Override -vs- Overload

By: Sheldon Pasciak

REFERENCES:

Polymorphism and Overriding

https://github.com/SkillDistillery/SD43/tree/2c26e5069ad973f0828100d893549d35d861d575/java1/Polymorphism

Method Overloading

https://github.com/SkillDistillery/SD43/blob/2c26e5069ad973f0828100d893549d35d861d575/jfop/Methods/overloading.md

Override -vs- Overload

Student Presentations

Every programmer needs to be able to communicate about code and programming.
Each student will prepare and deliver a presentation to the rest of the class.
The presentation will cover an assigned, specific Java topic that comes up in test questions.
The topics have already been covered in class, so this presentation is a review and summary to help classmates answer related questions.

- You will become the class's *SME* (subject-matter expert) in this topic, and hopefully will learn something along the way.

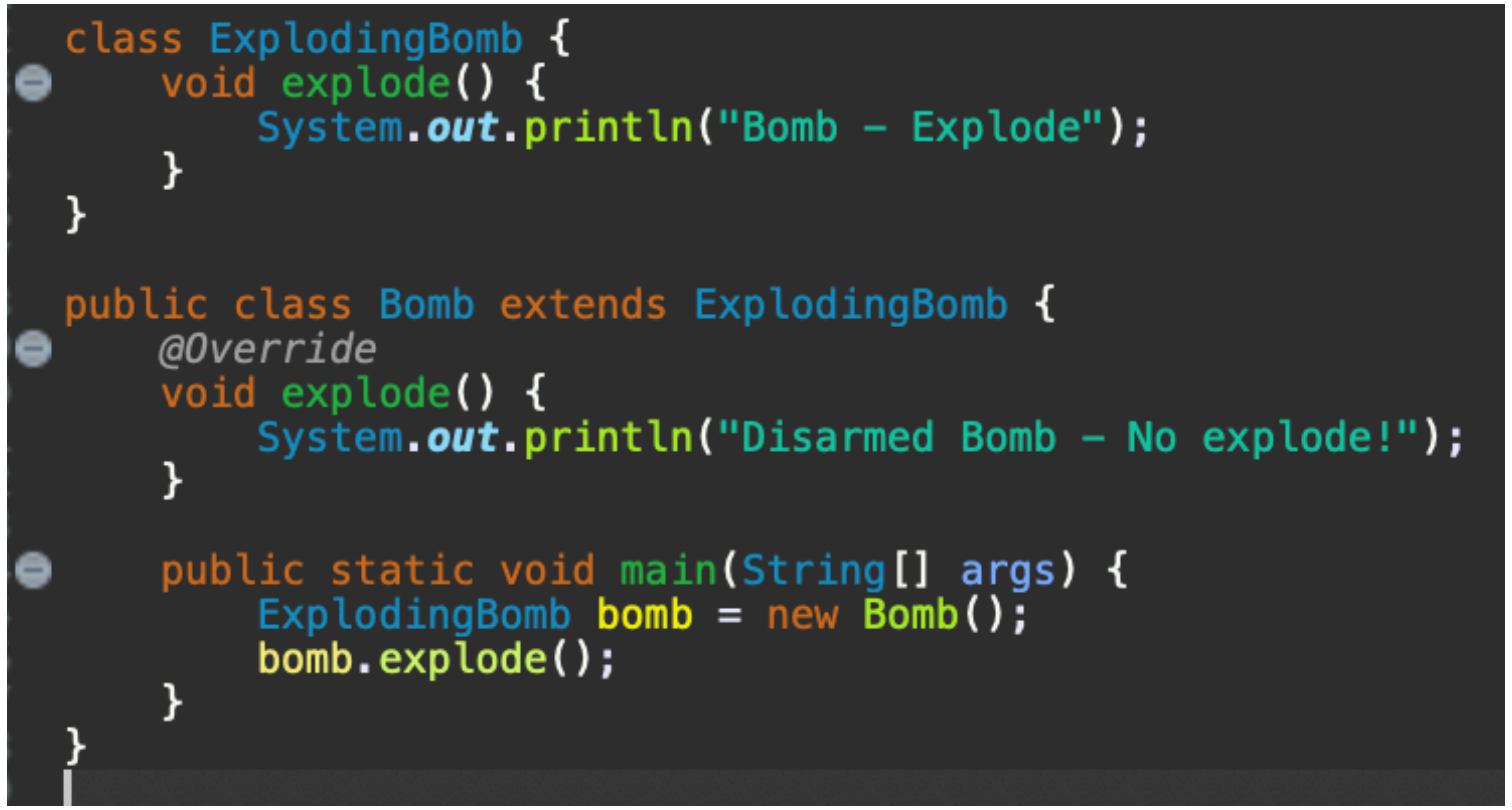
Requirements:

- You must create a prepared presentation in advance - Keynote slides, Google doc, etc. to guide your presentation.
- You must prepare running code in Eclipse to demonstrate one or more aspects of the topic.
- You must locate and present 1 - 3 (as part of your slides) examples of actual questions about the topic from the tests we've taken so far.
- Technical information as presented must be correct.

Target a presentation length of about 8-10 minutes - to do this, rehearse and time your presentation, and adjust as needed.

Override -vs- Overload

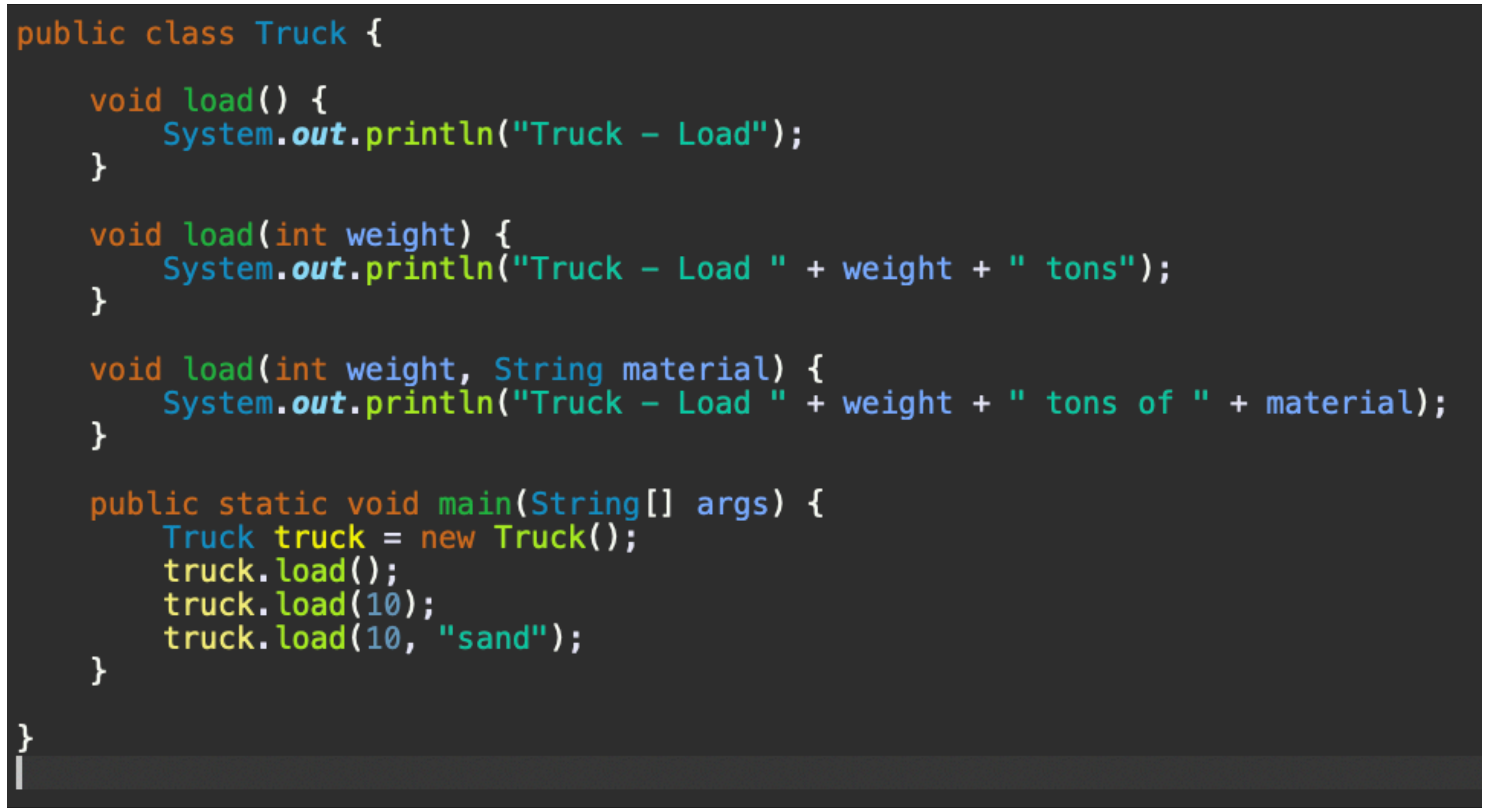
Override the Bomb so it doesn't Explode.



```java
class ExplodingBomb {
    void explode() {
        System.out.println("Bomb - Explode");
    }
}

public class Bomb extends ExplodingBomb {
    @Override
    void explode() {
        System.out.println("Disarmed Bomb - No explode!");
    }

    public static void main(String[] args) {
        ExplodingBomb bomb = new Bomb();
        bomb.explode();
    }
}
```

