ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

EXERCISE 1

THIẾT KẾ PHẦN MỀM – CQ2021/3

21120280 – Lý Minh Khuê

Thành phố Hồ Chí Minh – 2024

TABLE OF CONTENTS

INTRODUCTION

Abstract classes and interfaces are essential concepts in object-oriented programming (OOP), widely used in languages like Java, C#, and others. They provide a means to define contracts for classes and establish a blueprint for implementing functionality. In this report, we will delve into the differences between abstract classes and interfaces, their characteristics and use cases.

ABSTRACT CLASSES

An abstract class acts as a blueprint for other classes to inherit from. It defines a set of functionalities, some of which can be implemented directly, while others are left abstract, requiring subclasses to provide their specific implementations.

Abstract classes are classes that cannot be instantiated on their own and may contain one or more abstract methods. These methods are declared without implementation, serving as placeholders for concrete subclasses to implement.

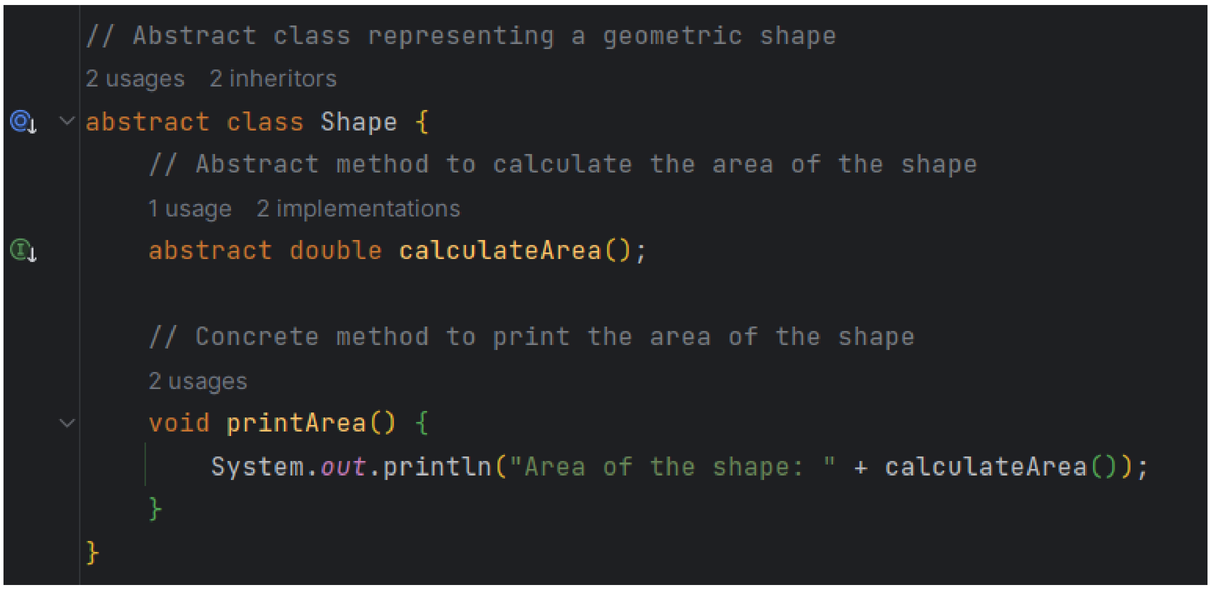
1. Key characteristics of abstract classes:

1.1. Partial Implementation:

Abstract classes can contain both abstract methods (without implementation) and concrete methods (with implementation).
Abstract methods define the expected behaviour of classes that inherit from the abstract class, while concrete methods define the actual implementation of that behaviour.

