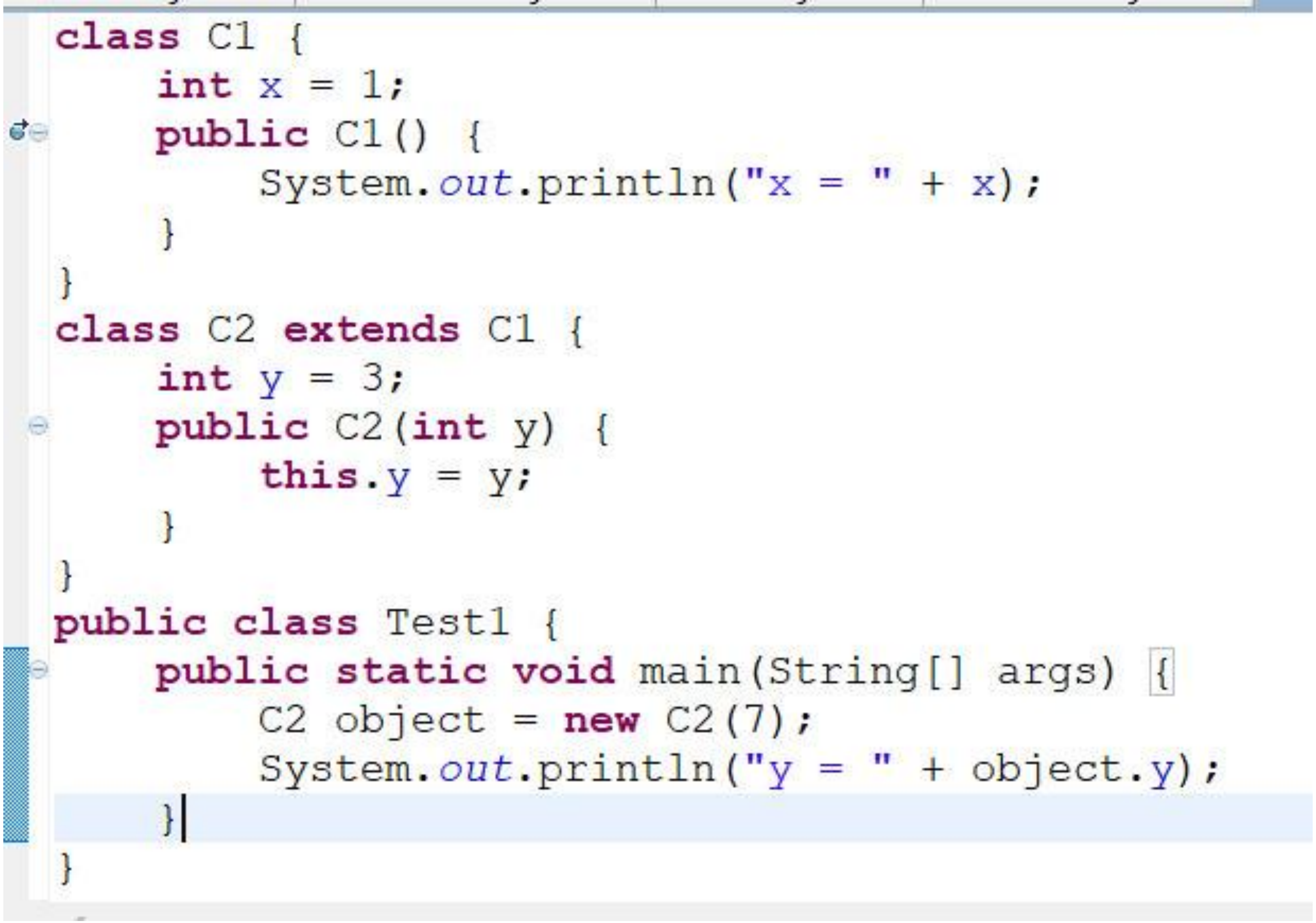
