Override -vs- Overload

By: Sheldon Pasciak

In Java, both override and overload are ways to achieve polymorphism, but they serve different purposes.

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

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.

The ticking timer on the bomb is running… RUNTIME

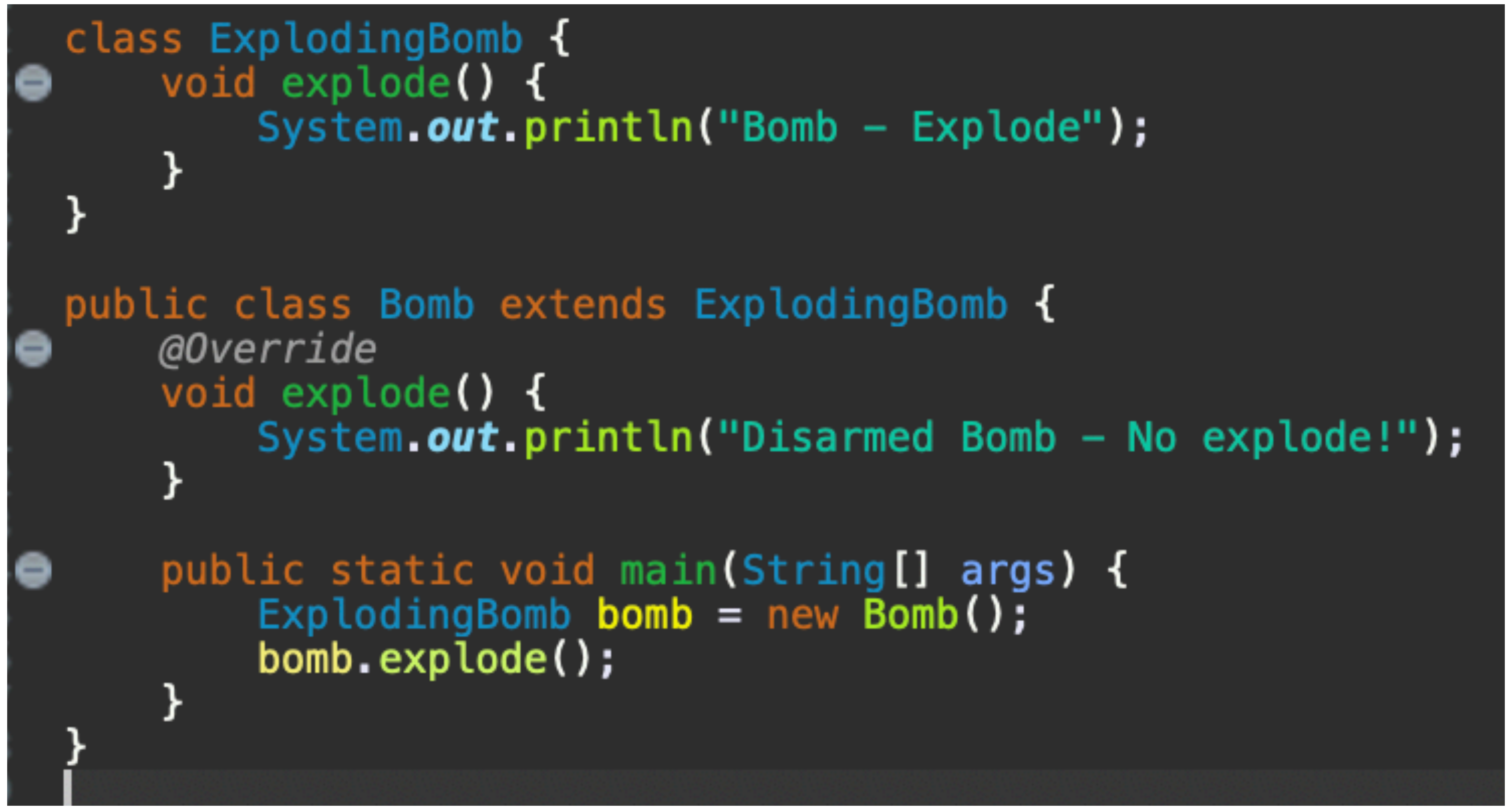
Compile-time Polymorphism (Method Overloading):

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 load for the truck is being compiled… COMPILE TIME

Override -vs- Overload

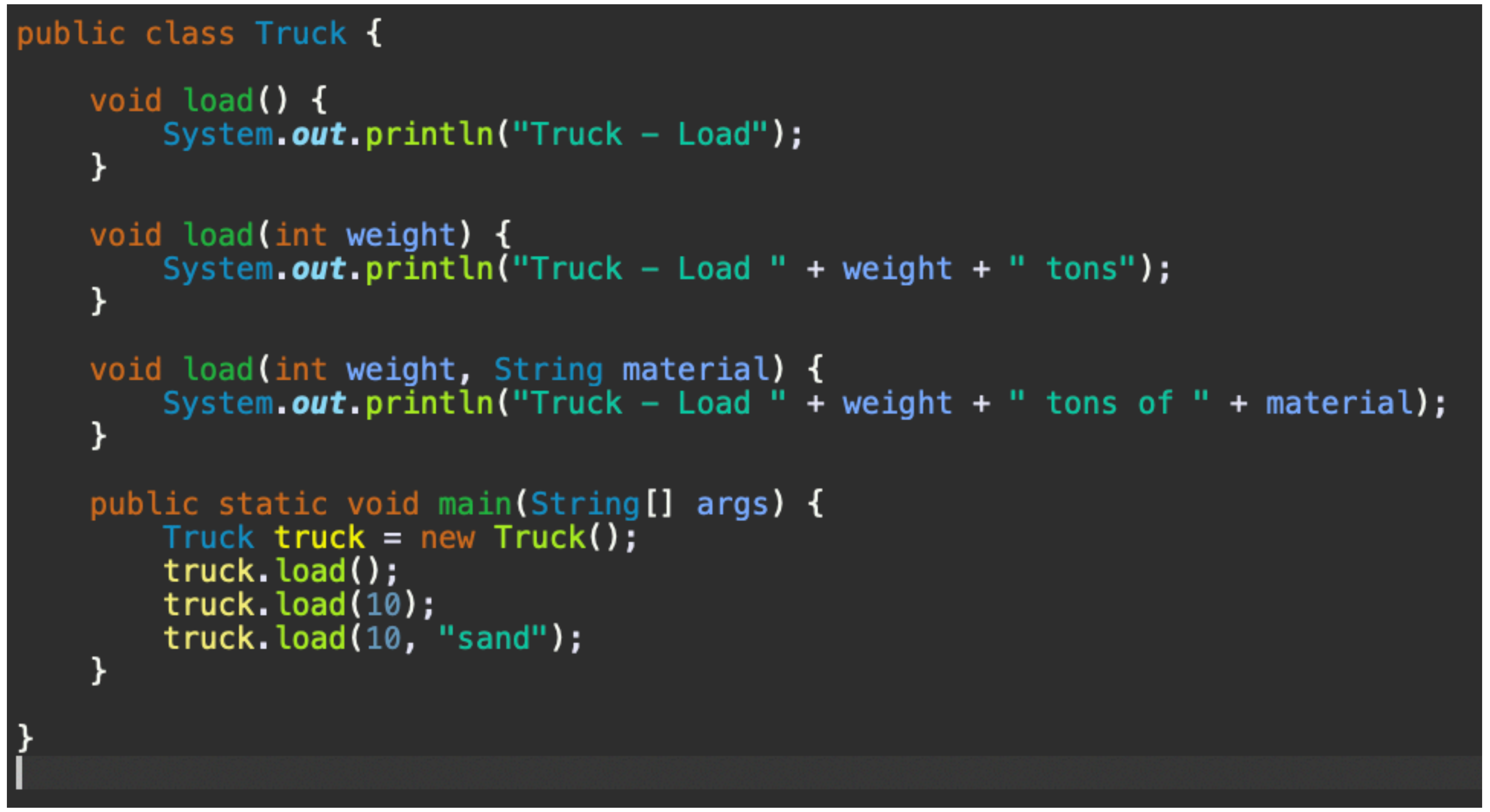
Override the Bomb so it doesn't Explode.



```
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();
    }
}
```

The ticking timer on the bomb is running… RUNTIME

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

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

```java
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));
    }
}
```

