介面

TYIC 桃高資訊社

介面

介面(interface)大部分時候可以視為抽象類別

但介面是專門用來定義方法，且介面通常定義動態抽象方法

交由實作(implement)該介面的類別去定義實際要執行什麼

介面可以定義欄位，且皆為公開靜態不可變

介面的靜態方法預設為公開靜態方法、動態方法預設為公開動態抽象方法

一個類別可以實作多個介面，一個介面可以繼承多個介面

```
修飾子 interface 介面名稱 {
    ...
}
```
java

```
修飾子 class 類別名稱 implements 實作介面 {
    ...
}
```
java

```
修飾子 interface 介面名稱 extends 父介面1, 父介面2, ..., 父介面n {
    ...
}
```
java

```
修飾子 class 類別名稱 implements 實作介面1, 實作介面2, ..., 實作介面n {
    ...
}
```
java

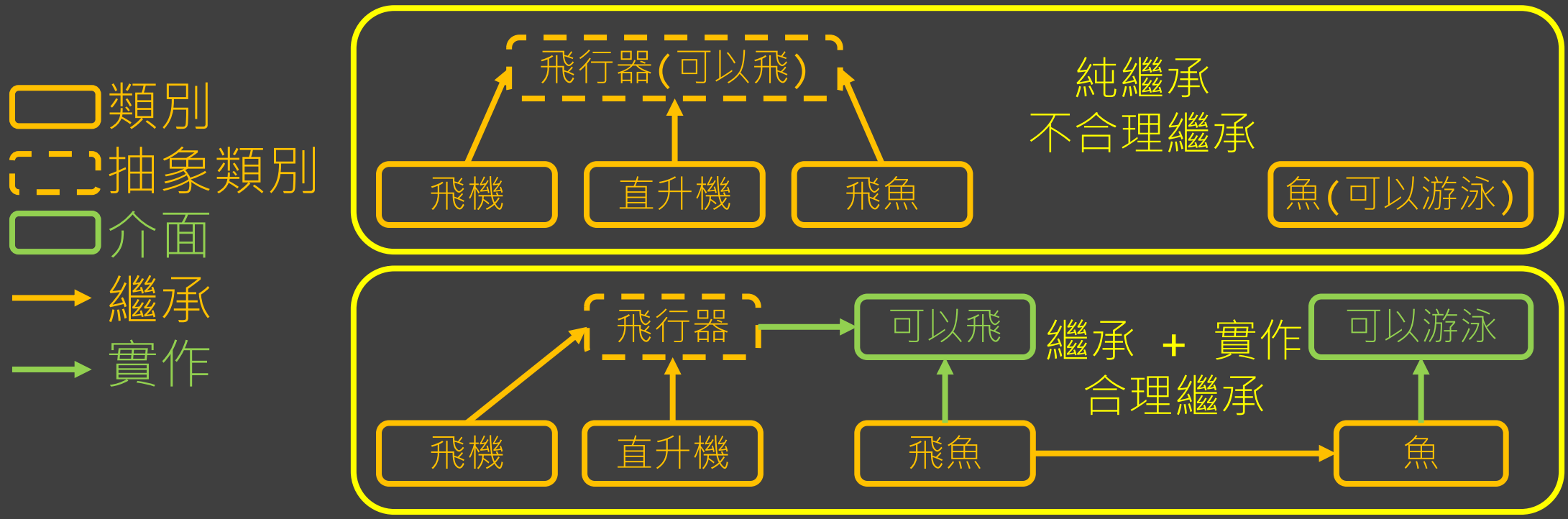
實作與繼承

實作是定義介面方法實際要做的事

被實作的介面與實作該介面的類別之間為「有沒有功能」關係

而被繼承的類別與繼承的類別之間則為「是不是一種」關係

在設計程式時應該優先考慮實作而非繼承，避免濫用繼承



實作與繼承

```java
public class Main {
    public static void main(String[] args) {
        FlyingFish flyingFish = new FlyingFish();
        flyingFish.fly();
        flyingFish.swim();
        Aircraft aircraft1 = new Airplane();
        aircraft1.fly();
        Aircraft aircraft2 = new Helicopter();
        aircraft2.fly();
    }
}

interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

class Fish implements CanSwim {
    @Override
    public void swim() {
        System.out.println("魚兒魚兒水中遊");
    }
}

class Airplane extends Aircraft {
    @Override
    public void fly() {
        System.out.println("飛機用引擎飛");
    }
}

class Helicopter extends Aircraft {
    @Override
    public void fly() {
        System.out.println("直升機用螺旋槳飛");
    }
}

class FlyingFish extends Fish implements CanFly {
    @Override
    public void fly() {
        System.out.println("飛魚用魚翅飛");
    }
}
```

```
飛魚用魚翅飛
魚兒魚兒水中遊
飛機用引擎飛
直升機用螺旋槳飛
```

output

java

介面多型

介面也是一種型別，所以也可用在變數宣告上，實作該介面都可以填入

```java
public class Main {
    public static void main(String[] args) {
        CanFly[] canFlies =
                {new Helicopter(), new Airplane(), new FlyingFish()};
        for (CanFly canFly : canFlies) {
            canFly.fly();
        }
    }
}
```

```java
interface CanFly {
    void fly();
}
```

```java
abstract class Aircraft implements CanFly {
}
```

```java
class Airplane extends Aircraft {
    @Override
    public void fly() {
        System.out.println("飛機用引擎飛");
    }
}
```

```java
class Helicopter extends Aircraft {
    @Override
    public void fly() {
        System.out.println("直升機用螺旋槳飛");
    }
}
```

```java
class FlyingFish implements CanFly {
    @Override
    public void fly() {
        System.out.println("飛魚用魚翅飛");
    }
}
```

```
直升機用螺旋槳飛
飛機用引擎飛
飛魚用魚翅飛
```