Ex:

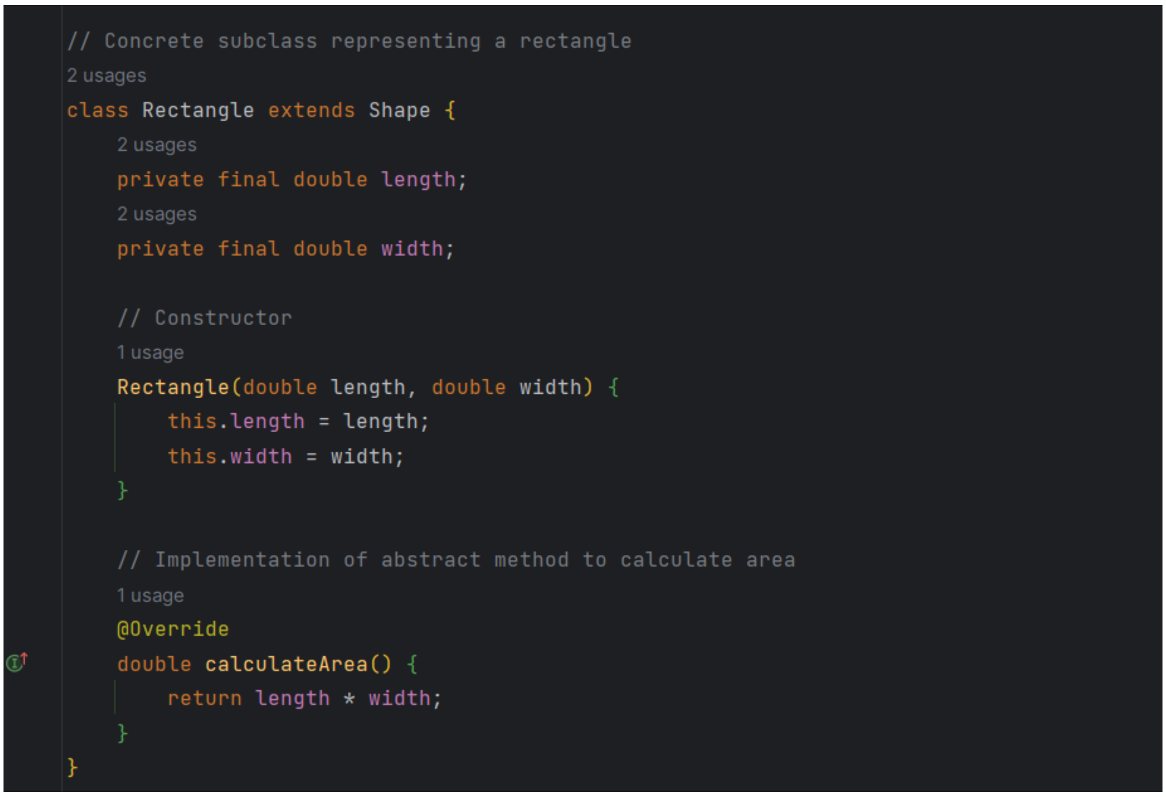
The `Shape` class is an abstract class with an abstract method `calculateArea()` and a concrete method `printArea()`.



```
// Abstract class representing a geometric shape
2 usages   2 inheritors
abstract class Shape {
    // Abstract method to calculate the area of the shape
    1 usage   2 implementations
    abstract double calculateArea();

    // Concrete method to print the area of the shape
    2 usages
    void printArea() {
        System.out.println("Area of the shape: " + calculateArea());
    }
}
```