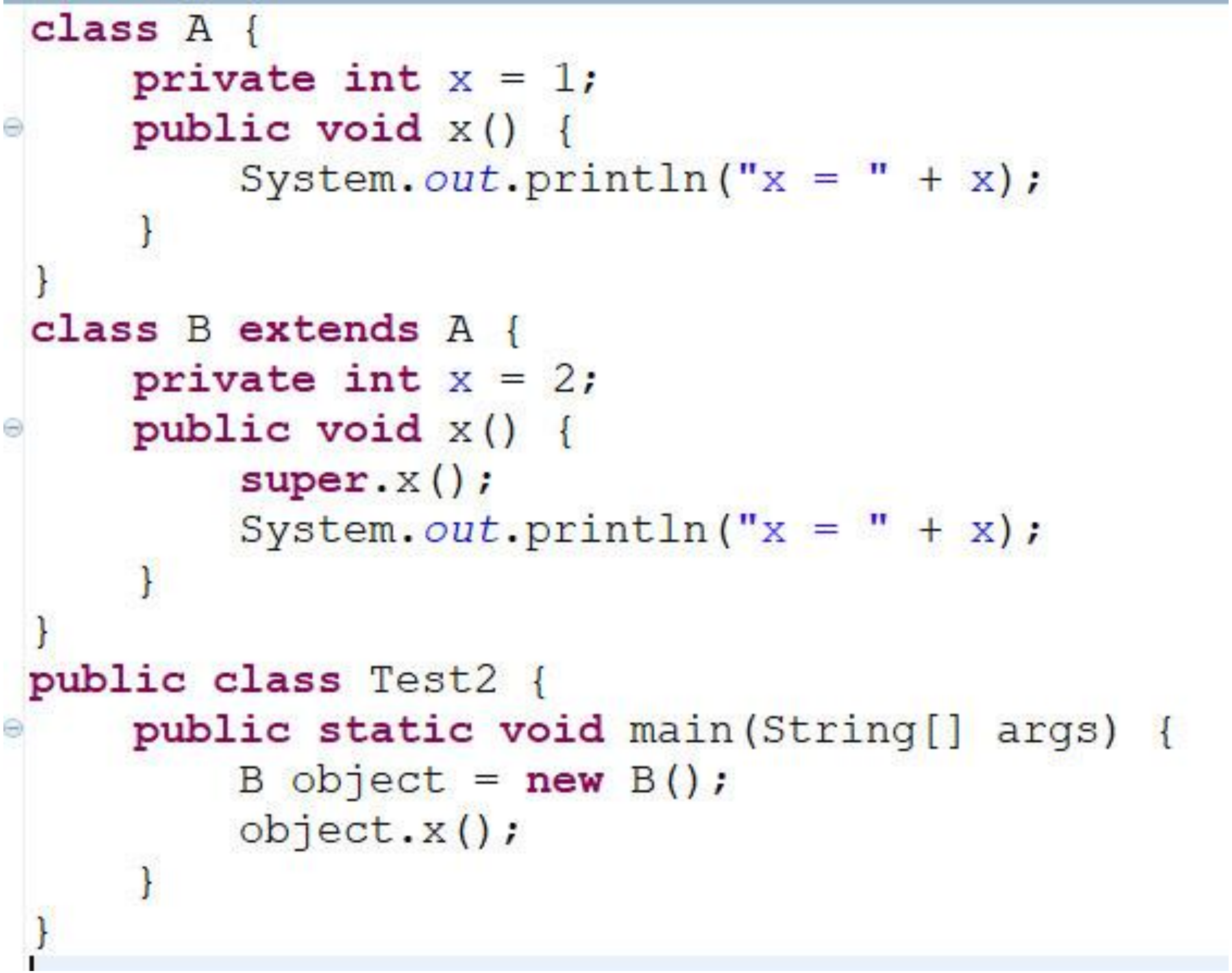
Inheritance
Abstract Classes

