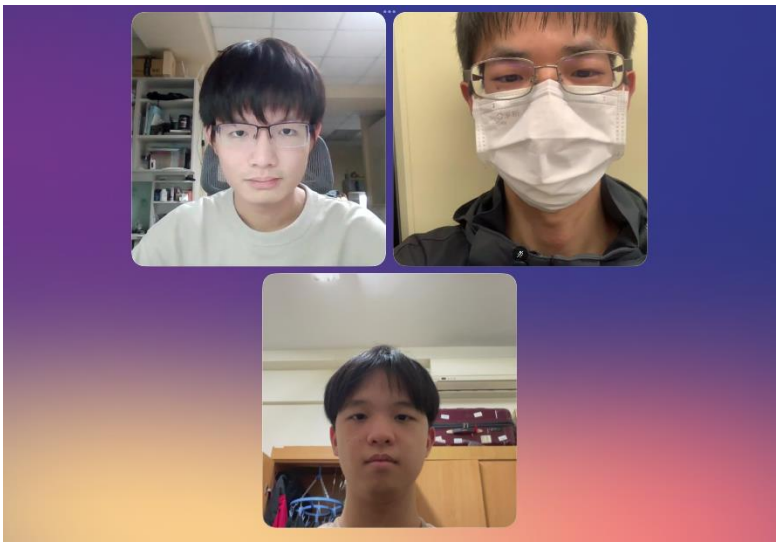
CH4

111360138 電子三甲 蔣安聖



讀書會

組員:
111360127 林煒哲
111360137 許鎧晏
111360138 蔣安聖
討論時間:2024/11/27 下午 3 點 30
地點:Discord

詢問 GPT 問題

1. 在 JAVA 中，可不可以覆寫不同回傳值的函式?

在 Java 中，**不能僅以不同的回傳值**覆寫 (override) 一個方法。要成功覆寫一個方法，方法的名稱、參數列表（包括順序與數量）、以及修飾符（如 `public` 或 `protected`）都必須與父類中的方法完全一致。這是因為覆寫的本質是替代父類中定義的方法，回傳值的不同會導致潛在的衝突。

Java 方法覆寫的規則

1. **方法名稱和參數列表**必須完全相同。
2. **回傳值**
   - 可以相同，或者可以是父類回傳型別的**協變型別**（Covariant Return Type）。例如，父類回傳型別是 `Object`，子類覆寫方法可以回傳 `String`（因為 `String` 是 `Object` 的子類）。
   - 不允許完全不同的回傳型別。
3. **存取修飾符**
   - 子類覆寫的方法存取權限不能比父類更小（例如，父類是 `public`，覆寫方法不能是 `protected` 或 `private`）。
4. **例外拋出**
   - 子類的方法不能拋出比父類更廣泛的檢查型例外（Checked Exception）。例如，父類方法拋出 `IOException`，覆寫方法不能拋出 `Exception`。

↓

如果只改回傳值會怎樣？

假設你在父類中有一個方法：

```java
class Parent {
    public int method() {
        return 0;
    }
}
```

如果你嘗試在子類中以不同回傳值覆寫：

```java
class Child extends Parent {
    @Override
    public String method() { // 編譯錯誤！
        return "Hello";
    }
}
```

這段程式碼會出現 **編譯錯誤**，因為 Java 無法將兩個方法區分開來（名稱與參數完全一致），但回傳型別不匹配。

如何解決？

1. **使用不同的方法名稱**
   - 如果你希望有不同的回傳值，可以定義一個新方法，而非覆寫原方法：

     ```java
     class Child extends Parent {
         public String methodWithString() {
             return "Hello";
         }
     }
     ```

2. 協變回傳型別

- 如果父類方法回傳的是一個類型，子類覆寫時可以回傳其子類型：

```java
class Parent {
    public Object method() {
        return new Object();
    }
}

class Child extends Parent {
    @Override
    public String method() { // 合法
        return "Hello";
    }
}
```

3. 方法多載 (Overloading)

- 你可以定義一個參數列表不同的方法，這不算覆寫，而是多載：

```java
class Parent {
    public int method() {
        return 0;
    }
}

class Child extends Parent {
    public String method(int x) { // 多載 (Overloading)
        return "Hello";
    }
}
```

小結

- **覆寫 (Override)** 不允許僅僅改變回傳值。
- 如果需要不同的回傳值，可以考慮 **協變回傳型別** 或 **多載 (Overloading)**。
- 合理使用這些特性可以避免設計衝突，並遵循 Java 的語言規範。

雖然這個問題看似很白癡，但我仍然好奇有沒有方式可以做到在子類別中，宣告與父類別相同名稱，但回傳值卻不同，雖然最後結果是不可以，但我學到一個新的東西，如果你在子類別中宣告的函式回傳值類型是父類別中宣告的函式

回傳值類別的子類別，那麼這樣其實是沒錯的，就如 GPT 回答中「怎麼解決」的第二個範例程式碼一樣。

2. abstract 會在哪些地方使用?

