Java Course

Object-Oriented Programming

Part 2

Table of Content

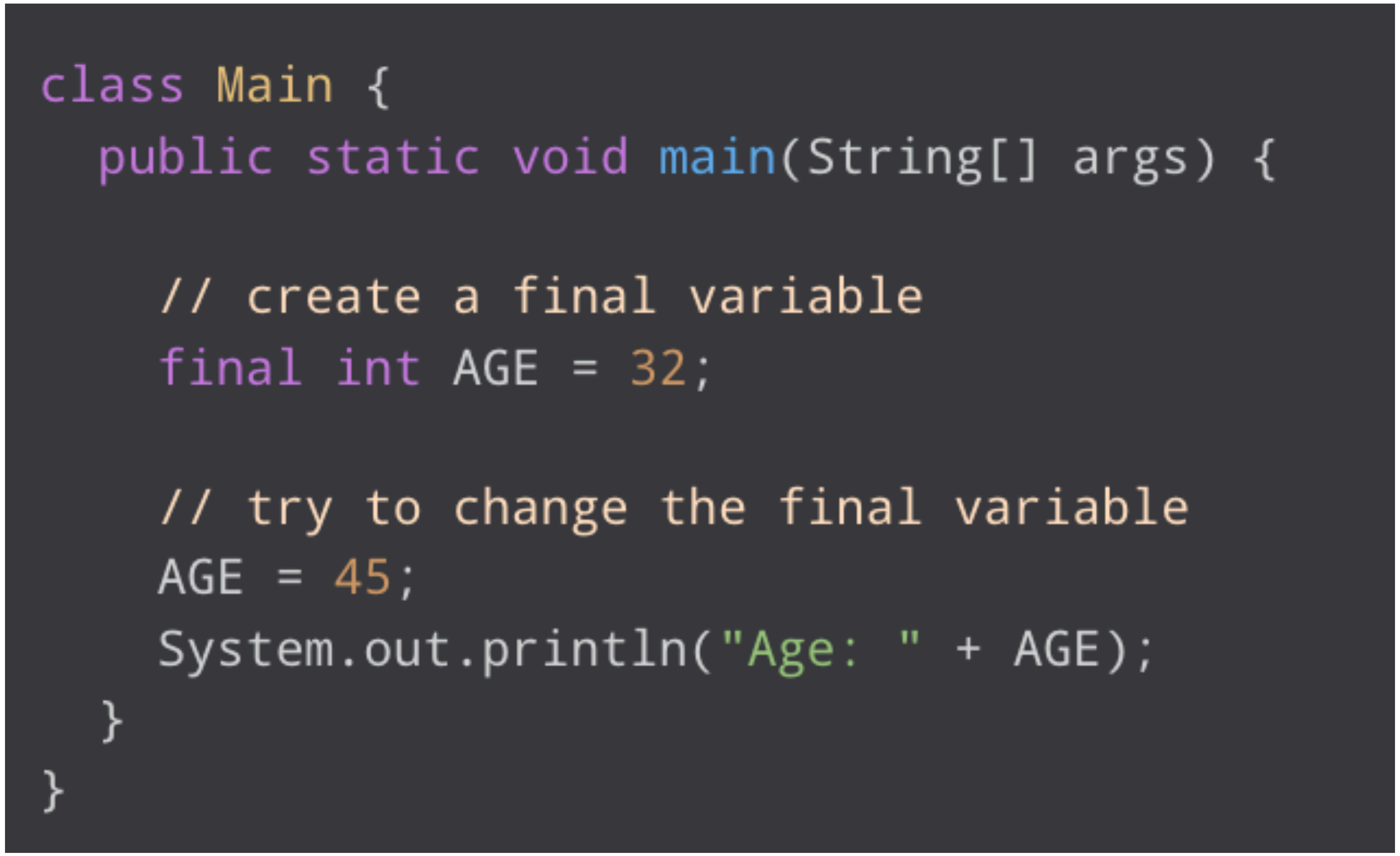
- `final` keyword
- `instanceof`
- Java Inheritance
- Types of Inheritance in Java
- Method Overriding
- `super` keyword

`final`

- `final` keyword denotes constants
- It can be used with variables, methods and classes
- Once any entity (variable, method or a class) is declared `final`, it can be assigned only once. That is:
    - the `final` variable cannot be reinitialised with another value
    - the `final` method cannot be overridden
    - the `final` class cannot be extended