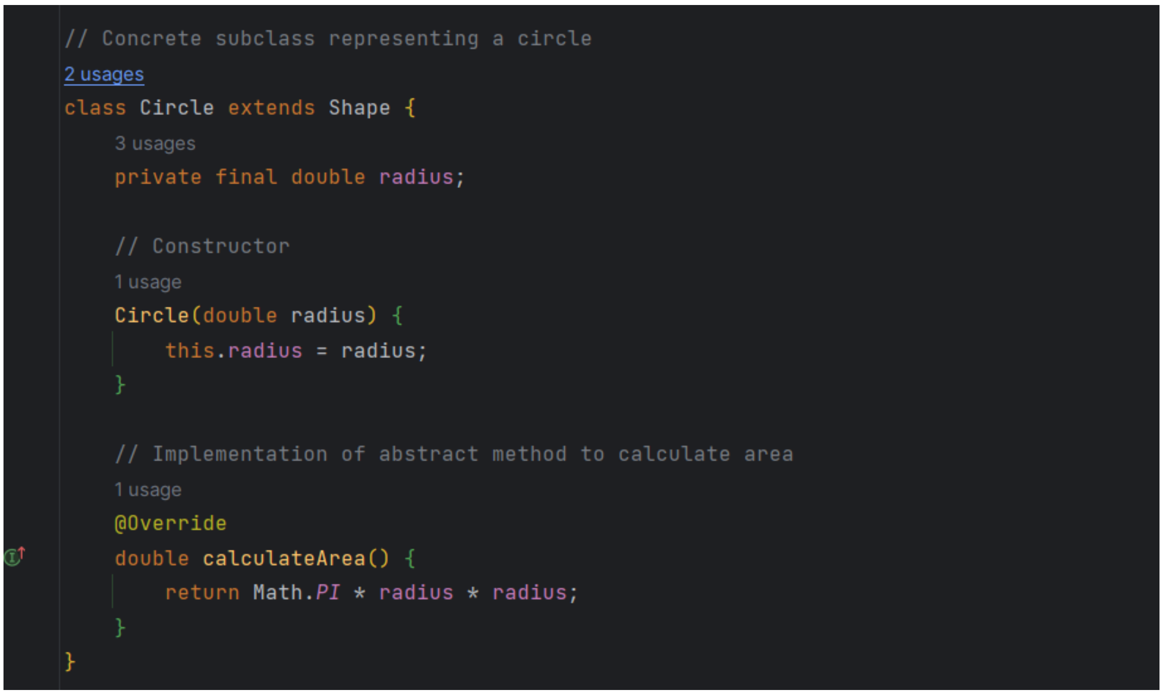
Figure 1. Shape abstract class

The `Rectangle` and `Circle` classes are concrete subclasses of `Shape`, each provides its implementation for the `calculateArea()` method according to its specific shape.



```
// Concrete subclass representing a rectangle
2 usages
class Rectangle extends Shape {
    2 usages
    private final double length;
    2 usages
    private final double width;

    // Constructor
    1 usage
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Implementation of abstract method to calculate area
    1 usage
    @Override
    double calculateArea() {
        return length * width;
    }
}
```