Inheritance

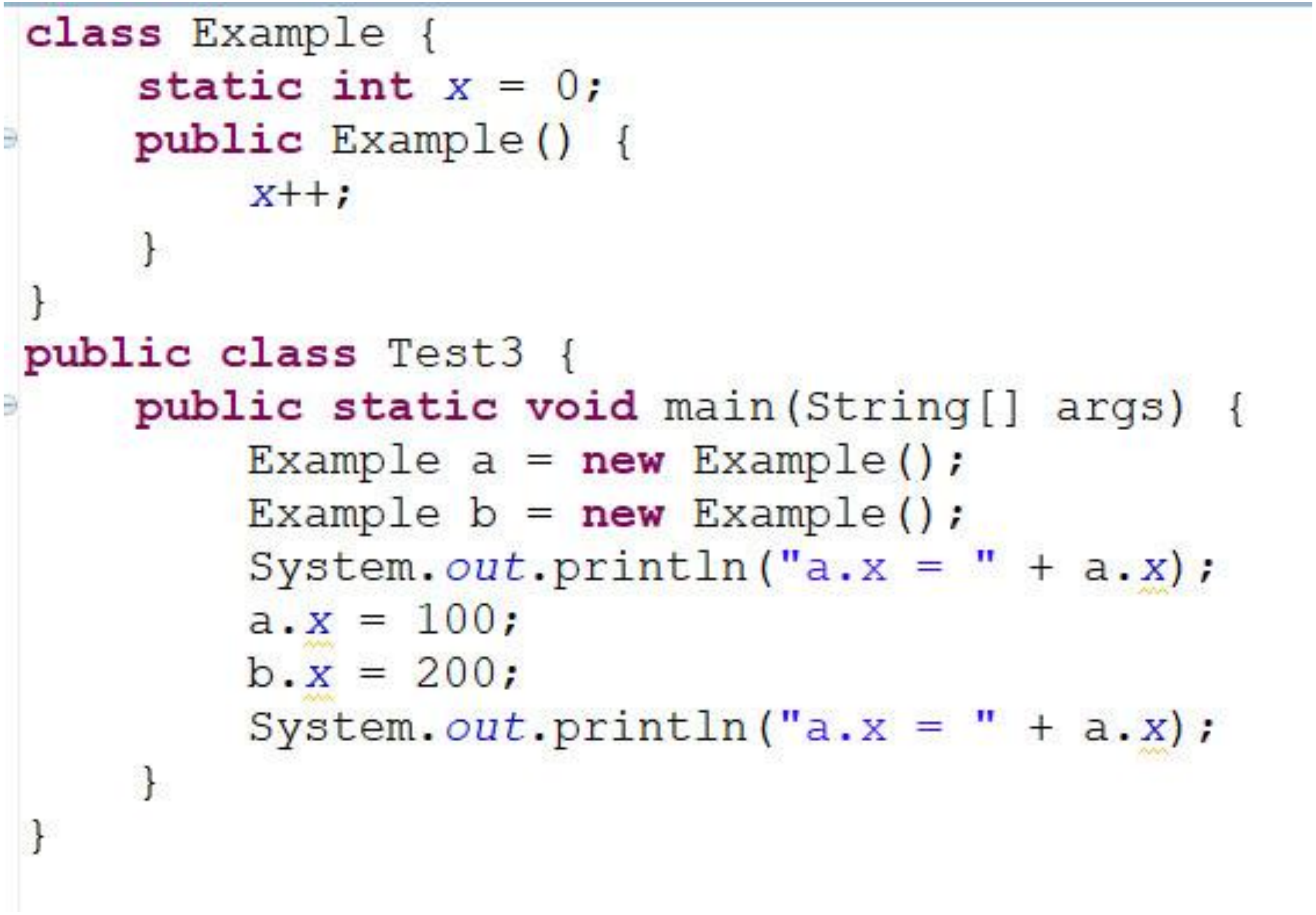
- Object oriented languages have a feature called **inheritance**
- Inheritance enables you to define a new class, based upon an existing class
- The new class is similar to the existing class, but it has additional member variables and methods
- This makes programming easier, because you can build upon an existing class, instead of starting out from scratch
- Programming in Java consists mostly of creating class hierarchies and instantiating objects from them