Java `final` variable

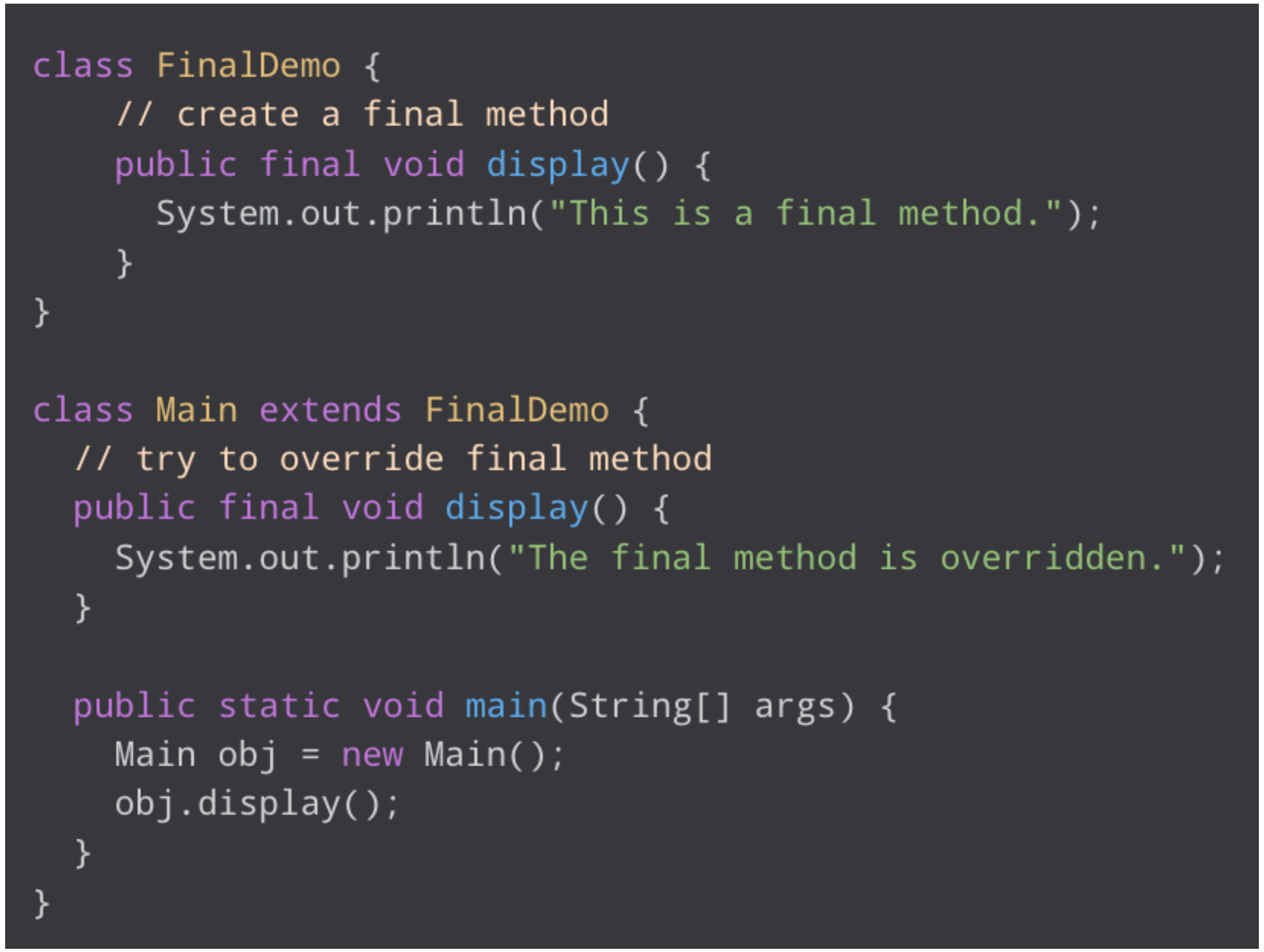
- Value of a `final` variable cannot be changed after initializing



```java
class Main {
  public static void main(String[] args) {

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

Java `final` method

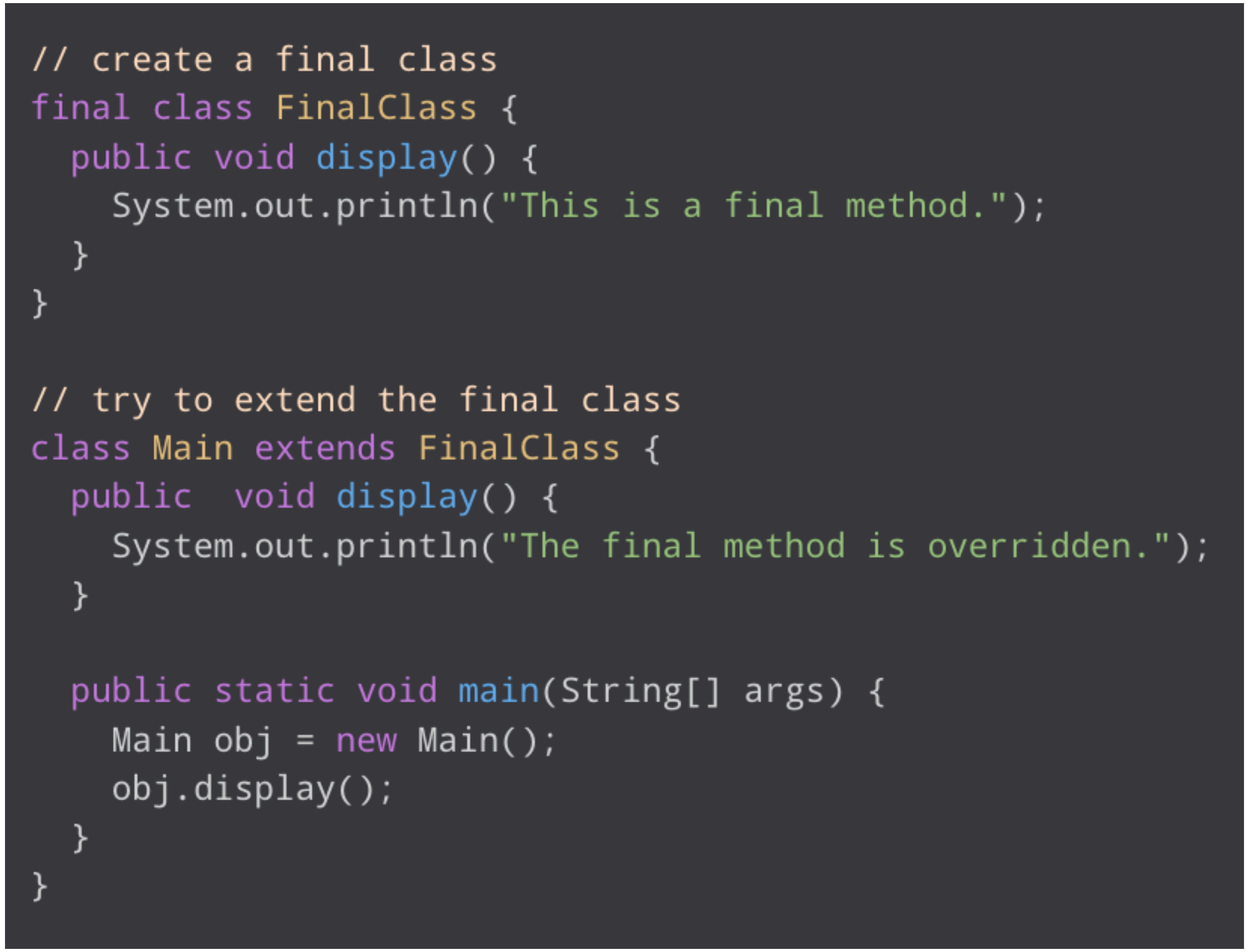
- In Java, the `final` method cannot be overridden by the child class