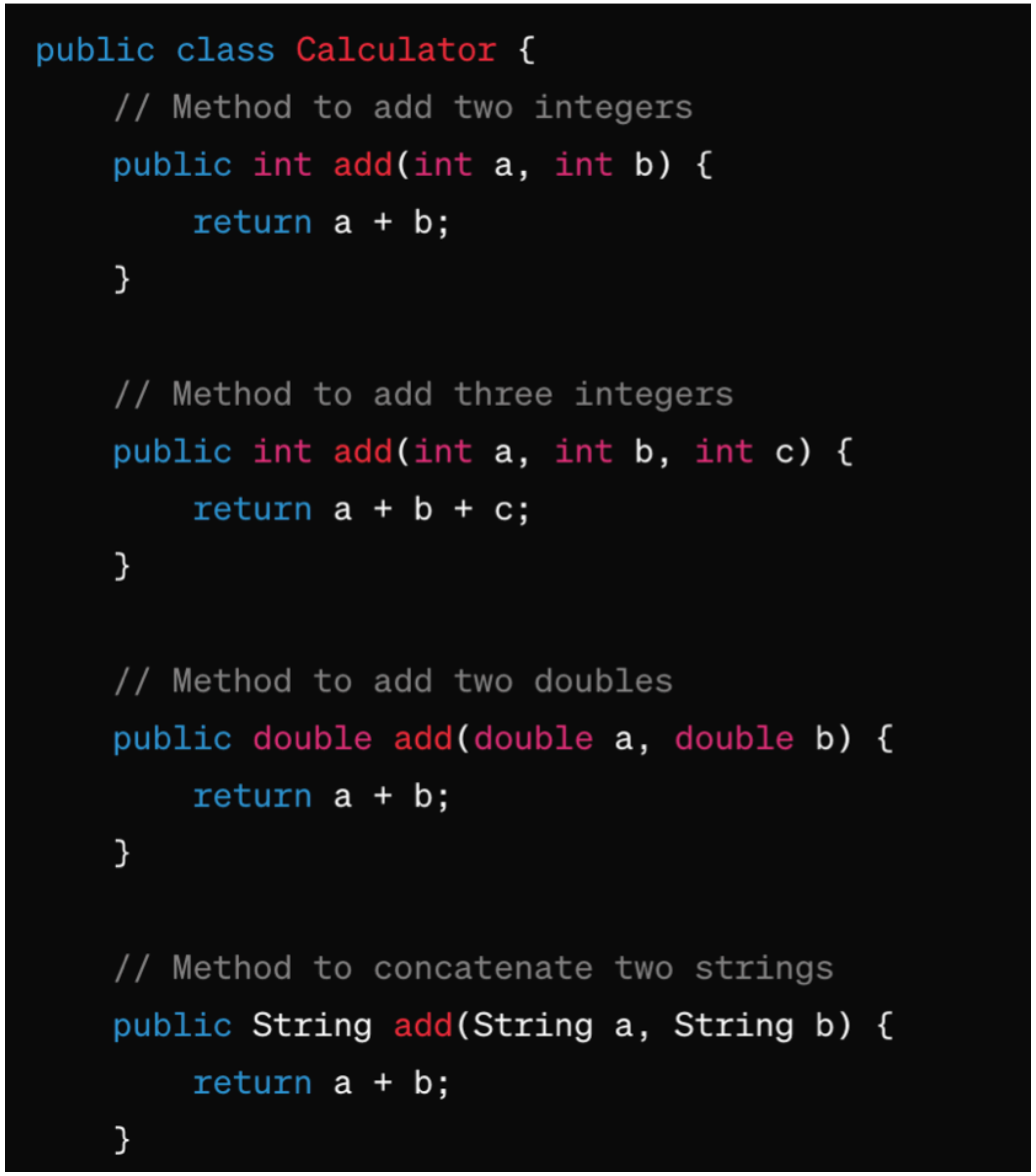
```java
public class Calculator {

    // 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

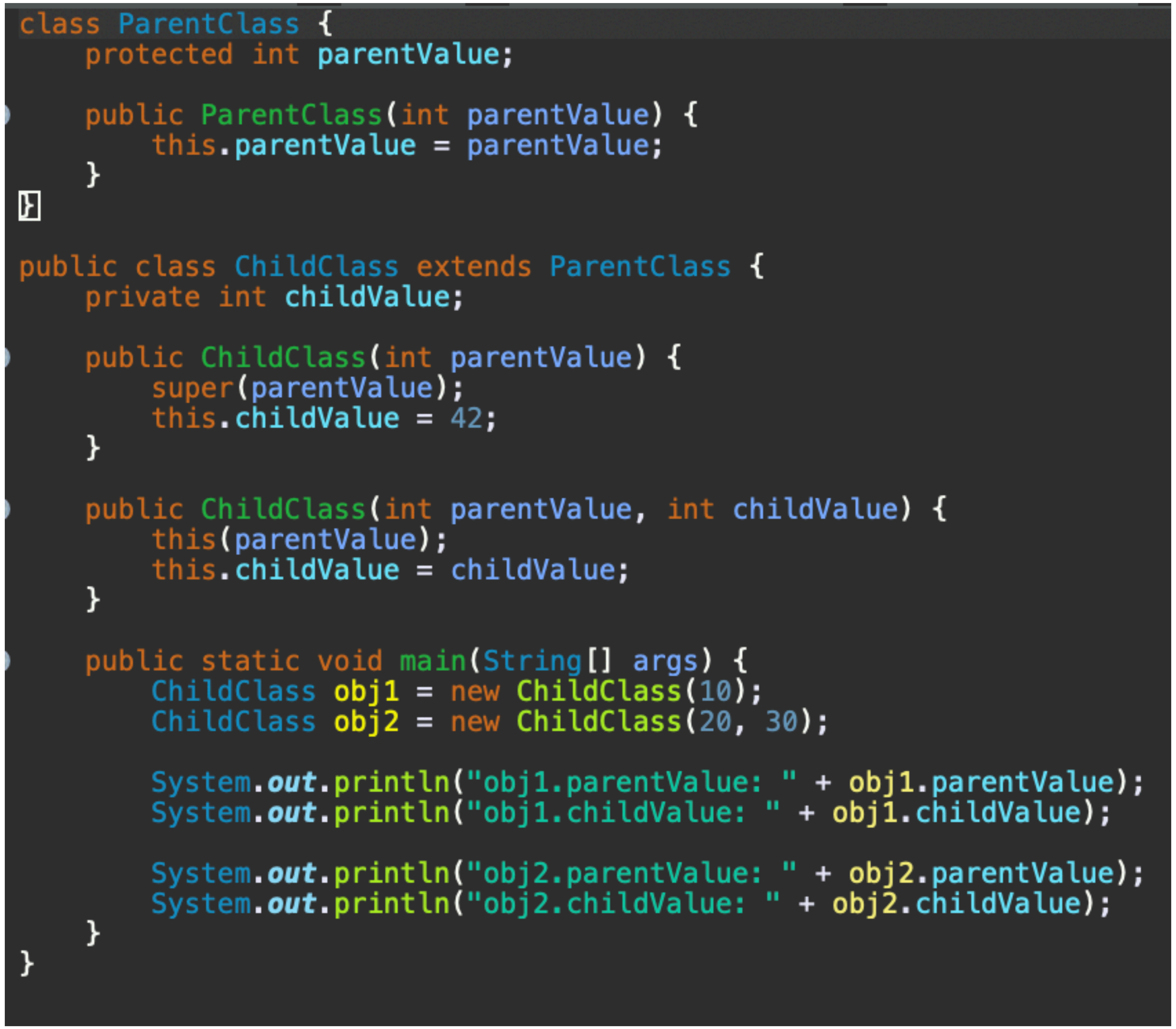
```java
public class Calculator {

    // 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;
    }

    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.