output

java

預設方法

介面也可以定義預設方法

這樣實作的類別就不一定要覆寫該方法

```
interface 介面名稱 {
    default 返回值型別 方法名稱(...) {
        ...
    }
}
```
java

```
public class Main {
    public static void main(String[] args) {
        CanFly[] canFlies =
                {new Helicopter(), new Airplane(), new FlyingFish()};
        for (CanFly canFly : canFlies) {
            canFly.fly();
        }
    }
}

interface CanFly {
    default void fly() {
        System.out.println("飛");
    }
}

class Helicopter extends Aircraft {
}

abstract class Aircraft implements CanFly {
}

class Airplane extends Aircraft {
    @Override
    public void fly() {
        System.out.println("飛機用引擎飛");
    }
}

class FlyingFish implements CanFly {
    @Override
    public void fly() {
        System.out.println("飛魚用魚翅飛");
    }
}
```
java

```
飛
飛機用引擎飛
飛魚用魚翅飛
```
output

匿名內部類別

匿名內部類別也可以用在介面上

實際上這會創建直接繼承 Object

並實作該介面的匿名類別

這常用在將方法作為引數進行傳遞

```
new 已存在介面() {
    匿名內部類別定義...
}
```
java

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        ArrayHelper.map(arr,
                new ArrayHelper.MapFunction() {
                    @Override
                    public int map(int value) {
                        return value + 6;
                    }
                });
        System.out.println(Arrays.toString(arr));
    }
}
```

```java
abstract class ArrayHelper {
    public interface MapFunction {
        int map(int value);
    }

    public static void map(int[] array,
                           MapFunction mapFunction) {
        for (int i = 0; i < array.length; i++) {
            array[i] = mapFunction.map(array[i]);
        }
    }
}
```

```
[7, 8, 9, 10, 11, 12, 13]
```
output

java

函式介面與 Lambda

顯然的，為了覆寫一個方法需要實例化、權限修飾等程式碼太冗長

所以在函式介面，即只要覆寫一個方法的介面，可以使用 Lambda

```
(參數1, 參數2, ..., 參數n) -> 返回值
```
java

```
(參數1, 參數2, ..., 參數n) -> {
    陳述式...
}
```
java

箭號後方可以接返回值或是區塊

若是使用區塊，則須使用 return 返回結果

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7};
        ArrayHelper.map(arr, (value) -> value + 6);
        System.out.println(Arrays.toString(arr));
    }
}
```

```
[7, 8, 9, 10, 11, 12, 13]
```
output

```
abstract class ArrayHelper {
    @FunctionalInterface
    public interface MapFunction {
        int map(int value);
    }

    public static void map(int[] array,
                           MapFunction mapFunction) {
        for (int i = 0; i < array.length; i++) {
            array[i] = mapFunction.map(array[i]);
        }
    }
}
```
java

方法參考

若是要將現有的方法傳入，則不應該使用 Lambda，而是方法參考

```java
類別名稱::方法名稱
```

若是靜態方法，就會將所有參數填入靜態方法

若是動態方法，則會將第一個傳入的參數作為物件呼叫該動態方法

若是想要呼叫建構子，則必須使用以下格式：

```java
類別名稱::new
```

方法參考

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] intArr = {1, 2, 3, 4, 5, 6, 7};
        Num[] numArr = new Num[7];
        ArrayHelper.map(intArr, Num::new, numArr);
        System.out.println(Arrays.toString(numArr));
        ArrayHelper.map(numArr, Num::square, numArr);
        System.out.println(Arrays.toString(numArr));
        ArrayHelper.map(numArr, Num::negative, numArr);
        System.out.println(Arrays.toString(numArr));
    }
}

class Num {
    int number;

    public Num(int number) {
        this.number = number;
    }

    public Num square() {
        return new Num(number * number);
    }

    public static Num negative(Num num) {
        return new Num(-num.number);
    }

    @Override
    public String toString() {
        return String.valueOf(number);
    }
}
```

```java
abstract class ArrayHelper {
    @FunctionalInterface
    public interface IntMapper {
        Object map(int value);
    }

    @FunctionalInterface
    public interface StudentMapper {
        Object map(Num value);
    }

    public static void map(int[] srcArray,
                           IntMapper mapFunction, Object[] dstArray) {
        for (int i = 0; i < Math.max(srcArray.length, dstArray.length); i++) {
            dstArray[i] = mapFunction.map(srcArray[i]);
        }
    }

    public static void map(Num[] srcArray,
                           StudentMapper mapFunction, Object[] dstArray) {
        for (int i = 0; i < Math.max(srcArray.length, dstArray.length); i++) {
            dstArray[i] = mapFunction.map(srcArray[i]);
        }
    }
}
```

```
[1, 2, 3, 4, 5, 6, 7]
[1, 4, 9, 16, 25, 36, 49]
[-1, -4, -9, -16, -25, -36, -49]
```
output

java