```java
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

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

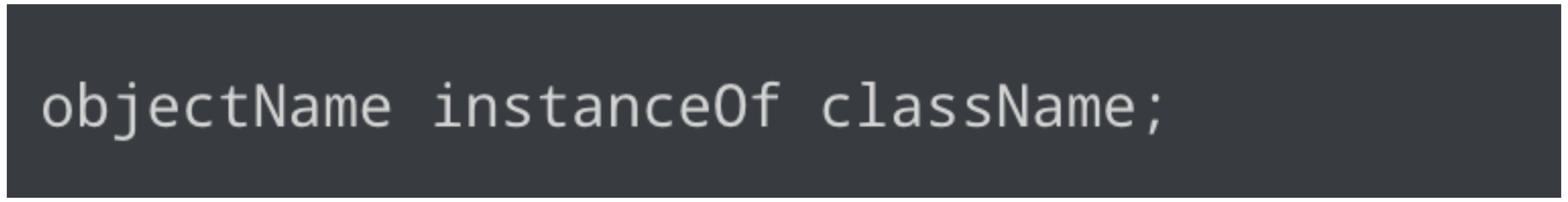
Java `final` Class

- In Java, the `final` class cannot be inherited by another class



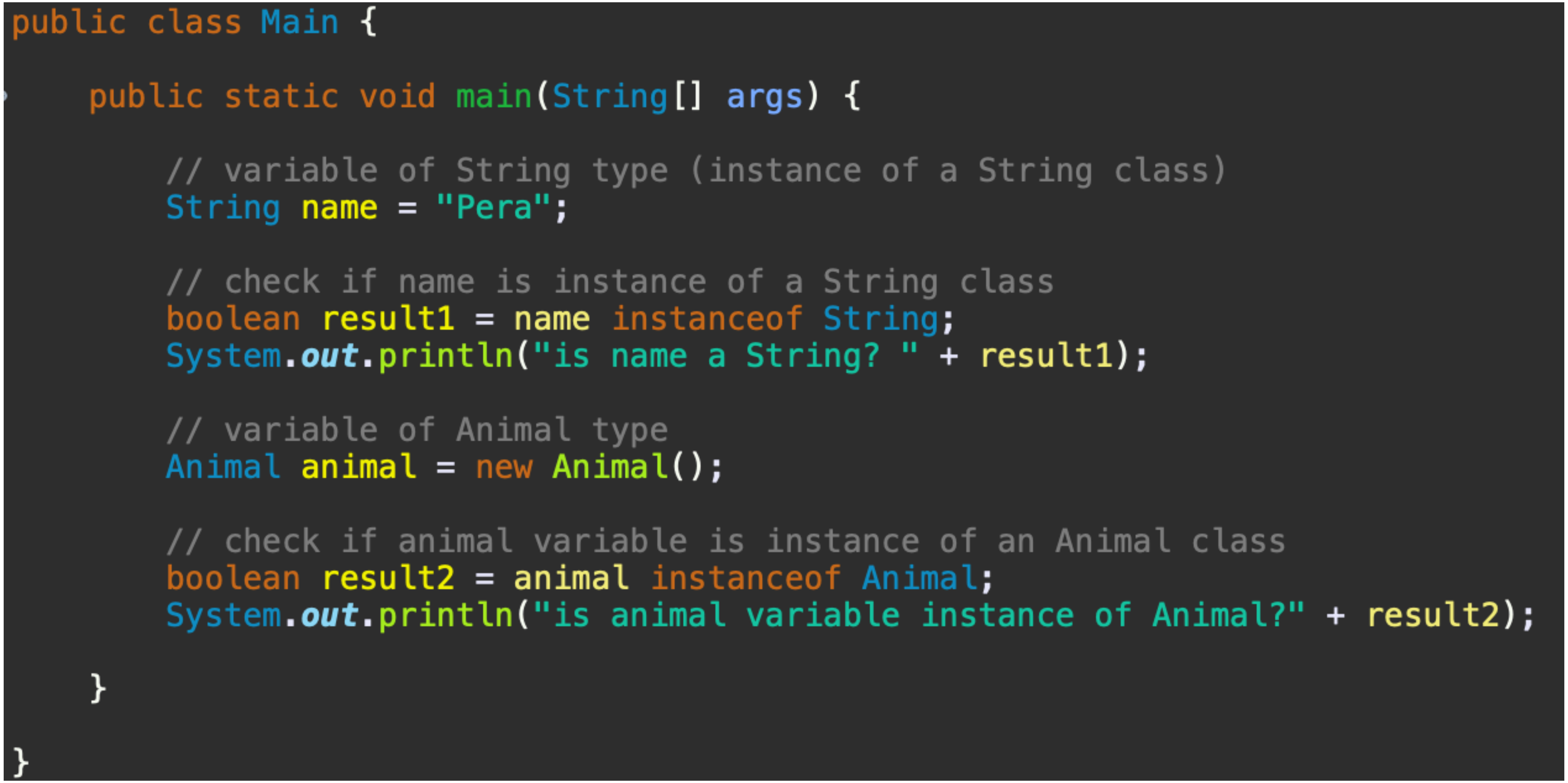
```java
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

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

Java `instanceof` operator



```
objectName instanceOf className;
```

- The instanceof operator is used to check whether an object is an instance of a particular class or not
- Operator returns a boolean value (object is either an instance of a particular class or it is not)



```java
public class Main {

    public static void main(String[] args) {

        // variable of String type (instance of a String class)
        String name = "Pera";

        // check if name is instance of a String class
        boolean result1 = name instanceof String;
        System.out.println("is name a String? " + result1);

        // variable of Animal type
        Animal animal = new Animal();

        // check if animal variable is instance of an Animal class
        boolean result2 = animal instanceof Animal;
        System.out.println("is animal variable instance of Animal?" + result2);

    }

}
```

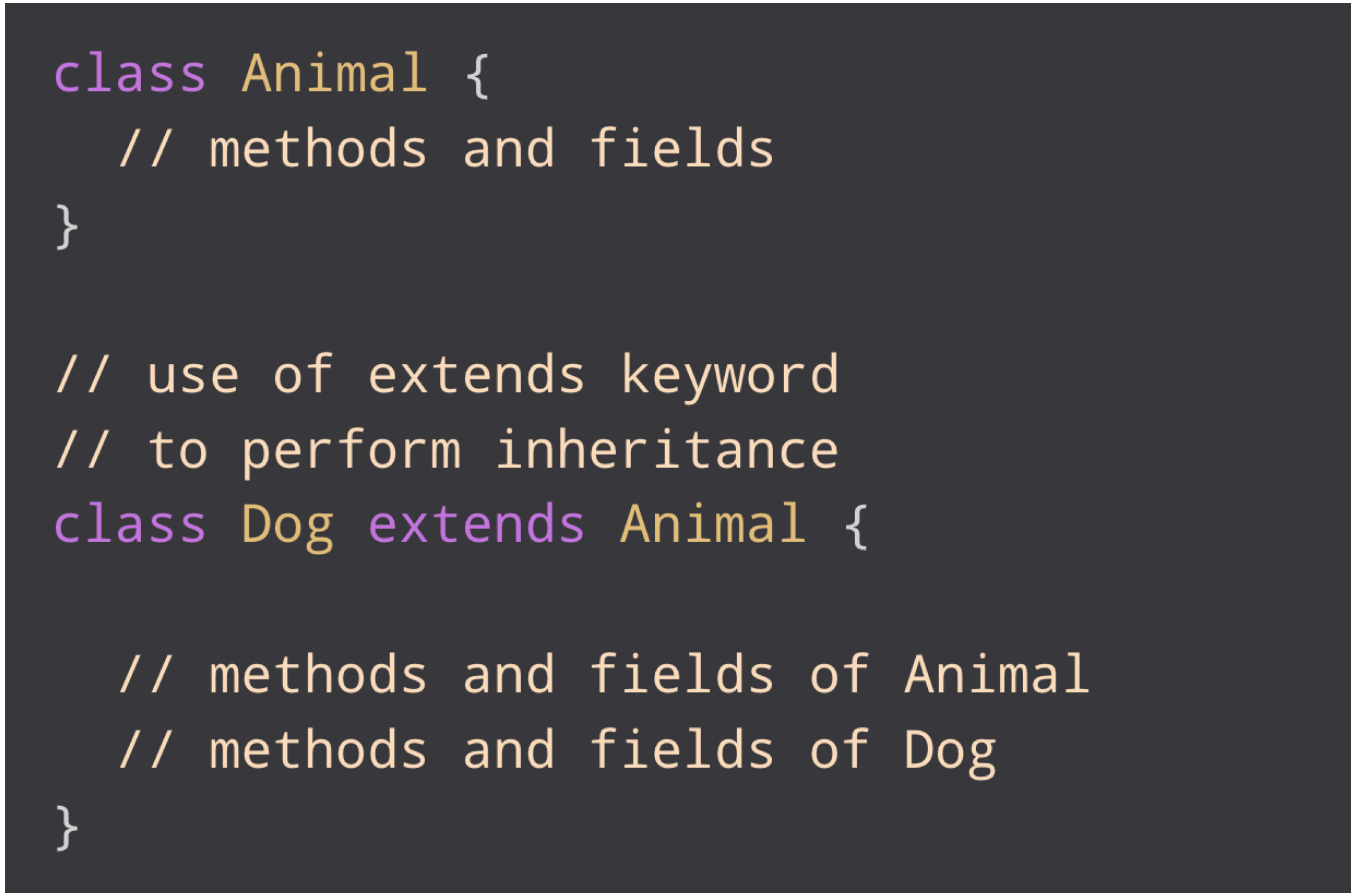
Inheritance in Java

- Mechanism in which one object acquires all the properties and behaviours of a parent object
- The idea is to create new classes that are built upon existing classes
- When you inherit from an existing class, you can reuse methods and fields of the parent class
- Reusability (**DRY** Principle - **D**on’t **R**epeat **Y**ourself)
- `extends` keyword

Inheritance in Java

- The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class)

- The `extends` keyword is used to perform inheritance



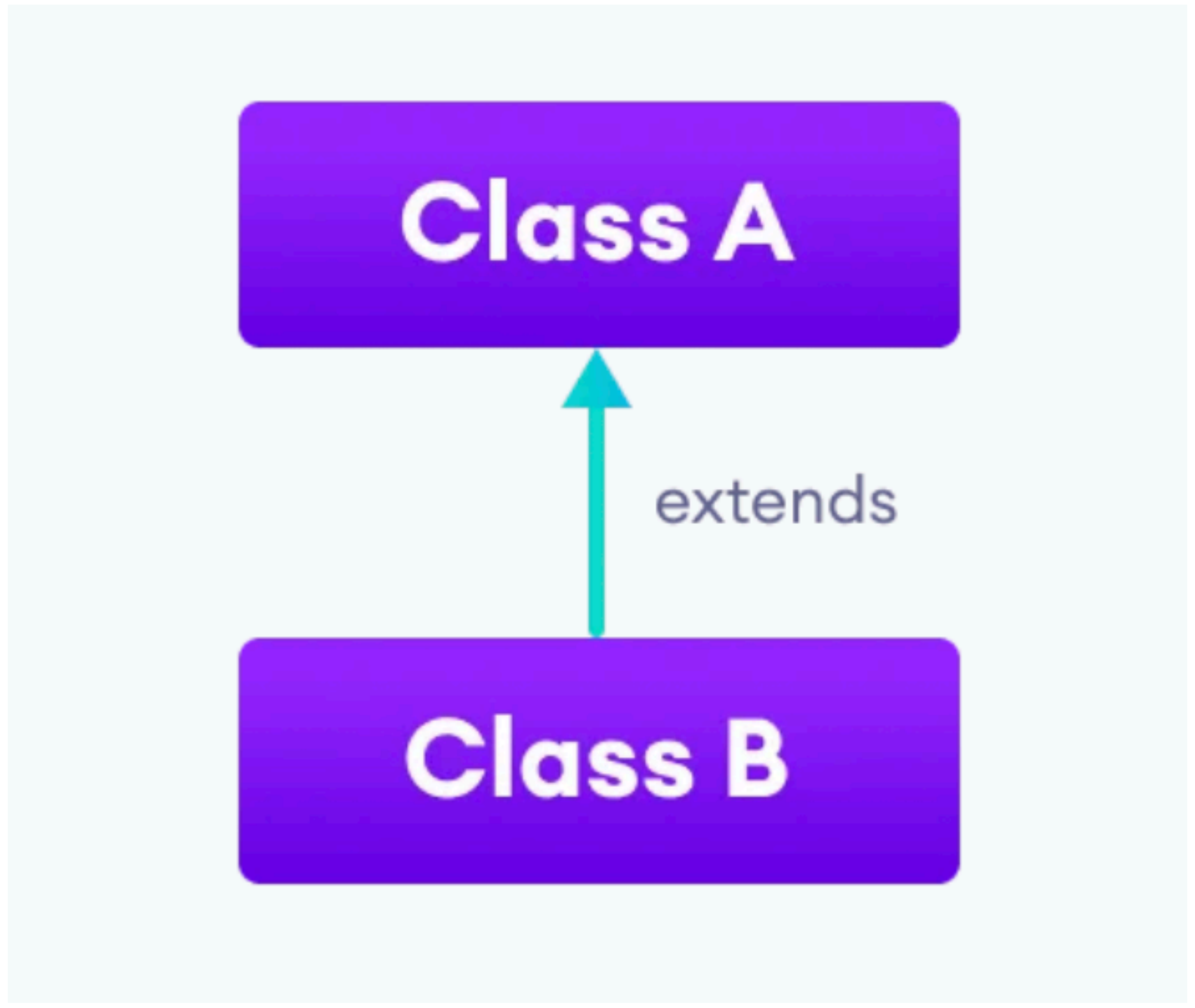
```java
class Animal {
  // methods and fields
}