R E V I E W - S U M M A R Y

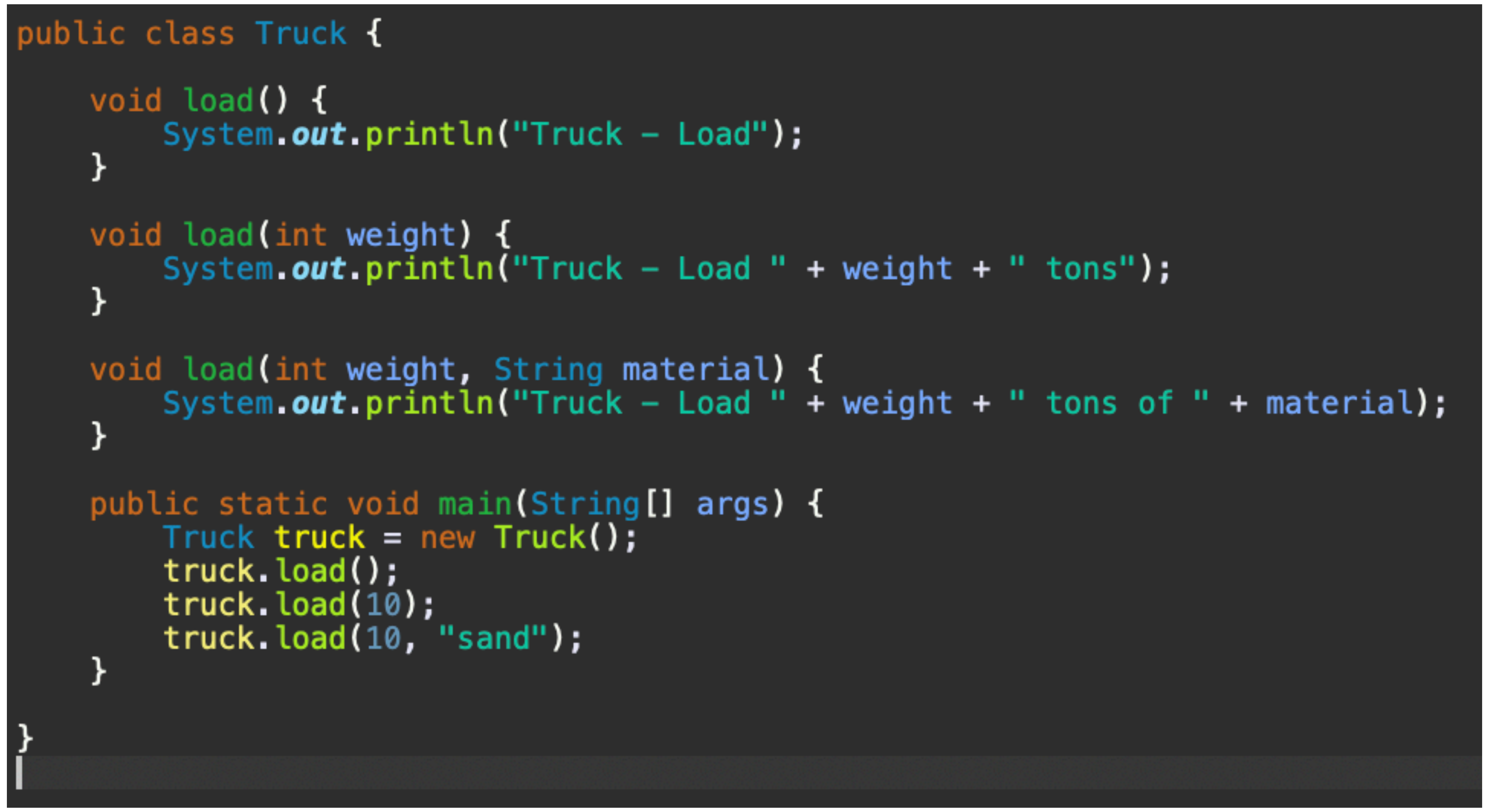
Overload

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.



