

讀書會

- 組員：龍品勳、張耀文、時顯文
- 討論時間：2024/11/25 10:00 AM
- 地點：圖書館

照片：



心得報告

- 心得：Java 的物件導向使用起來比 C++ 還要簡單一點。
- ChatGPT：

1. 請問物件導向中，多型的意思



在物件導向程式設計 (OOP) 中，多型 (Polymorphism) 是一種允許物件以多種形式表現的特性。它的核心概念是：

- 相同的介面 (interface) 或基類別 (base class) 可以有多種具體的實現 (implementation) 。
- 使用者可以透過統一的方式來操作不同類型的物件，而不需要知道它們的具體類型。

多型的主要形式

1. 函式重載 (Function Overloading)：

- 同一個函式名稱可以根據參數數量或類型的不同來實現不同的行為。
- 通常用於靜態型別語言 (如 C++) 。

cpp

複製程式碼

```
void print(int i) { cout << "Integer: " << i; }  
void print(double d) { cout << "Double: " << d; }  
void print(string s) { cout << "String: " << s; }
```

2. 運算子重載 (Operator Overloading)：

- 自定義運算符 (如 `+`、`-`) 的行為，讓它適用於自定義的類型。

cpp

複製程式碼

```
class Complex {
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }
};
```

3. 函式覆寫 (Function Overriding) :

- 子類別覆蓋父類別的方法，並提供自己的實現。
- 利用虛擬函式 (virtual function) 和多型性，父類型的指標或參考可以呼叫子類型的行為。

cpp

複製程式碼

```
class Animal {
public:
    virtual void speak() { cout << "Animal speaks"; }
};
class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks"; }
};
```

4. 介面多型 (Interface Polymorphism) :

- 基於介面或抽象類別，實現不同類別的多態性。
- 常見於語言如 Java。

java

複製程式碼

```
interface Animal {
    void speak();
}

class Dog implements Animal {
    public void speak() { System.out.println("Dog barks"); }
}

class Cat implements Animal {
    public void speak() { System.out.println("Cat meows"); }
}
```

優點

- **提高程式的彈性和可擴展性**：能夠輕鬆新增新的類型和行為，而無需修改現有代碼。
- **簡化代碼**：使用統一的介面處理不同類型的物件，減少條件分支判斷。

實際應用

- GUI 框架中，按鈕、滑桿等元件繼承同一個基類，根據具體類別表現不同的行為。
- 資料庫操作，使用相同的介面處理不同的資料來源（如 MySQL、PostgreSQL）。

這種特性是 OOP 的核心概念之一，也是讓代碼更具彈性與模組化的關鍵。

2. 請問在 Java 中，函式複寫是否一定需要 @Override?



在 Java 中，**函式覆寫 (override)** 並不一定需要 `@Override` 註解，但建議使用它。以下是細節解釋：

不使用 `@Override` 的情況

- 如果不加 `@Override`，只要方法的簽名正確（名稱、參數類型和數量一致），覆寫仍然可以正常工作。
- 然而，若方法簽名不完全匹配，程式不會報錯，而會將此方法視為新的方法（多載/overload），這可能導致潛在的錯誤。

使用 `@Override` 的好處

1. **編譯時檢查**：編譯器會檢查方法簽名是否正確覆寫了父類方法。如果不匹配，會報錯提示。
2. **增強可讀性**：讓其他開發者清楚這是覆寫而非新的方法，增加程式的可維護性。

範例說明

正確覆寫：

java

複製程式碼

```
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    @Override
    void display() { // 正確覆寫
        System.out.println("Child display");
    }
}
```

忘記加 `@Override`，但仍正確覆寫：

java

複製程式碼

```
class Child extends Parent {
    void display() { // 正確覆寫，但少了 @Override
        System.out.println("Child display");
    }
}
```

錯誤的覆寫（不加 `@Override` 時容易忽略的錯誤）：

java

複製程式碼

```
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    // 錯誤的簽名，實際是定義新方法
    void display(String msg) {
        System.out.println(msg);
    }
}
```

上例中，由於 `display(String msg)` 並非覆寫，而是多載，父類的方法 `display()` 仍然存在，且程式不會報錯。但如果加上 `@Override`，編譯器會立刻提示簽名不匹配的錯誤。

總結

- 不一定需要 `@Override`，但強烈建議使用。
- 它可以幫助避免覆寫錯誤，並提升程式碼的可讀性和穩定性。

3. 請問在 Java 中， interface 可以加上 abstract 嗎



在 Java 中，`interface` 本身就是抽象的，因此不需要也不應該在 `interface` 宣告中顯式地加上 `abstract` 關鍵字。這是因為 Java 語言設計中，`interface` 的抽象性是內建的。

例子：

java

複製程式碼

```
public interface MyInterface {  
    void method1();  
    void method2();  
}
```

這裡的 `method1` 和 `method2` 都隱含地是 `abstract` 的，即使你不寫 `abstract`，Java 編譯器也會將它們視為抽象方法。

如果你試圖加上 `abstract`，也不會報錯，因為這只是多餘的修飾。以下程式碼是有效的，但不建議：

java

複製程式碼

```
public abstract interface MyInterface {  
    void method1();  
    void method2();  
}
```

注意：

- 如果你正在使用 Java 8 或更高版本，`interface` 中可以包含以下幾種類型的成員：
 - 抽象方法 (預設行為)
 - 預設方法 (使用 `default` 關鍵字)
 - 靜態方法
 - 常量 (隱含為 `public static final`)

範例：

```
java 複製程式碼

public interface MyInterface {
    // 抽象方法
    void abstractMethod();

    // 預設方法
    default void defaultMethod() {
        System.out.println("This is a default method.");
    }

    // 靜態方法
    static void staticMethod() {
        System.out.println("This is a static method.");
    }
}
```

總結來說，在 `interface` 上加上 `abstract` 是冗餘的，建議省略。

- GitHub 程式連結：<https://github.com/WoemTheCat/CH4>
- GitHub 的截圖：

