Think and Tell

- Different banks use different methods to calculate interest rates. However, it is important for the banks to provide a concrete definition for interest calculation.

- How can we implement this programmatically?

Slide Note

- What makes it easier for you to locate a particular tweet?
- What sequence of posts do you see on your Instagram account?
- How does Facebook display a record of events?

Gaming Console

In a game console, each game offers functionalities to play the game, compute and display the score, and exit the game. Some of these functionalities are common and some vary from game to game.

- How can you represent the common functionalities of an application and reuse them at multiple places?

Slide Note

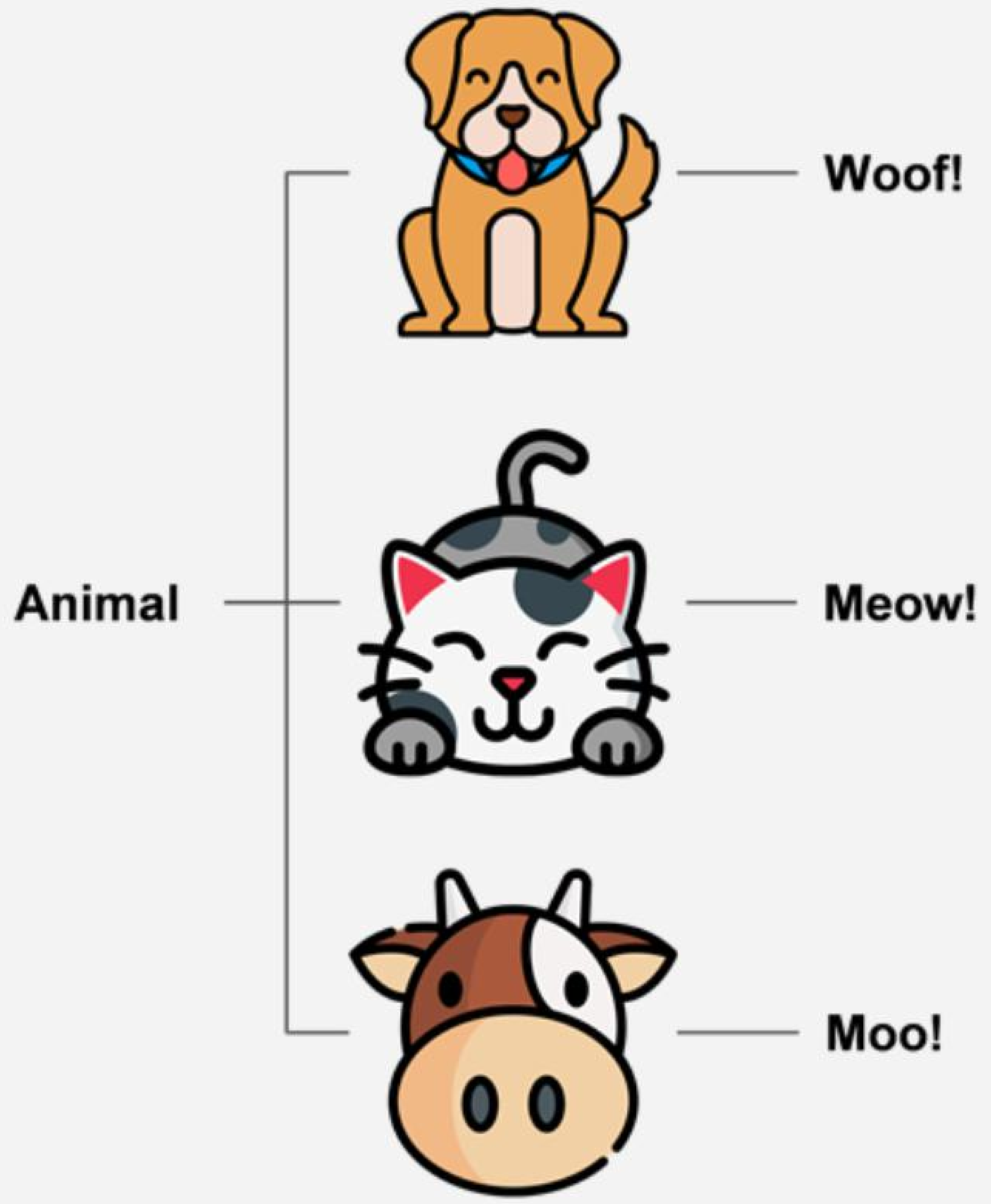
What is the common behavior among these animals?

They all can hear, see, and make sounds.

Different animals have different sounds.

Think and Tell

What behaviors do these animals have in common?

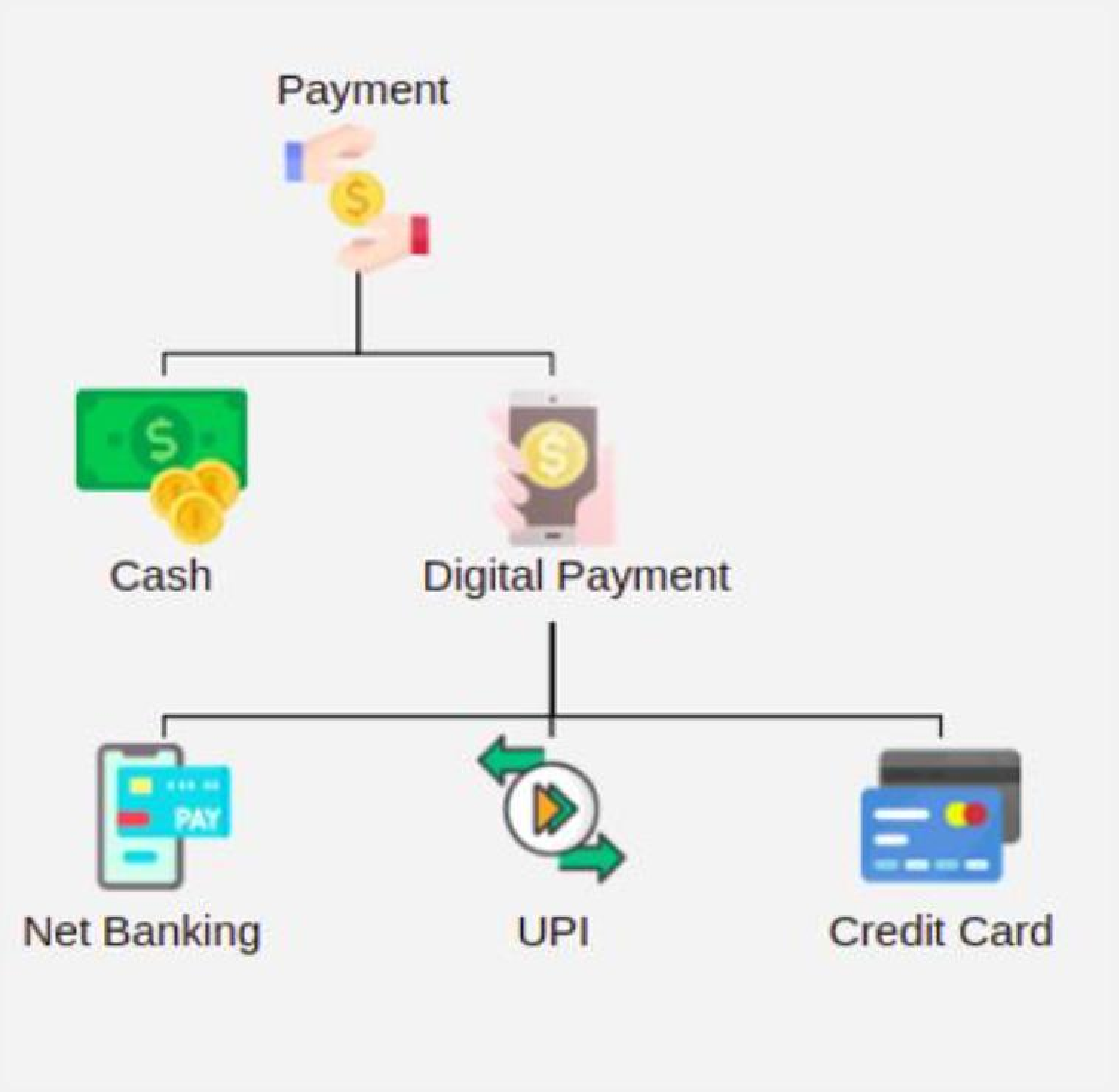


Slide Note

Most of us have cell phones these days.

How would you like to arrange your contact details?

Why would you like to arrange these numbers in this order?



Application Behavior

- What is the common behavior of these applications?

