

介面與多型與列舉

學習目標

- 使用介面定義行為
- 瞭解介面的多型操作
- 利用介面列舉常數
- 利用ENUM列舉常數

介面

介面變數

- 定義了一個介面，相當於定義了一種新的資料類型，就可以用這種新的資料類型定義變數。用介面定義的變數稱為介面變數。

介面變數的定義形式：

介面名 介面變數表列；

介面

1. 介面的定義

介面的定義形式：

```
interface 介面名  
{  
    //符號變數定義  
    //方法宣告  
}
```

介面 (Interface)

定義介面

- 介面(**interface**)與抽象類別(**abstract class**)的概念非常類似，而介面比抽象類別更抽象。
- 介面僅具有方法與變數的宣告，不包含任何方法的定義區塊。
- 介面必須藉由其他類別使用**implements**關鍵字來實作，所謂實作就是為介面中宣告的方法加上定義區塊。
- 可以將**implements**「實作」當作「繼承」來看。
- 類別可以實作(**implements**)許多介面，而一個類別只能繼承一個類別而已，介面也可以繼承(**extends**)一個以上的介面。

介面（Interface）

介面（**interface**）與抽象類別（**abstract class**）

- 介面裡不能實作任何 **methods**；**abstract class** 可以實作方法。
- 一個類別可以同時實作多個介面；但一個類別只能有一個 **superclass**（**Java** 不支援多重繼承）。
- 介面不屬於類別階層架構。不相干的類別也可以實作相同介面。
- 因為多重繼承會有問題，但實際上又有多重繼承的需要，所以 **interface** 作為折衷的替代方案。

介面（Interface）

- 注意事項
 - 如果某介面繼承另一個介面，則 **subclass** 能從 **superclass** 繼承到的只有常數與函式宣告而已。
 - 介面開始使用後若有更改，已經使用此介面的程式會因無法實作新功能，而不能執行。
 - 如果真要更改需宣告新介面，就可以決定要實作一個介面或兩個介面都實作。

介面的繼承

1. 介面的定義

介面還可以有多個父介面，介面的完整定義如下形式：

```
interface 介面名 extends 父介面表列
{
    //符號常量定義
    //方法宣告
}
```

介面的繼承

- 定義
 - 一段只有常數與函式宣告，但沒有函式實作的程式碼。
- 宣告
 - `[public] interface` 介面名稱 `[extends` 父介面名稱]
- 作用
 - 讓某個功能，不論由誰實作，都能夠有相同的函式名稱，傳入值，傳出值，與存取範圍。

介面的實現

介面的使用

介面必須通過類別才能使用。如果一個類別定義了介面中的所有方法，稱這個類別實現了介面。實現介面的形式：

```
class 類別名 implements 介面名稱[, 其他介面名稱, 其他介面名稱...,  
...] {  
    //類體  
}
```

可以用介面變數表示實現它的類別（可以看作子類別）的物件，介面變數可以稱作是物件的上轉型物件。

介面的實現

implements 關鍵字

使用 `implements` 關鍵字可以同時繼承多個介面（介面跟介面之間採用逗號分隔）。

```
public interface A {  
    public void eat();  
    public void sleep();  
}
```

```
public interface B {  
    public void show();  
}
```

```
public class C implements A,B { }
```

介面的實現

```
package ch7_1;
public interface Animal {
    public void eat();
    public void run();
}

package ch7_1;
public class Dog implements Animal {
    public void eat() {
        System.out.println("Dog eats");
    }

    public void run() {
        System.out.println("Dog runs");
    }

    public int noOfLegs() {
        return 0;
    }
}
```

```
package ch7_1;
public class ch7_2 {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
        d.run();
    }
}
```

結果:

Dog eats
Dog runs

介面的繼承與實現

```
package ch7_2;
public interface AreaInterface {
    // 屬性開始
    public final double PI = 3.1415926;
    // 屬性結尾
    // 方法開始
    public void area();
    // 方法結尾
}
```

```
package ch7_2;
public abstract class shape {
    // 屬性開始
    public double x;
    public double y;
    // 屬性結尾
}
```

介面的繼承與實現

求圓面積,圓周長?

```
package ch7_2;  
public interface ShapeInterface extends AreaInterface {  
    // 方法開始  
    public void perimeter();  
    // 方法結尾  
}
```