// use of extends keyword
// to perform inheritance
class Dog extends Animal {

  // methods and fields of Animal
  // methods and fields of Dog
}
```

Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance ???

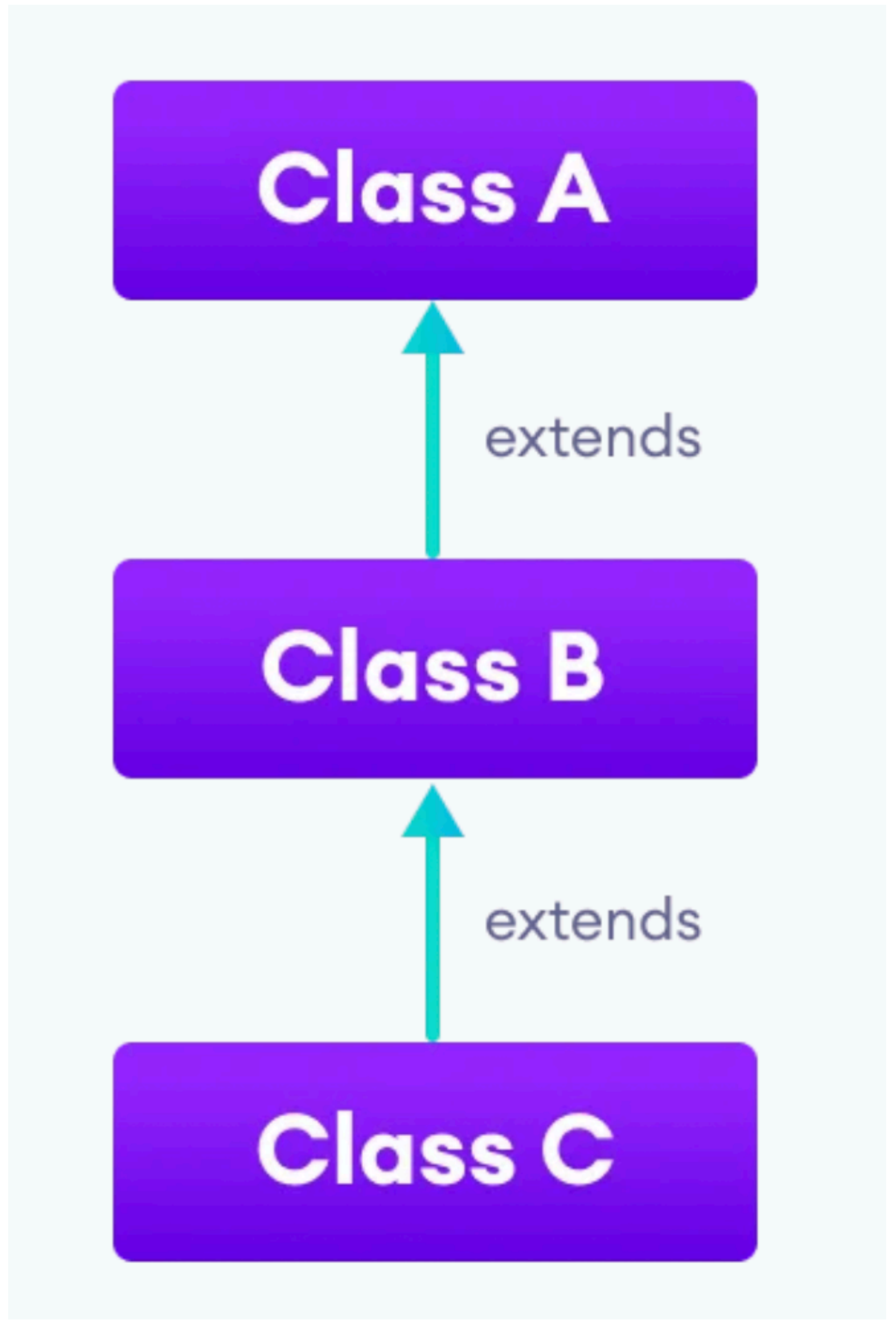
Single Inheritance

- A single subclass extends from a single superclass



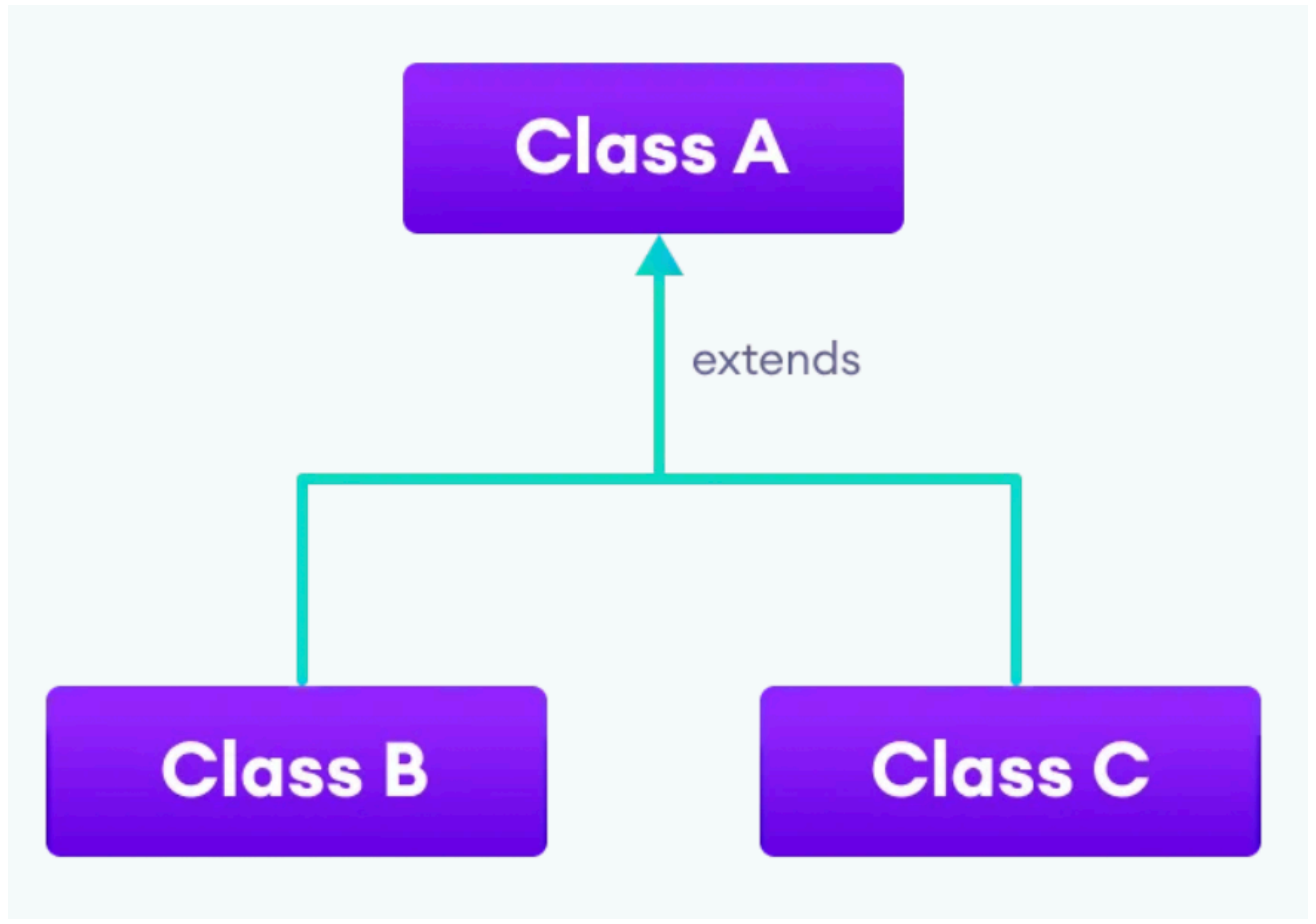
Multilevel Inheritance

- A subclass extends from a superclass and then the same subclass acts as a superclass for another class



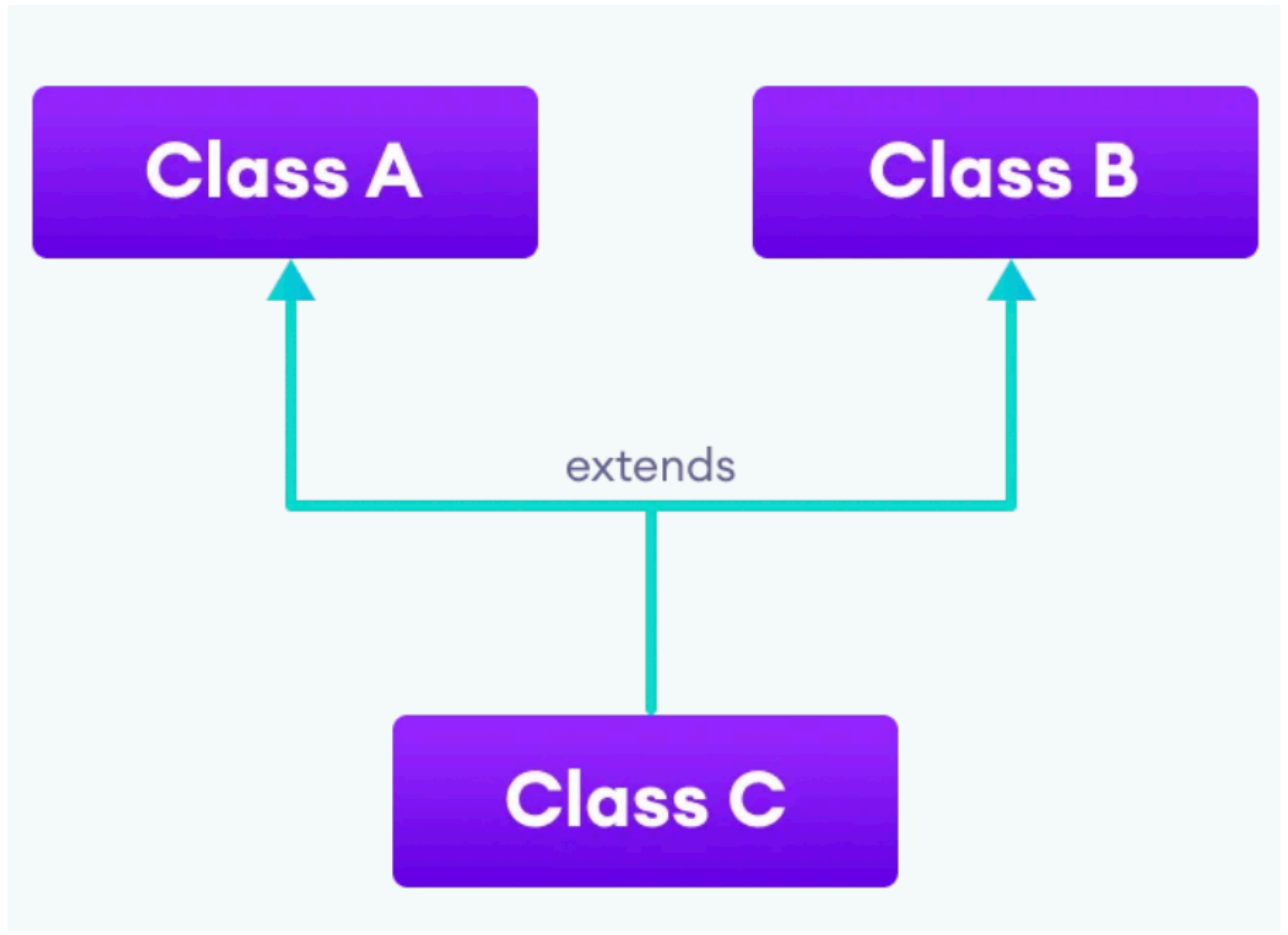
Hierarchical Inheritance