The ticking timer on the bomb is running… RUNTIME

Overload the Truck so it can load different things.



```java
public class Truck {

    void load() {
        System.out.println("Truck - Load");
    }

    void load(int weight) {
        System.out.println("Truck - Load " + weight + " tons");
    }

    void load(int weight, String material) {
        System.out.println("Truck - Load " + weight + " tons of " + material);
    }

    public static void main(String[] args) {
        Truck truck = new Truck();
        truck.load();
        truck.load(10);
        truck.load(10, "sand");
    }

}
```

The load for the truck is being compiled… COMPILE TIME

Override -vs- Overload

Runtime Polymorphism (Method Overriding): Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows objects of the subclass to be treated as objects of the superclass, but they can exhibit different behaviors based on their specific implementation.

Compile-time Polymorphism (Method Overloading): Method overloading allows a class to have multiple methods with the same name but different parameter lists. The Java compiler determines which method to call based on the number and types of arguments passed to the method.

The ticking timer on the bomb is running… RUNTIME

The load for the truck is being compiled… COMPILE TIME

Overloading

Overloading in Java refers to the ability to define multiple methods in the same class (or subclass) with the same name but different parameter lists (number, type, or order of parameters).

- Overloaded methods provide flexibility in method invocation based on the arguments passed.
- Overloaded methods are differentiated during compile-time based on the method signature.

Overloading

Calculator

```
public int addTwoIntegers(int num1, int num2) {}

public int addIntegers(int num1, int num2) {}

public double addTwoDecimals(double num1, double num2) {}

public int addThreeIntegers(int num1, int num2, int num3) {}

public double addThreeDecimals(double num1, double num2, double num3) {}
```

NOTE: All these methods have different names.

Overloading

Implemented by using different **method signatures**

In Java, method overloading is based on the **method signature**, which includes the method name and the parameter types, order and number of parameters, but not the return type.

```
public class Example {

   public int add(int a, int b) {
       return a + b;
   }

    public static void main(String[] args) {

        Example example = new Example();

        System.out.println(example.add(5, 3));

    }
}
```

Overloading

```
public class Calculator {

    // Method to add two integers
    public int addIntegers(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    public double addDoubles(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        System.out.println("Sum of 5 and 10: " + calculator.addIntegers(5, 10));

        System.out.println("Sum of 2.5 and 3.5: " + calculator.addDoubles(2.5, 3.5));
    }
}
```

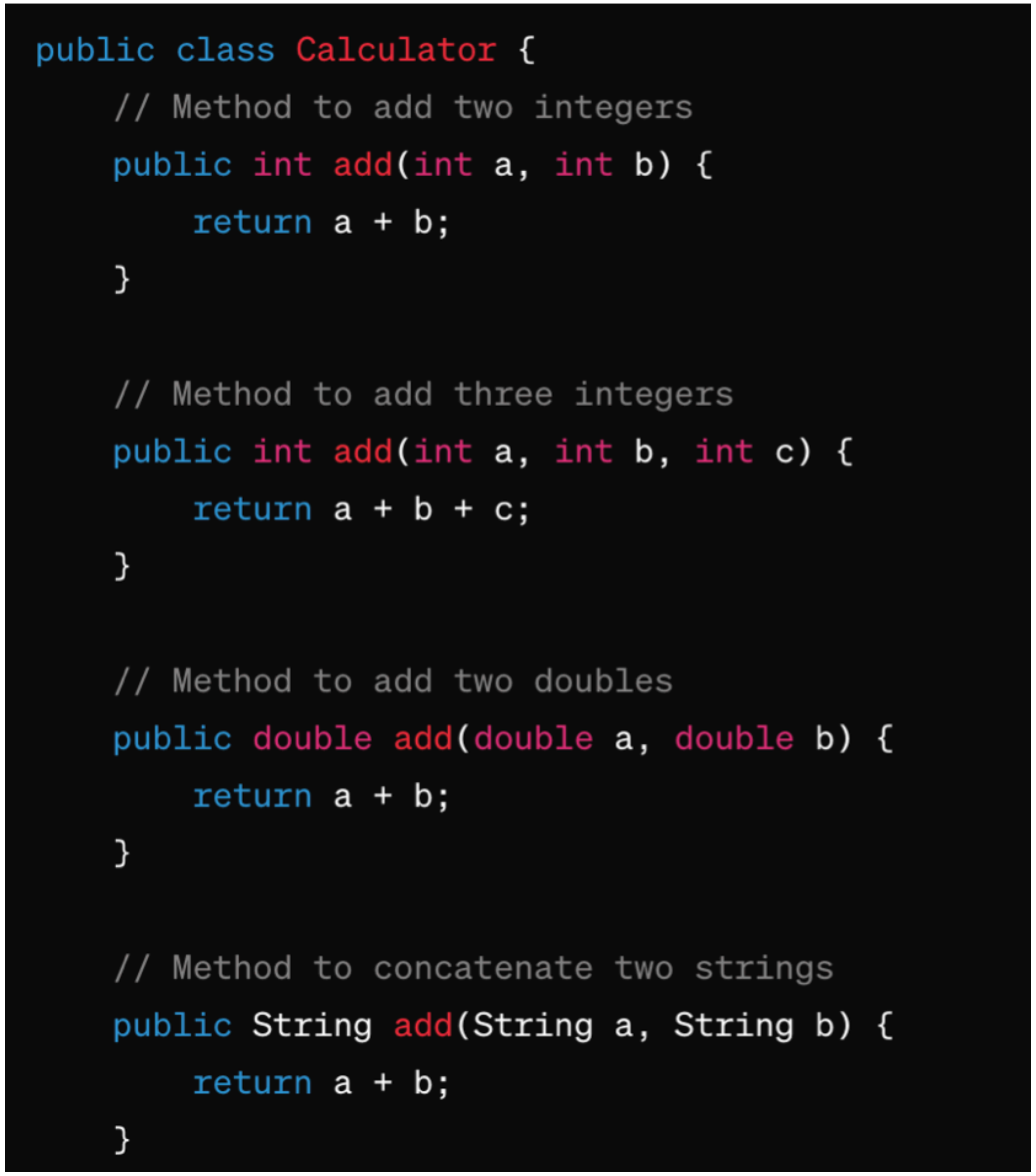
```
public class Calculator {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        System.out.println("Sum of 5 and 10: " + calculator.add(5, 10));

        System.out.println("Sum of 2.5 and 3.5: " + calculator.add(2.5, 3.5));
    }
}
```