`abstract` 類別在設計程式架構時扮演重要角色，特別適合用於定義**基礎行為或特徵**，讓子類別可以繼承並實作詳細的功能。以下是一些典型的使用情境：

---

1. 建立基類 (Base Class)

`abstract` 類別通常用來作為其他類別的基礎，提供通用的屬性與方法，同時允許子類別實作特定行為。

範例：
假設有動物的應用，`Animal` 是抽象類別，定義了所有動物的共通行為（如吃飯、睡覺）和一些抽象行為（如發出聲音），由具體動物類別實作。

java 複製程式碼

```java
abstract class Animal {
    abstract void makeSound(); // 子類別必須實作
    void sleep() { // 具體方法，所有子類可用
        System.out.println("Sleeping...");
    }
}

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

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

2. 強制子類別實作特定功能

當你希望所有繼承的子類別都必須實作某些方法時，`abstract` 類別是一個很好的選擇。

範例：
在繪圖應用中，每種形狀（如圓形、矩形）都應該實作計算面積的方法。

java 複製程式碼

```java
abstract class Shape {
    abstract double calculateArea(); // 抽象方法
}

class Circle extends Shape {
    private double radius;

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

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

class Rectangle extends Shape {
    private double width, height;

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

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

3. 提供部分實作，讓子類別專注於特定細節

`abstract` 類別可以包含具體方法（有實作）和抽象方法（無實作），因此可以處理一些通用邏輯，而將細節交給子類別。

範例：
假設一個線上支付系統，`Payment` 是抽象類別，處理支付的通用流程，而具體支付方式（如信用卡支付、PayPal 支付）由子類別實作。

java 複製程式碼

```java
abstract class Payment {
    void processPayment() {
        validate(); // 通用邏輯
        makePayment(); // 抽象方法，需由子類別實作
        confirm(); // 通用邏輯
    }

    private void validate() {
        System.out.println("Validating payment...");
    }

    abstract void makePayment(); // 不同支付方式的具體實作

    private void confirm() {
        System.out.println("Payment confirmed!");
    }
}

class CreditCardPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing credit card payment...");
    }
}

class PayPalPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing PayPal payment...");
    }
}
```

4. 定義框架 (Framework) 或模板模式 (Template Pattern)

在設計模式中，`abstract` 類別經常用於模板模式，讓子類別填補框架中定義的抽象方法。

範例：
定義一個資料處理框架，包含資料的讀取、處理和寫入步驟，但具體細節由子類別決定。

java 複製程式碼

```java
abstract class DataProcessor {
    void process() {
        readData();
        processData();
        writeData();
    }

    abstract void readData(); // 抽象方法
    abstract void processData(); // 抽象方法

    void writeData() { // 通用邏輯
        System.out.println("Writing data to output...");
    }
}

class CsvDataProcessor extends DataProcessor {
    @Override
    void readData() {
        System.out.println("Reading CSV data...");
    }
```

```
    @Override
    void processData() {
        System.out.println("Processing CSV data...");
    }
}

class JsonDataProcessor extends DataProcessor {
    @Override
    void readData() {
        System.out.println("Reading JSON data...");
    }

    @Override
    void processData() {
        System.out.println("Processing JSON data...");
    }
}
```

---

5. 抽象類別與設計模式

在許多設計模式中（例如**模板模式**、**工廠模式**），`abstract` 類別被廣泛應用，用於定義框架結構或抽象工廠的基類。

範例：
抽象工廠模式中，`abstract` 類別可以定義工廠方法。

```java
abstract class AnimalFactory {
    abstract Animal createAnimal(); // 工廠方法
}

class DogFactory extends AnimalFactory {
    @Override
    Animal createAnimal() {
        return new Dog();
    }
}

class CatFactory extends AnimalFactory {
    @Override
    Animal createAnimal() {
        return new Cat();
    }
}
```

