OOP in Java

Object-oriented programming is a programming style associated with concepts like class, object, Inheritance, Encapsulation, Abstraction, and Polymorphism. Java, C++, C# and Ruby are the most popular programming languages.

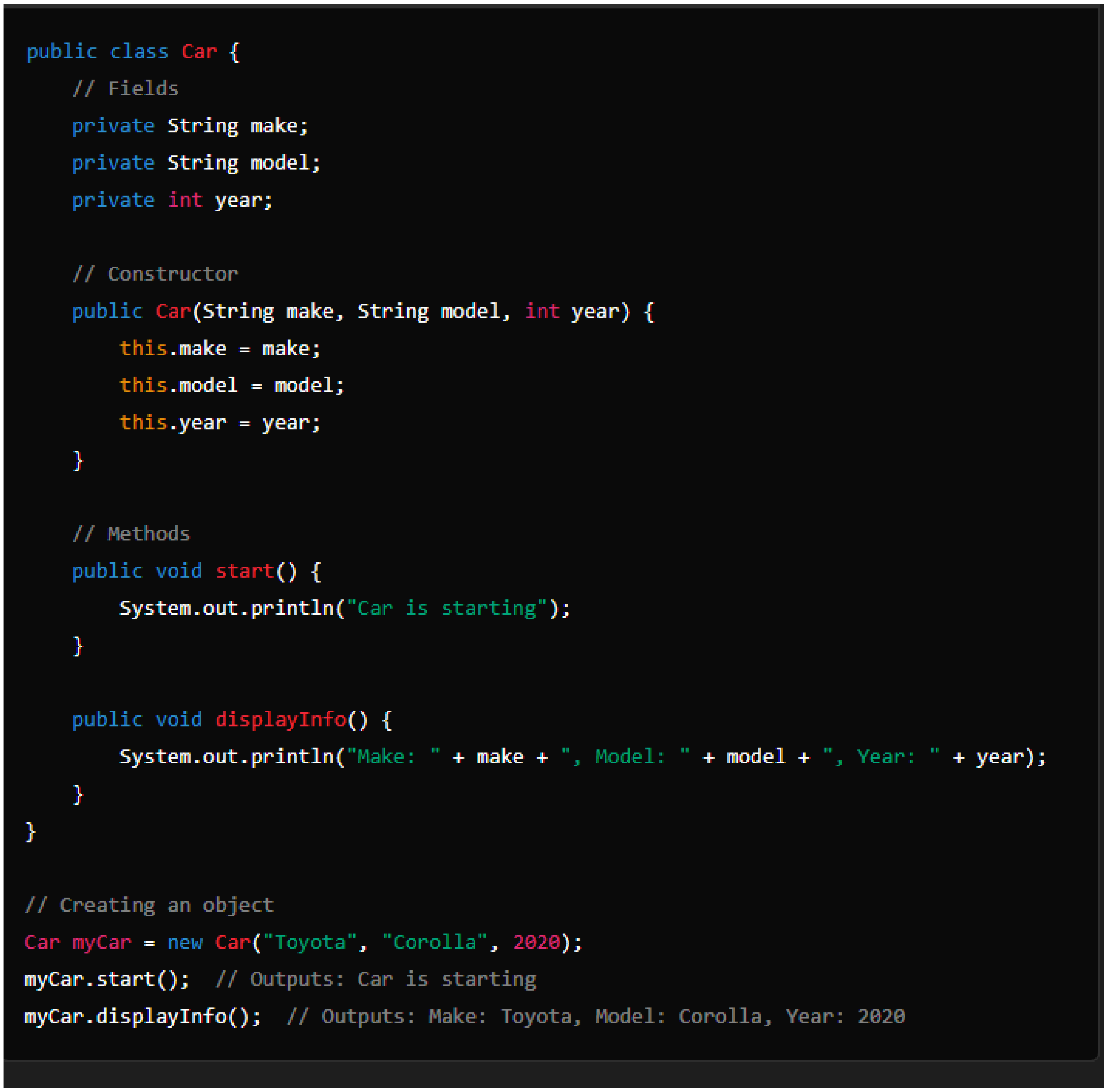
Key Concepts of OOP:

1. **Classes and Objects:** Classes are blueprints or templates that define objects' properties (attributes) and behaviors (methods). Objects are instances of classes that hold specific data and can perform actions.

   An object in Java is a real-world entity with a state and behavior. It is an instance of a class created using the `new` keyword followed by a constructor call. Objects are instances of classes and represent specific instances of that class in memory.

   A class is a blueprint for the object. Before we create an object, we first need to define the class. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, we built the house. House is the object. Since many homes can be made from the same description, we can create many objects from a class.