Overloading

```
public class Calculator {

    // Method to add two integers
    public int addIntegers(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    public double addDoubles(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        System.out.println("Sum of 5 and 10: " + calculator.addIntegers(5, 10));

        System.out.println("Sum of 2.5 and 3.5: " + calculator.addDoubles(2.5, 3.5));
    }
}
```

```
public class Calculator {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        System.out.println("Sum of 5 and 10: " + calculator.add(5, 10));

        System.out.println("Sum of 2.5 and 3.5: " + calculator.add(2.5, 3.5));
    }
}
```

Overloading



```java
public class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    // Method to concatenate two strings
    public String add(String a, String b) {
        return a + b;
    }
```

Overloading

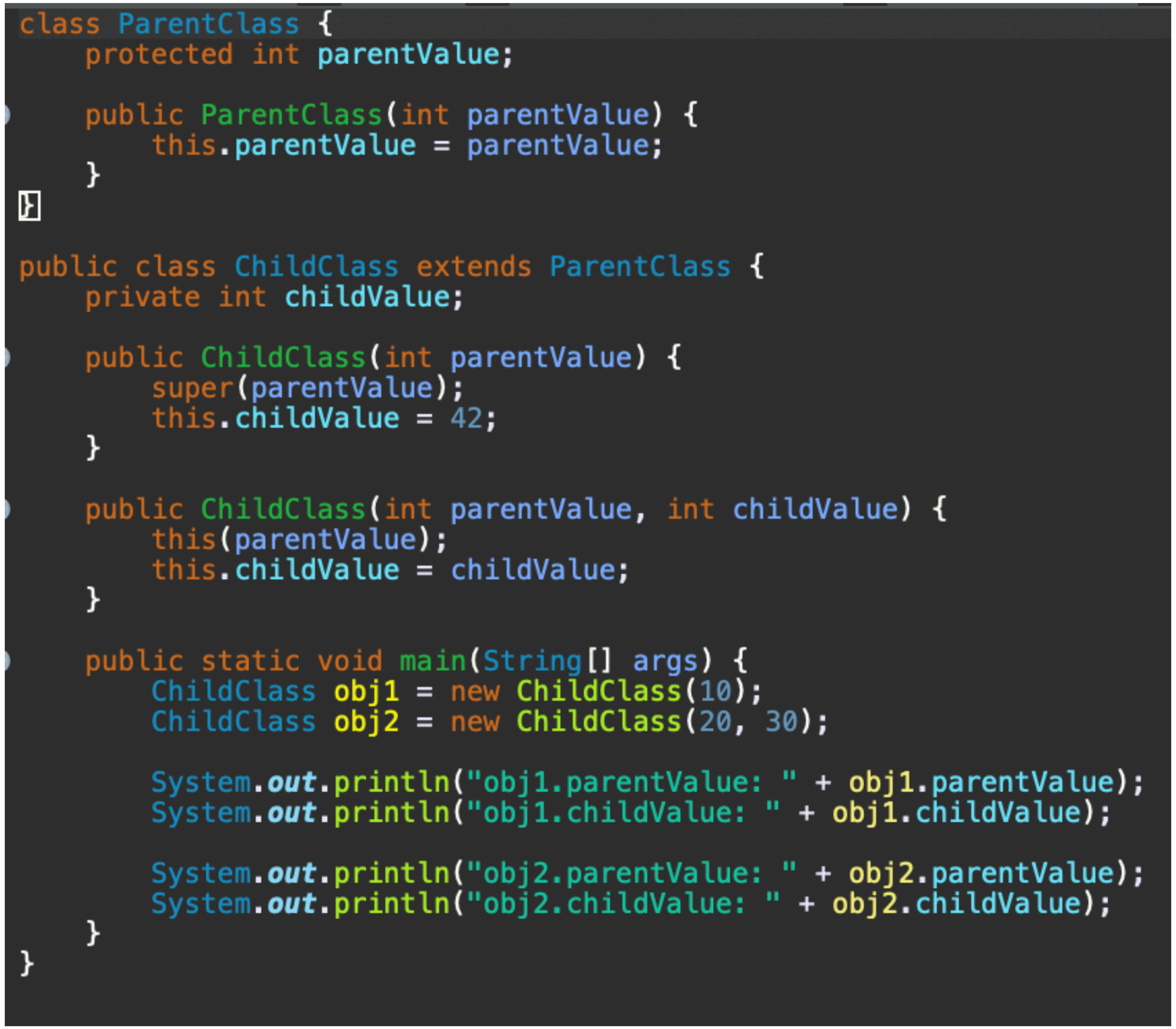
```
public class Calculator {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    // Method to concatenate two strings
    public String add(String a, String b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        // Calling different overloaded methods
        System.out.println("Sum of 5 and 10: " + calculator.add(5, 10));
        System.out.println("Sum of 5, 10, and 15: " + calculator.add(5, 10, 15));
        System.out.println("Sum of 2.5 and 3.5: " + calculator.add(2.5, 3.5));
        System.out.println("Concatenation of 'Hello' and 'World': " + calculator.add("Hello", "World"));
    }
}
```

Overloading in Java refers to the ability to define multiple methods in the same class with the same name but with different parameters.

This allows you to use the same method name for different behaviors depending on the arguments passed to it.

The combination of arguments, differing by both number, type and order is what makes up a unique method signature that is used to determine which method is called.

Overloading


```
class ParentClass {
    protected int parentValue;

    public ParentClass(int parentValue) {
        this.parentValue = parentValue;
    }
}

public class ChildClass extends ParentClass {
    private int childValue;

    public ChildClass(int parentValue) {
        super(parentValue);
        this.childValue = 42;
    }

    public ChildClass(int parentValue, int childValue) {
        this(parentValue);
        this.childValue = childValue;
    }

    public static void main(String[] args) {
        ChildClass obj1 = new ChildClass(10);
        ChildClass obj2 = new ChildClass(20, 30);

        System.out.println("obj1.parentValue: " + obj1.parentValue);
        System.out.println("obj1.childValue: " + obj1.childValue);

        System.out.println("obj2.parentValue: " + obj2.parentValue);
        System.out.println("obj2.childValue: " + obj2.childValue);
    }
}
```

Overloading

```
class ParentClass {
    protected int parentValue;

    public ParentClass(int parentValue) {
        this.parentValue = parentValue;
    }
}

public class ChildClass extends ParentClass {
    private int childValue;

   public ChildClass(int parentValue) {
        super(parentValue);
        this.childValue = 42;
    }

   public ChildClass(int parentValue, int childValue) {
        this(parentValue);
        this.childValue = childValue;
    }

    public static void main(String[] args) {
       ChildClass obj1 = new ChildClass(10);
        ChildClass obj2 = new ChildClass(20, 30);
    }
}
```

- **Flexibility in Object Initialization**: Constructor overloading allows objects of a class to be initialized in different ways.
- **Improved Readability and Maintainability**: By providing multiple constructors with different parameter lists, the intent of object initialization becomes clearer. This enhances code readability as developers can easily identify which constructor is being used based on the arguments passed.
- **Reduction of Repetitive Code**: Constructor overloading helps in avoiding code duplication by allowing common initialization logic to be encapsulated in one or more constructors.
- **Enables Default Parameter Values**: Through constructor overloading, default parameter values can be provided for optional arguments.
- **Supports Polymorphism**: Constructor overloading is an example of polymorphism, where multiple constructors can have the same name but differ in the number or types of parameters they accept.