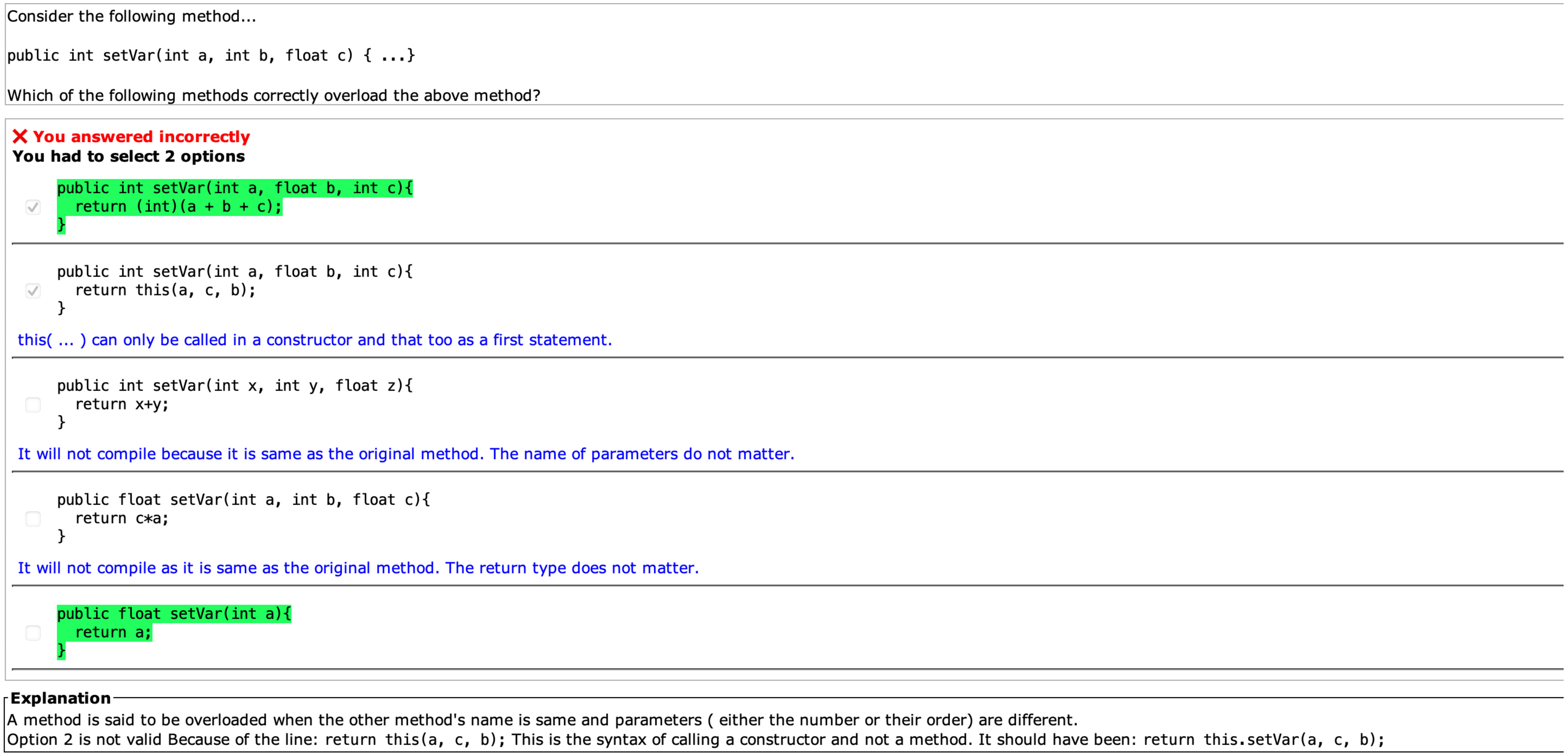
```
public class Truck {

    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

OCP - QUIZ EXAMPLE

Overloading

OCP - QUIZ EXAMPLE



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);`

Override

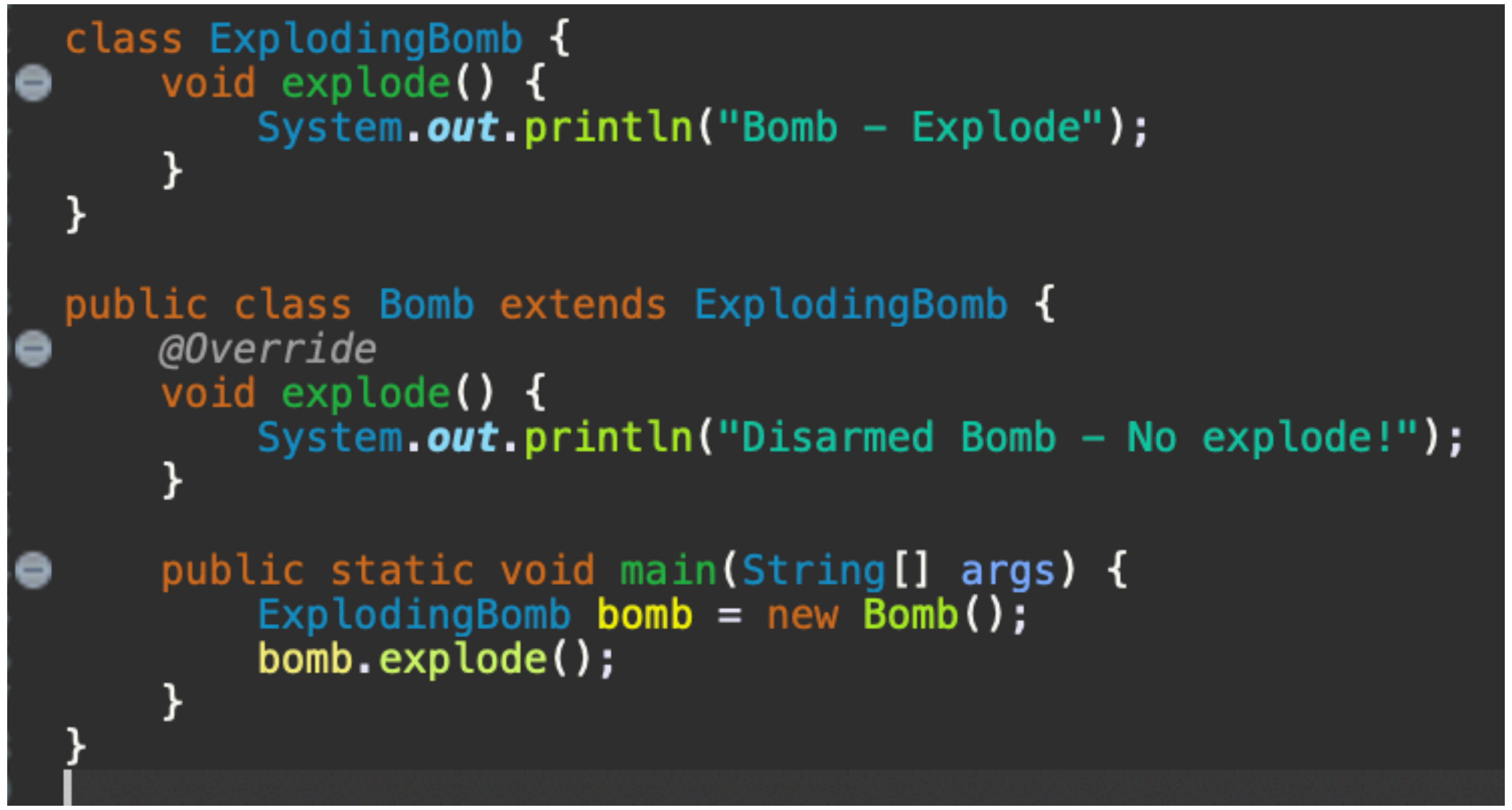
In Java, overriding is used to provide a specific implementation of a method that is already defined in a superclass (parent class) within a subclass (child class).

This allows subclasses to provide their own implementation of methods inherited from their superclass, tailoring behavior to suit their specific needs, while still adhering to the inheritance hierarchy.

Overall, overriding in Java is a powerful mechanism that enables flexible, extensible, and maintainable object-oriented design by allowing subclasses to tailor the behavior of inherited methods to meet their specific requirements.

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

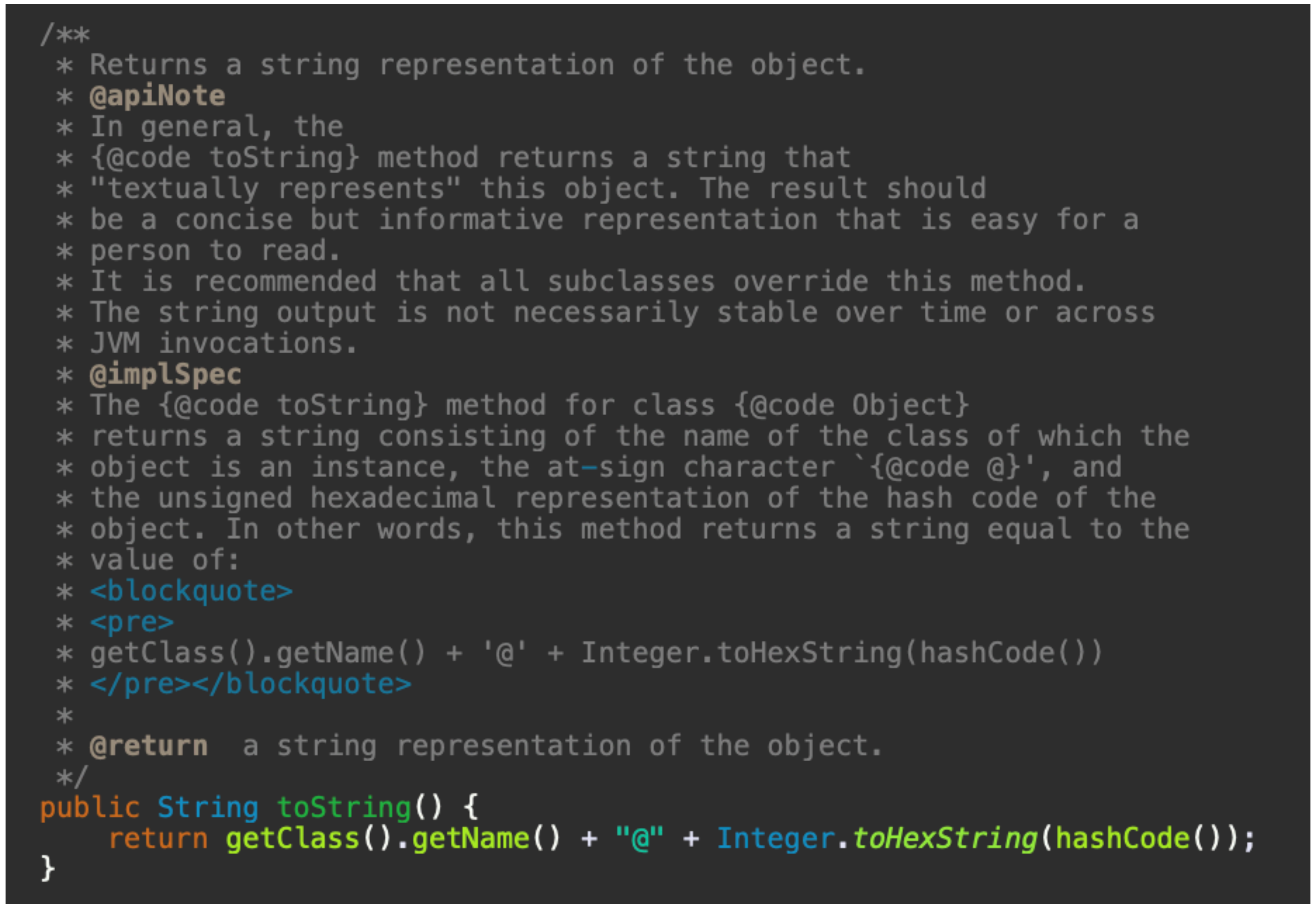
What's the purpose of overriding ?

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

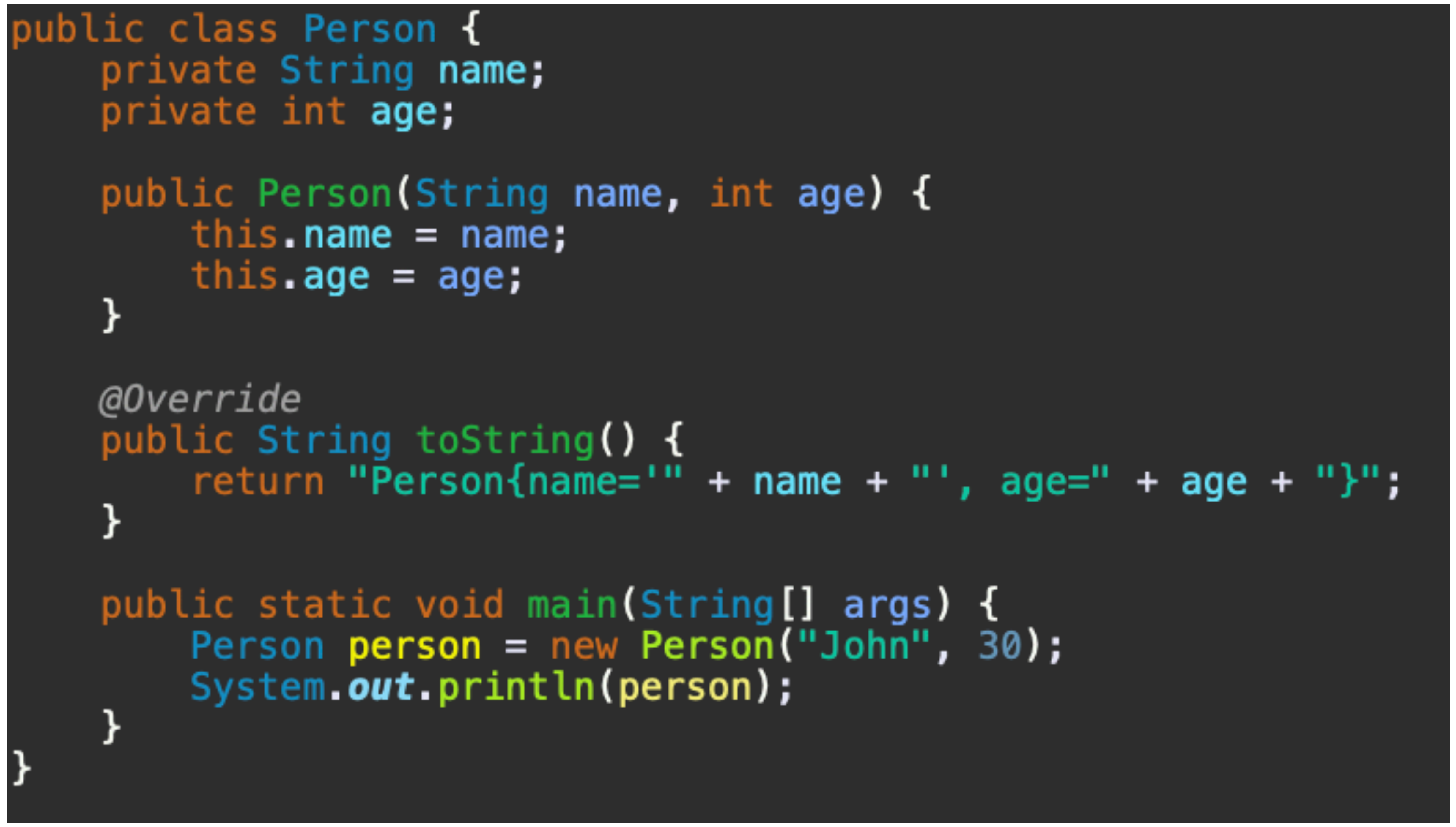
Runtime Polymorphism (Method Overriding): Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows objects of the subclass to be treated as objects of the superclass, but they can exhibit different behaviors based on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

Override