Menu

Implement Polymorphism

Learning Objectives

- Explore interfaces
- Implement an interface
- Define polymorphism
- Explain static polymorphism
- Explain dynamic polymorphism

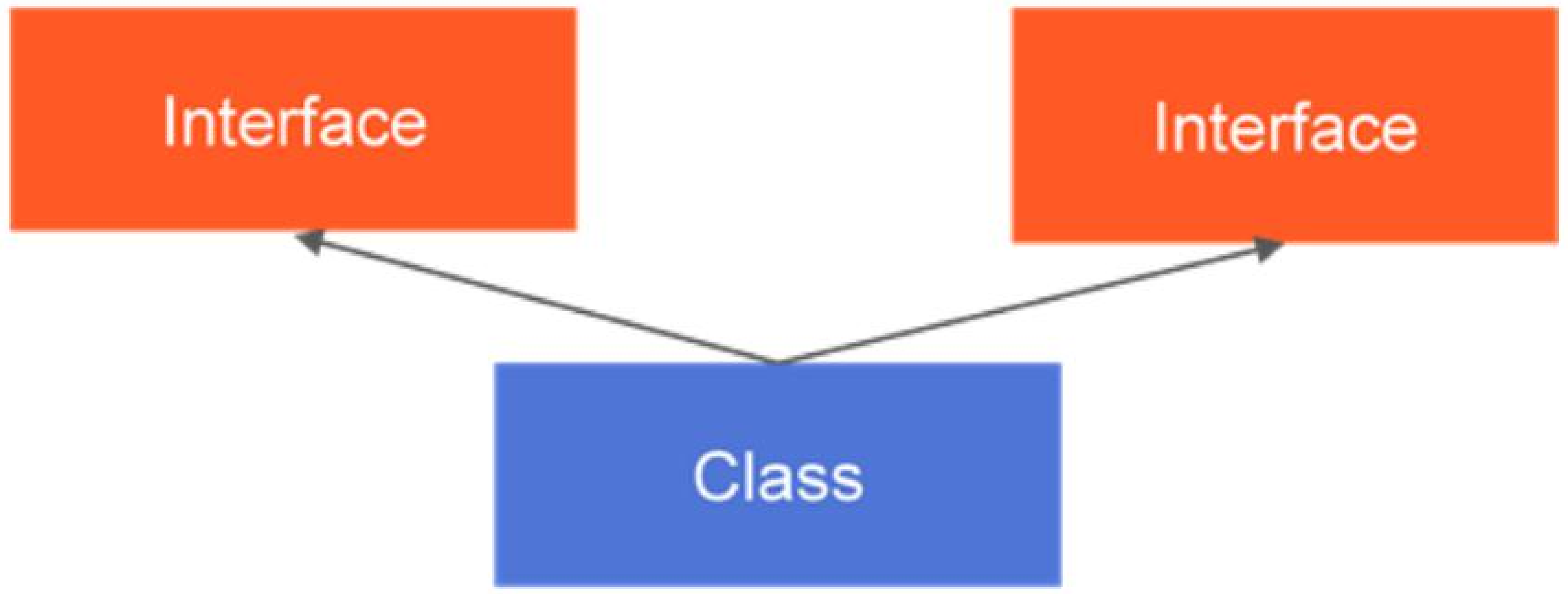
Think and Tell

- In previous sprints, you learned about inheritance.
- Java does not support multiple inheritance but suppose there is a scenario where a manager needs to implement some features of the `Employee` class. Also, the same manager wants to implement some features of another class.
- How can this scenario be implemented?

Explore Interfaces

What Are Interfaces?

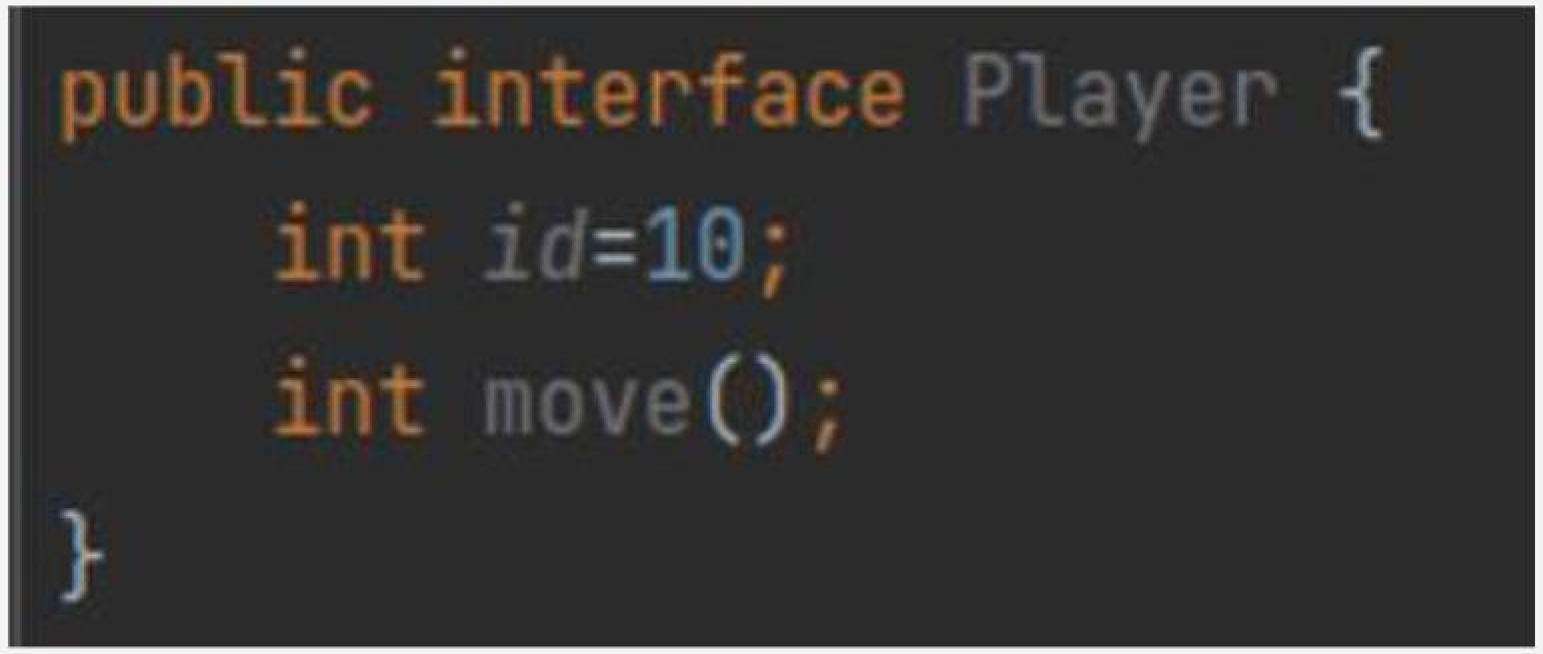
- The interface in Java is a mechanism to achieve abstraction and multiple inheritance.



- Interfaces in Java have only constant (static and final) variables and public abstract methods that do not have any implementation.
- Java interfaces also represent an **is-a** relationship.
- Interfaces impose a set of rules for the classes that implement them through the abstract methods.
- The class that implements that interface must provide implementation for all its abstract methods.

Understanding Interfaces

- An interface is defined using the `interface` keyword.
- The following rules need to be followed to implement an interface:
  - Variables in an interface are implicitly public, static, and final. Protected or private access specifiers cannot be used.
  - Methods that are unimplemented are implicitly public and abstract. They cannot be declared protected or private.
  - An interface does not have a constructor.
  - Interfaces do not have any instance variables.
  - **Like the abstract class, interfaces cannot be instantiated.**

