

Java^{SE 7} 技術手冊

- 涵蓋 OCP/JP (原 SCJP) 考試範圍
- Coin 專案、JSR166y、JDBC 4.1、NIO.2 等 Java SE 7 新功能介紹
- JDK 基礎與 IDE 操作交相對照
- 提供 Lab 檔案與操作錄影教學

CHAPTER 7

• 介面與多型



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

介面定義行為

- 老闆今天想開發一個海洋樂園遊戲，當中所有東西都會游泳 ...
- 剛學過繼承？

```
public abstract class Fish {  
    protected String name;  
    public Fish(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public abstract void swim();  
}
```

介面定義行為

```
public class Anemonefish extends Fish {  
    public Anemonefish(String name) {  
        super(name);  
    }  
    @Override  
    public void swim() {  
        System.out.printf("小丑魚 %s 游泳%n", name);  
    }  
}
```

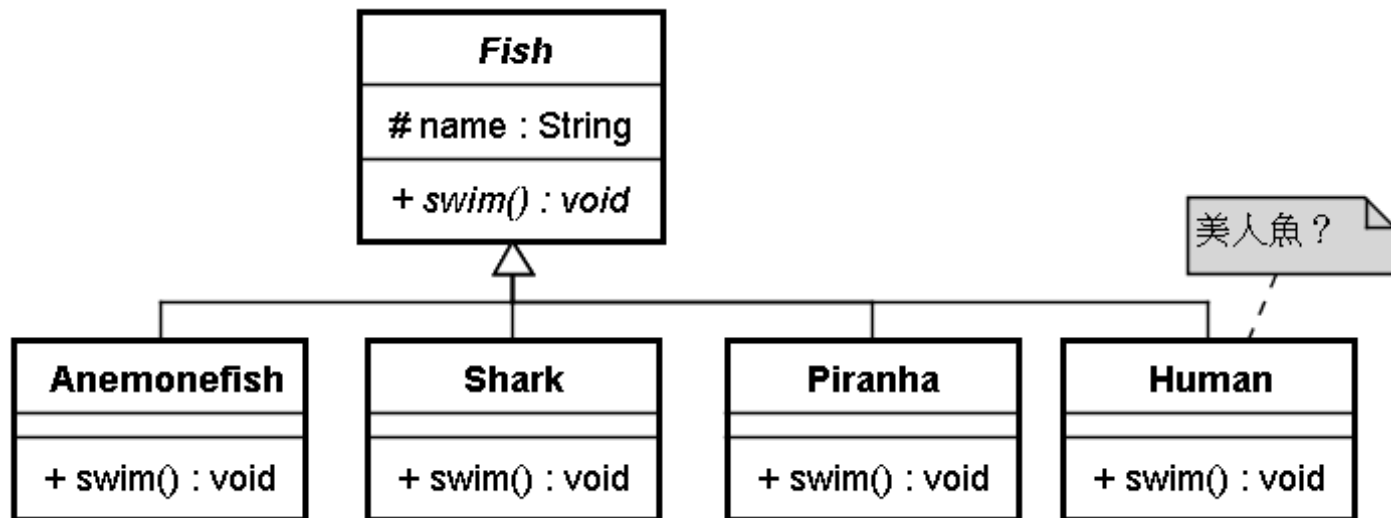
介面定義行為

```
public class Shark extends Fish {  
    public Shark(String name) {  
        super(name);  
    }  
    @Override  
    public void swim() {  
        System.out.printf("鯊魚 %s 游泳%n", name);  
    }  
}
```

```
public class Piranha extends Fish {  
    public Piranha(String name) {  
        super(name);  
    }  
    @Override  
    public void swim() {  
        System.out.printf("食人魚 %s 游泳%n", name);  
    }  
}
```

介面定義行為

- 老闆說話了，為什麼都是魚？人也會游泳啊！怎麼沒寫？
- 於是你就再定義Human類別繼承Fish...等一下！...