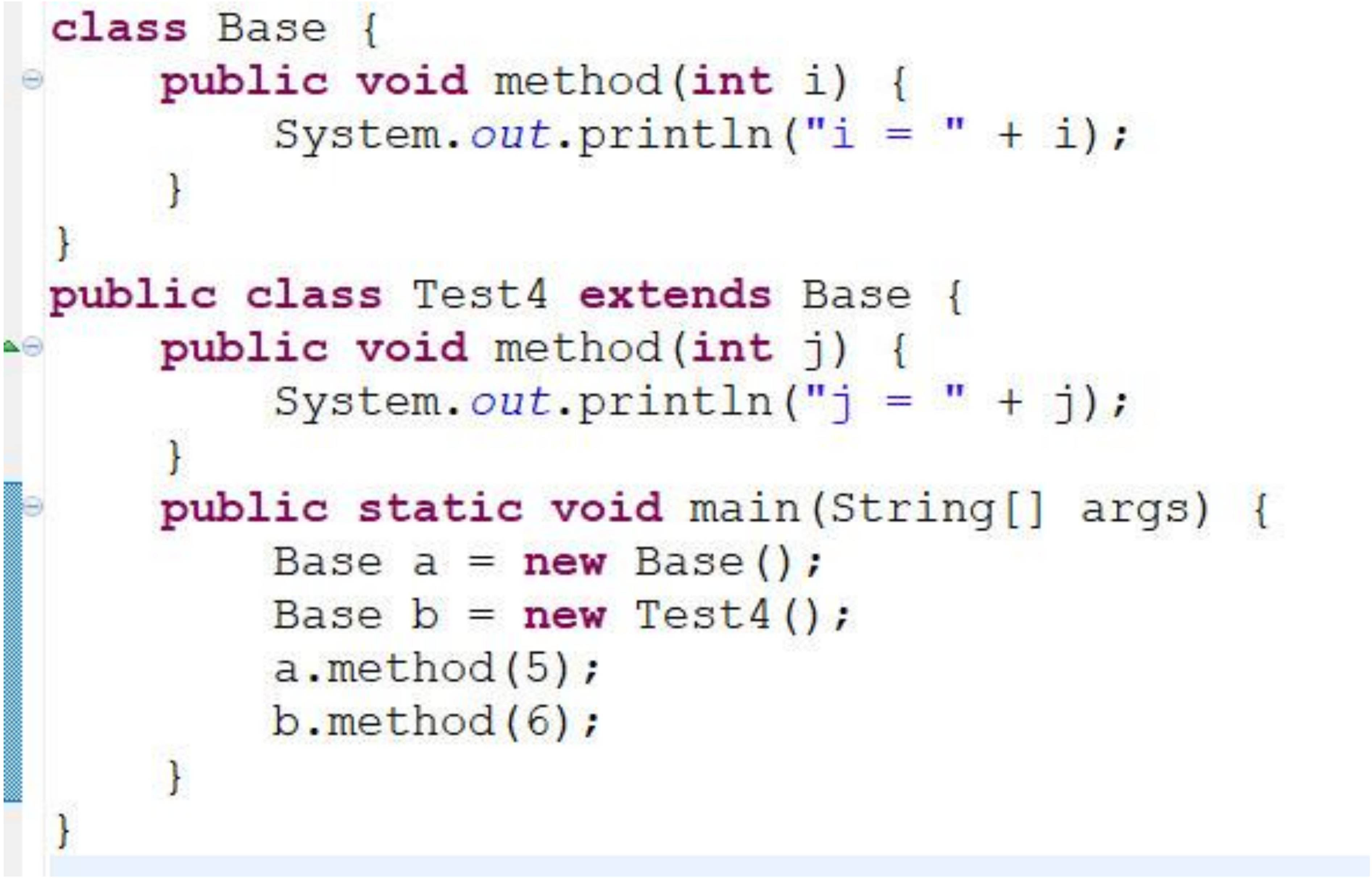
```
class C1 {
    int x = 1;
    public C1() {
        System.out.println("x = " + x);
    }
}
class C2 extends C1 {
    int y = 3;
    public C2(int y) {
        this.y = y;
    }
}
public class Test1 {
    public static void main(String[] args) {
        C2 object = new C2(7);
        System.out.println("y = " + object.y);
    }
}
```