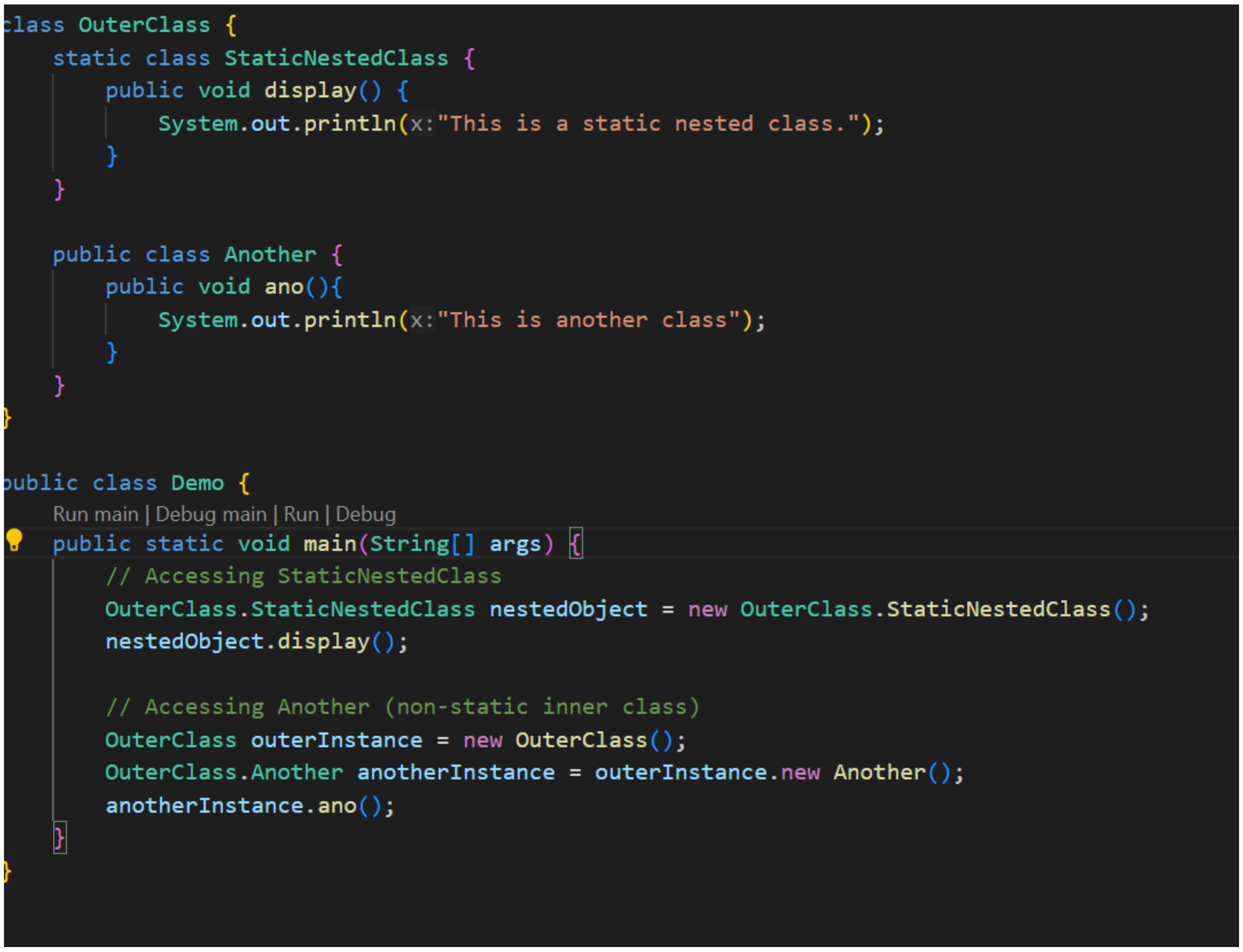
- **Components of a Class:**
  - **Fields:** Variables that hold the state of the object.
  - **Methods:** Functions that define the behavior of the object.
  - **Constructors:** Special methods invoked when an object is instantiated, initializing the object.



```java
public class Car {
    // Fields
    private String make;
    private String model;
    private int year;

    // Constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    // Methods
    public void start() {
        System.out.println("Car is starting");
    }

    public void displayInfo() {
        System.out.println("Make: " + make + ", Model: " + model + ", Year: " + year);
    }
}

// Creating an object
Car myCar = new Car("Toyota", "Corolla", 2020);
myCar.start();  // Outputs: Car is starting
myCar.displayInfo();  // Outputs: Make: Toyota, Model: Corolla, Year: 2020
```