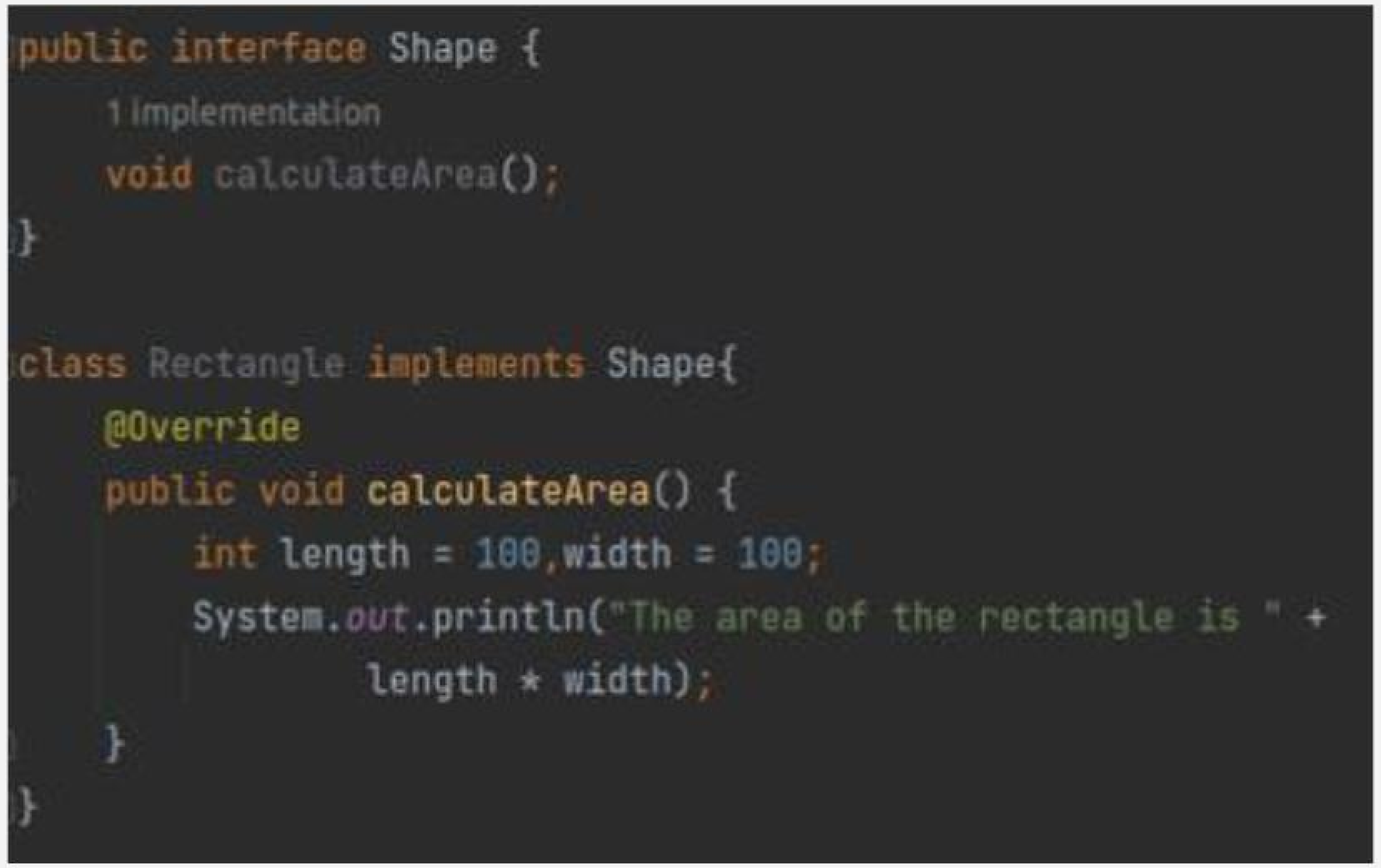


```
public interface Player {
    int id=10;
    int move();
}
```

Implement an Interface

Implementing an Interface

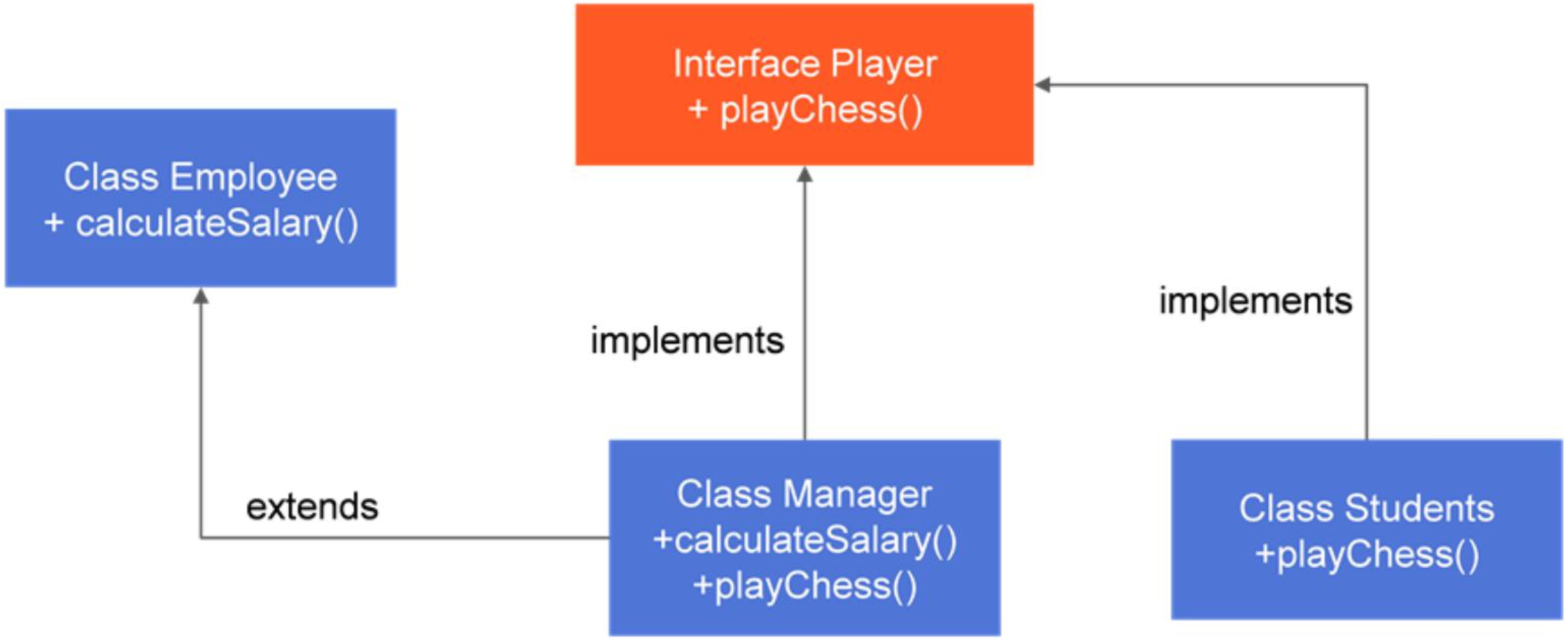
- An interface `Shape` is declared. The interface contains one method: `calculateArea()`.
- Here, the `Rectangle` class implements `Shape` and provides the implementation of the `calculateArea()` method.
- Here `calculateArea()` calculates the area of shapes, but the way area is calculated is different for different shapes.
- Hence, the implementation of `calculateArea()` is independent of other shapes.
- The rule is specified in the interface `Shape` through the abstract method `calculateArea()`.
- Any class that implements the `Shape` interface must provide an implementation for the `calculateArea()` method.



```
public interface Shape {
    1 implementation
    void calculateArea();
}

class Rectangle implements Shape{
    @Override
    public void calculateArea() {
        int length = 100,width = 100;
        System.out.println("The area of the rectangle is " +
                length * width);
    }
}
```

Unrelated Classes - Implement a Common Interface

- Interfaces are meant to share behavior across independent or different objects.
- The `Manager` and `Student` are two different objects and can implement the `Player` interface.



Gaming Console

Create an interface as GameConsole, having an abstract method computeScore.

Create Badminton and Chess classes that implement the functionality of the GameConsole interface.

Create a Main class, creating objects of the Badminton and Chess classes and calling the computeScore functionality.

Click here for the solution.
Intellij IDE -must be used for the demonstration.

DEMO

Think and Tell

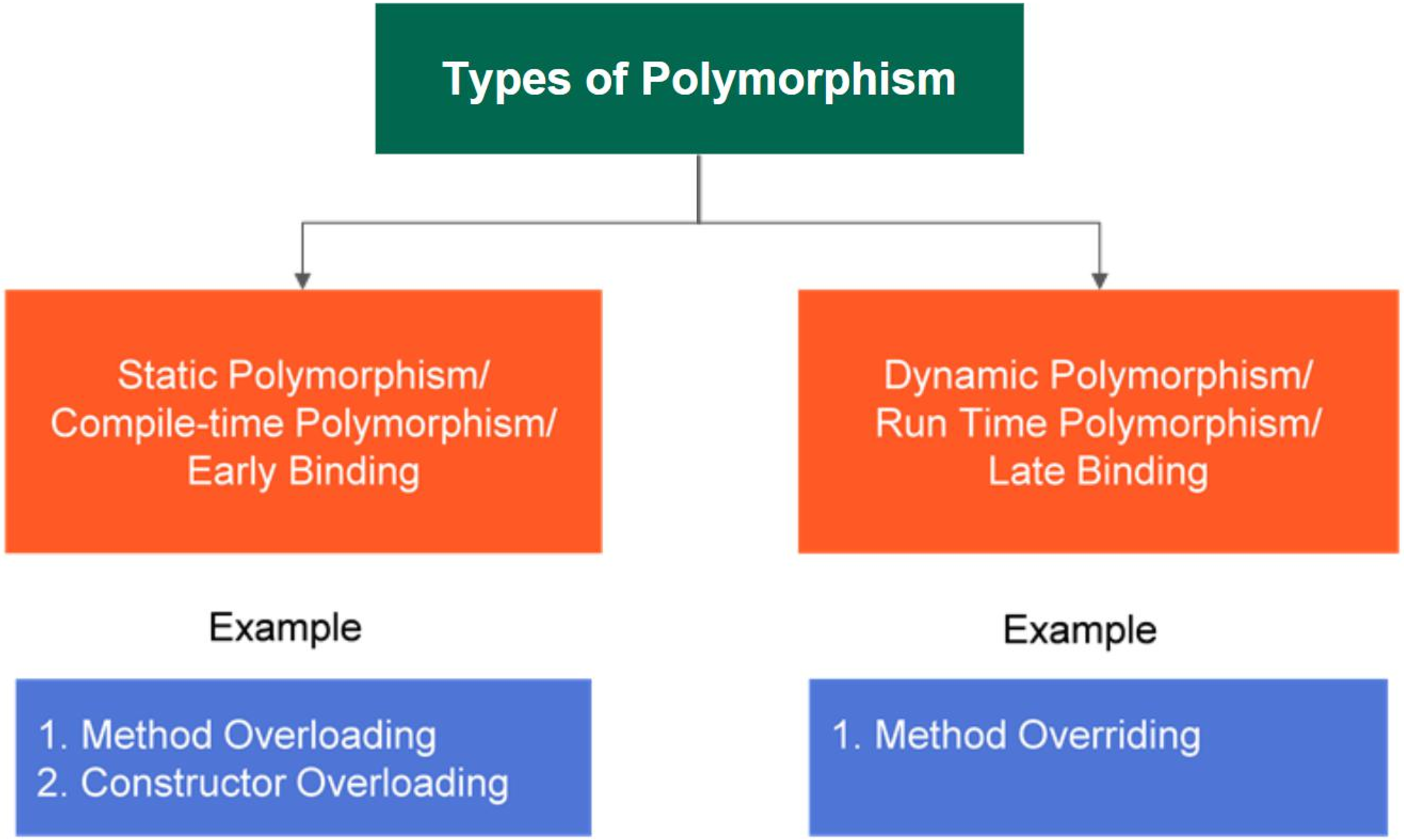
- How does a person's behavior change in different environments and in different roles?
  - At home, a person can be a father, son, etc.
  - At school, the same person can be a student or teacher.
  - At the market, the same person can be a shopkeeper or consumer.
  - At the office, the same person can be a manager or a trainee.
- A person is an object whose behavior changes based on the environment they are in.
- Can the same be done with Java objects?
- Does the behavior of Java objects change depending on the environment they are in?