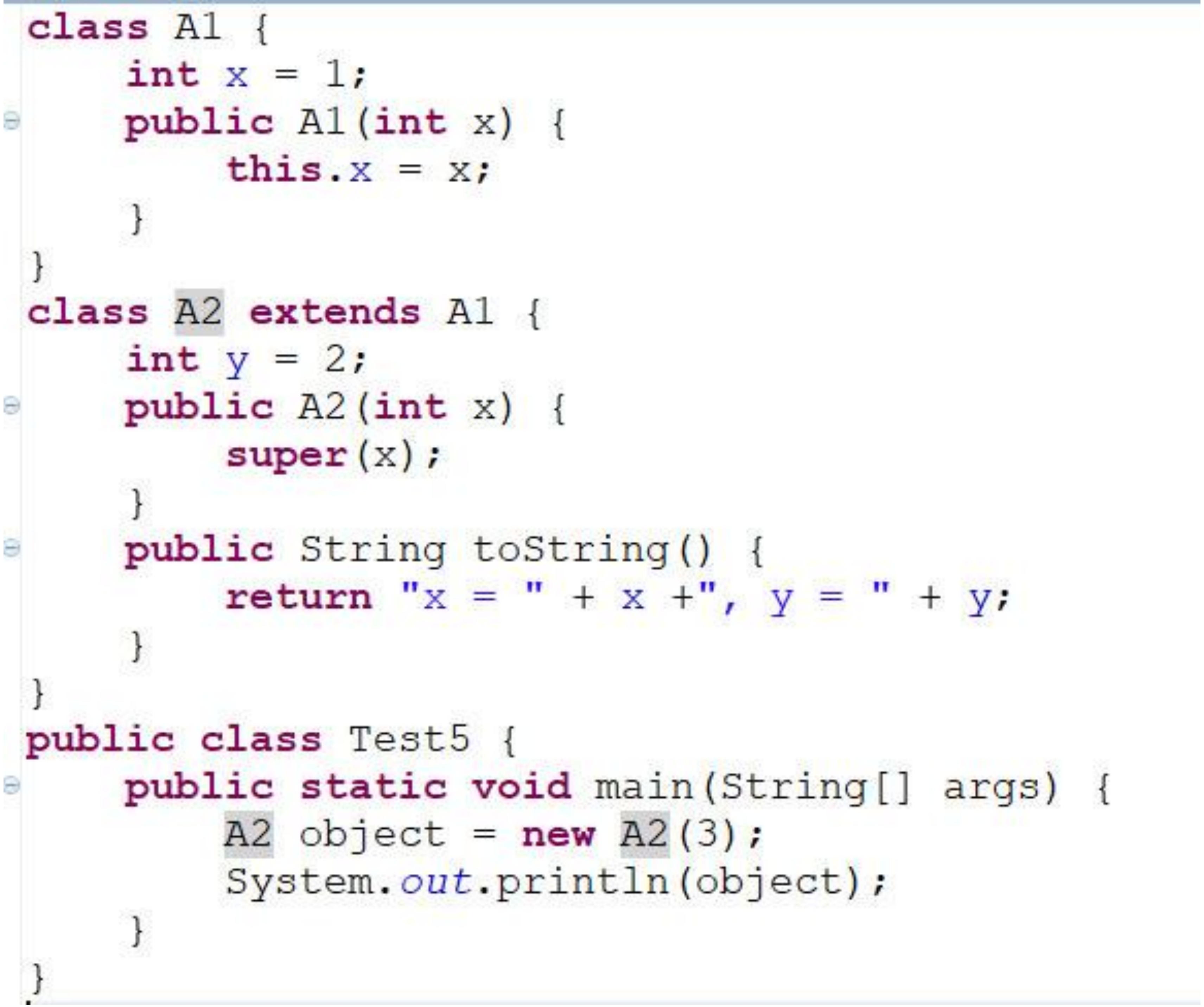
```java
class A {
    private int x = 1;
    public void x() {
        System.out.println("x = " + x);
    }
}
class B extends A {
    private int x = 2;
    public void x() {
        super.x();
        System.out.println("x = " + x);
    }
}
public class Test2 {
    public static void main(String[] args) {
        B object = new B();
        object.x();
    }
}
```



```
class Example {
    static int x = 0;
    public Example() {
        x++;
    }
}
public class Test3 {
    public static void main(String[] args) {
        Example a = new Example();
        Example b = new Example();
        System.out.println("a.x = " + a.x);
        a.x = 100;
        b.x = 200;
        System.out.println("a.x = " + a.x);
    }
}
```