Static Classes & Methods

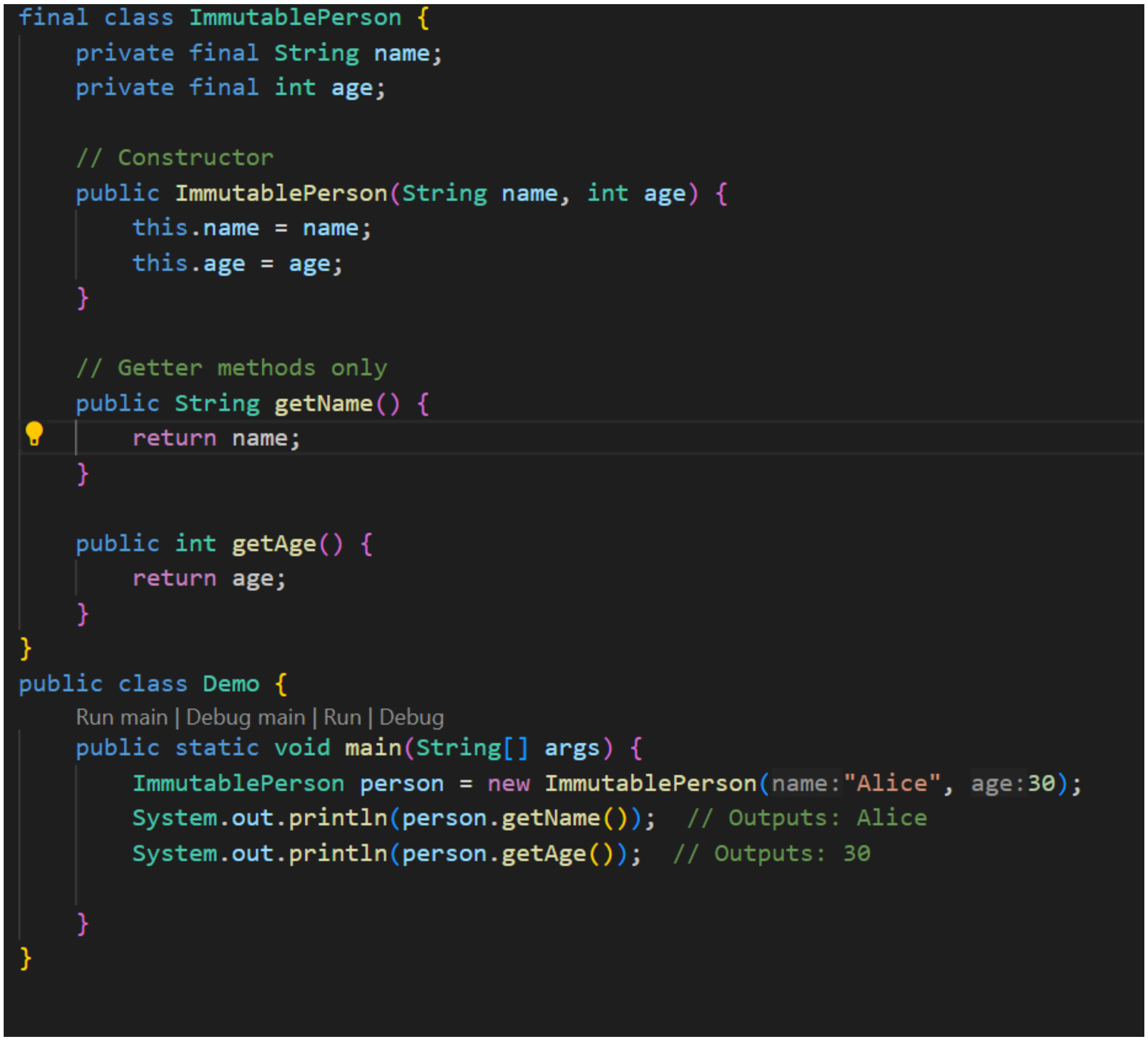
A static class is a class that belongs to the class itself rather than to any specific instance of the class. This means you don’t need to create an object of the class to access its members (fields and methods). Instead, you can access them directly using the class name.



```
class OuterClass {
    static class StaticNestedClass {
        public void display() {
            System.out.println(x:"This is a static nested class.");
        }
    }

    public class Another {
        public void ano(){
            System.out.println(x:"This is another class");
        }
    }
}

public class Demo {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        // Accessing StaticNestedClass
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
        nestedObject.display();

        // Accessing Another (non-static inner class)
        OuterClass outerInstance = new OuterClass();
        OuterClass.Another anotherInstance = outerInstance.new Another();
        anotherInstance.ano();
    }
}
```