Overall, constructor overloading in Java enhances code flexibility, readability, and maintainability while promoting code reuse and supporting object initialization in diverse scenarios.

Overloading

```
class ParentClass {
    protected int parentValue;

    public ParentClass(int parentValue) {
        this.parentValue = parentValue;
    }
}

public class ChildClass extends ParentClass {
    private int childValue;

   public ChildClass(int parentValue) {
        super(parentValue);
        this.childValue = 42;
    }

   public ChildClass(int parentValue, int childValue) {
        this(parentValue);
        this.childValue = childValue;
    }

    public static void main(String[] args) {
       ChildClass obj1 = new ChildClass(10);
        ChildClass obj2 = new ChildClass(20, 30);
    }
}
```

- **Flexibility in Object Initialization**: Constructor overloading allows objects of a class to be initialized in different ways.
- **Improved Readability and Maintainability**: By providing multiple constructors with different parameter lists, the intent of object initialization becomes clearer. This enhances code readability as developers can easily identify which constructor is being used based on the arguments passed.
- **Reduction of Repetitive Code**: Constructor overloading helps in avoiding code duplication by allowing common initialization logic to be encapsulated in one or more constructors.
- **Enables Default Parameter Values**: Through constructor overloading, default parameter values can be provided for optional arguments.
- **Supports Polymorphism**: Constructor overloading is an example of polymorphism, where multiple constructors can have the same name but differ in the number or types of parameters they accept.

Overall, constructor overloading in Java enhances code flexibility, readability, and maintainability while promoting code reuse and supporting object initialization in diverse scenarios.

Overloading

```
class ParentClass {
    protected int parentValue;

    public ParentClass(int parentValue) {
        this.parentValue = parentValue;
    }
}

public class ChildClass extends ParentClass {
    private int childValue;

   public ChildClass(int parentValue) {
        super(parentValue);
        this.childValue = 42;
    }

   public ChildClass(int parentValue, int childValue) {
        this(parentValue);
        this.childValue = childValue;
    }

    public static void main(String[] args) {
       ChildClass obj1 = new ChildClass(10);
        ChildClass obj2 = new ChildClass(20, 30);
    }
}
```

- **Flexibility in Object Initialization**: Constructor overloading allows objects of a class to be initialized in different ways.
- **Improved Readability and Maintainability**: By providing multiple constructors with different parameter lists, the intent of object initialization becomes clearer. This enhances code readability as developers can easily identify which constructor is being used based on the arguments passed.
- **Reduction of Repetitive Code**: Constructor overloading helps in avoiding code duplication by allowing common initialization logic to be encapsulated in one or more constructors.
- **Enables Default Parameter Values**: Through constructor overloading, default parameter values can be provided for optional arguments.
- **Supports Polymorphism**: Constructor overloading is an example of polymorphism, where multiple constructors can have the same name but differ in the number or types of parameters they accept.

Overall, constructor overloading in Java enhances code flexibility, readability, and maintainability while promoting code reuse and supporting object initialization in diverse scenarios.

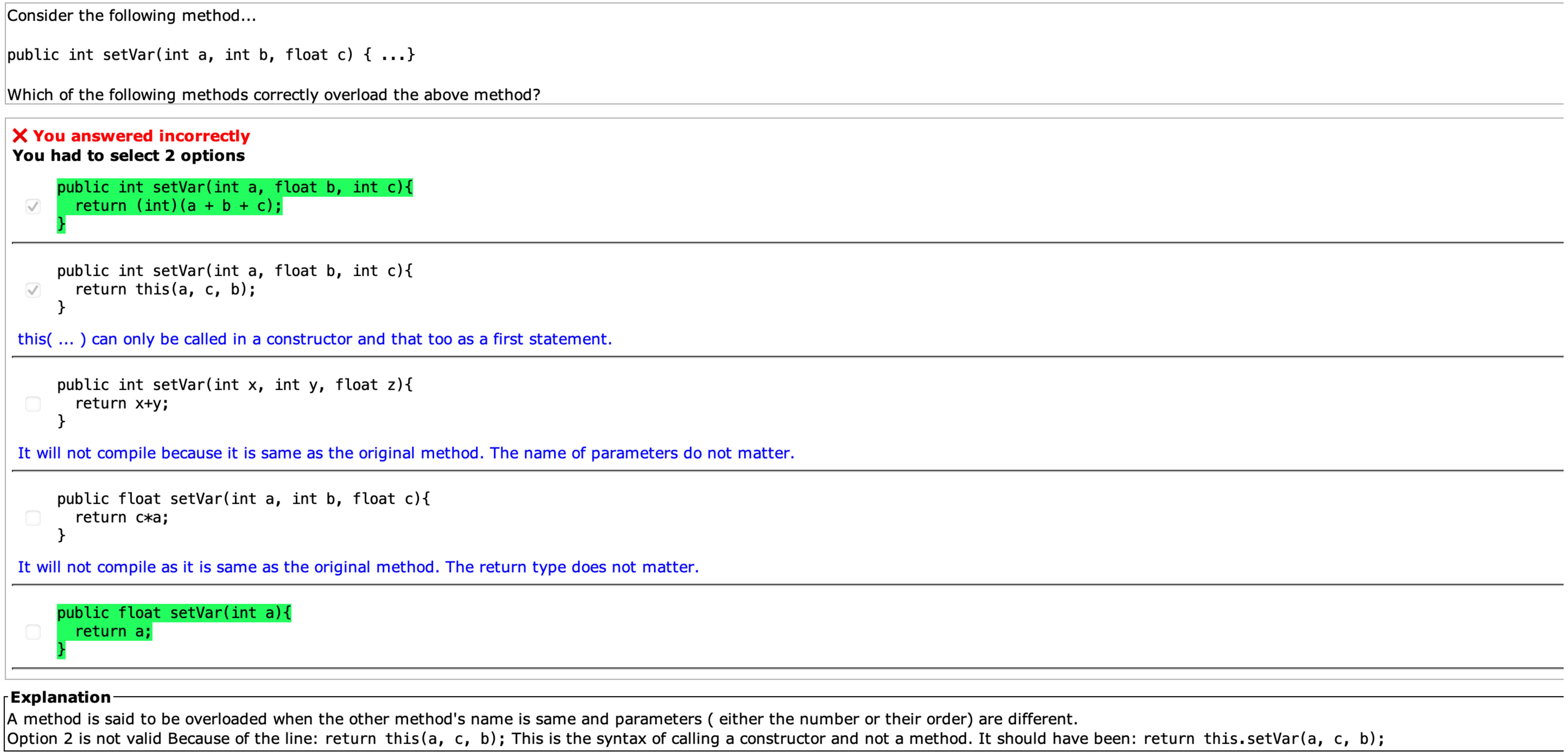
Overloading

```
class ParentClass {
    protected int parentValue;

    public ParentClass(int parentValue) {
        this.parentValue = parentValue;
    }
}

public class ChildClass extends ParentClass {
    private int childValue;

   public ChildClass(int parentValue) {
        super(parentValue);
        this.childValue = 42;
    }

   public ChildClass(int parentValue, int childValue) {
        this(parentValue);
        this.childValue = childValue;
    }

    public static void main(String[] args) {
       ChildClass obj1 = new ChildClass(10);
        ChildClass obj2 = new ChildClass(20, 30);
    }
}
```

- **Flexibility in Object Initialization**: Constructor overloading allows objects of a class to be initialized in different ways.
- **Improved Readability and Maintainability**: By providing multiple constructors with different parameter lists, the intent of object initialization becomes clearer. This enhances code readability as developers can easily identify which constructor is being used based on the arguments passed.
- **Reduction of Repetitive Code**: Constructor overloading helps in avoiding code duplication by allowing common initialization logic to be encapsulated in one or more constructors.
- **Enables Default Parameter Values**: Through constructor overloading, default parameter values can be provided for optional arguments.
- **Supports Polymorphism**: Constructor overloading is an example of polymorphism, where multiple constructors can have the same name but differ in the number or types of parameters they accept.