- Multiple subclasses extend from a single superclass



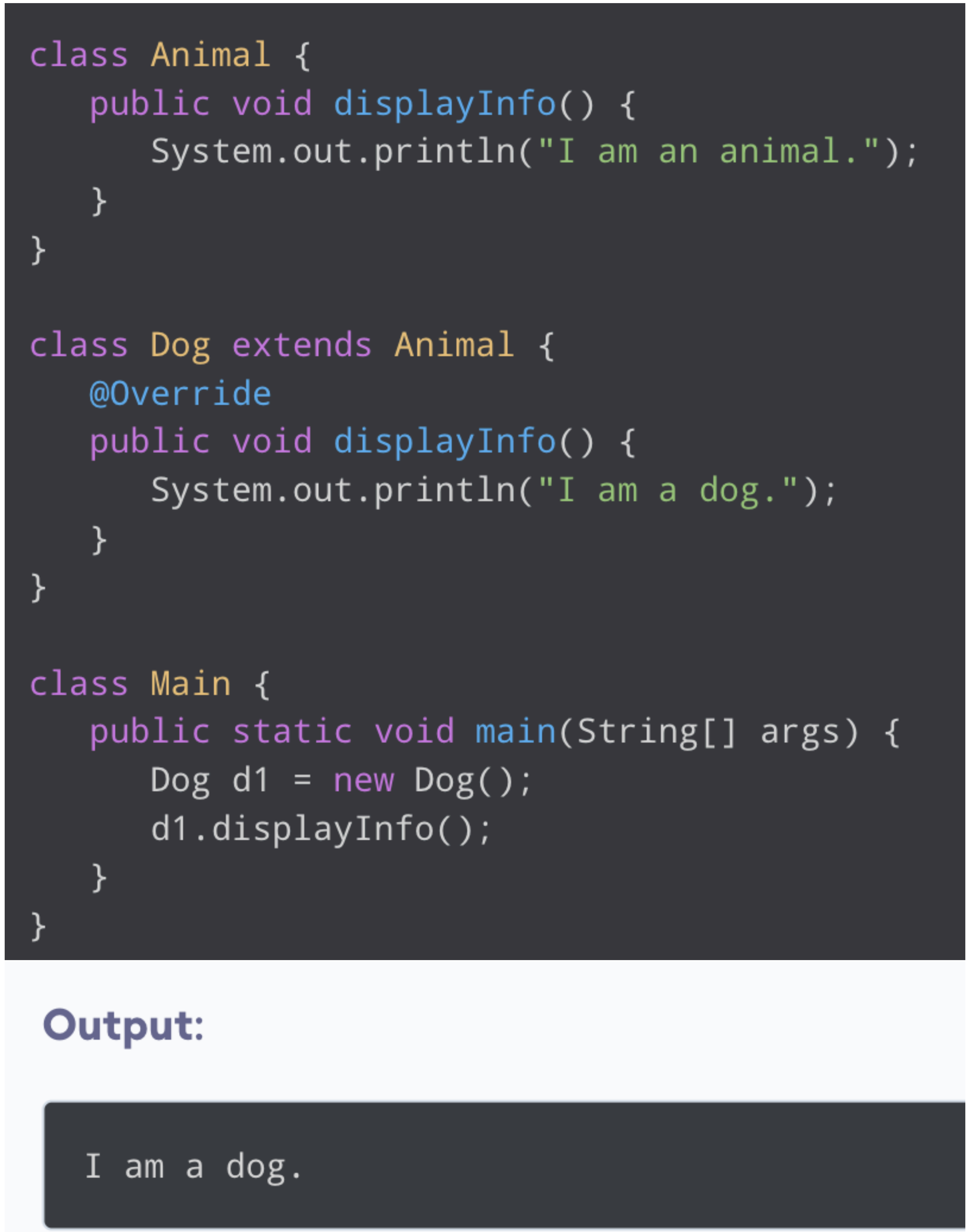
Multiple Inheritance

- A single subclass extends from multiple superclasses
- Java doesn’t support multiple inheritance, however, we can achieve this using interfaces (more on this later)



Method Overriding

- With inheritance, we can see that object of a subclass can access methods of the superclass
- What will happen if we have the same method in both superclass and subclass?
- If the same method is defined in both superclass and subclass, then the method of the subclass overrides the method of the superclass