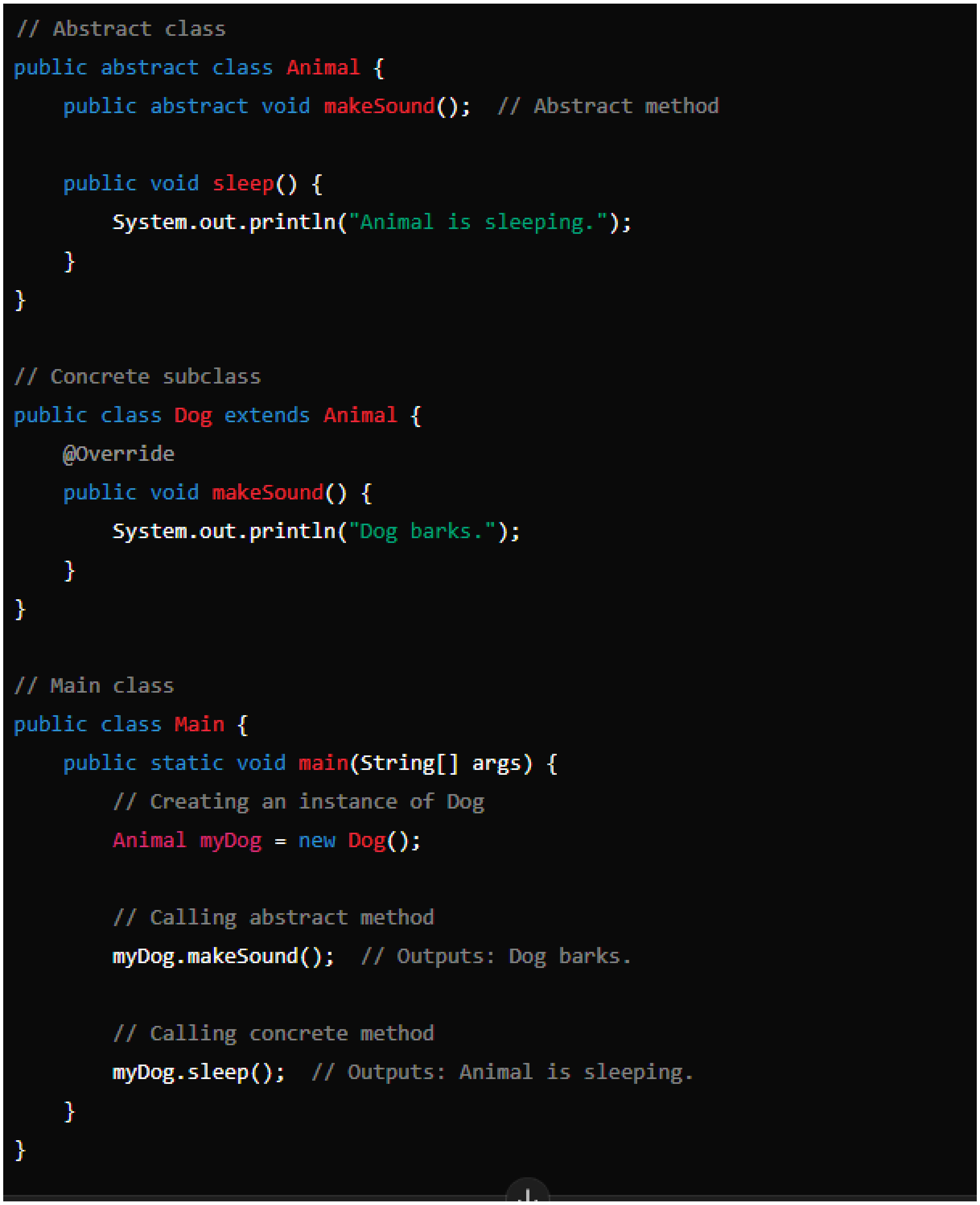
Immutable Classes

An immutable class is a class whose instances cannot be modified after creation. This means that its state cannot be changed once the object is constructed.


```java
final class ImmutablePerson {
    private final String name;
    private final int age;

    // Constructor
    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter methods only
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
public class Demo {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        ImmutablePerson person = new ImmutablePerson(name:"Alice", age:30);
        System.out.println(person.getName());  // Outputs: Alice
        System.out.println(person.getAge());  // Outputs: 30

    }
}
```