In the Java **Object** class, the toString() method is used to get a string representation of an object. By default, it returns a string containing the class name along with the hash code of the object. However, it's often useful to override this method to provide a more meaningful string representation of the object.



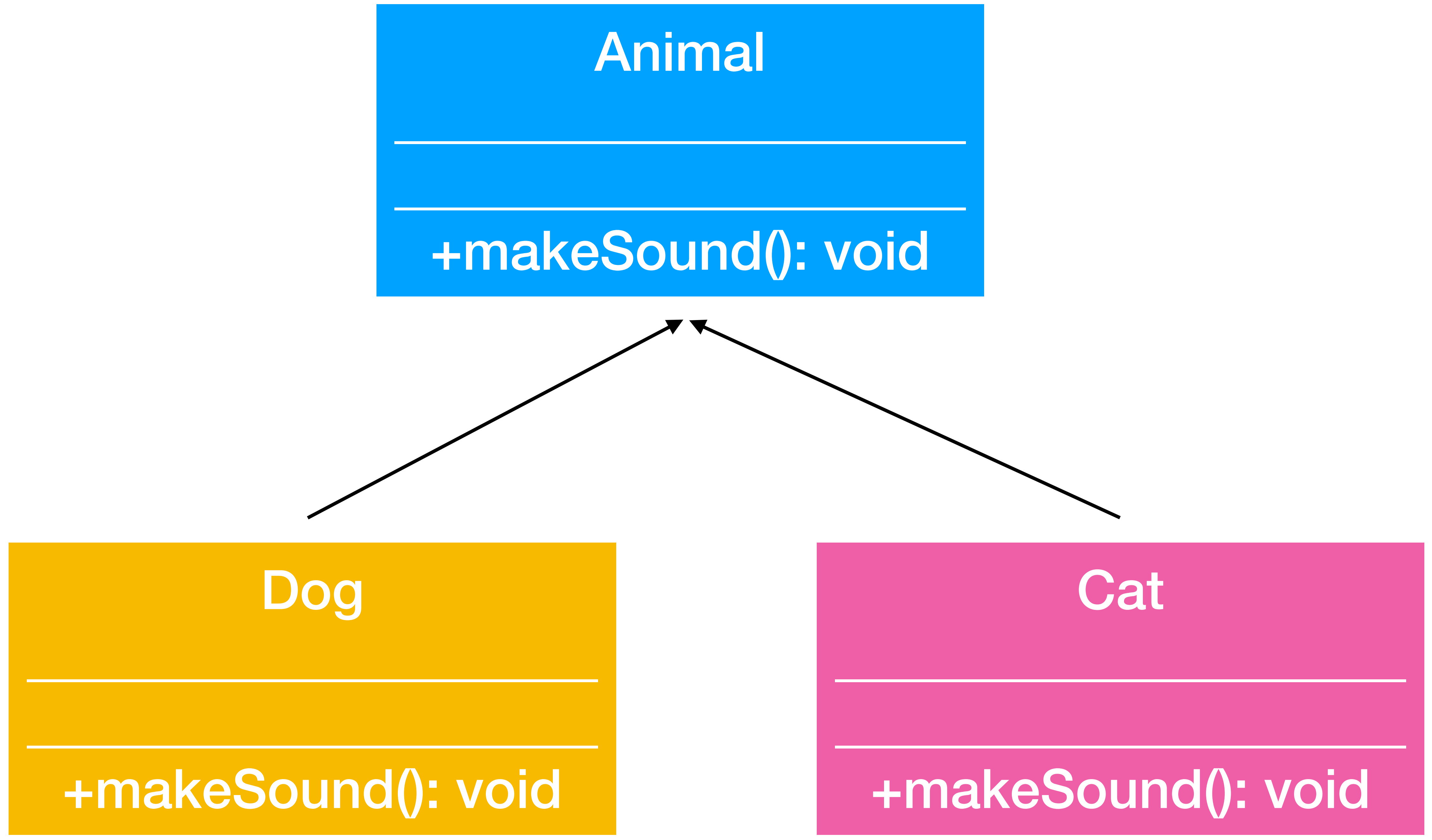
```java
/**
 * Returns a string representation of the object.
 * @apiNote
 * In general, the
 * {@code toString} method returns a string that
 * "textually represents" this object. The result should
 * be a concise but informative representation that is easy for a
 * person to read.
 * It is recommended that all subclasses override this method.
 * The string output is not necessarily stable over time or across
 * JVM invocations.
 * @implSpec
 * The {@code toString} method for class {@code Object}
 * returns a string consisting of the name of the class of which the
 * object is an instance, the at-sign character `{@code @}', and
 * the unsigned hexadecimal representation of the hash code of the
 * object. In other words, this method returns a string equal to the
 * value of:
 * <blockquote>
 * <pre>
 * getClass().getName() + '@' + Integer.toHexString(hashCode())
 * </pre></blockquote>
 *
 * @return  a string representation of the object.
 */
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```



```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }

    public static void main(String[] args) {
        Person person = new Person("John", 30);
        System.out.println(person);
    }
}
```