介面定義行為

- Java中只能繼承一個父類別，所以更強化了「是一種」關係的限制性
- 如果今天老闆突發奇想，想把海洋樂園變為海空樂園，有的東西會游泳，有的東西會飛，有的東西會游也會飛 ...

介面定義行為

- 老闆今天想開發一個海洋樂園遊戲，當中所有東西都會游泳。「所有東西」都會「游泳」，而不是「某種東西」都會「游泳」
- 「所有東西」都會「游泳」，代表了「游泳」這個「行為」可以被所有東西擁有，而不是「某種」東西專屬

介面定義行為

- 對於「定義行為」，在Java中可以使用 **interface** 關鍵字定義

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

介面定義行為

- 物件若想擁有Swimmer定義的行為，就必須實作Swimmer介面

```
public abstract class Fish implements Swimmer {  
    protected String name;  
    public Fish(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    @Override  
    public abstract void swim();  
}
```

介面定義行為

- 類別要實作介面，必須使用**`implements`**關鍵字，實作某介面時，對介面中定義的方法有兩種處理方式
 - 實作介面中定義的方法
 - 再度將該方法標示為`abstract`

介面定義行為

- Anemonefish、Shark與Piranha繼承Fish
- 如果Human要能游泳呢？

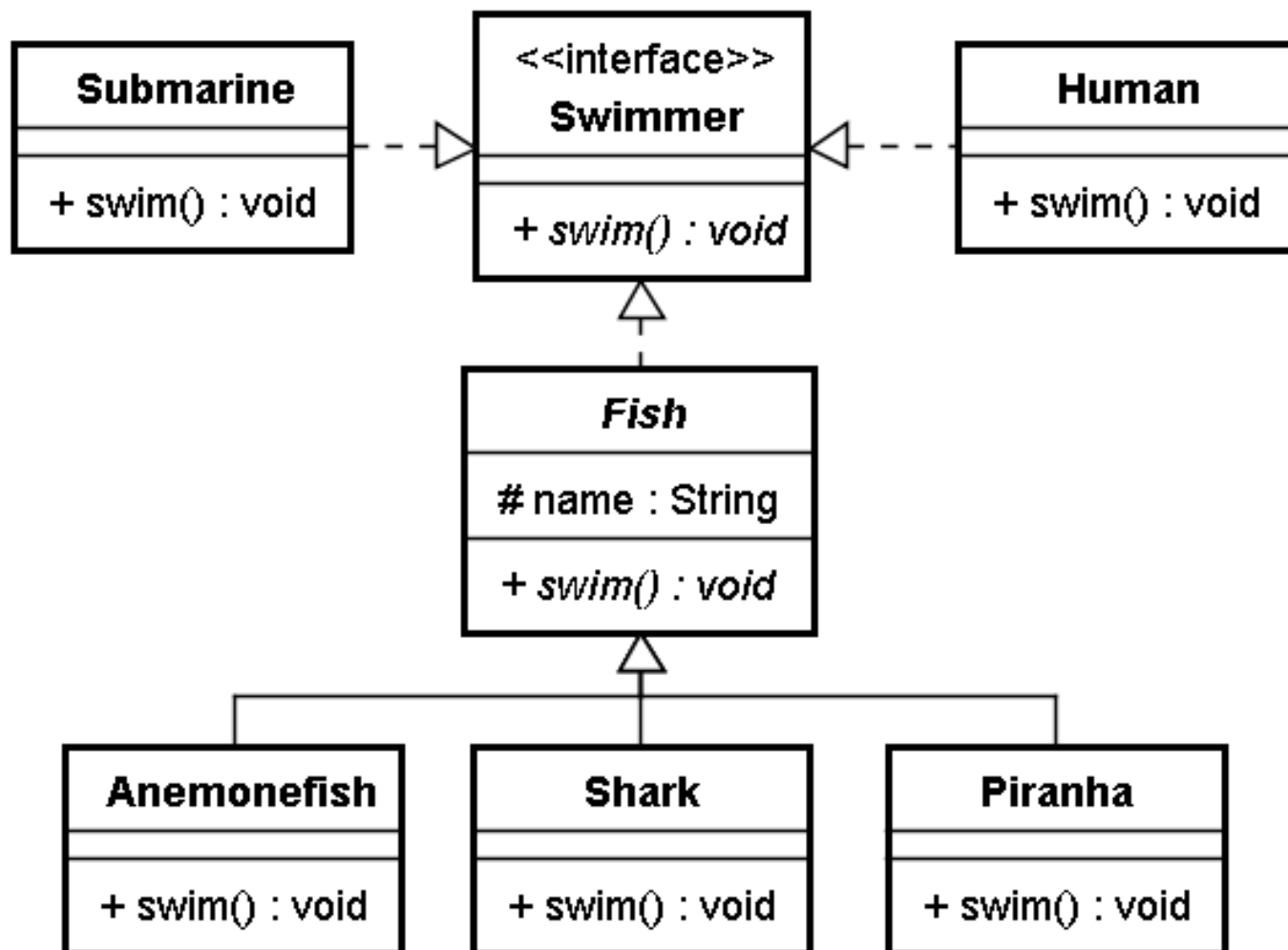
```
public class Human implements Swimmer {  
    private String name;  
    public Human(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
  
    @Override  
    public void swim() {  
        System.out.printf("人類 %s 游泳%n", name);  
    }  
}
```