介面的繼承與實現

```
package ch7_2;
public class Circle extends shape implements ShapeInterface {
    // 屬性開始
    private double r;
    // 屬性結尾
    public Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    // 方法開始
    public void area() {
        System.out.println("圓面積: " + PI*r*r);
    }
    public void perimeter() {
        System.out.println("圓周長: " + 2.0*PI*r);
    }
    // 方法結尾
}
```

介面的繼承與實現

```
package ch7_2;  
public class ch7_2 {  
    public static void main(String[] args) {  
        Circle c = new Circle(5.0, 10.0, 5.0);  
        c.area();  
    }  
}
```

結果:

圓面積: 78.539815

多介面的繼承與實現3-1

求圓面積,圓周長,X座標,Y座標,圓半徑?

```
package ch7_3;
public abstract class Shape {
    // 屬性開始
    public double x;
    public double y;
    // 屬性結尾
}
```

```
package ch7_3;
public interface ShapeInfo {
    // 方法開始
    public void show();
    // 方法結尾
}
```

```
package ch7_3;
public interface ShapeInterface extends AreaInterface, ShapeInfo {
    // 方法開始
    public void perimeter();
    // 方法結尾
}
```

```
package ch7_3;
public interface AreaInterface {
    // 屬性開始
    public final double PI = 3.1415926;
    // 屬性結尾
    // 方法開始
    public void area();
    // 方法結尾
}
```


多介面的繼承與實現3-2

```
package ch7_3;
public class Circle extends Shape implements ShapeInterface {
    // 屬性開始
    private double r;
    // 屬性結尾
    public Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    // 方法開始
    public void area() {
        System.out.println("圓面積: " + PI*r*r);
    }
    public void perimeter() {
        System.out.println("圓周長: " + 2.0*PI*r);
    }
}

    public void show() {
        System.out.println("X座標: " + x);
        System.out.println("Y座標: " + y);
        System.out.println("圓半徑: " + r);
    }
    // 方法結尾
}
```

多介面的繼承與實現3-3

```
package ch7_3;
public class ch7_3 {
    public static void main(String[] args) {
        // 宣告Circle類別型態的變數且建立物件
        Circle c = new Circle(5.0, 10.0, 5.0);
        c.area();          // 呼叫介面方法area()
        c.perimeter();     // 呼叫介面方法perimeter()
        c.show();          // 呼叫介面方法show()
    }
    // 方法結尾
}
```

結果:

圓面積: 78.539815

圓周長: 31.415926

X座標: 5.0

Y座標: 10.0

圓半徑: 5.0

介面的多型 (Polymorphism)

- 重載(Overloading) 是一個類別中定義了多個方法名相同,而他們的參數的數量不同或數量相同而類型和次序不同。
- 重新定義(Overriding)是在子類別存在方法與父類別的方法的名字相同,而且參數的個數與類型一樣,返回值也一樣的方法。
- 方法重載是一個類的多態性表現,而方法重新定義是子類與父類的一種多態性表現。

介面重新定義Override 3-1

```
package ch7_4;  
public interface Swimmer {  
    public abstract void swim();  
}
```

```
package ch7_4;  
public class Submarine implements Swimmer {  
    private String name;  
  
    public Submarine(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void swim() {  
        System.out.printf("潛水艇 %s 潛行%n", name);  
    }  
}
```

介面重新定義Override 3-2

```
package ch7_4;
public class Fish implements Swimmer {
    protected String name;

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

    public String getName() {
        return name;
    }

    public void swim() {
    }
}
```

```
package ch7_4;
public class Piranha extends Fish {
    public Piranha(String name) {
        super(name);
    }

    public void swim() {
        System.out.printf("食人魚 %s 游泳%n", name);
    }
}
```

介面重新定義Override 3-3

```
package ch7_4;
public class Shark extends Fish {
    public Shark(String name) {
        super(name);
    }

    public void swim() {
        System.out.printf("鯊魚 %s 游泳%n", name);
    }
}
```

```
package ch7_4;
public class Ocean {
    public static void main(String[] args) {
        Piranha a = new Piranha("尼莫");
        Shark b = new Shark("蘭尼");
        Submarine c = new Submarine("黃色一號");
        a.swim();
        b.swim();
        c.swim();
    }
}
```

結果:

食人魚 尼莫 游泳
鯊魚 蘭尼 游泳
潛水艇 黃色一號 潛行

介面重載 Overloading

```
package ch7_5;
public interface Computable {
    final int MAX=100;
    int f(int x);
    int g(int x,int y);
}
```

```
package ch7_5;
public class A implements Computable{
    public int f(int x){
        return x*x;
    }
    public int g(int x,int y){
        return x+y;
    }
}
```

```
package ch7_5;
public class B extends A{
    public int f(int x,int y){
        return x*y;
    }
    public int g(int x){
        return x/2;
    }
}
```

```
package ch7_5;
public class ch7_4 {
    public static void main(String[] args) {
        A a=new A();
        B b=new B();
        System.out.println(a.MAX);
        System.out.println(""+a.f(10)+" "+a.g(12,6));
        System.out.println(b.MAX);
        System.out.println(""+b.f(10,2)+" "+b.g(28));
    }
}
```

介面重載 Overloading

結果:

100

100 18

100

20 14

匿名類別

- 物件導向程式設計的特性之一，就是軟體元件（類別）的重複使用性。
- 類別可在程式中建立多個物件、必要時則可由既有類別來衍生新類別。
- 但類別在程式中只會用到一次，費時去定義一個類別，再來建立物件，就顯得沒有必要，此時就能利用匿名類別的技巧來建立此類別物件，同時也有使程式碼較簡化的功效。

匿名類別

- 類別去掉類別宣告後的類體,稱作匿名類別。
- 匿名類別就是一個子類別,由於無名可用,所以不可能用匿名類別宣告物件,但卻可以直接用匿名類別創建一個物件。
- 匿名類可以繼承類別的方法也可以重寫類別的方法。
- 匿名類別可以訪問外部類別中的成員變數和方法,匿名類別不可以聲明static成員變數和static方法。匿名類別的主要用途就是方法的參數傳值。
- 使用匿名類別時,必然是在某個類別中直接用匿名類別創建物件,因此匿名類別一定是內部類別。

內部類別

內部類別

如果在**A**類別裡面再定義一個**B**類別，這種結構稱為巢狀類別(**nested classes**)，此時的類別**A**稱為外部類別(**outer class**)，而類別**B**則稱為內部類別(**inner class**)。

內部類別也可以分為**public**和**private**，當宣告成**public**和**private**時，其存取的限制與資料成員或函數成員相同。

```
修飾子 class 外部類別的名稱{  
    //外部類別的成員  
    修飾子 class 內部類別名稱{  
        //內部類別的成員  
    }  
}
```

匿名內部類別

匿名內部類別(**anonymous inner class**)可以方便利用內部類別建立不具名稱的物件，並利用他存取類別裡的成員。

```
(  
    new 類別名稱(引數){  
        傳回值型態 函數名稱(引數1,引數2,...,引數n){  
            函數敘述;  
        }  
    }  
).函數名稱(引數1,引數2,...,引數n);
```

- 建立匿名內部類別的用意，主要是直接用**new**建立類別，並捕捉內部類別裡沒有定義到的函數，有效的簡化程式碼。
- 當程式中有使用匿名內部類別時，**JAVA**會用"\$"再加上數字編號，用以區分該內部類別是屬於哪一個外部類別。

匿名內部類別

```
package ch7_6;
public class ch7_6 {
    public static void main(String[] args) {
        final int a= 10;

        (new Object() { // 匿名類別
            int b =10000; // 匿名類別的成員
            public void show() { // 匿名類別的方法
                System.out.println ("匿名類別:");
                System.out.println ("this ->b= " +b);
                System.out.println ("main()->a= " +a);
            }
        }).show(); // 產生匿名類別物件後即呼叫其 show() 方法
    }
}
```

結果:

匿名類別：

this ->b= 10000

main()->a= 10

匿名內部類別

```
package ch7_7;
abstract class Student {
    abstract void speak();
}

package ch7_7;
public class Teacher {
    void look(Student stu){
        System.out.println("這是Teacher類別中的方法");
        stu.speak(); //執行匿名類體中重新定義的speak()方法
    }
}

package ch7_7;
public class ch7_7 {
    public static void main(String[] args) {
        Teacher z = new Teacher();
        z.look(new Student() { // 匿名類的類體，即Student的子類別的類體
            void speak() {
                System.out.println("這是匿名類別中的方法");
            }
        }); // 匿名類類體結束
    }
}
```