Define Polymorphism

Introduction to Polymorphism

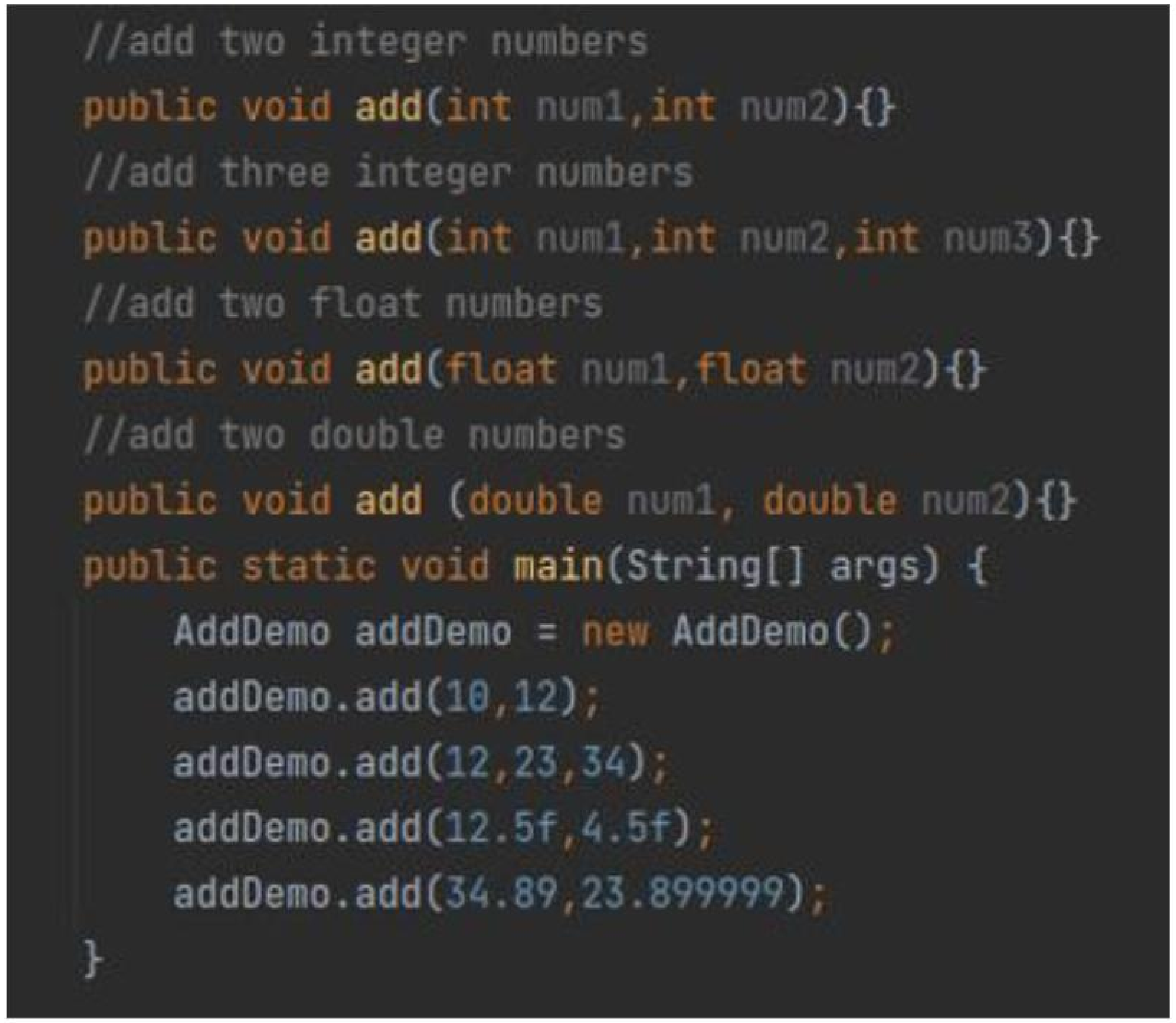
- Polymorphism is a combination of two words: “poly” means “many” and “morph” means “forms.”
- Polymorphism means “the state of existing in different forms.”
- In OOP, polymorphism defines the ability of an object, method, or variable to take on different forms based on the characteristics required.
- Polymorphism enables an object to change its behavior based on the method being called.
- Similar~~,~~ to inheritance, polymorphism is also a core OOP concept; however, it primarily aims to improve code readability and provide a common interface to interact with objects.

Types of Polymorphism


Types of Polymorphism
Static Polymorphism/ Compile-time Polymorphism/ Early Binding
Dynamic Polymorphism/ Run Time Polymorphism/ Late Binding
Example
1. Method Overloading
2. Constructor Overloading
Example
1. Method Overriding

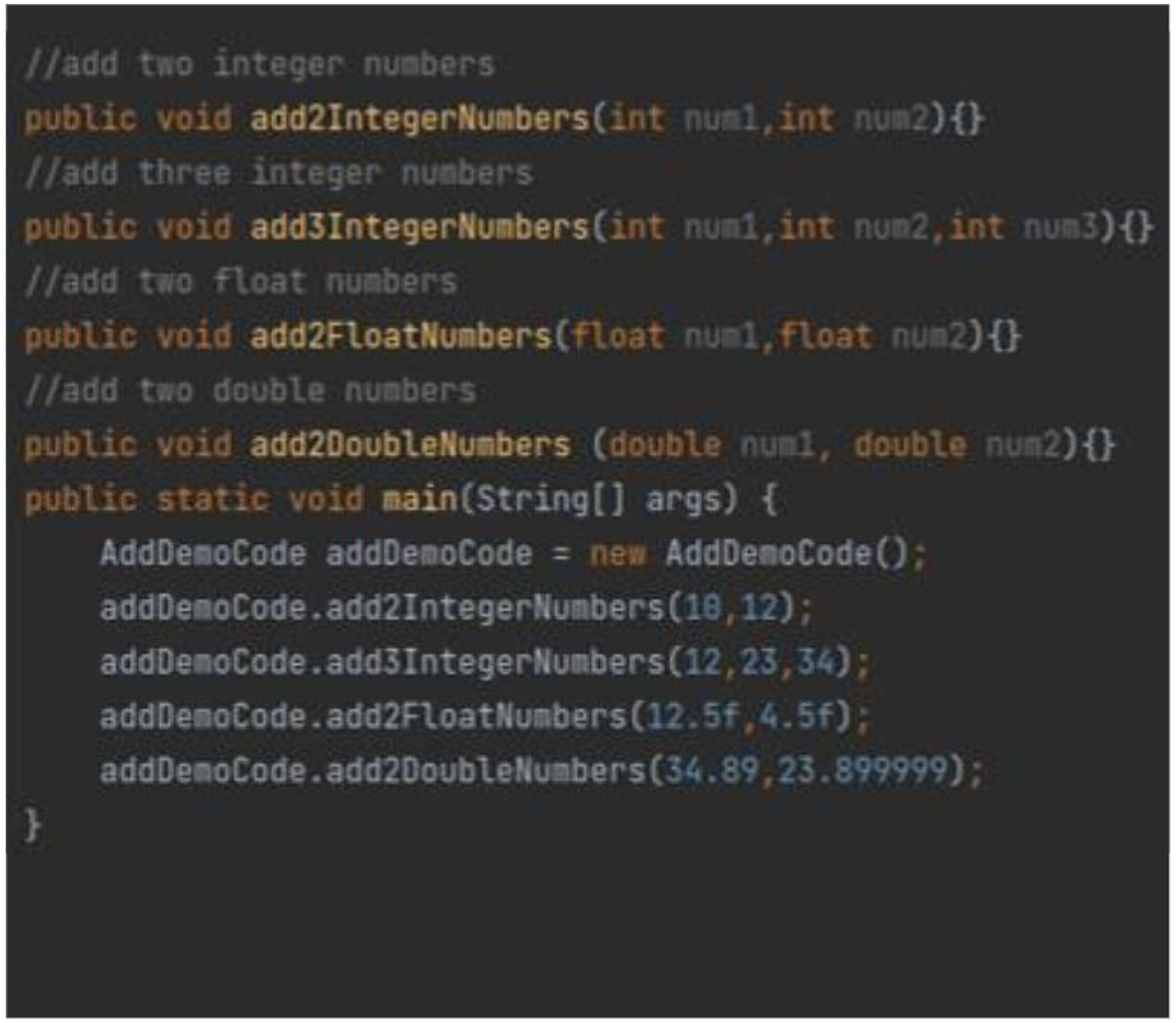
Which Code Is Easy, Maintainable, and Reusable?