---

6. 無法實例化

`abstract` 類別無法直接實例化（不能 `new AbstractClass()`），這避免了使用不完整的類別。它的目的是作為模板或父類別，指導子類別完成具體實作。

7. 總結適用情境

使用 `abstract` 類別的典型場合：

- 當需要定義子類別的**共通行為與屬性**。
- 當需要部分邏輯由父類提供，部分邏輯交給子類別實作。
- 當需要使用模板方法模式來控制執行的步驟，但允許細節靈活。
- 當需要限制繼承結構（因為 Java 不支援多重繼承，`abstract` 類別更適合定義基礎層級的結構）。

如果需要多重繼承或只需定義**行為規範**（而非具體屬性或邏輯），則更適合用 `interface`。

在上述 GPT 給的範例中，我們可以了解到，abstract 主要用於多種不同物件之間有大量相同的內容，例如動物都有移動、進食、發聲等相同的行為，但細節卻不一樣，此時使用 abstract 定義這些不同的動物各種行為應傳入什麼類型，回傳值是什麼類型等等，只參與概念定義，不參與實作。

3. interface 與 abstract 有什麼差別?

以下是 `interface` 與 `abstract` 的主要差別：

特性	Interface	Abstract Class
目的	定義行為規範（方法簽名）	定義基礎類別（可以有具體邏輯與方法）
方法	只能有抽象方法（Java 8 開始可有 default/static 方法）	可以有抽象方法與具體方法
屬性	只能有 `public static final` 常數	可以有一般屬性
繼承	支援多重繼承	只支援單一繼承
實例化	不能被實例化	不能被實例化，但可以有構造方法供子類使用
使用場景	定義功能規範，讓多個類別實現	當有共同行為、需要部分邏輯由子類別實作時使用

1. 定義與使用目的

- `interface`（介面）：
  - 定義一組行為規範，描述「類別應該做什麼」，但不會規定「怎麼做」。
  - 主要用於實現多重繼承，讓一個類別可以實作多個行為規範。
  - 用於實現「完全抽象」，沒有狀態或邏輯。
  - 例子：

```java
public interface Flyable {
    void fly();  // 定義所有可以飛的物體都要實現 fly 方法
}

public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}

public class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Airplane is flying");
    }
}
```

- `abstract class`（抽象類別）：
  - 定義類別的基礎結構，表示「子類別是什麼」且「應該做什麼」。
  - 用於實現部分邏輯，提供子類別繼承與擴展。
  - 適合用來描述一組**有共用邏輯**與**特性**的類別。
  - 例子：

```java
public abstract class Animal {
    String name; // 屬性：所有動物都有名字

    public Animal(String name) {
        this.name = name;
    }

    // 抽象方法：要求子類別實作具體行為
    public abstract void makeSound();

    // 具體方法：提供通用邏輯
    public void eat() {
        System.out.println(name + " is eating");
    }
}
```

```java
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }
}
```

---

2. 方法與屬性

特性	Interface	Abstract Class
方法	只能有抽象方法（Java 8 後可以有 default/static 方法）	可以有抽象方法和具體方法
屬性	只能是 `public static final`（常數），不能有一般屬性	可以有一般屬性
構造方法	沒有構造方法，因為不能有狀態	可以有構造方法，供子類別初始化屬性

3. 繼承與實作

- `interface` 支援多重繼承：
  - 一個類別可以實作多個介面。
  - 例子：

```java
public interface Swimmable {
    void swim();
}

public interface Runnable {
    void run();
}

public class Frog implements Swimmable, Runnable {
    @Override
    public void swim() {
        System.out.println("Frog is swimming");
    }

    @Override
    public void run() {
        System.out.println("Frog is running");
    }
}
```

- `abstract class` 支援單一繼承：
  - 一個類別只能繼承一個抽象類別，但可以同時實作多個介面。
  - 例子：

```java
public abstract class Vehicle {
    public abstract void start();
}

public interface Flyable {
    void fly();
}

public class Helicopter extends Vehicle implements Flyable {
    @Override
    public void start() {
        System.out.println("Helicopter is starting");
    }

    @Override
    public void fly() {
        System.out.println("Helicopter is flying");
    }
}
```