結果:

這是Teacher類別中的方法
這是匿名類中的方法

介面匿名類別

介面匿名類別創建物件語法：

```
NEW 介面名 () {  
    實現介面的匿名類別的類體  
}
```

- 如果某個方法的參數是介面類別型，那麼我們可以使用介面名和類體組合創建一個匿名物件傳遞給方法的參數。
- 類體必須要實現介面中的全部方法。

介面匿名類別

```
package ch7_8;
    public interface Show {
        public void show();
    }

package ch7_8;
    public class A {
        void f(Show s){
            s.show();    //s呼叫匿名類別中實現的介面方法
        }
    }

package ch7_8;
    public class ch7_8 {
        public static void main(String[] args) {
            A a = new A();
            a.f(new Show() {
                public void show() {
                    System.out.println("這是實現介面的匿名類別");
                }
            });
        }
    }
```

結果:

這是實現介面的匿名類別

列舉

- 如果一個物件的所有值可以一一列舉出來，則稱為列舉。
- 在Java中，列舉類型也是一個類別，用列舉類的變數表示列舉值。
- Java定義了一個基礎列舉類Enum，任何列舉類別都是這個類別的子類。列舉可以提高程式的清晰性和可讀性。

簡單的列舉類別

- 列舉類別定義的語法形式：

```
Enum 列舉名{  
    列舉常量表列 }
```

- 列舉變數的定義形式：

```
列舉名 列舉變數表列;
```

使用enum列舉

```
package ch7_9;  
public enum Color {  
    RED, GREEN, BLUE;  
}
```

```
package ch7_9;  
public class ch7_9 {  
    public static void main(String[] args) {  
        Color c1 = Color.RED;  
        System.out.println(c1);  
    }  
}
```

結果:

RED

使用enum列舉 2-1

```
package ch7_10;
public interface Action {
    public static final int RIGHT = 0;
    public static final int LEFT = 1;
}
```

```
package ch7_10;
public enum Action2 {
    RIGHT, LEFT
}
```

```
package ch7_10;
public interface Command {
    void execute();
}
```

```
package ch7_10;
import static java.lang.System.out;
public enum Action3 implements Command {
    RIGHT {
        public void execute() {
            out.println("播放右轉動畫");
        }
    },
    LEFT {
        public void execute() {
            out.println("播放左轉動畫");
        }
    },
}
```

使用enum列舉 2-2

```
package ch7_11;
import static java.lang.System.out;
public class ch7_11 {
    public static void main(String[] args) {
        play(Action.RIGHT);
    }

    public static void play(int action) {
        switch (action) {
            case Action.RIGHT:
                out.println("播放向右動畫");
                break;
            case Action.LEFT:
                out.println("播放向左動畫");
                break;
            default:
                out.println("不支援此動作");
        }
    }
}
```

結果:

播放向右動畫

複雜的列舉類別

- 列舉類別中除了列舉常量外，還可以定義區域、方法、建構式方法，main方法也可以定義在列舉類別中。
- 如果在列舉類別中定義了區域和方法，則列舉常量必須放在列舉體內的第一行，而且最後一個列舉常量後要加一個“;”。

enum列舉常用方法

位於 java.lang.Enum 類別中：

- values() 返回列舉類別中所有的值。
- ordinal() 找到每個列舉變數的索引。
- compareTo() 依據ordinal成員進行排序。
- valueOf() 返回指定字串值的列舉變數。
- name() 取得列舉成員名稱字串。
- equals() 作為物件相等性比較。

enum列舉常用方法

```
package ch7_12;
public enum Size {
    SMALL, MEDIUM, LARGE
}
```

```
package ch7_12;
public class ch7_12 {
    public static void main(String[] args) {
        for (Size size : Size.values()) {
            System.out.println(size + "的序數是：" + size.ordinal());
            System.out.print(size.compareTo(Size.SMALL) + " ");
            System.out.print(size.equals(Size.SMALL) + " ");
            System.out.println(size == Size.SMALL);
            System.out.println(size.name());
            System.out.println("*****");
        }
    }
}
```

結果:

SMALL的序數是：0

0 true true

SMALL

MEDIUM的序數是：1

1 false false

MEDIUM

LARGE的序數是：2

2 false false

LARGE