Here the method name is the same [that is, `add()`], and it is adding different numbers by taking different parameters.



```
//add two integer numbers
public void add(int num1,int num2){}
//add three integer numbers
public void add(int num1,int num2,int num3){}
//add two float numbers
public void add(float num1,float num2){}
//add two double numbers
public void add (double num1, double num2){}
public static void main(String[] args) {
    AddDemo addDemo = new AddDemo();
    addDemo.add(10,12);
    addDemo.add(12,23,34);
    addDemo.add(12.5f,4.5f);
    addDemo.add(34.89,23.899999);
}
```

Here, the method name is not the same. These methods are also doing the same task of adding different numbers.

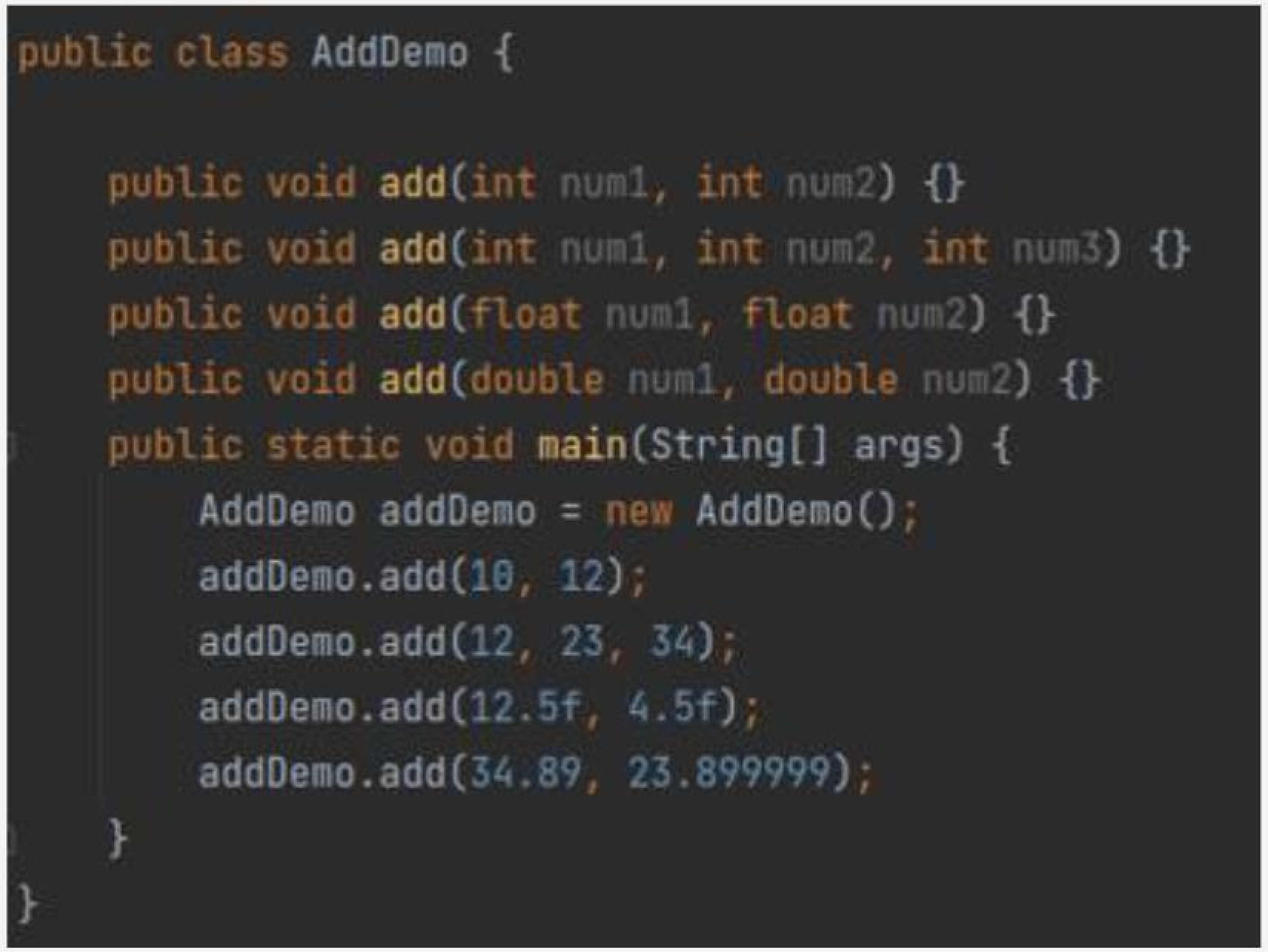


```
//add two integer numbers
public void add2IntegerNumbers(int num1,int num2){}
//add three integer numbers
public void add3IntegerNumbers(int num1,int num2,int num3){}
//add two float numbers
public void add2FloatNumbers(float num1,float num2){}
//add two double numbers
public void add2DoubleNumbers (double num1, double num2){}
public static void main(String[] args) {
    AddDemoCode addDemoCode = new AddDemoCode();
    addDemoCode.add2IntegerNumbers(10,12);
    addDemoCode.add3IntegerNumbers(12,23,34);
    addDemoCode.add2FloatNumbers(12.5f,4.5f);
    addDemoCode.add2DoubleNumbers(34.89,23.899999);
}
```