介面定義行為

- Submarine 也有 Swimmer 的行為：

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

介面定義行為



介面定義行為


- 繼承會有「是一種」關係，實作介面則表示「擁有行為」，但不會有「是一種」的關係

行為的多型

- 再來當編譯器，看看哪些是合法的多型語法


```
Swimmer swimmer1 = new Shark();  
Swimmer swimmer2 = new Human();  
Swimmer swimmer3 = new Submarine();
```

擁有 **Swimmer** 行為



```
Swimmer swimmer1 = new Shark();
```

實作 **Swimmer** 介面



```
Swimmer swimmer2 = new Human();
```


行為的多型

- 底下的程式碼是否可通過編譯？

```
Swimmer swimmer = new Shark();  
Shark shark = swimmer;
```

- 加上扮演（Cast）語法，編譯器別再囉嗦：

```
Swimmer swimmer = new Shark();  
Shark shark = (Shark) swimmer;
```

行為的多型

- 底下的程式片段編譯失敗：

```
Swimmer swimmer = new Shark();
```

```
Fish fish = swimmer;
```

- 如果加上扮演語法：

```
Swimmer swimmer = new Shark();
```

```
Fish fish = (Fish) swimmer;
```

行為的多型

- 下面例子就會拋出**ClassCastException**錯誤：

```
Swimmer swimmer = new Human();  
Shark shark = (Shark) swimmer;
```

- 底下的例子也會出錯：

```
Swimmer swimmer = new Submarine();  
Fish fish = (Fish) swimmer;
```

行為的多型

- 寫個static的swim()方法，讓會游的東西都游起來
- 在不會使用介面多型語法時，也許會寫下：

```
public static void doSwim(Fish fish) {  
    fish.swim();  
}  
public static void doSwim(Human human) {  
    human.swim();  
}  
public static void doSwim(Submarine submarine) {  
    submarine.swim();  
}
```

行為的多型

- 問題是，如果「種類」很多怎麼辦？

```
public static void doSwim(Swimmer swimmer) { ← ❶ 參數是 Swimmer 型態  
    swimmer.swim();  
}
```

```
public static void main(String[] args) {  
    doSwim(new Anemonefish("尼莫"));  
    doSwim(new Shark("蘭尼"));  
    doSwim(new Human("賈斯汀"));  
    doSwim(new Submarine("黃色一號"));  
}
```