Overall, constructor overloading in Java enhances code flexibility, readability, and maintainability while promoting code reuse and supporting object initialization in diverse scenarios.

Overloading



Consider the following method...

```
public int setVar(int a, int b, float c) { ...}
```

Which of the following methods correctly overload the above method?

❌ **You answered incorrectly**

You had to select 2 options

[✓]
```
public int setVar(int a, float b, int c){
  return (int)(a + b + c);
}
```

[✓]
```
public int setVar(int a, float b, int c){
  return this(a, c, b);
}
```

this(...) can only be called in a constructor and that too as a first statement.

```
public int setVar(int x, int y, float z){
  return x+y;
}
```

It will not compile because it is same as the original method. The name of parameters do not matter.

```
public float setVar(int a, int b, float c){
  return c*a;
}
```

It will not compile as it is same as the original method. The return type does not matter.

```
public float setVar(int a){
  return a;
}
```

Explanation

A method is said to be overloaded when the other method's name is same and parameters (either the number or their order) are different.
Option 2 is not valid Because of the line: `return this(a, c, b);` This is the syntax of calling a constructor and not a method. It should have been: `return this.setVar(a, c, b);`

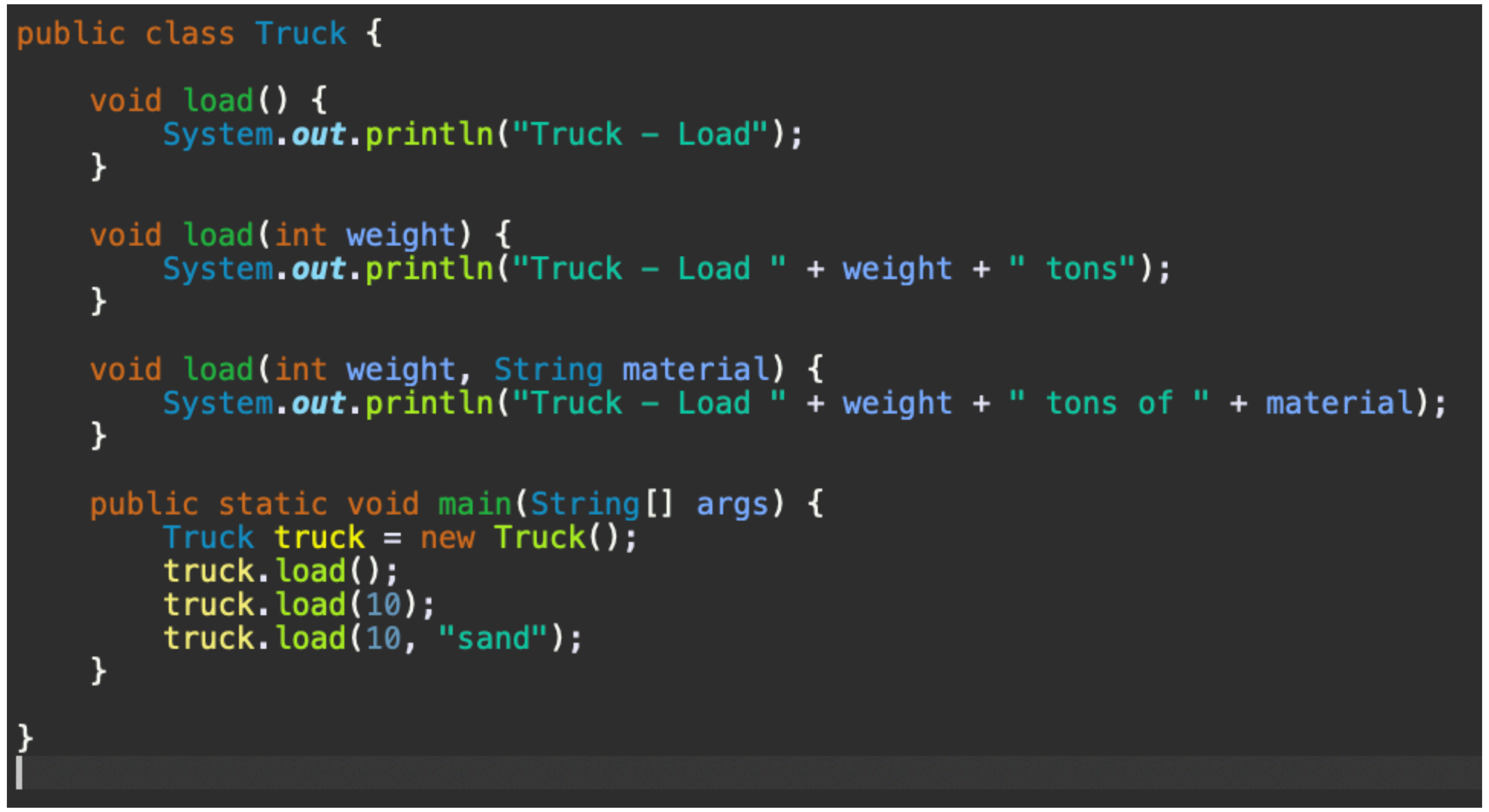
Overload

Overloading in Java refers to the ability to define multiple methods in the same class with the same name but with different parameters.

This allows you to use the same method name for different behaviors depending on the arguments passed to it.

The combination of arguments, differing by both number, type and order is what makes up a unique method signature that is used to determine which method is called.

Overload the Truck so it can load different things.