```java
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am a dog.
```

Method Overriding

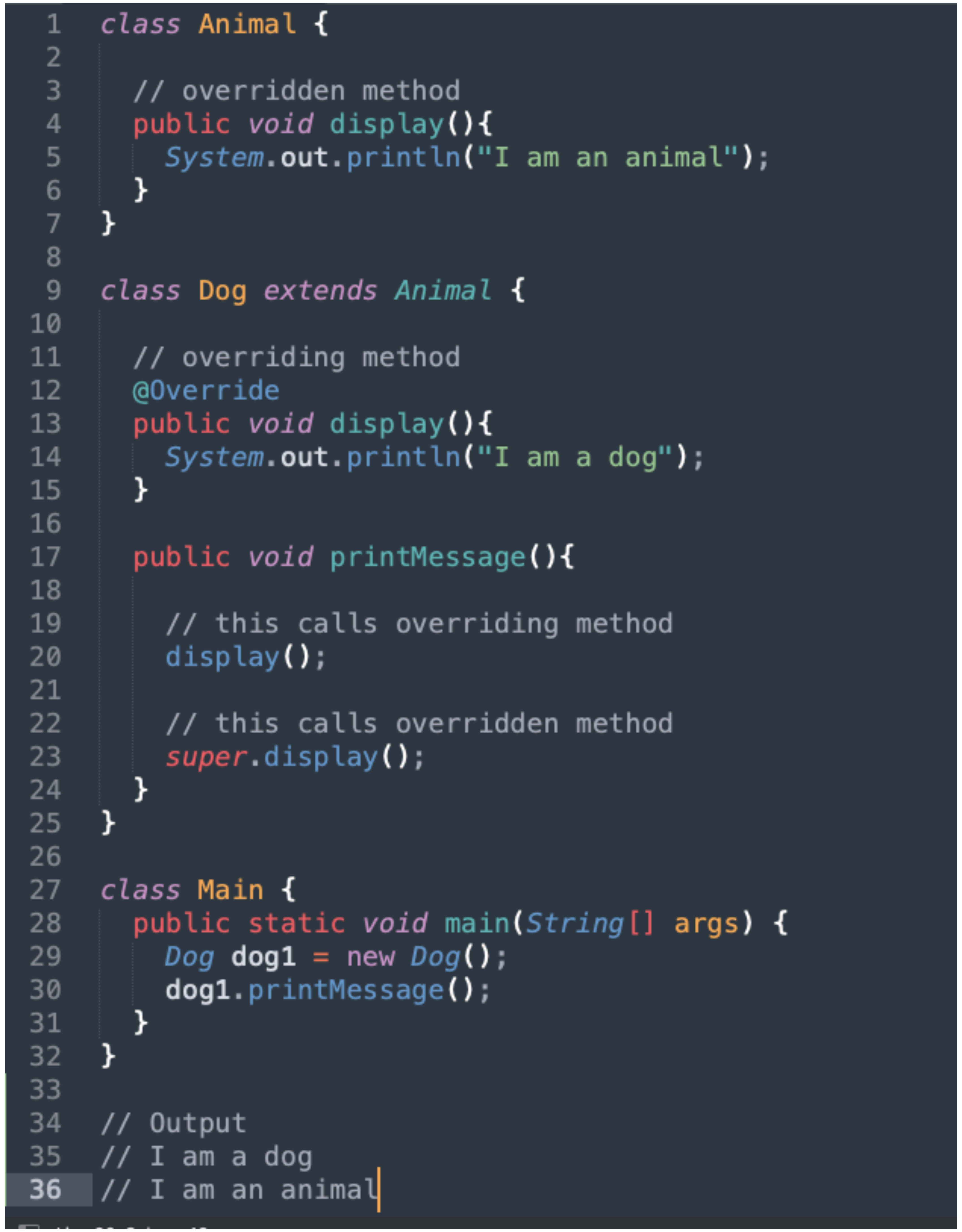
- Overriding rules:
  - Both superclass and subclass must have the same method name, the same return type and the same parameter list
  - We cannot override methods declared as `final` or `static`
- `@Override` annotation
  - Not mandatory to use, but it makes sure all the rules are followed
  - Otherwise, the compiler will generate an error

`super` **keyword**

- Used for accessing members of the parent class (attributes, constructors and methods) from the child class
- Use of super:
  - Access overridden methods of the superclass
  - Access attributes of the superclass
  - Access constructors of the superclass

Access overridden methods of the superclass

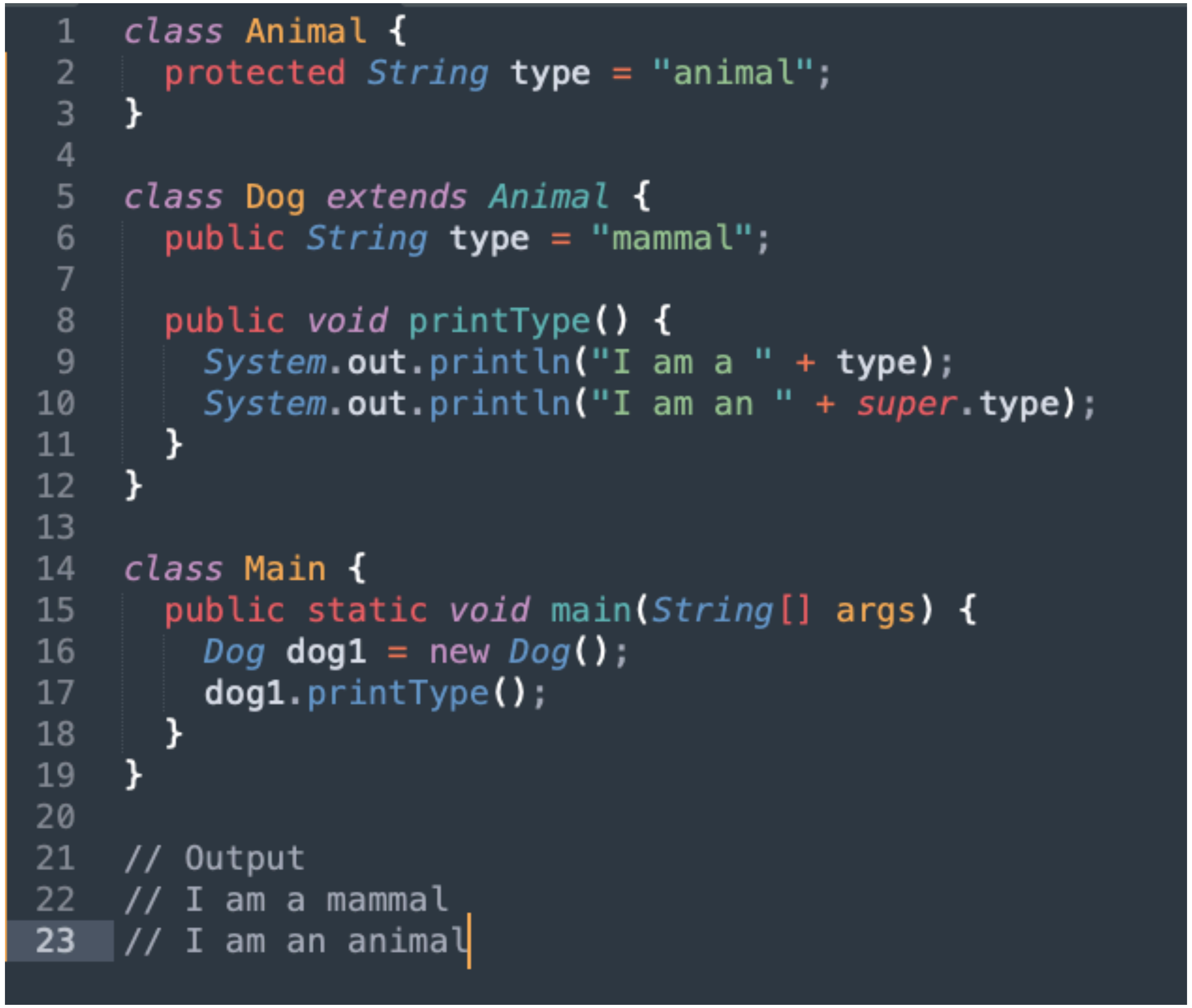
- If we have the same method declared in both superclass and subclass, the method in the subclass **overrides** the method in the superclass
- What if we want to call the method from the superclass instead of overridden subclass method?
- We can access it using `super` keyword