Explain Static Polymorphism

Static Polymorphism - Method Overloading

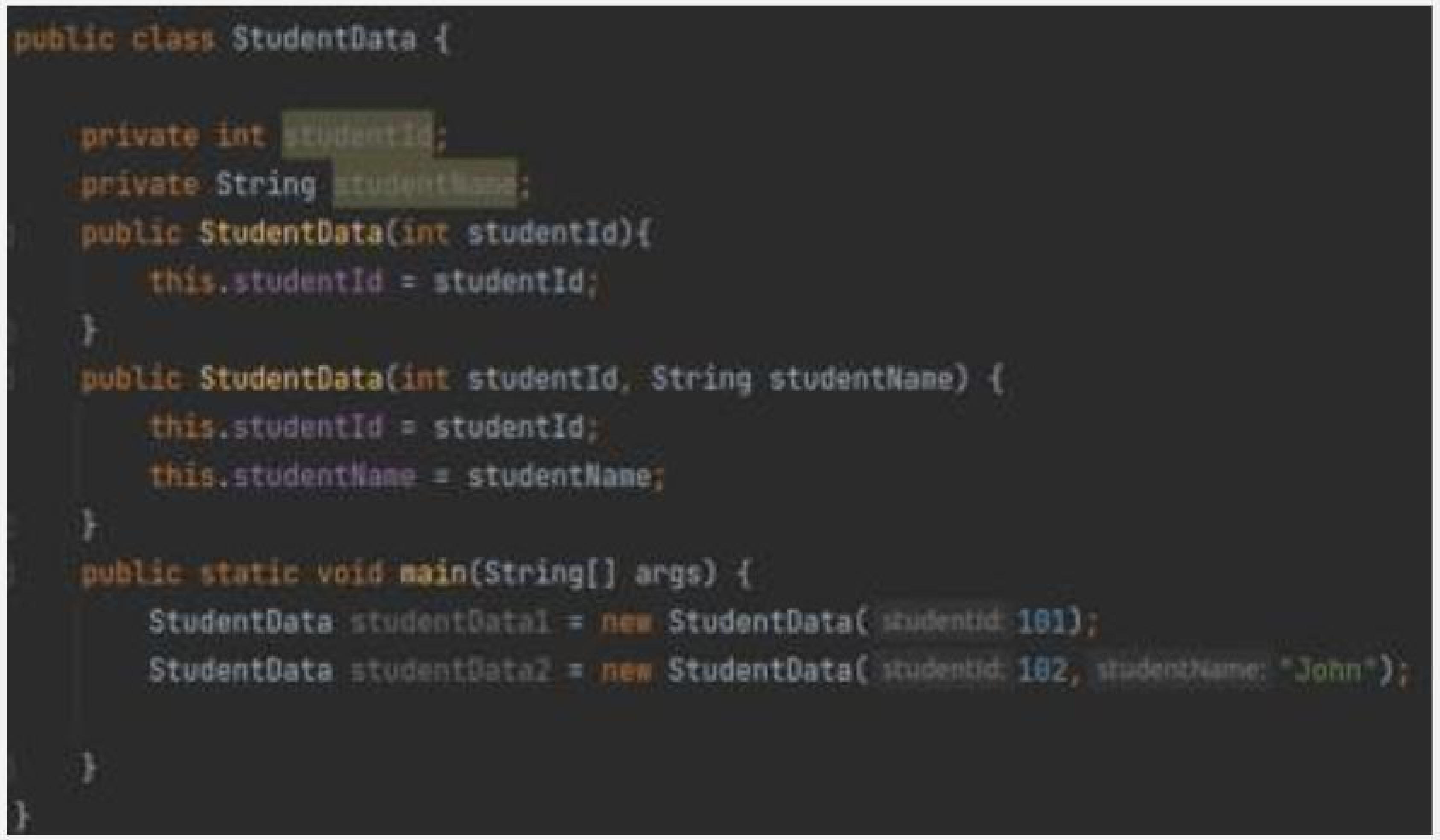
- In overloaded methods, a class has multiple methods with the same name but different parameters. This feature is called method overloading.
- The `add()` method is overloaded. These methods have the same name but accept different arguments.
- Here, the same method, `add()`, will perform different operations based on the parameter.
- The `add()` method that is called is determined by the compiler. Hence, it is also known as **compile-time polymorphism.**
- Method overloading makes the code more readable and easier to maintain.
- Method overloading is achieved by either:
  - Changing the number of arguments, or
  - Changing the data types of arguments.



```
public class AddDemo {

    public void add(int num1, int num2) {}
    public void add(int num1, int num2, int num3) {}
    public void add(float num1, float num2) {}
    public void add(double num1, double num2) {}
    public static void main(String[] args) {
        AddDemo addDemo = new AddDemo();
        addDemo.add(10, 12);
        addDemo.add(12, 23, 34);
        addDemo.add(12.5f, 4.5f);
        addDemo.add(34.89, 23.899999);
    }
}
```

Static Polymorphism – Constructor Overloading

- As with method overloading, you can also create two or more constructors with different parameters. This is called constructor overloading.
- Constructor overloading enables the creation of the object of a specific class in several ways:
    - Based on the parameter passed, different constructors are called, and different values are assigned.
    - Objects are initialized with different data types.
- Here, `studentData(int studentId)` is an overloaded constructor, with another constructor `studentData(int studentId,String studentName)`.
- In the main method, two objects are created by calling the respective overloaded constructor.



```
public class StudentData {

    private int studentId;
    private String studentName;
    public StudentData(int studentId){
        this.studentId = studentId;
    }
    public StudentData(int studentId, String studentName) {
        this.studentId = studentId;
        this.studentName = studentName;
    }
    public static void main(String[] args) {
        StudentData studentData1 = new StudentData( studentId: 101);
        StudentData studentData2 = new StudentData( studentId: 102, studentName: "John");

    }
}
```

Add Different Numbers

Write a Java program to implement method overloading and overload the `add()` method with different parameters.

1) Add three integer values.
2) Add two float values.
3) Add three integer values.

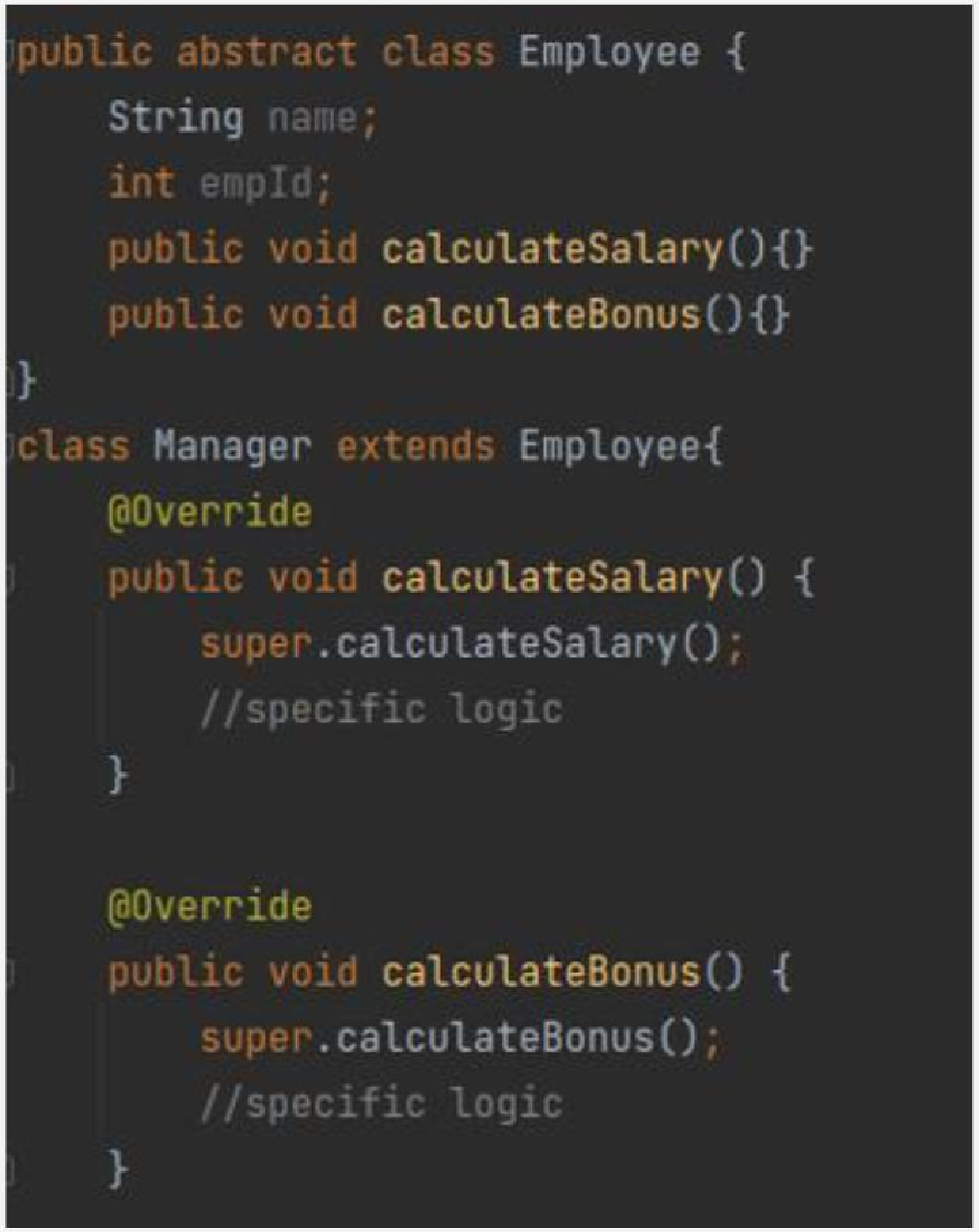
Click here for the solution.
Intellij IDE must be used for the demonstration.