Abstract Classes

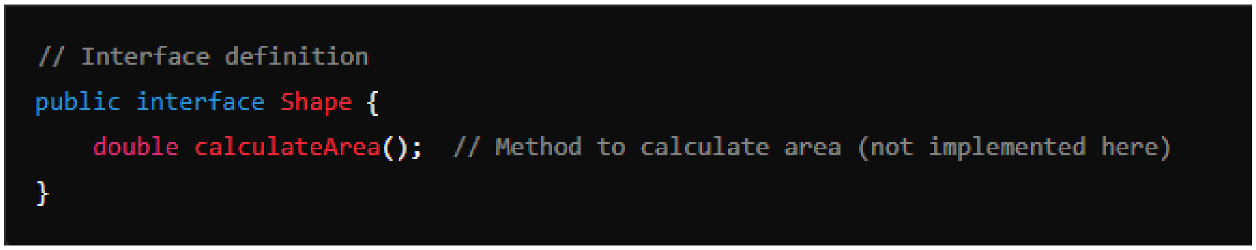
Abstract classes cannot be instantiated and are intended to be subclassed. They can include abstract methods (methods without implementation) that must be implemented by subclasses.



```java
// Abstract class
public abstract class Animal {
    public abstract void makeSound();  // Abstract method

    public void sleep() {
        System.out.println("Animal is sleeping.");
    }
}

// Concrete subclass
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Creating an instance of Dog
        Animal myDog = new Dog();

        // Calling abstract method
        myDog.makeSound();  // Outputs: Dog barks.

        // Calling concrete method
        myDog.sleep();  // Outputs: Animal is sleeping.
    }
}
```

Interfaces

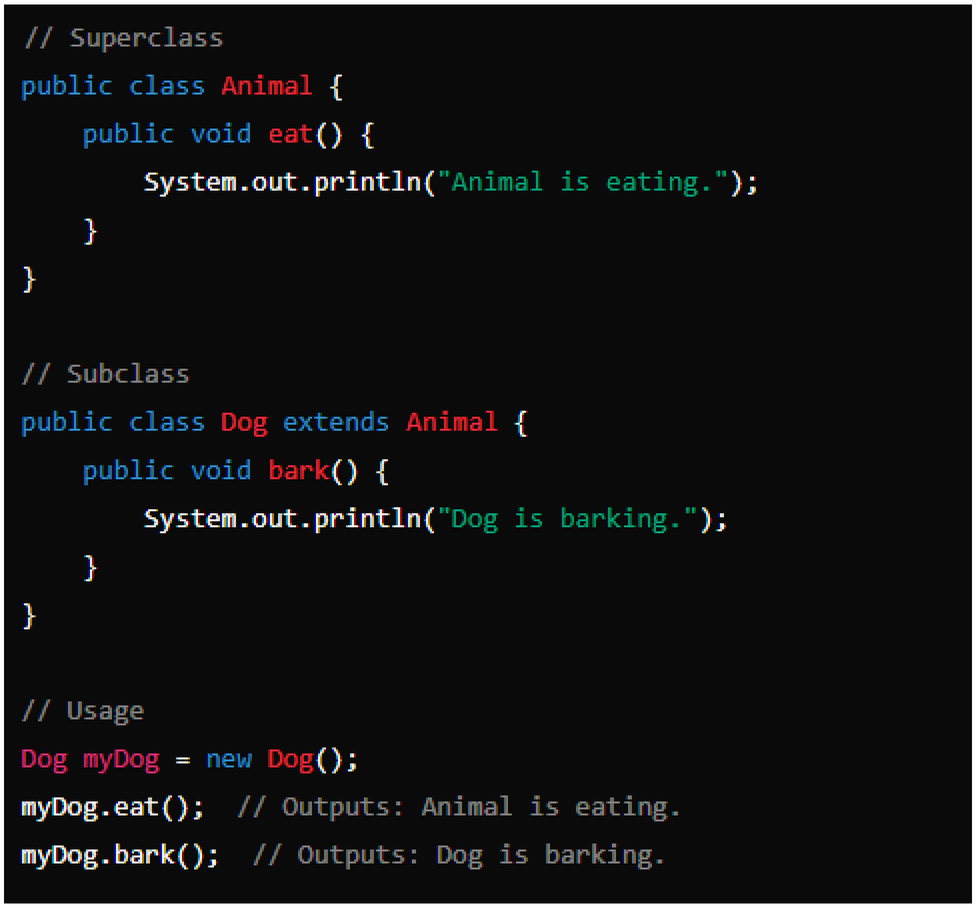
In Java, when you define methods within an interface, those methods are simply declarations or signatures. They do not contain any implementation details (i.e., the code inside the methods). Therefore, if you want those methods to perform some action when called, you must implement the interface in a class and provide the specific implementation for each method defined in the interface.



```java
// Interface definition
public interface Shape {
    double calculateArea();  // Method to calculate area (not implemented here)
}
```

```java
// Implementing classes
public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
    }
}
```