Figure 2. Rectangle concrete subclass



```
// Concrete subclass representing a circle
2 usages
class Circle extends Shape {
    3 usages
    private final double radius;

    // Constructor
    1 usage
    Circle(double radius) {
        this.radius = radius;
    }

    // Implementation of abstract method to calculate area
    1 usage
    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

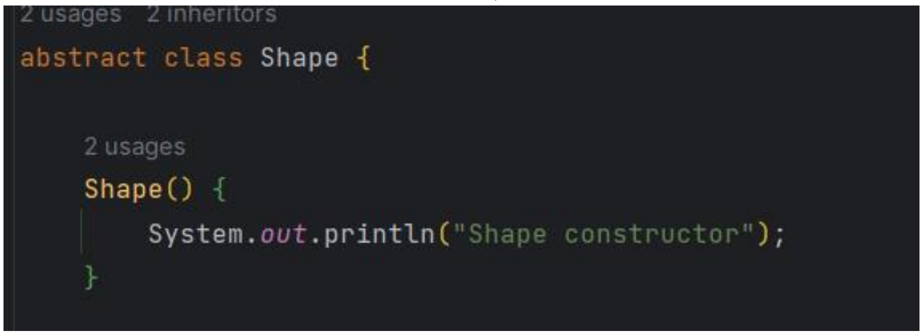
Figure 3. Circle concrete subclass

1.2. Inheritance:

Abstract classes can be extended by subclasses, allowing for code reuse and hierarchical organization. Subclasses benefit from the functionality defined in the abstract class.

1.3. Constructor Usage:

Abstract classes can have constructors, which are invoked when a subclass is instantiated.



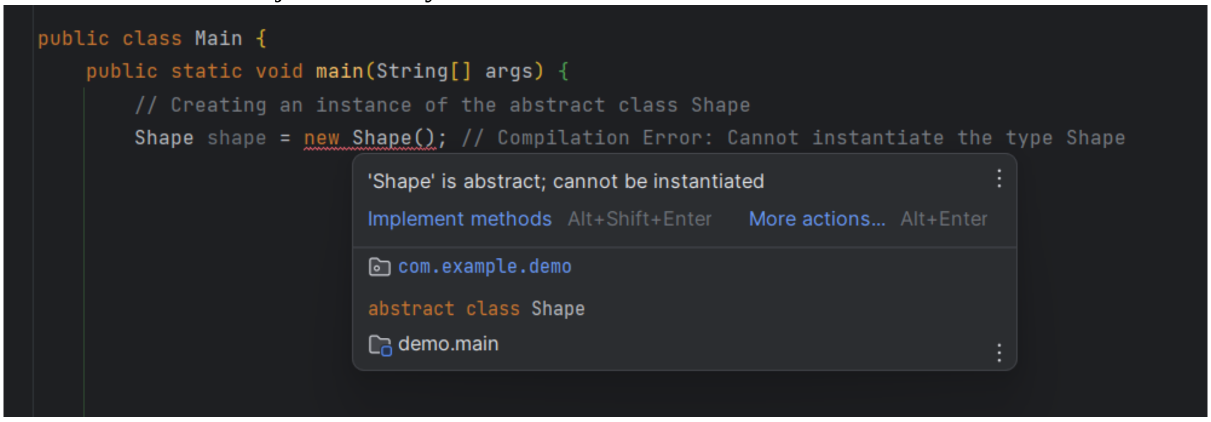
```
2 usages  2 inheritors
abstract class Shape {

    2 usages
    Shape() {
        System.out.println("Shape constructor");
    }
```

Figure 4. Constructor Usage

When creating instances of ‘Rectangle’ and ‘Circle’, the superclass constructor are called, and the message “Shape constructor” will be printed.

1.4. Cannot be instantiated:

You cannot create objects directly from an abstract class.



```
public class Main {
    public static void main(String[] args) {
        // Creating an instance of the abstract class Shape
        Shape shape = new Shape(); // Compilation Error: Cannot instantiate the type Shape
```