DEMO

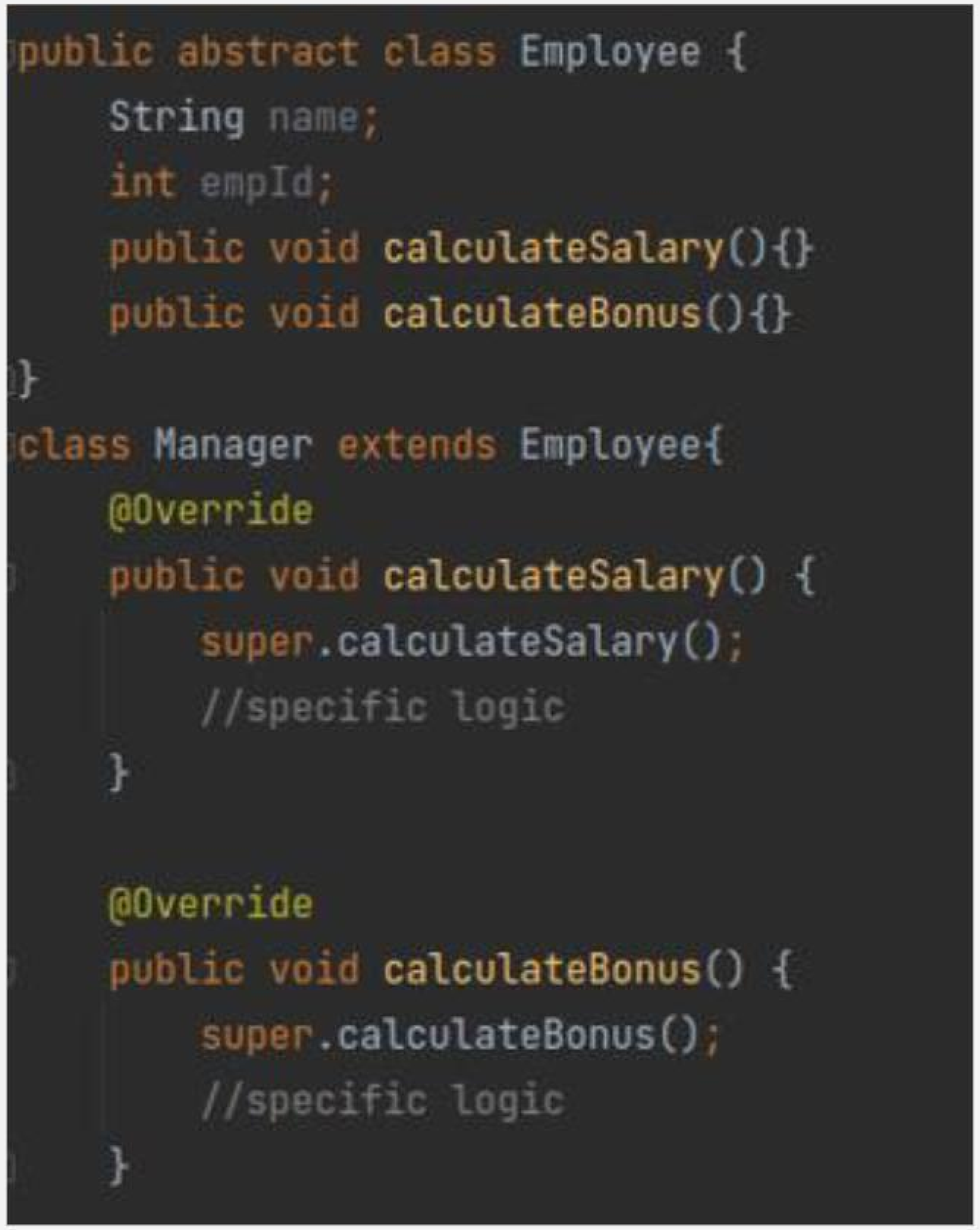
Explain Dynamic Polymorphism

Dynamic Polymorphism – Method Overriding

- Dynamic polymorphism is implemented in Java by method overriding.
- When the same method is defined in a superclass and in a subclass, then the method of the subclass overrides the method of the superclass. This is known as method overriding.
- Method overriding enables a subclass to provide its own implementation of a method that already has an implementation defined in its superclass.
- To override a method~~;~~ present in the superclass, the subclass method should have:
  - The same name.
  - The same parameters.
  - The same return type as the method in the superclass.



```
public abstract class Employee {
    String name;
    int empId;
    public void calculateSalary(){}
    public void calculateBonus(){}
}
class Manager extends Employee{
    @Override
    public void calculateSalary() {
        super.calculateSalary();
        //specific logic
    }

    @Override
    public void calculateBonus() {
        super.calculateBonus();
        //specific logic
    }
```



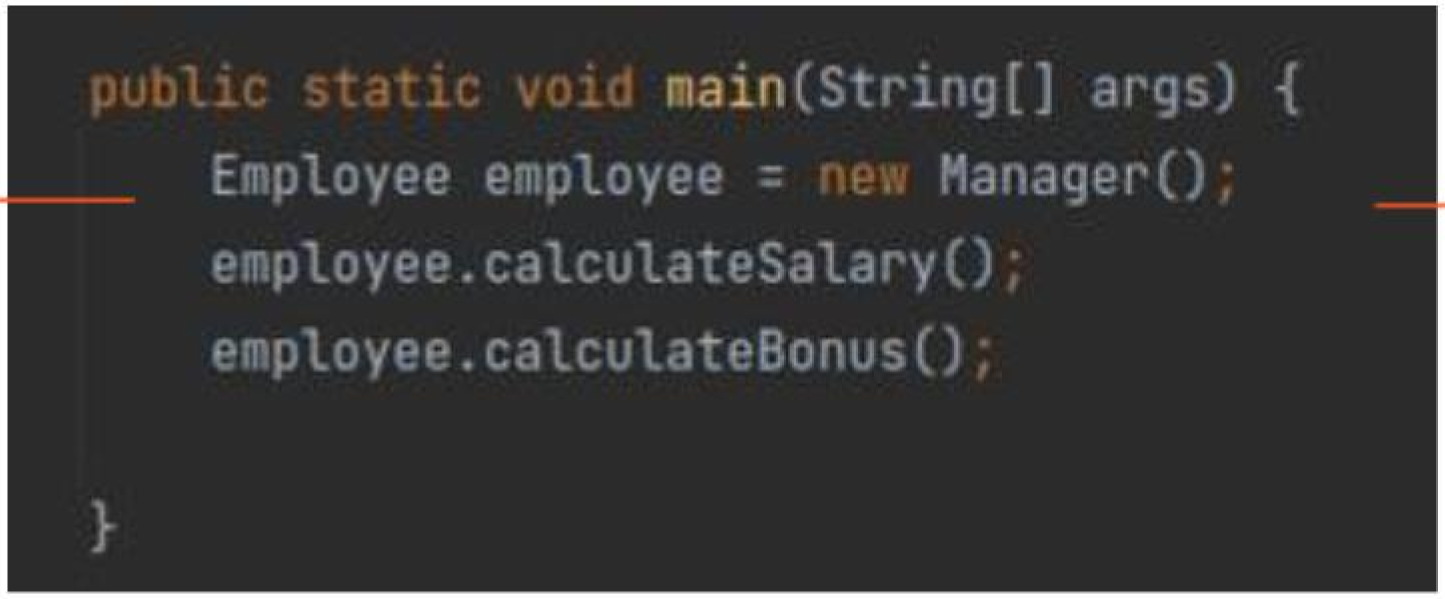
```
public abstract class Employee {
    String name;
    int empId;
    public void calculateSalary(){}
    public void calculateBonus(){}
}
class Manager extends Employee{
    @Override
    public void calculateSalary() {
        super.calculateSalary();
        //specific logic
    }

    @Override
    public void calculateBonus() {
        super.calculateBonus();
        //specific logic
    }
```

Method Overriding

- The `Manager` class extends the `Employee` class.
- So, as per inheritance, all the methods from the parent class can be inherited to the child class.
- Here, `calculateSalary()` and `calculateBonus()` are overridden methods.
- Method overriding supports code reusability by allowing slight changes in the implementation of certain methods in the derived class while keeping the rest the same as in the parent class.
- The `@Override` annotation tells the compiler that the child class method overrides the parent class method.
- Method overriding can be achieved through the abstract method and without the abstract method.

Method Overriding and Dynamic Method Dispatch

The employee variable is a reference for the Employee class.



```
public static void main(String[] args) {
    Employee employee = new Manager();
    employee.calculateSalary();
    employee.calculateBonus();

}
```

The employee variable is an object for the Manager class.