Override

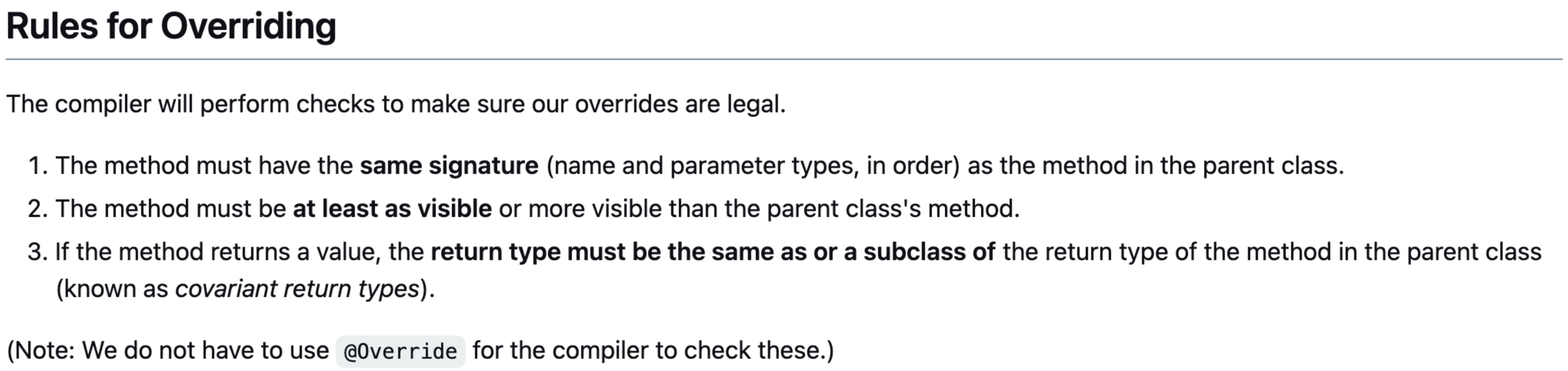
```
class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}

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
    }
}
```



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.

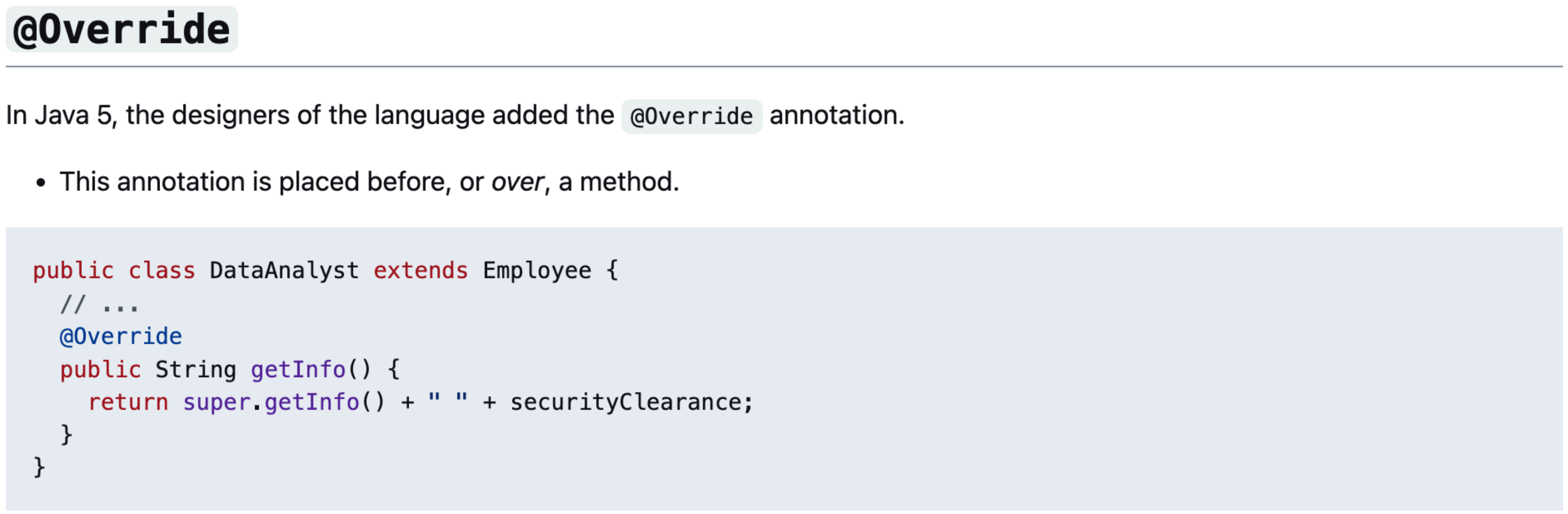
1. The method must have the **same signature** (name and parameter types, in order) as the method in the parent class.
2. The method must be **at least as visible** or more visible than the parent class's method.
3. If the method returns a value, the **return type must be the same as or a subclass of** 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



`@Override`

In Java 5, the designers of the language added the `@Override` annotation.

- This annotation is placed before, or *over*, a method.

```
public class DataAnalyst extends Employee {
  // ...
  @Override
  public String getInfo() {
    return super.getInfo() + " " + securityClearance;
  }
}
```

Reference:

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

Override

`private` Methods Cannot Be Overridden

There is no such thing as overriding a `private` method.

- The subclass cannot see the superclass's `private` method, so it doesn't know there is a method to override.

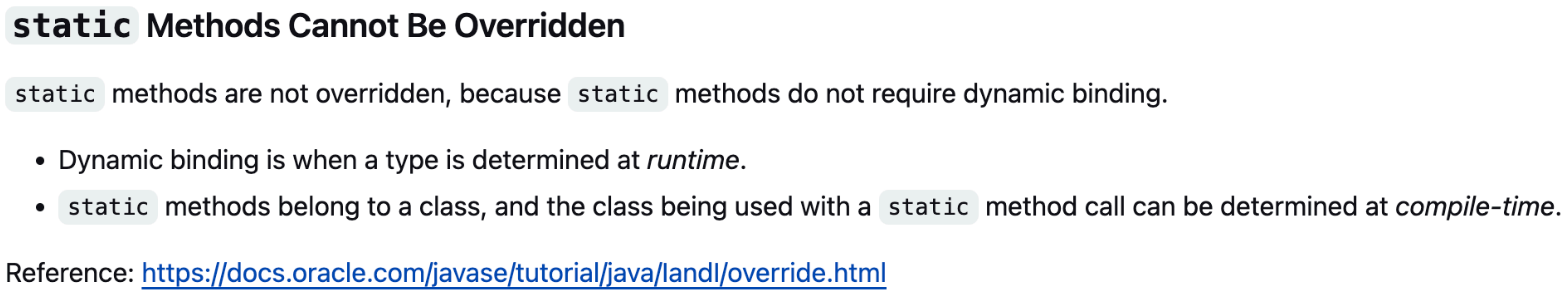
`Organization` has a `private` method `lookupEmployee`.

```
public class Organization {
  // ...
  // private method
  private Employee lookupEmployee(int id) { /* ... */ }
}
```

Reference:

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

Override



`static` Methods Cannot Be Overridden

`static` methods are not overridden, because `static` methods do not require dynamic binding.

- Dynamic binding is when a type is determined at *runtime*.
- `static` methods belong to a class, and the class being used with a `static` method call can be determined at *compile-time*.

Reference: https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Reference:

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

Override

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

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.