```java
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}

// Output
// I am a dog
// I am an animal
```

Access attributes of the superclass

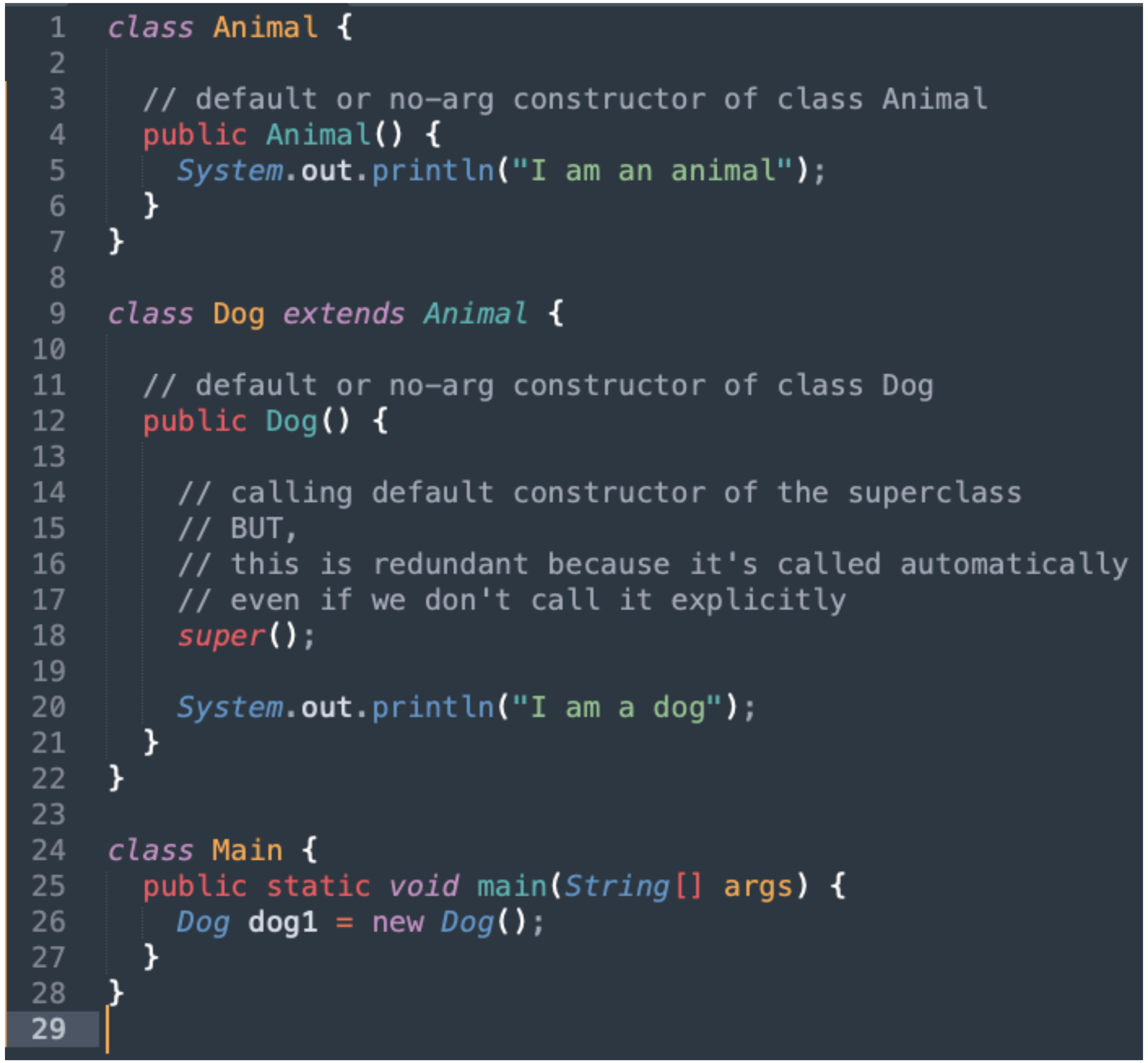
- The superclass and subclass can have attributes with the same name
- We can use `super` to access the attributes of the superclass