4. 何時使用？

- 使用 `interface`：
  - 當需要定義一組**行為規範**，並讓多個不相關的類別實作時。
  - 適合應用在功能層面，例如：`Flyable`、`Swimmable`、`Runnable`。
- 使用 `abstract class`：
  - 當需要建立一個**基礎類別**，讓相關的類別繼承，並共用一些屬性與方法時。
  - 適合應用在層次結構中，例如：`Animal`、`Vehicle`。

簡單對比總結：

特性	Interface	Abstract Class
關鍵字	`interface`	`abstract`
抽象層級	完全抽象	部分抽象
多重繼承	支援	不支援
設計理念	「行為規範」	「父類別模板」
靈活性	適合多個類別實作功能	適合子類別共用邏輯

結論

如果需要**行為規範**並實現多重繼承，用 `interface`；如果需要**基礎邏輯**並提供單一繼承，用 `abstract class`。

Interface 相比於 abstract 來說更加的抽象，interface 基本上完全就是用於定義以及規範，並且使用範圍更加廣泛，可以想像成是高層的定義。

4. 可以同時 extends 又 implementation 嗎?

是的，在 Java 中，一個類別可以同時使用 `extends` 繼承一個父類別，並使用 `implements` 實作一個或多個介面。

語法範例：

java 複製程式碼

```java
public class MyClass extends ParentClass implements Interface1, Interface2 {
    // 必須實作所有介面中的抽象方法
    @Override
    public void methodFromInterface1() {
        System.out.println("Implementing method from Interface1");
    }

    @Override
    public void methodFromInterface2() {
        System.out.println("Implementing method from Interface2");
    }

    // 可以使用從 ParentClass 繼承的屬性和方法
    public void useParentMethod() {
        super.parentMethod();
    }
}
```

詳解：

1. **繼承父類別（`extends`）：**
   - Java 不支援多重繼承，但一個類別可以繼承一個父類別，獲得其屬性與方法。
   - 使用 `extends` 繼承父類別的功能與結構。
2. **實作介面（`implements`）：**
   - 一個類別可以實作多個介面，並需要提供介面中所有抽象方法的具體實作。
   - 使用 `implements` 讓類別能完成多種行為規範。

注意事項：

1. `extends` 必須放在前面：
   - Java 語法規定，`extends`（父類別）一定要放在 `implements`（介面）之前。
   - 正確：

```java
public class MyClass extends ParentClass implements Interface1, Interface2 { ... }
```

   - 錯誤：

```java
public class MyClass implements Interface1, Interface2 extends ParentClass { ... }
```

2. 多個介面：
   - 可以同時實作多個介面，用逗號分隔。
   - 例子：

```java
public class MyClass extends ParentClass implements Interface1, Interface2, Interf
```

3. 需要實作所有抽象方法：
   - 子類別必須實作所有介面中的抽象方法，否則該類別也必須標記為 `abstract`。

結論：

- Java 支援單一繼承（`extends`）和多介面實作（`implements`）。
- 可以同時使用 `extends` 和 `implements`，藉此繼承父類別的功能，並實作多個介面的行為規範，達到靈活的設計目標。

在 Java 中是可以同時使用 extends 以及 implementation 的，不過 extends 要寫在 implementation 前面，除此之外沒有實作到的抽象方法，要用 abstract 標記。

心得

在這次的實驗中，我學到什麼叫做繼承，前幾個禮拜在寫 android studio 的時候就不太了解 public、static、final、extends 等等，而在這章中講述了 extends 與 final 的用途，原來 extends 是繼承的意思，在宣告新物件的時候使用 extends 就可以把另一個物件繼承到自己身上，節省許多麻煩。而

Overriding 也是一個很重要的概念，在寫 android studio 也一定會看到 @Overriding，這代表覆寫的意思，目的在於將繼承過來的函式覆蓋掉，以達到相同函式名稱，而做不同的事情，我覺得這個方法很厲害，當你覆寫後，便可以客製化函式，當你使用列別陣列，並且賦予不同的子類別到陣列中，就可以使用 for 迴圈直接呼叫相同名稱的函式，卻可以執行客製化的內容，我覺得這方式超級厲害，也很有用。不過有些人不會想讓自己類別中的函式被覆蓋掉，因此誕生了 final 這個東西來保護它，讓之後的人不能覆寫這個函式。除此之外還有目前我還覺得用不太到的抽象跟介面，感覺也是個很厲害的東西，但我還想不到要怎麼用。

GitHub 程式連結與截圖

https://github.com/DolphinBlast/JavaHW08