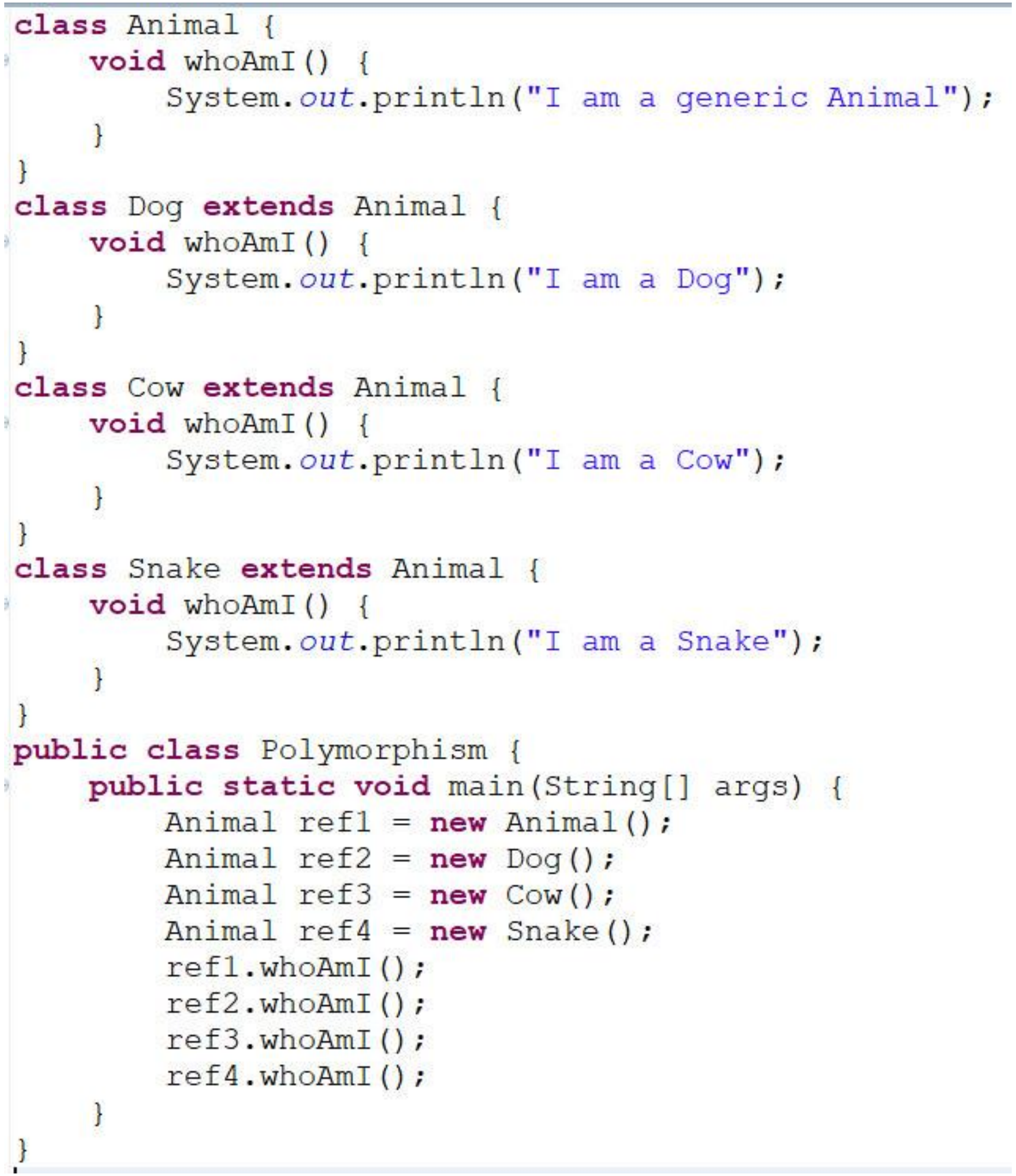
```
class Base {
    public void method(int i) {
        System.out.println("i = " + i);
    }
}
public class Test4 extends Base {
    public void method(int j) {
        System.out.println("j = " + j);
    }
    public static void main(String[] args) {
        Base a = new Base();
        Base b = new Test4();
        a.method(5);
        b.method(6);
    }
}
```



```java
class A1 {
    int x = 1;
    public A1(int x) {
        this.x = x;
    }
}
class A2 extends A1 {
    int y = 2;
    public A2(int x) {
        super(x);
    }
    public String toString() {
        return "x = " + x +", y = " + y;
    }
}
public class Test5 {
    public static void main(String[] args) {
        A2 object = new A2(3);
        System.out.println(object);
    }
}
```