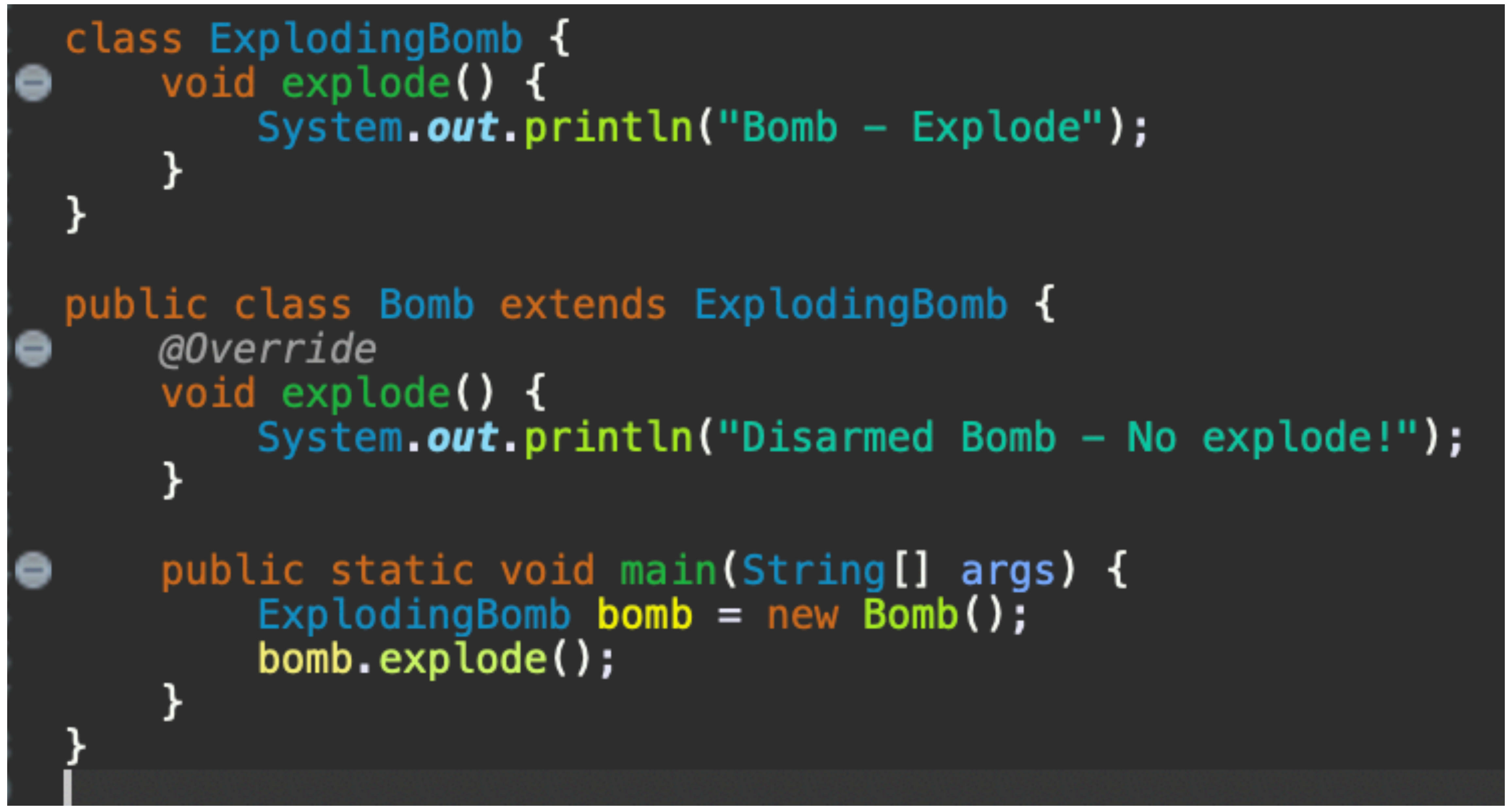


```
public class Truck {

    void load() {
        System.out.println("Truck - Load");
    }

    void load(int weight) {
        System.out.println("Truck - Load " + weight + " tons");
    }

    void load(int weight, String material) {
        System.out.println("Truck - Load " + weight + " tons of " + material);
    }

    public static void main(String[] args) {
        Truck truck = new Truck();
        truck.load();
        truck.load(10);
        truck.load(10, "sand");
    }

}
```

The load for the truck is being compiled… COMPILE TIME

Override

Override the Bomb so it doesn't Explode.



```
class ExplodingBomb {
    void explode() {
        System.out.println("Bomb - Explode");
    }
}

public class Bomb extends ExplodingBomb {
    @Override
    void explode() {
        System.out.println("Disarmed Bomb - No explode!");
    }

    public static void main(String[] args) {
        ExplodingBomb bomb = new Bomb();
        bomb.explode();
    }
}
```

The ticking timer on the bomb is running… RUNTIME

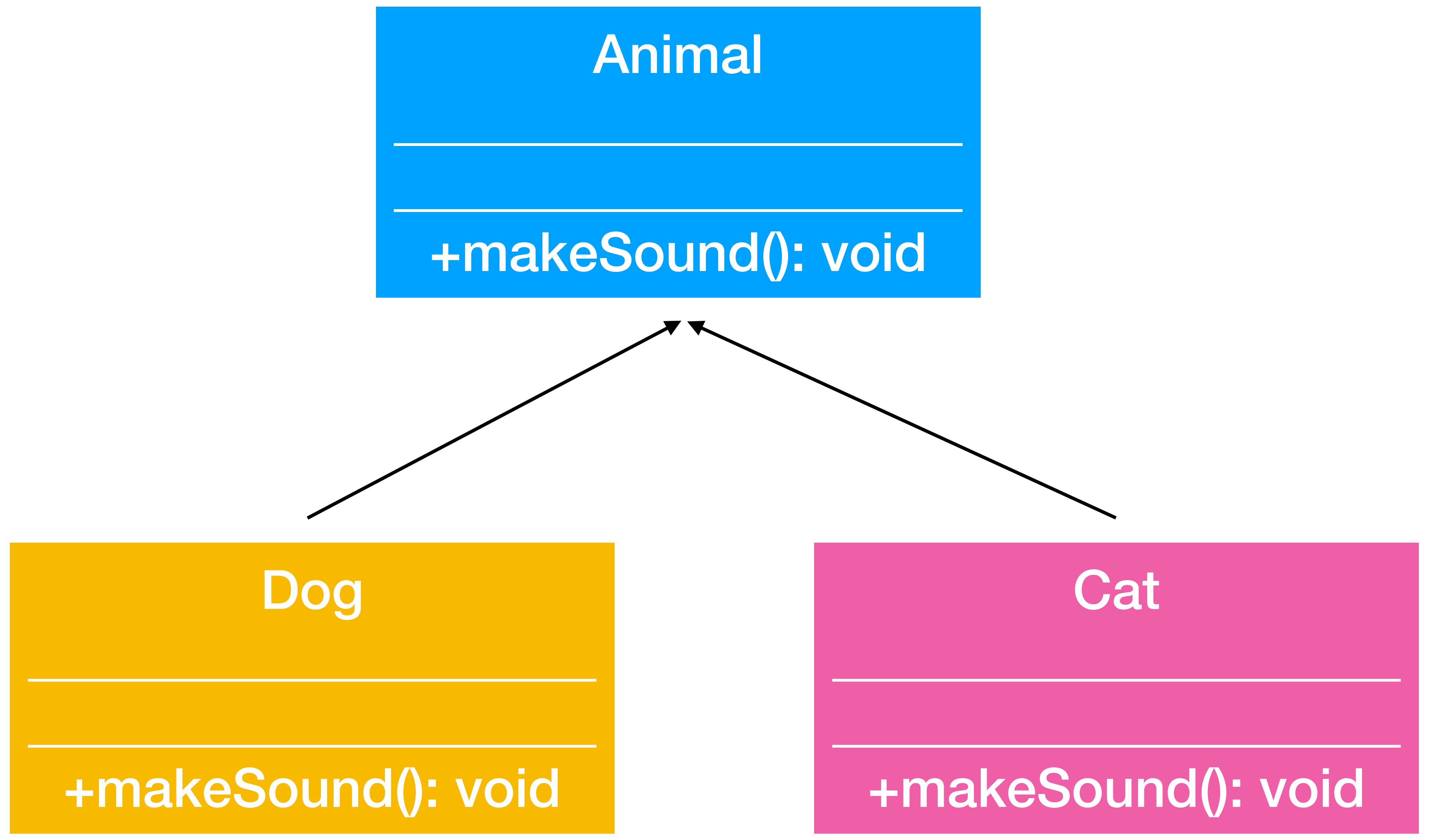
What's the purpose of overriding ?

Anthony - "So we can have children that have modified behaviors of the parent class."

Runtime Polymorphism (Method Overriding): Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows objects of the subclass to be treated as objects of the superclass, but they can exhibit different behaviors based on their specific implementation.

```java
class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        // Calls overridden methods
        animal1.makeSound(); // Output: Bark
        animal2.makeSound(); // Output: Meow
    }
}
```

Override



Method **overriding** occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

This allows a subclass to provide a specialized version of a method that is tailored to its own behavior.