2. **Inheritance:** Inheritance allows one class (subclass or derived class) to inherit properties and behaviors from another (superclass or base class), promoting code reuse and establishing a hierarchical relationship.



```java
// Superclass
public class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

// Subclass
public class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking.");
    }
}

// Usage
Dog myDog = new Dog();
myDog.eat();  // Outputs: Animal is eating.
myDog.bark();  // Outputs: Dog is barking.
```

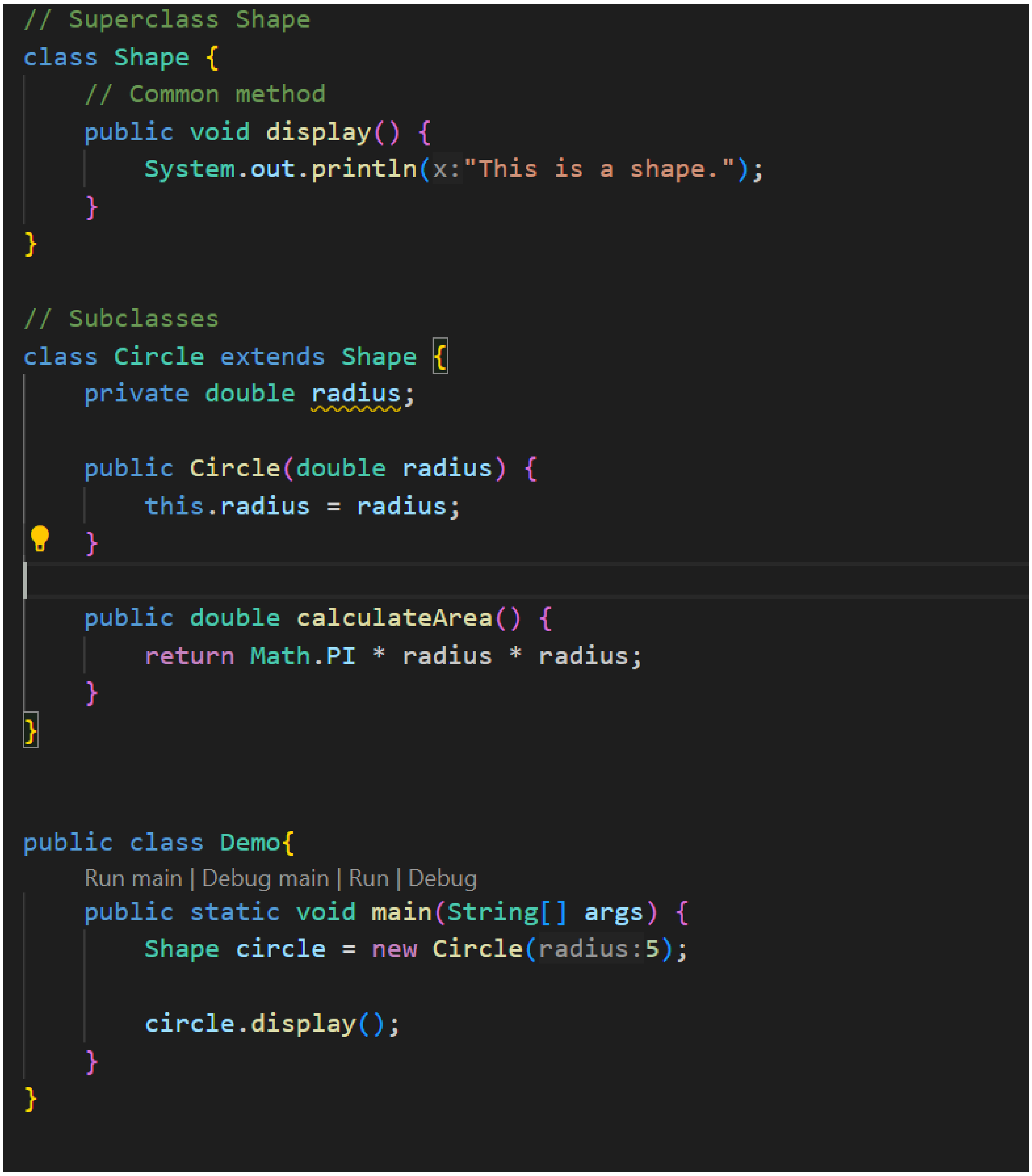
3. **Encapsulation**: In Java, encapsulation means hiding things by making them private or inaccessible.

   Why would we want to hide things in Java?

   To protect data integrity on an object, we may hide or restrict access to some of the data and operations.