```java
class Animal {
  protected String type = "animal";
}

class Dog extends Animal {
  public String type = "mammal";

  public void printType() {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}

// Output
// I am a mammal
// I am an animal
```

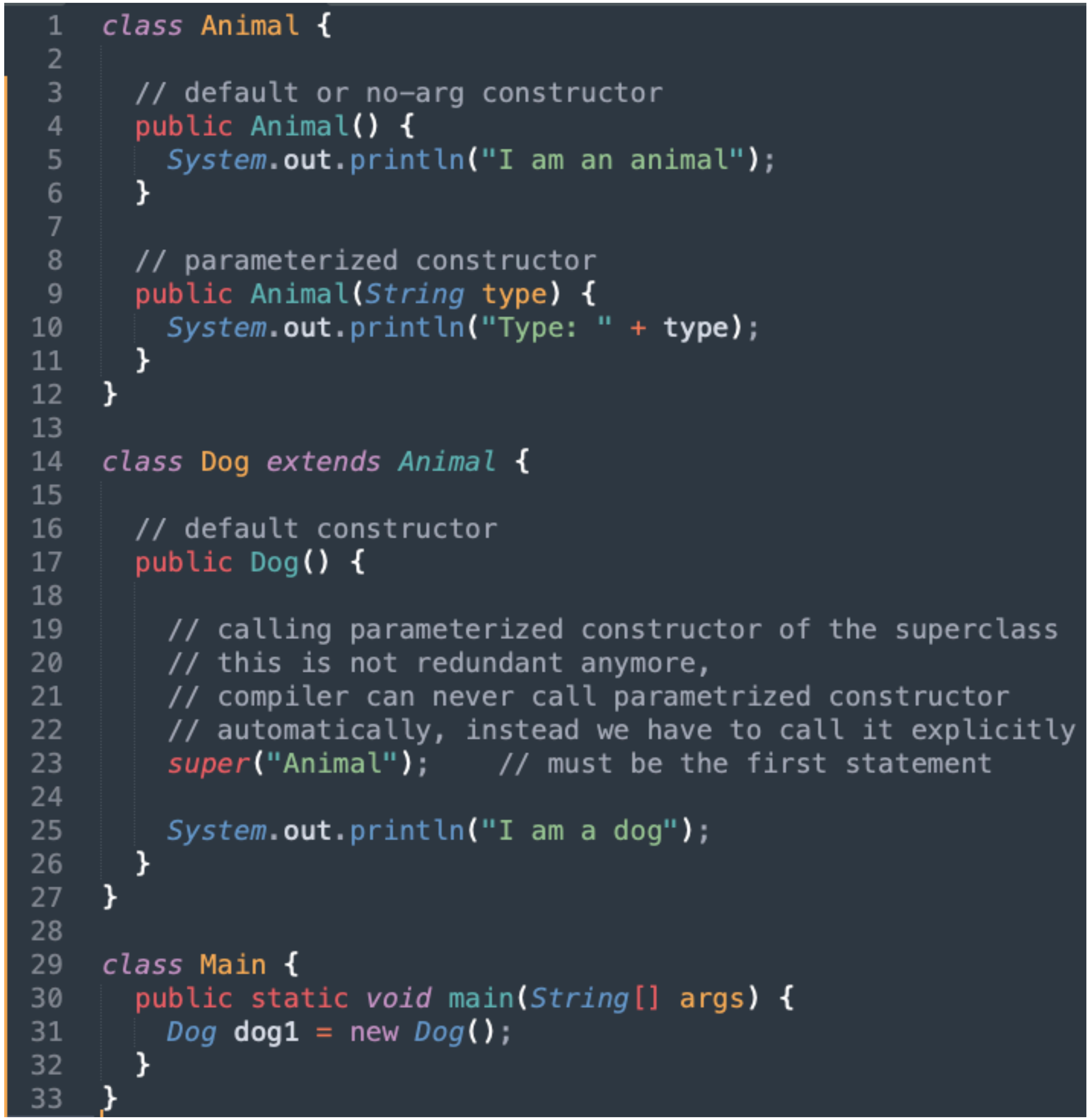
Access constructors of the superclass

- When an object of inherited class is instantiated, the default constructor of the superclass is called automatically
- What if we want to call parametrized constructor of the superclass instead?
- To explicitly call the constructor of the superclass, we use `super()`
- `super()` can be used only inside the subclass constructor and must be the first statement in it



```java
class Animal {

  // default or no-arg constructor of class Animal
  public Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // default or no-arg constructor of class Dog
  public Dog() {

    // calling default constructor of the superclass
    // BUT,
    // this is redundant because it's called automatically
    // even if we don't call it explicitly
    super();

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}

```

- What if we want to call parametrized constructor?



```java
class Animal {

  // default or no-arg constructor
  public Animal() {
    System.out.println("I am an animal");
  }

  // parameterized constructor
  public Animal(String type) {
    System.out.println("Type: " + type);
  }
}

class Dog extends Animal {

  // default constructor
  public Dog() {

    // calling parameterized constructor of the superclass
    // this is not redundant anymore,
    // compiler can never call parametrized constructor
    // automatically, instead we have to call it explicitly
    super("Animal");    // must be the first statement

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Exercise 1

- Let’s model Employees
- Every employee is a person and a person is defined with it’s first name, last name and JMBG
- Every employee is defined with it’s annual salary, a year employee started working (cannot be in the future or before 1900) and insurance number (some random string)
- In the main method, let’s create a few employees and print their information in the console (we want to override toString methods so we can print employees just by passing objects to `System.out.println(..)`)

Exercise 2

- Let’s model a Book entity
- Book has a name, a price, a year it was written, and an author
- Author has first name, last name and email address
- Every book has exactly one author (how are we gonna implement this?)
- When we print book object in the console, we want it to look like:
    - Ivo Andric - Na Drini Cuprija (1945) [890.0 RSD]
- Let’s create a few books in the main method and print them out