Override

Rules for Overriding

The compiler will perform checks to make sure our overrides are legal.

1. The method must have the **same signature** (name and parameter types, in order) as the method in the parent class.

2. The method must be **at least as visible** or more visible than the parent class's method.

3. If the method returns a value, the **return type must be the same as or a subclass of** the return type of the method in the parent class (known as *covariant return types*).

(Note: We do not have to use @Override for the compiler to check these.)

Reference:

https://github.com/SkillDistillery/SD43/blob/2c26e5069ad973f0828100d893549d35d861d575/java1/Polymorphism/override-rules.md

Override

(Note: We do not have to use @Override for the compiler to check these.)

Methods marked with @Override but don't actually override a parent method will not compile.

```
class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

```
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
   @Override
    public void doMore() {
        System.out.println("Extra");
    }
}
```

Must override or implement a super-type method

Reference:

https://github.com/SkillDistillery/SD43/blob/2c26e5069ad973f0828100d893549d35d861d575/java1/Polymorphism/override-rules.md

Override

In Java, the `@Override` annotation is used to indicate that a method is being overridden from a superclass. This annotation is optional but highly recommended because it helps improve code readability and maintainability by explicitly stating that a method is intended to override a method in a superclass.

What: You use the `@Override` annotation to indicate that a method in a subclass is overriding a method in its superclass.

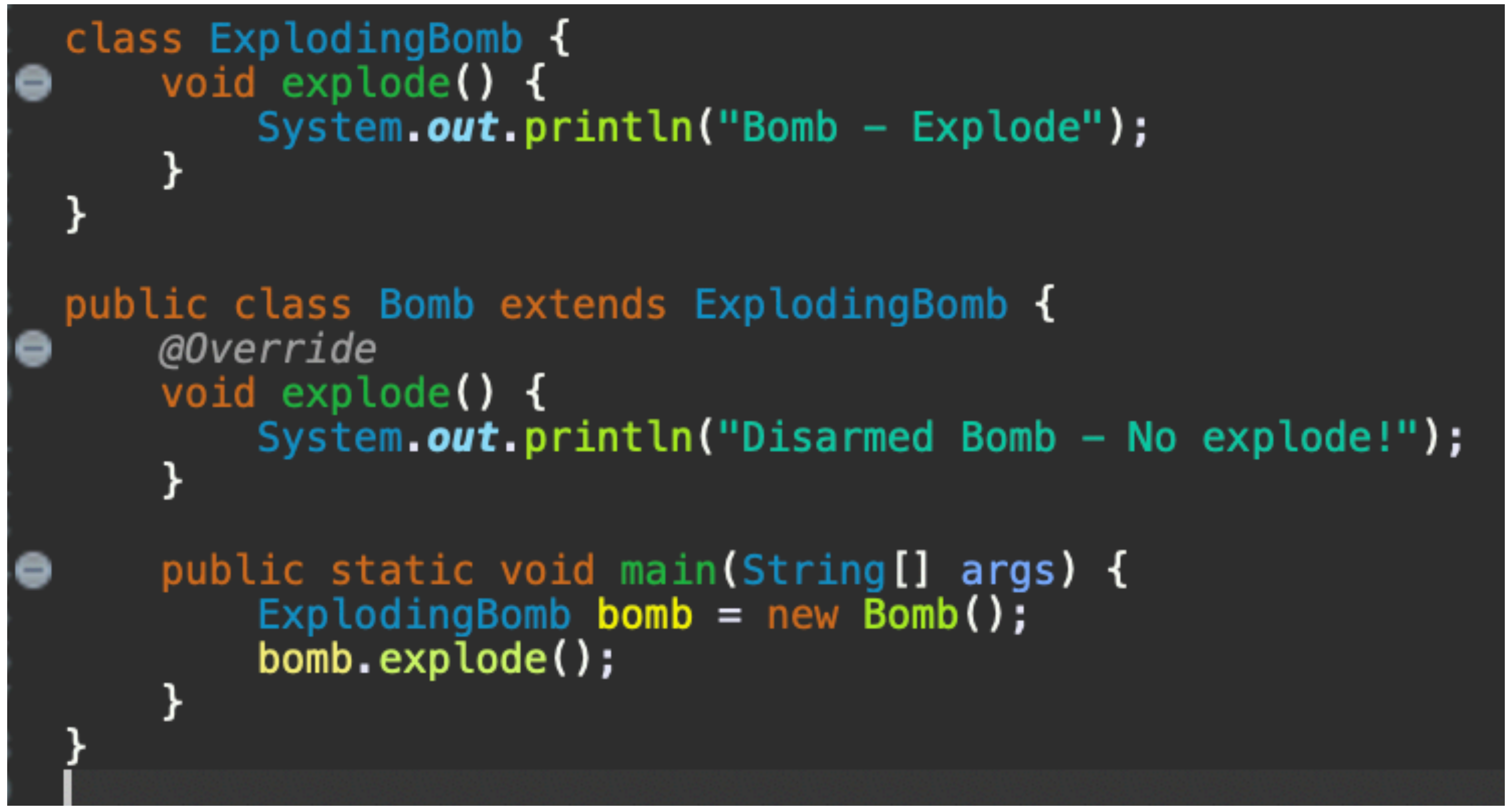
When: You should use `@Override` whenever you intend to override a method from a superclass.

Why: This helps prevent accidental method name changes or changes to the method signature in the superclass, which might not be detected by the compiler.

Where: You place the `@Override` annotation directly before the method you're overriding in the subclass.

Override

Override the Bomb so it doesn't Explode.



```
class ExplodingBomb {
    void explode() {
        System.out.println("Bomb - Explode");
    }
}

public class Bomb extends ExplodingBomb {
    @Override
    void explode() {
        System.out.println("Disarmed Bomb - No explode!");
    }