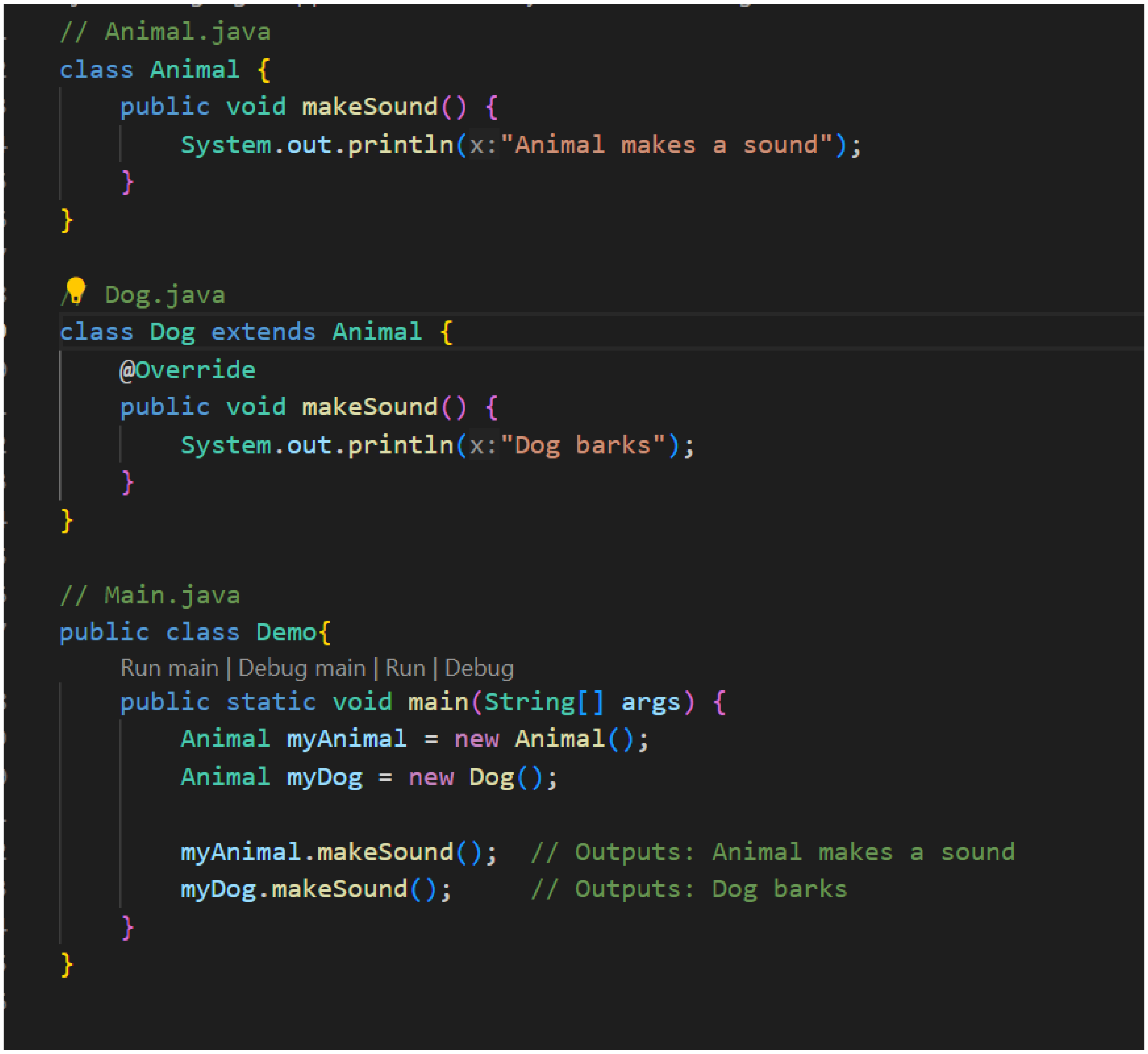
```java
public class UnencapsulatedStudent {
    public String name;
    public int age;

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

public class UnencapsulatedExample {
    public static void main(String[] args) {
        UnencapsulatedStudent student = new UnencapsulatedStudent("Bob", 25);
        student.age = -5;  // Directly modifying age without any restrictions
        System.out.println("Name: " + student.name);
        System.out.println("Age: " + student.age);  // Negative age can be set
    }
}

public class EncapsulatedStudent {
    private String name;
    private int age;

    public EncapsulatedStudent(String name, int age) {
        setName(name);
        setAge(age);
    }

public class EncapsulatedExample {
    public static void main(String[] args) {
        EncapsulatedStudent student = new EncapsulatedStudent("Alice", -20);  /
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());
    }
}
```