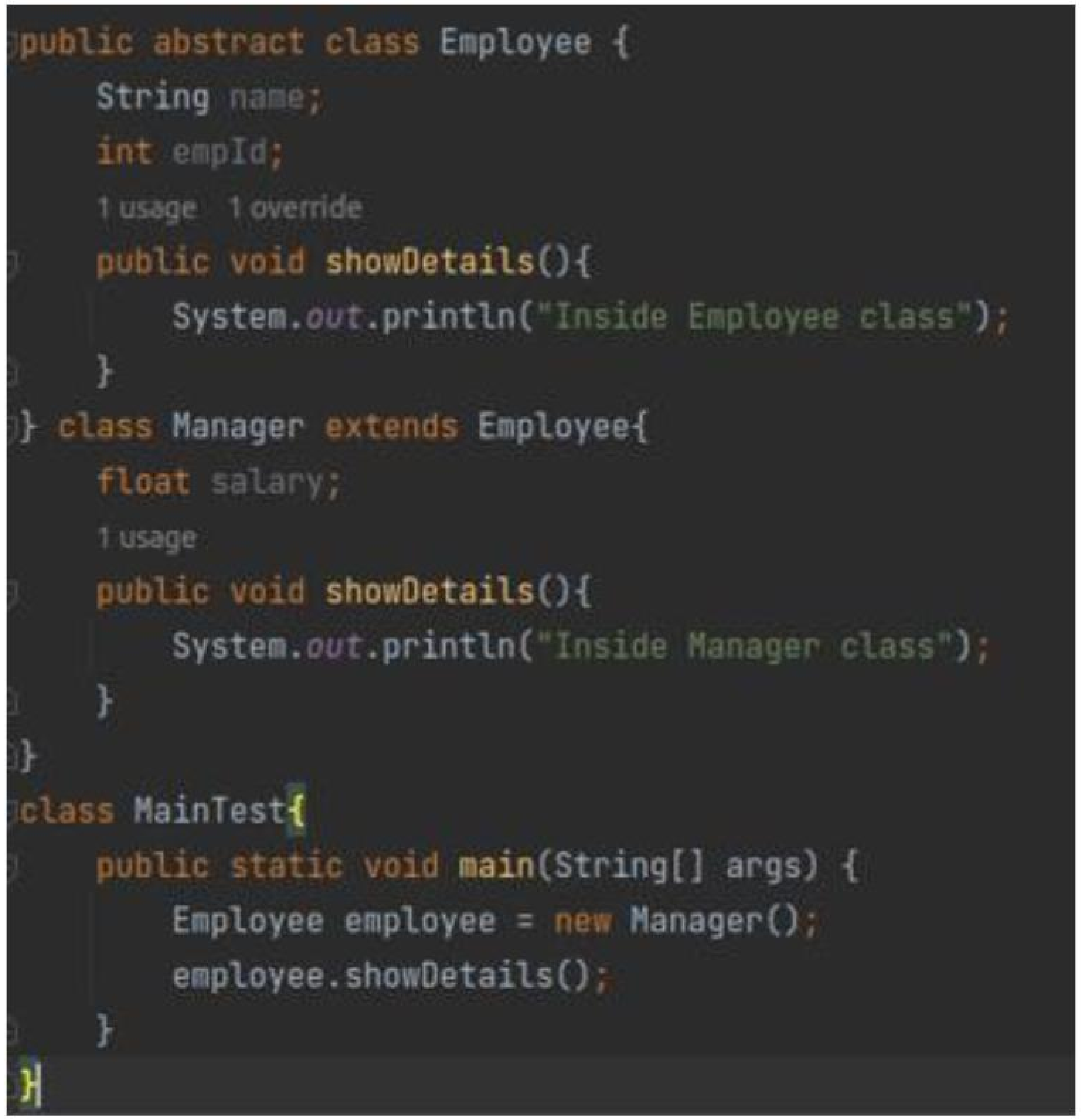
- Here, the employee variable is a reference of the Employee class but an object of the Manager class.
  - Here, the employee variable will check the calculateSalary() and calculateBonus() methods in the Employee class during compile time. This is called early binding.
  - But at runtime, it will become an object of the Manager class. Hence, at run time, it will call the calculateSalary() and calculateBonus() methods of the Manager class. This is also called late binding.
- Since method invocation is determined by the JVM at execution time, it is known as **runtime polymorphism or dynamic polymorphism.**
- The process whereby the call to the overridden method is resolved at runtime is called the dynamic method dispatch.

Method Overloading vs. Method Overriding

Method Overloading	Method Overriding
This is also called static polymorphism or compile-time polymorphism.	This is also called dynamic polymorphism or runtime polymorphism.
Overloading occurs between the methods in the same class.	Overriding occurs between the superclass and subclass.
Overloaded method names are the same, but the parameters are different.	Overriding methods have the same signature, i.e., the same name and method arguments.
In method overloading, the method to call is determined at compile time.	In method overriding, the method to call is determined at runtime based on the object type.

Quick Check

Predict the output for the following code:



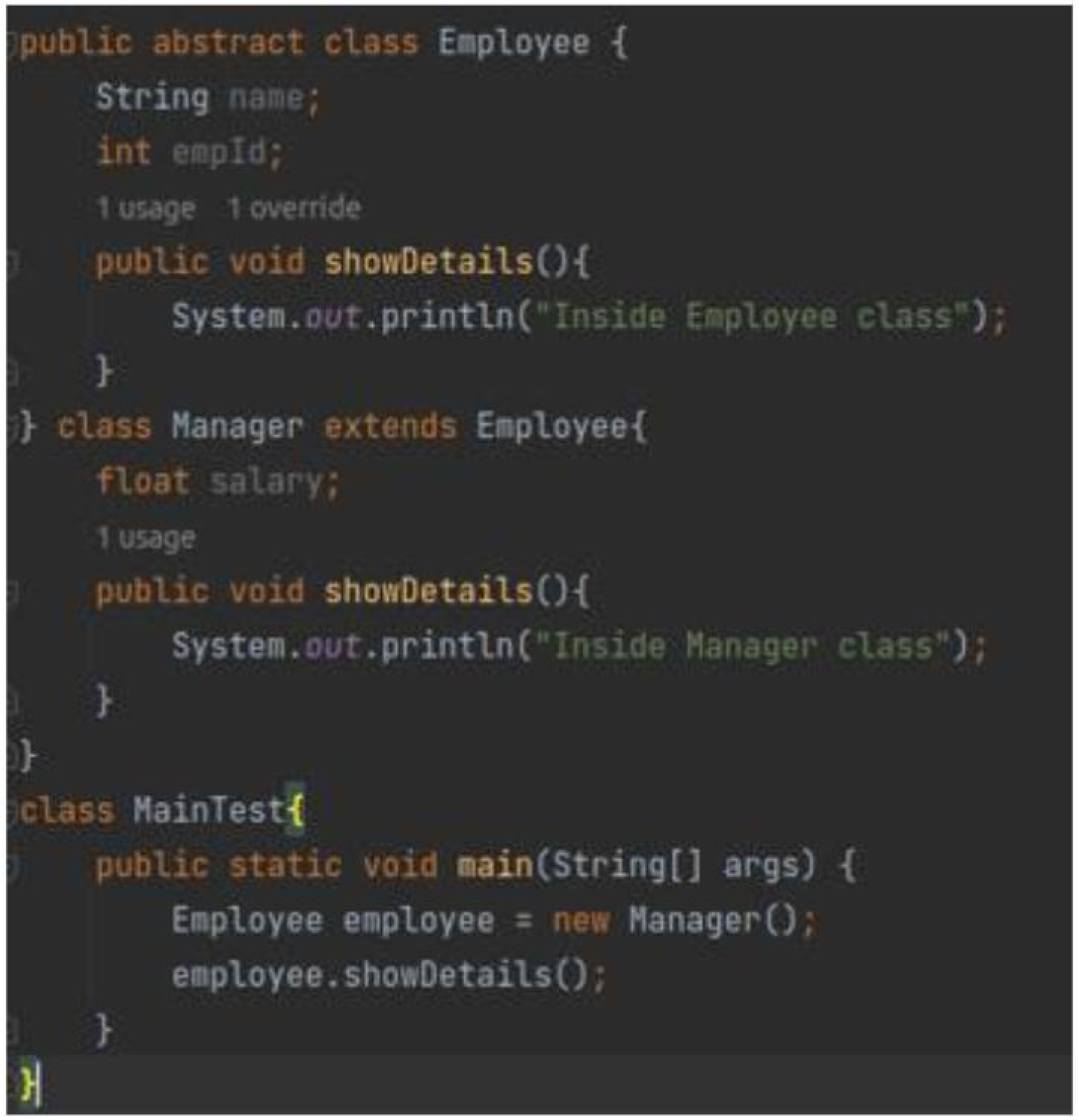
```
public abstract class Employee {
    String name;
    int empId;
    1 usage   1 override
    public void showDetails(){
        System.out.println("Inside Employee class");
    }
} class Manager extends Employee{
    float salary;
    1 usage
    public void showDetails(){
        System.out.println("Inside Manager class");
    }
}
class MainTest{
    public static void main(String[] args) {
        Employee employee = new Manager();
        employee.showDetails();
    }
}
```

1. Inside `Employee` class
2. Inside `Manager` class
3. Runtime exception
4. Inside `Employee` class

   Inside `Manager` class

Quick Check: Solution

Predict the output for the following code:



```java
public abstract class Employee {
    String name;
    int empId;
    1 usage   1 override
    public void showDetails(){
        System.out.println("Inside Employee class");
    }
} class Manager extends Employee{
    float salary;
    1 usage
    public void showDetails(){
        System.out.println("Inside Manager class");
    }
}
class MainTest{
    public static void main(String[] args) {
        Employee employee = new Manager();
        employee.showDetails();
    }
}
```

1. Inside `Employee` class
2. **Inside `Manager` class**
3. Runtime exception
4. Inside `Employee` class

   Inside `Manager` class

Method Overriding

Write a Java program to calculate the salaries of `SalesManager` and `TechLead`.
`SalesManager` and `TechLead` extend the `Employee` class and override all its method.

Inside the main method, create the object of the `SalesManager` and `TechLead` classes, call the calculate salary method, and print the respective salary.

Click here for the solution.
Intellij IDE must be used for the demonstration.

DEMO