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

```
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

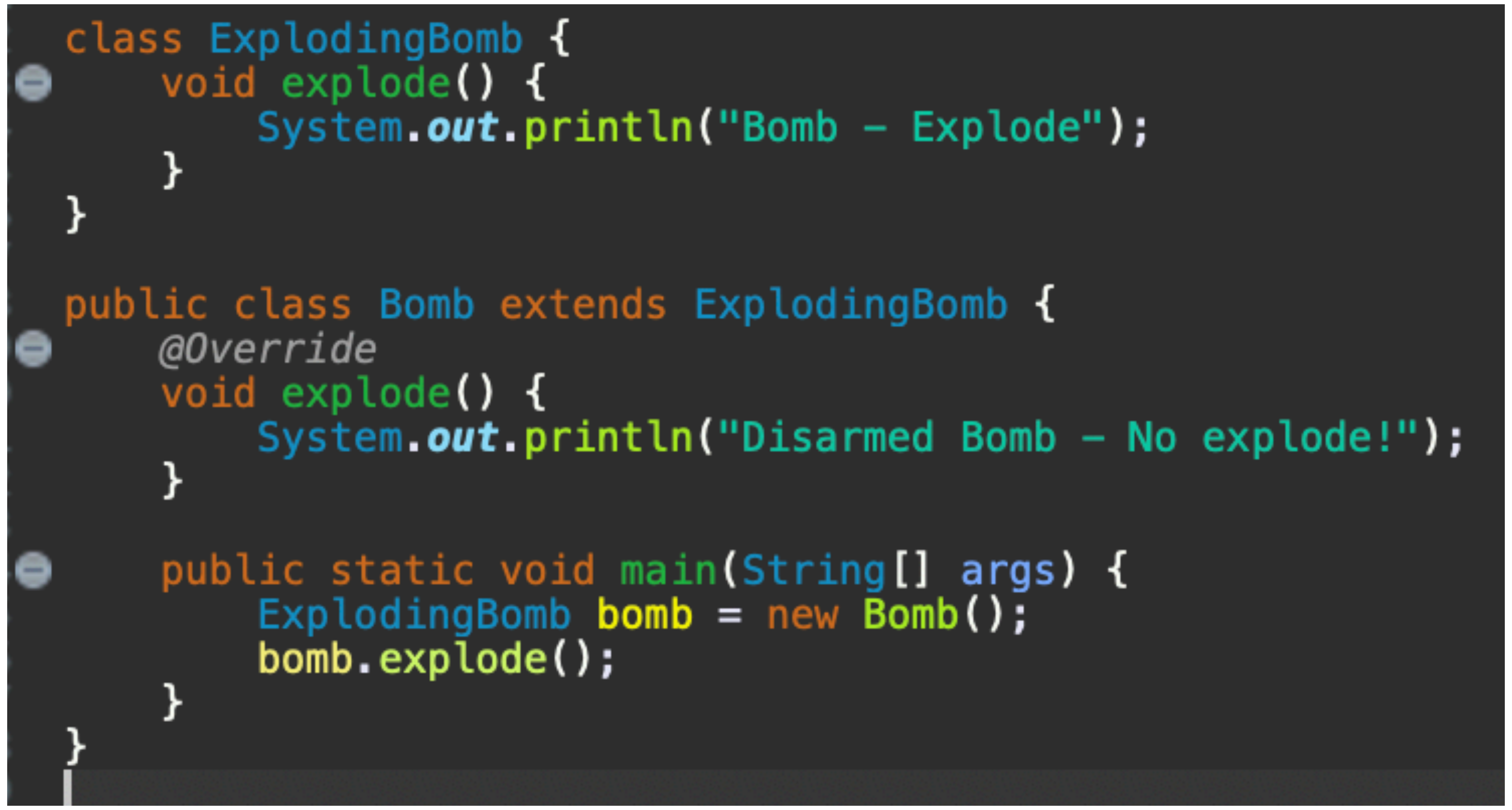
Reference:

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

R E V I E W - S U M M A R Y

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.

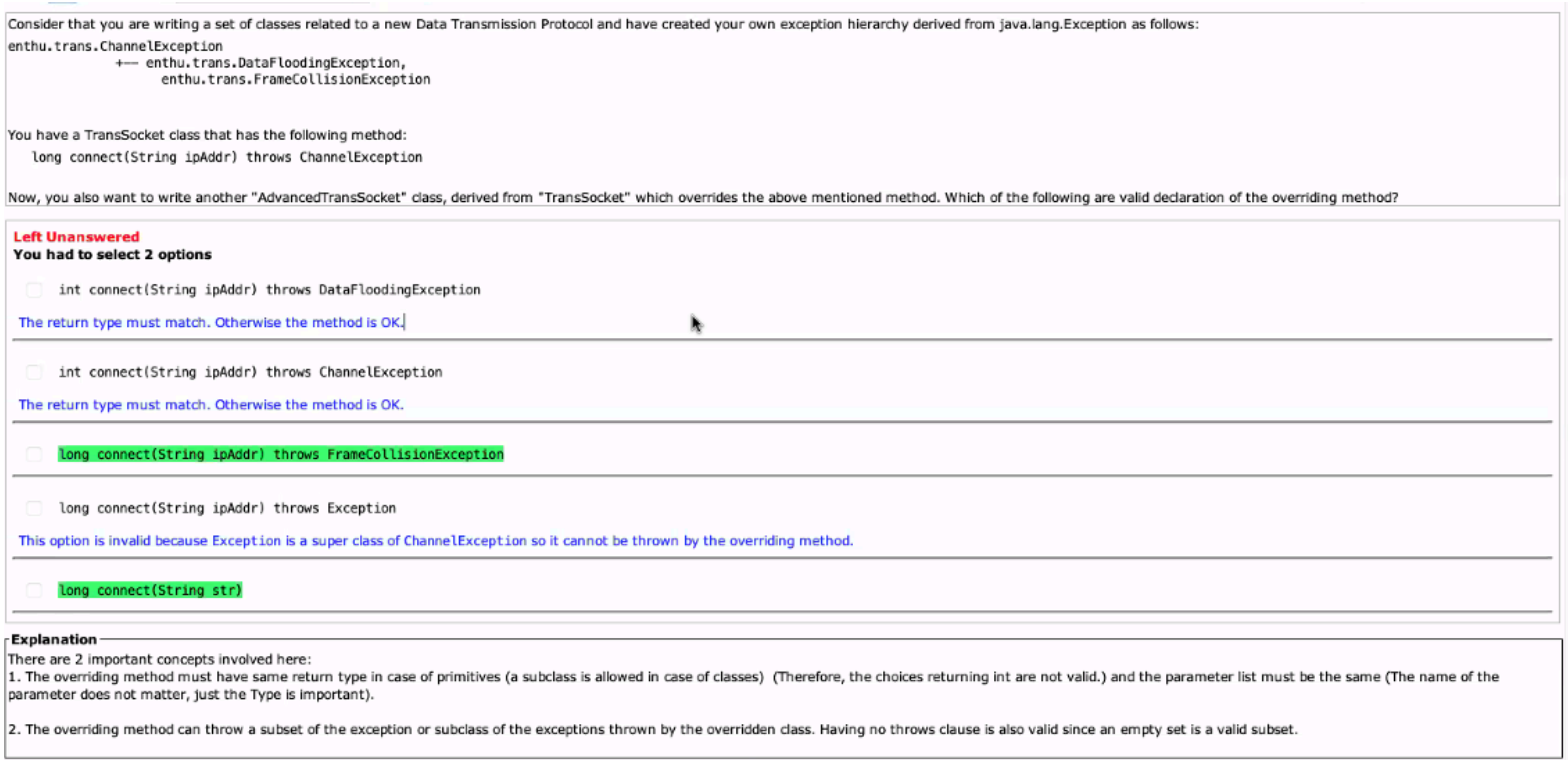
The purpose of overriding is to provide a specialized implementation of a method in a subclass, which is more appropriate for the subclass's context.

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

OCP - QUIZ EXAMPLE

Override

OCP - QUIZ EXAMPLE



Consider that you are writing a set of classes related to a new Data Transmission Protocol and have created your own exception hierarchy derived from java.lang.Exception as follows:

```
enthu.trans.ChannelException
             +-- enthu.trans.DataFloodingException,
                   enthu.trans.FrameCollisionException
```

You have a TransSocket class that has the following method:

```
   long connect(String ipAddr) throws ChannelException
```

Now, you also want to write another "AdvancedTransSocket" class, derived from "TransSocket" which overrides the above mentioned method. Which of the following are valid declaration of the overriding method?

Left Unanswered
You had to select 2 options

`int connect(String ipAddr) throws DataFloodingException`

The return type must match. Otherwise the method is OK.

`int connect(String ipAddr) throws ChannelException`

The return type must match. Otherwise the method is OK.

`long connect(String ipAddr) throws FrameCollisionException`

`long connect(String ipAddr) throws Exception`

This option is invalid because Exception is a super class of ChannelException so it cannot be thrown by the overriding method.

`long connect(String str)`

Explanation

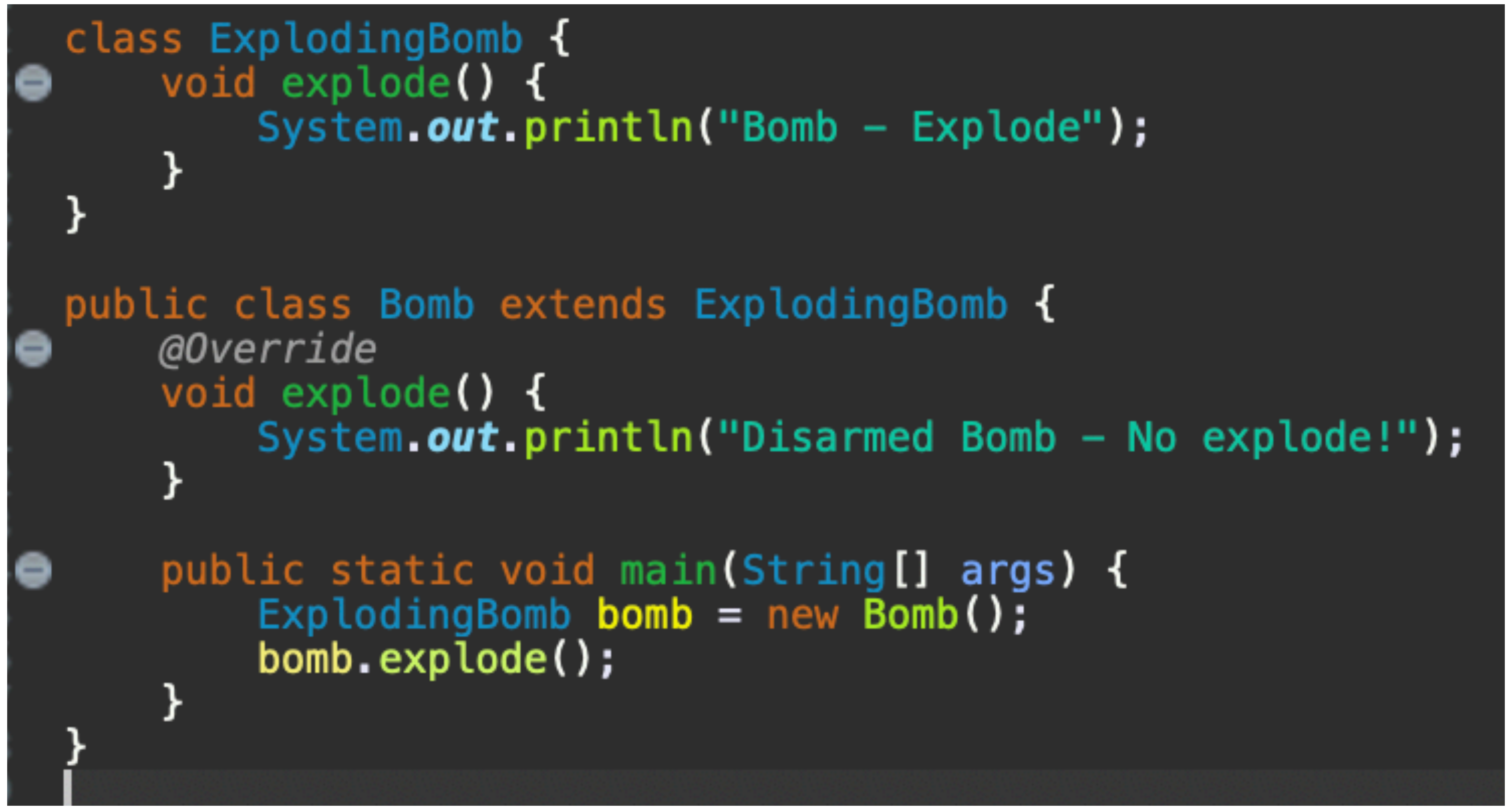
There are 2 important concepts involved here:
1. The overriding method must have same return type in case of primitives (a subclass is allowed in case of classes) (Therefore, the choices returning int are not valid.) and the parameter list must be the same (The name of the parameter does not matter, just the Type is important).

2. The overriding method can throw a subset of the exception or subclass of the exceptions thrown by the overridden class. Having no throws clause is also valid since an empty set is a valid subset.

SUMMARY

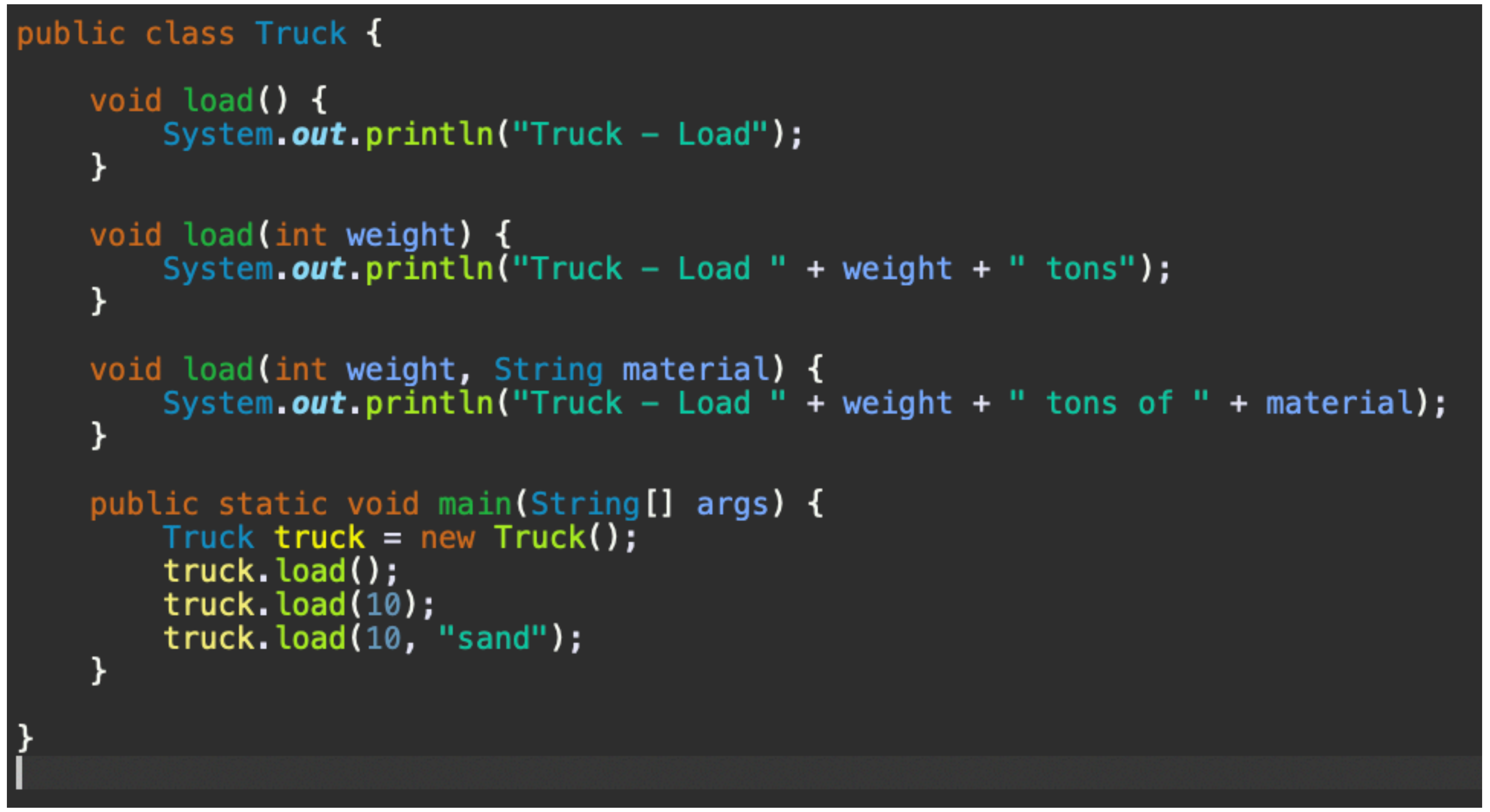
Override -vs- Overload

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).