    public static void main(String[] args) {
        ExplodingBomb bomb = new Bomb();
        bomb.explode();
    }
}
```

The ticking timer on the bomb is running… RUNTIME

Overriding in Java occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

When you override a method, you maintain the same method signature (name, parameters, return type) as the method in the superclass.

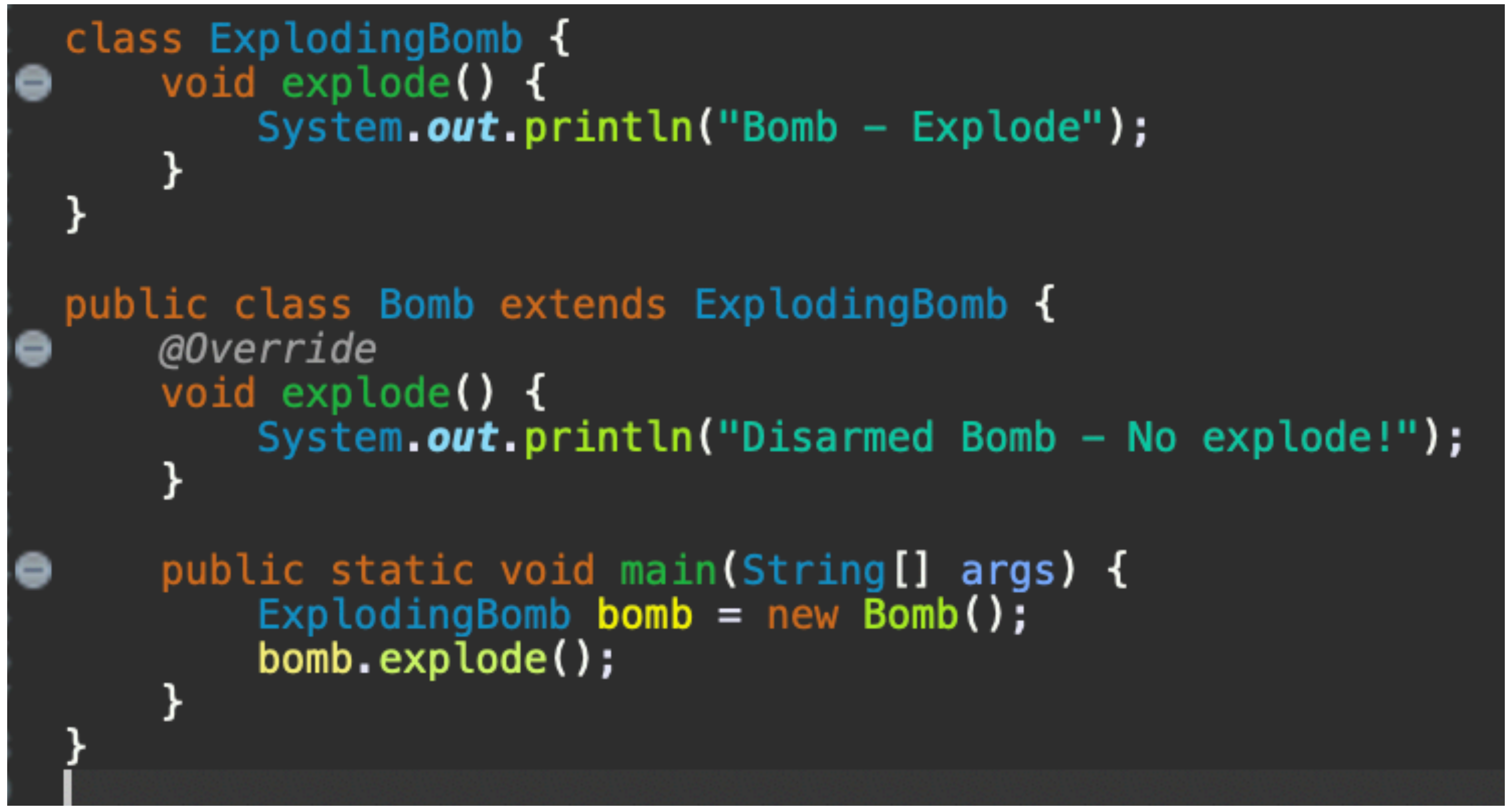
The purpose of overriding is to provide a specialized implementation of a method in a subclass, which is more appropriate for the subclass's context.

Method overriding is essential for achieving polymorphism in Java, where different objects can be treated uniformly through a common interface or superclass reference.

SUMMARY

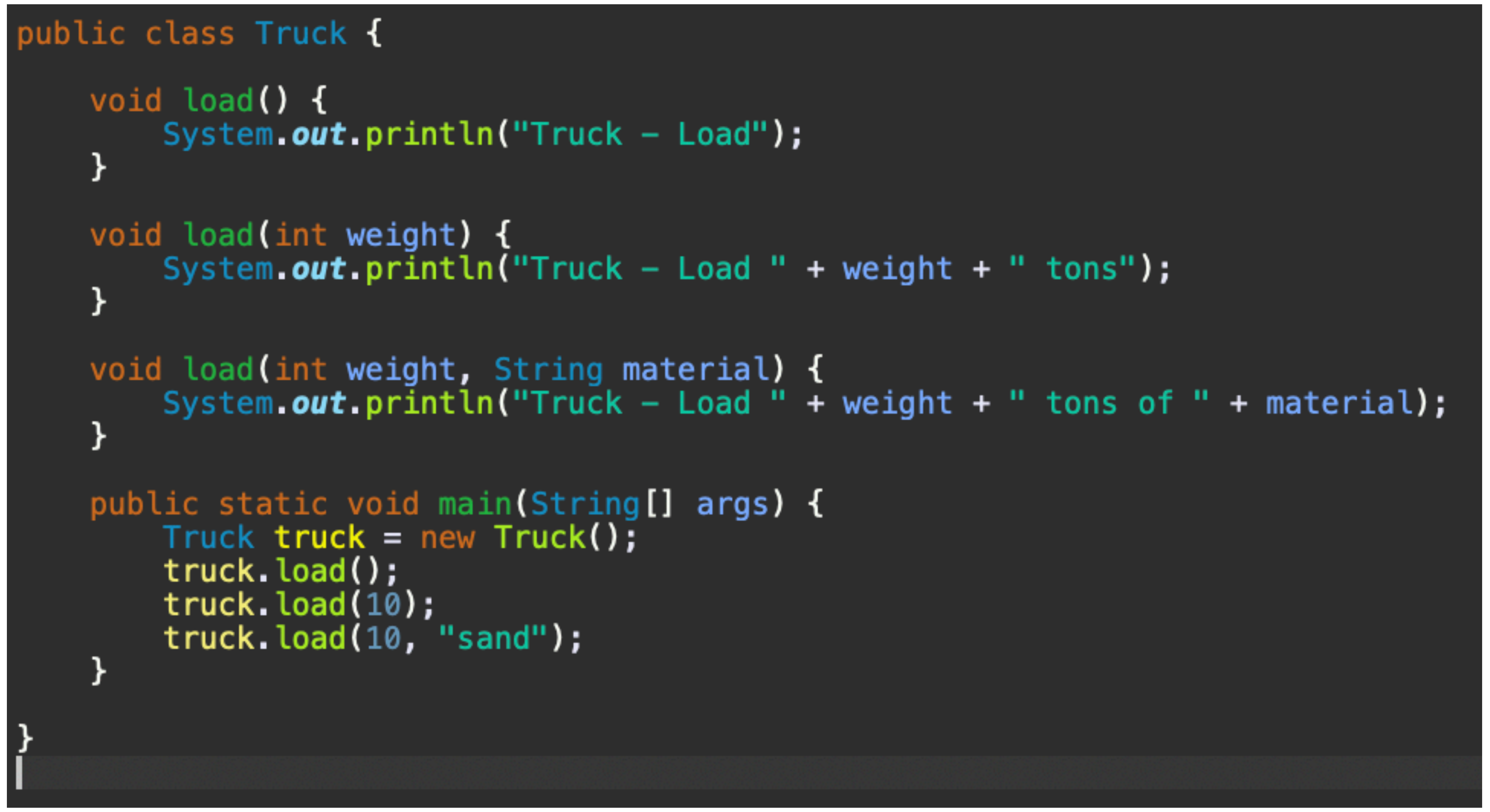
Override -vs- Overload

Override the Bomb so it doesn't Explode.



```
class ExplodingBomb {
    void explode() {
        System.out.println("Bomb - Explode");
    }
}

public class Bomb extends ExplodingBomb {
    @Override
    void explode() {
        System.out.println("Disarmed Bomb - No explode!");
    }

    public static void main(String[] args) {
        ExplodingBomb bomb = new Bomb();
        bomb.explode();
    }
}
```

Overload the Truck so it can load different things.



```
public class Truck {

    void load() {
        System.out.println("Truck - Load");
    }

    void load(int weight) {
        System.out.println("Truck - Load " + weight + " tons");
    }

    void load(int weight, String material) {
        System.out.println("Truck - Load " + weight + " tons of " + material);
    }

    public static void main(String[] args) {
        Truck truck = new Truck();
        truck.load();
        truck.load(10);
        truck.load(10, "sand");
    }

}
```

In summary, overriding is about providing a new implementation for a method inherited from a superclass, while overloading is about having multiple methods with the same name but different parameters within the same class (or subclass).