4. **Polymorphism:** Polymorphism allows objects to be treated as instances of their superclass, enabling flexibility and extensibility in software design. It allows methods to be overridden in subclasses to provide specific implementations.



```java
// Superclass Shape
class Shape {
    // Common method
    public void display() {
        System.out.println(x:"This is a shape.");
    }
}

// Subclasses
class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}


public class Demo{
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Shape circle = new Circle(radius:5);

        circle.display();
    }
}
```

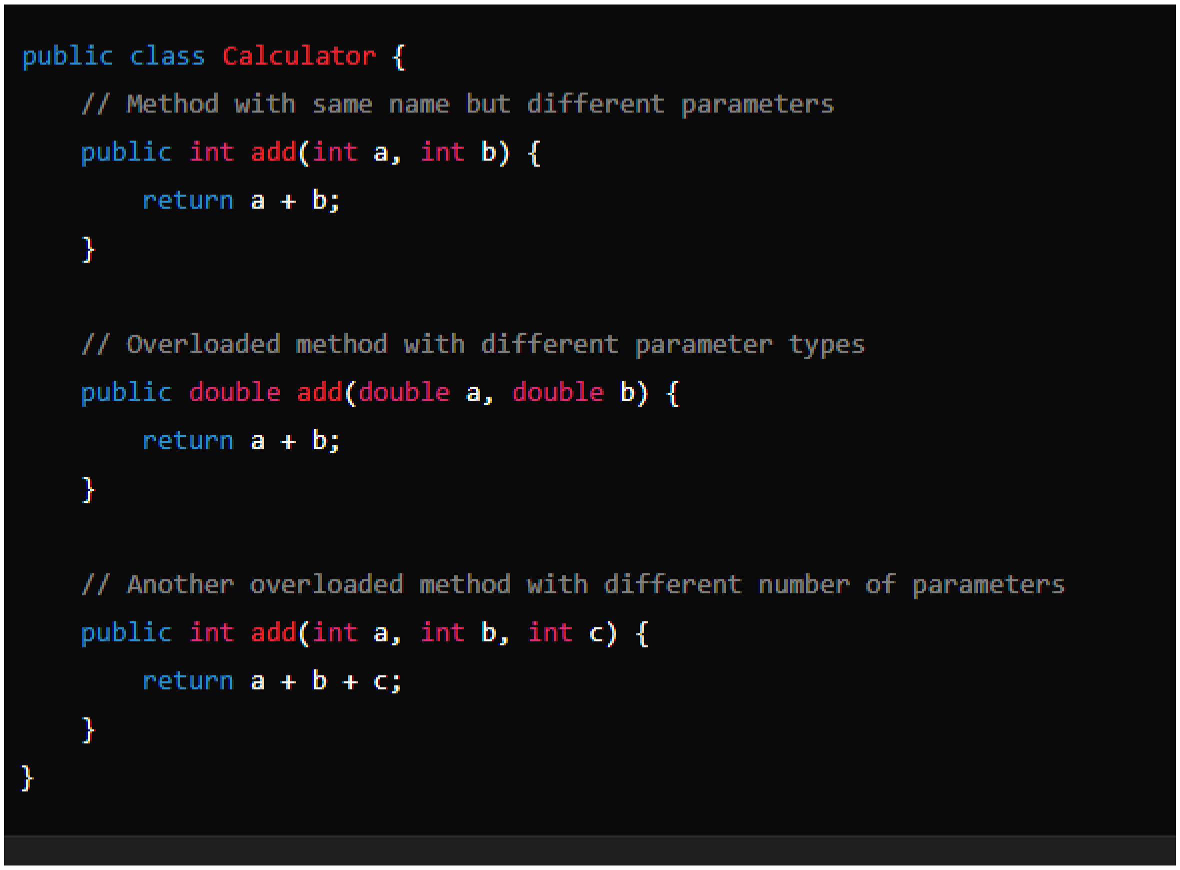
Method Overriding: The subclass can override (replace) a superclass method with its implementation, provided the method signatures are the same (same method name, parameters, and return type).



```
// Animal.java
class Animal {
    public void makeSound() {
        System.out.println(x:"Animal makes a sound");
    }
}

// Dog.java
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println(x:"Dog barks");
    }
}

// Main.java
public class Demo{
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();

        myAnimal.makeSound();  // Outputs: Animal makes a sound
        myDog.makeSound();     // Outputs: Dog barks
    }
}
```

Method Overloading

Method overloading occurs when a class has two or more methods with the same name but different parameters (either different types of parameters or a different number of parameters). The compiler determines which method to call based on the number and types of arguments passed.



```
public class Calculator {
    // Method with same name but different parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with different parameter types
    public double add(double a, double b) {
        return a + b;
    }

    // Another overloaded method with different number of parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```