```
'Shape' is abstract; cannot be instantiated
Implement methods  Alt+Shift+Enter    More actions...  Alt+Enter
com.example.demo
abstract class Shape
demo.main
```

Figure 5. Abstract class cannot be instatiated

2. Use cases for abstract classes:

2.1. Providing a common structure for related classes:

An abstract class for animals can define methods like "eat", “make a sound” and "move" which subclasses like "Dog" and "Bird" can implement in their own ways.

2.2. Enforcing partial implementation:

An abstract class for payment processing can define a core "processPayment" method, requiring subclasses like "CreditCardPayment" and "CashPayment" to handle specifics.

INTERFACES

An interface defines a contract – a set of methods that a class must implement. It specifies the functionalities a class must provide without dictating how those functionalities are achieved.

1. Key characteristics of interfaces:

1.1. Full Abstraction:

Interfaces only contain method signatures without any implementation details.

1.2. No Constructors:

Interfaces cannot have constructors as they do not define any state or behavior.

1.3. Cannot be instantiated:

Similar to abstract classes, you cannot create objects directly from interfaces.

1.4. Promote loose coupling:

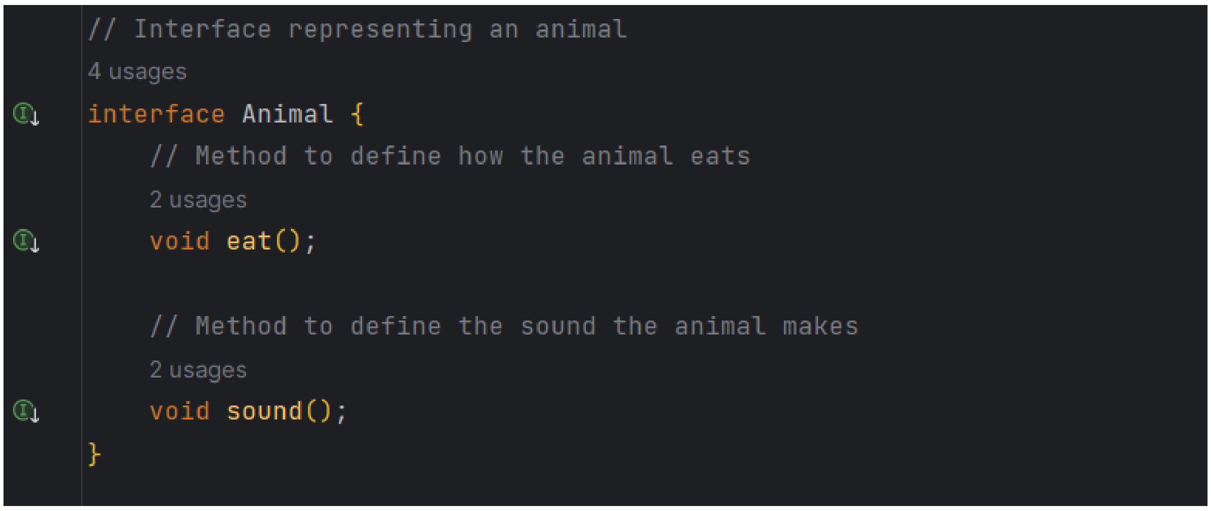
Classes implementing the interface can have different implementations, making the code more adaptable.

1.5. Support multiple inheritance:

A class can implement multiple interfaces, inheriting functionalities from each.

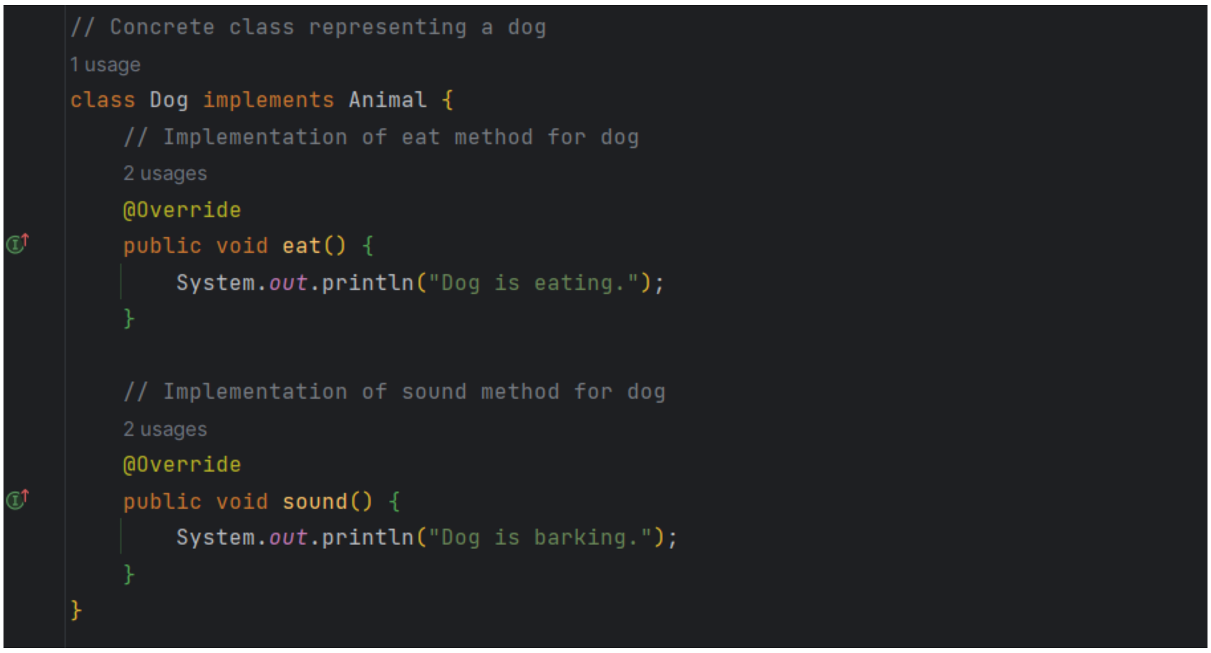
1.6. Example:

This `Animal` interface defines `eat()` and `sound()` methods to represent common behaviors of animals.



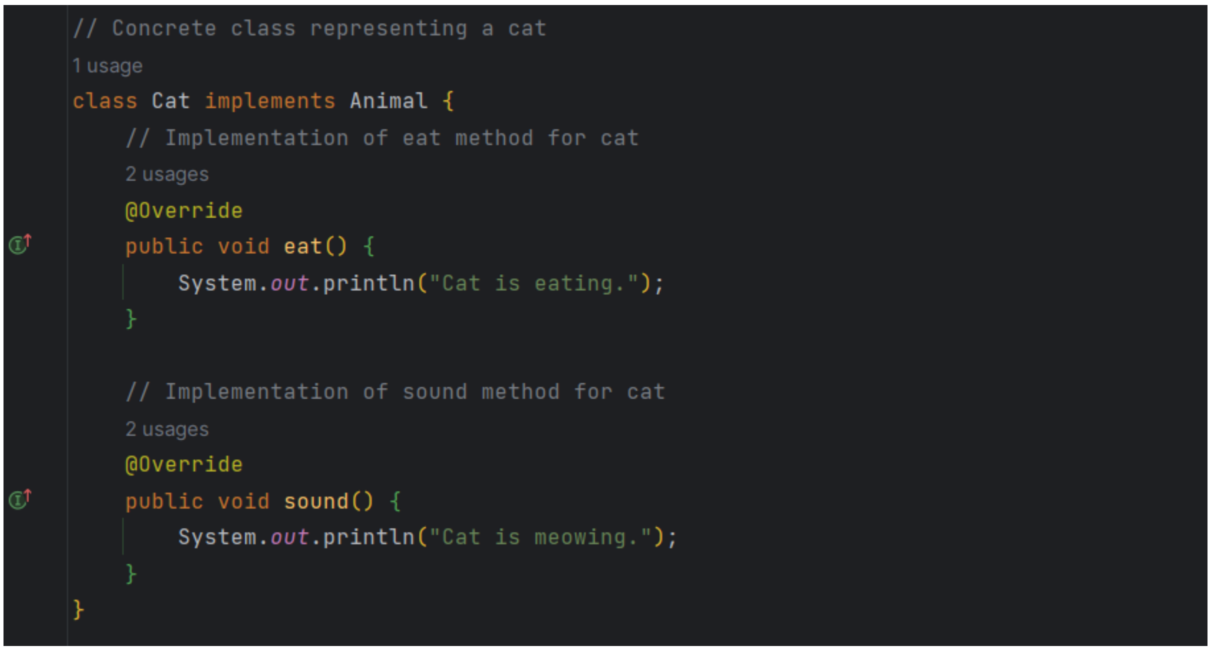
```java
// Interface representing an animal
4 usages
interface Animal {
    // Method to define how the animal eats
    2 usages
    void eat();