解決需求變化

- 寫程式要有彈性，要有可維護性！那麼什麼叫有彈性？何謂可維護？
- 最簡單的定義開始：如果增加新的需求，原有的程式無需修改，只需針對新需求撰寫程式，那就是有彈性、具可維護性的程式。

解決需求變化

- 如果今天老闆突發奇想，想把海洋樂園變為海空樂園，有的東西會游泳，有的東西會飛，有的東西會游也會飛，那麼現有的程式可以應付這個需求嗎？

解決需求變化

- 使用interface定義了Flyer介面：

```
public interface Flyer {  
    public abstract void fly();  
}
```


解決需求變化

```
public class Seaplane implements Swimmer, Flyer {
    private String name;

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

    @Override
    public void fly() {
        System.out.printf("海上飛機 %s 在飛%n", name);
    }

    @Override
    public void swim() {
        System.out.printf("海上飛機 %s 航行海面%n", name);
    }
}
```

解決需求變化

```
public class FlyingFish extends Fish implements Flyer {  
    public FlyingFish(String name) {  
        super(name);  
    }  
  
    @Override  
    public void swim() {  
        System.out.println("飛魚游泳");  
    }  
  
    @Override  
    public void fly() {  
        System.out.println("飛魚會飛");  
    }  
}
```