Polymorphism

- **Polymorphism** is the capability of an action or method to do different things, based on the object that it is acting upon
- In other words, polymorphism allows you to define one interface and have multiple implementation
- This is one of the **basic** principles of object oriented programming
- The **method overriding** is an example of **runtime polymorphism**
- You can have a method in a subclass which overrides the method in its superclass with the same name and signature
- Java virtual machine determines the proper method to call at runtime, not at compile time



```java
class Animal {
    void whoAmI() {
        System.out.println("I am a generic Animal");
    }
}
class Dog extends Animal {
    void whoAmI() {
        System.out.println("I am a Dog");
    }
}
class Cow extends Animal {
    void whoAmI() {
        System.out.println("I am a Cow");
    }
}
class Snake extends Animal {
    void whoAmI() {
        System.out.println("I am a Snake");
    }
}
public class Polymorphism {
    public static void main(String[] args) {
        Animal ref1 = new Animal();
        Animal ref2 = new Dog();
        Animal ref3 = new Cow();
        Animal ref4 = new Snake();
        ref1.whoAmI();
        ref2.whoAmI();
        ref3.whoAmI();
        ref4.whoAmI();
    }
}
```

- There are four variables of type *Animal*
- Only *ref1* refers to an instance of *Animal* class, all others refer to an instance of the subclasses of *Animal*
- From the output results, you can confirm which version of a method is invoked, based on the actually object’s type
- In Java, a variable declared type of class *A* can hold a reference to an object of class *A* or an object belonging to any subclasses of class *A*
- The program is able to resolve the correct method related to the subclass object at runtime
- This is called the runtime polymorphism in Java
- This provides the ability to override functionality already available in the class hierarchy tree
- At runtime, which version of the method will be invoked is based on the type of actual object stored in that reference variable and not on the type of the reference variable

Exercise 1

- Create a class, called **Video**, to represent videos available at a rental store
- The class **Video** has three *private* member variables:
    - *String title;* //name of the item
    - *int length;* //number of minutes
    - *boolean available;* //is the video in the store?