    // Method to define the sound the animal makes
    2 usages
    void sound();
}
```

Figure 6. Animal interface



```
// Concrete class representing a dog
1 usage
class Dog implements Animal {
    // Implementation of eat method for dog
    2 usages
    @Override
    public void eat() {
        System.out.println("Dog is eating.");
    }

    // Implementation of sound method for dog
    2 usages
    @Override
    public void sound() {
        System.out.println("Dog is barking.");
    }
}
```

Figure 7. Dog concrete class



```
// Concrete class representing a cat
1 usage
class Cat implements Animal {
    // Implementation of eat method for cat
    2 usages
    @Override
    public void eat() {
        System.out.println("Cat is eating.");
    }

    // Implementation of sound method for cat
    2 usages
    @Override
    public void sound() {
        System.out.println("Cat is meowing.");
    }
}
```

Figure 8. Cat concrete class

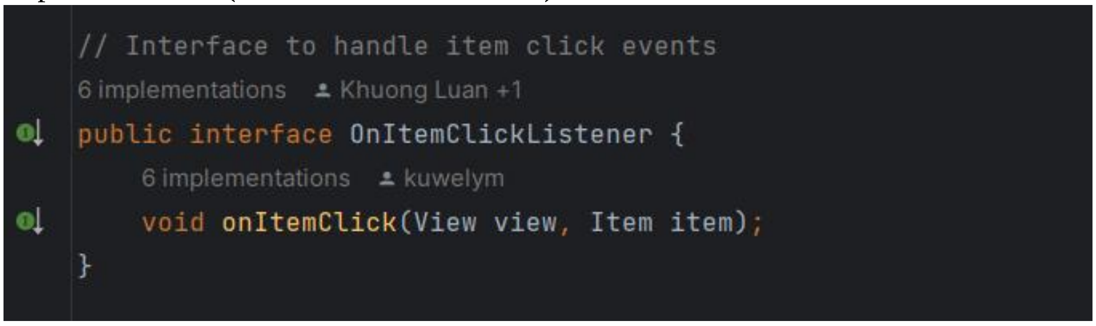
2. Use cases for interfaces:

2.1. Defining functionalities for different object types:

A "Drawable" interface can specify a "draw" method that both "Shape" and "Image" classes can implement for rendering purposes.

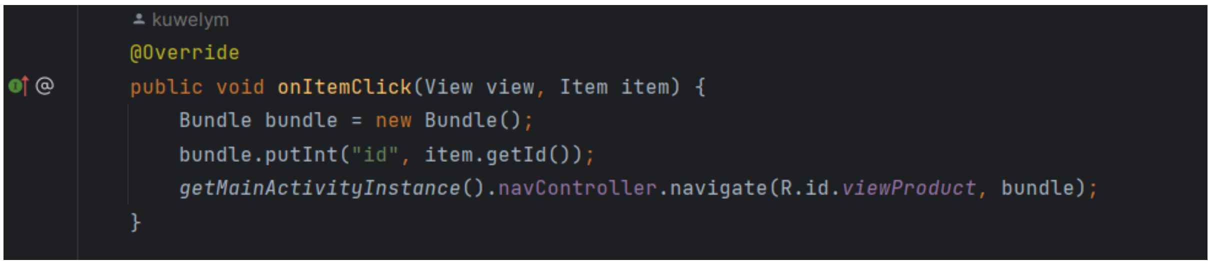
2.2. Enabling pluggable behavior:

A "OnItemClickListener" interface can define click method on an item that different items implementations ("CartItem" or "Product") can fulfill.



```
// Interface to handle item click events
6 implementations  Khuong Luan +1
public interface OnItemClickListener {
    6 implementations  kuwelym
    void onItemClick(View view, Item item);
}
```

Figure 9. OnItemClickListener interface



```
    kuwelym
    @Override
    public void onItemClick(View view, Item item) {
        Bundle bundle = new Bundle();
        bundle.putInt("id", item.getId());
        getMainActivityInstance().navController.navigate(R.id.viewProduct, bundle);
    }
```

Figure 10. Implementation of onItemClick method

DIFFERENCES

Feature	*Abstract Class*	*Interface*
Constructors	Have constructors	Cannot have constructors
Inheritance	Single inheritance (one abstract class)	Multiple inheritance (multiple interfaces)
Member variables	Can have final, non-final, static and non-static variables	Only have static and final variables
Keyword	Abstract class	Interface
Implementation	Can implement interface	Cannot implement abstract class

REFERENCES

https://viblo.asia/p/interface-vs-abstract-class-ke-tam-lang-nguoi-nua-can-07LKX9JeZV4
https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/
https://www.javatpoint.com/difference-between-abstract-class-and-interface