Exercise 1

- There are two constructors in the class
- The first one has only one parameter, the title of the video, and initializes the other member variables of the class with the following values: *length = 90* and *available = true*
- The second constructor has two parameters, the title of the video and its length, while the member variable *available* is initialized to *true*
- The method *show()* displays information regarding the video objects

Exercise 1

- In another class, called **VideoTest**, which contains the *main* method, create two objects of class **Video**
- The first object is created using the first constructor of the class, and the second object is created using the second constructor of the class
- Display on the screen the information regarding the video objects

Exercise 2

- Create a class, called **Movie**, which inherits the class **Video**, and has, in addition, two member variables: the *director* of a movie and the *rating* of a movie
- The class **Movie** is a subclass of **Video**
- The class **Movie** has a constructor that initializes the data of **Movie** objects
- The method *show()* displays information regarding the movie objects

Observations

- Use the keyword *super* to invoke the constructor of the parent class to initialize some of the data
- *super(…)* must be the first statement in the subclass’s constructor
- A constructor for a children class always starts with an invocation of one of the constructors in the parent class
- If the parent class has several constructors, then the one which is invoked is determined by matching argument lists
- Even though the parent class has a *show()* method, the new definition of *show()* in the children class will **override** the parent’s version
- A children’s method **overrides** a parent’s method when it has the same signature as a parent method

Exercise 2

- In another class, called **MovieTest**, which contains the *main* method, create an object of class **Video** and an object of class **Movie**
- Display on the screen the information regarding the two objects

Abstract Classes

- An **abstract class** in Java is a class that is never instantiated
- Its purpose is to be a parent to several related classes
- The children classes inherit from the abstract parent class
- The advantage of using an abstract class is that you can group several related classes together as siblings
- Grouping classes together is important in keeping a program organized and understandable
- Access modifiers such as *public* can be placed before *abstract*
- Even though it can not be instantiated, an abstract class can define methods and variables that children class inherit

Exercise 3

- Create an abstract class, called **Card**, which contains:
- a) the *protected* member variable of type *String* called *recipient* (representing the name of the person who gets the card)
- b) *public abstract void greeting();*
- Each class has its own version of the *greeting()* method
- Each class has a *greeting()*, but each one is implemented differently
- It is useful to put an abstract *greeting()* method in the parent class
- This says that each children inherits the “idea” of *greeting()*, but each implementation is different

Observations

- Since no constructor is defined in **Card**, the default no argument constructor is automatically supplied by the compiler
- However, this constructor cannot be used directly, because no **Card** object can be constructed
- Abstract classes are used to organize the “concept” of something that has several different versions in the children classes
- The abstract class can include abstract methods and non-abstract methods

Exercise 3

- Create a class, called **Holiday**, which is a non-abstract children of an abstract parent class
- The constructor of the class **Holiday** initializes the name of the *recipient* with the parameter received as argument
- The method body for *greeting()* is:
- *System.out.println("Dear " + recipient + ", ");*
- *System.out.println("Season's Greetings!");*

Exercise 3

- Create a class, called **Birthday**, which is a non-abstract children of an abstract parent class
- It contains the *private* member variable *age*, of type *int*
- The constructor of the class **Birthday** initializes the name of the *recipient* and the *age* with the parameters received as arguments
- The method body for *greeting()* is:
- *System.out.println("Dear " + recipient + ", ");*
- *System.out.println("Happy " + age + "th Birthday!");*

Exercise 3

```
public class CardTest {
public static void main(String[] args) {
Card card1 = new Holiday("John");
card1.greeting();
Card card2 = new Birthday("Betty", 18);
card2.greeting();
}
}
```

```
Dear John,
Season’s Greetings!
Dear Betty,
Happy 18th Birthday!
```