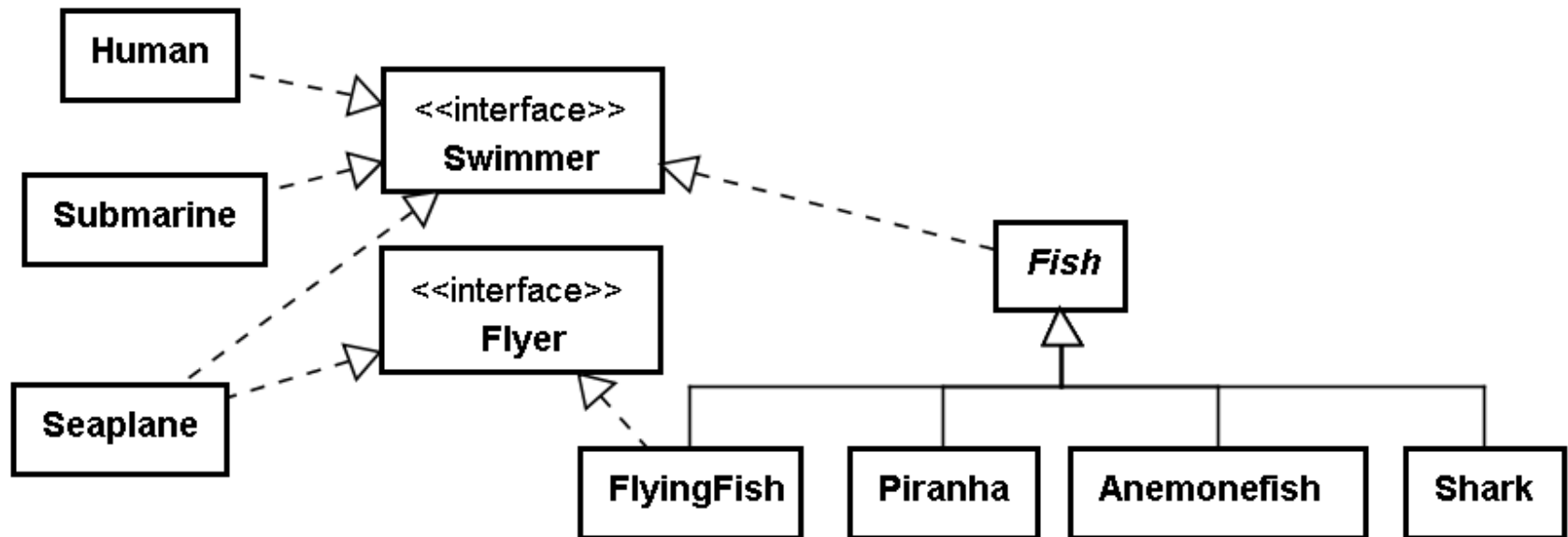
解決需求變化

- 如果現在要讓所有會游的東西游泳，那麼7.1.1節中的doSwim()方法就可滿足需求了

```
public class Ocean {  
    public static void doSwim(Swimmer swimmer) {  
        swimmer.swim();  
    }  
  
    public static void main(String[] args) {  
        略...  
        doSwim(new Seaplane("空軍零號"));  
        doSwim(new FlyingFish("甚平"));  
    }  
}
```

解決需求變化

- 就滿足目前需求來說，你所作的就是新增程式碼來滿足需求，但沒有修改舊有既存的程式碼，你的程式確實擁有某種程度的彈性與可維護性



解決需求變化

- 原有程式架構也許確實可滿足某些需求，但有些需求也可能超過了原有架構預留之彈性
- 一開始要如何設計才會有彈性，是必須靠經驗與分析判斷
- 不用為了保有程式彈性的彈性而過度設計，因為過大的彈性表示過度預測需求，有的設計也許從不會遇上事先假設的需求

解決需求變化

- 也許你預先假設會遇上某些需求而設計了一個介面，但從程式開發至生命週期結束，該介面從未被實作過，或者僅有一個類別實作過該介面，那麼該介面也許就不必存在，你事先的假設也許就是過度預測需求

解決需求變化

- 事先的設計也有可能因為需求不斷增加，而超出原本預留之彈性
- 老闆又開口了：不是所有的人都會游泳啊！有的飛機只會飛，不能停在海上啊！ ...

解決需求變化

```
public class Human {  
    protected String name;  
  
    public Human(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```


解決需求變化

```
public class SwimPlayer extends Human implements Swimmer {  
    public SwimPlayer(String name) {  
        super(name);  
    }  
  
    @Override  
    public void swim() {  
        System.out.printf("游泳選手 %s 游泳%n", name);  
    }  
}
```

解決需求變化

```
public class Airplane implements Flyer {
    protected String name;

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

    @Override
    public void fly() {
        System.out.printf("飛機 %s 在飛%n", name);
    }
}
```

解決需求變化

```
public class Seaplane extends Airplane implements Swimmer {
    public Seaplane(String name) {
        super(name);
    }

    @Override
    public void fly() {
        System.out.print("海上");
        super.fly();
    }

    @Override
    public void swim() {
        System.out.printf("海上飛機 %s 航行海面%n", name);
    }
}
```

解決需求變化

```
public class Helicopter extends Airplane {  
    public Helicopter(String name) {  
        super(name);  
    }  
  
    @Override  
    public void fly() {  
        System.out.printf("飛機 %s 在飛%n", name);  
    }  
}
```

解決需求變化

- 這一連串的修改，都是為了調整程式架構，這只是個簡單的示範，想像一下，在更大規模的程式中調整程式架構會有多麼麻煩，而且

```
doSwim(new Anemonefish("尼莫"));  
doSwim(new Shark("蘭尼"));  
doSwim(new Human("賈斯汀"));
```

```
method doSwim in class cc.openhome.Ocean cannot be applied to given types;  
  required: cc.openhome.Swimmer  
  found:    cc.openhome.Human  
  reason:   actual argument cc.openhome.Human cannot be converted to  
cc.openhome.Swimmer by method invocation conversion  
-----  
(Alt-Enter shows hints)
```

解決需求變化

- 也許老闆又想到了：水裡的話，將淺海游泳與深海潛行分開好了！ ...

```
public interface Diver extends Swimmer {  
    public abstract void dive();  
}
```

解決需求變化

```
public class Boat implements Swimmer {  
    protected String name;  
  
    public Boat(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public void swim() {  
        System.out.printf("船在水面 %s 航行%n", name);  
    }  
}
```

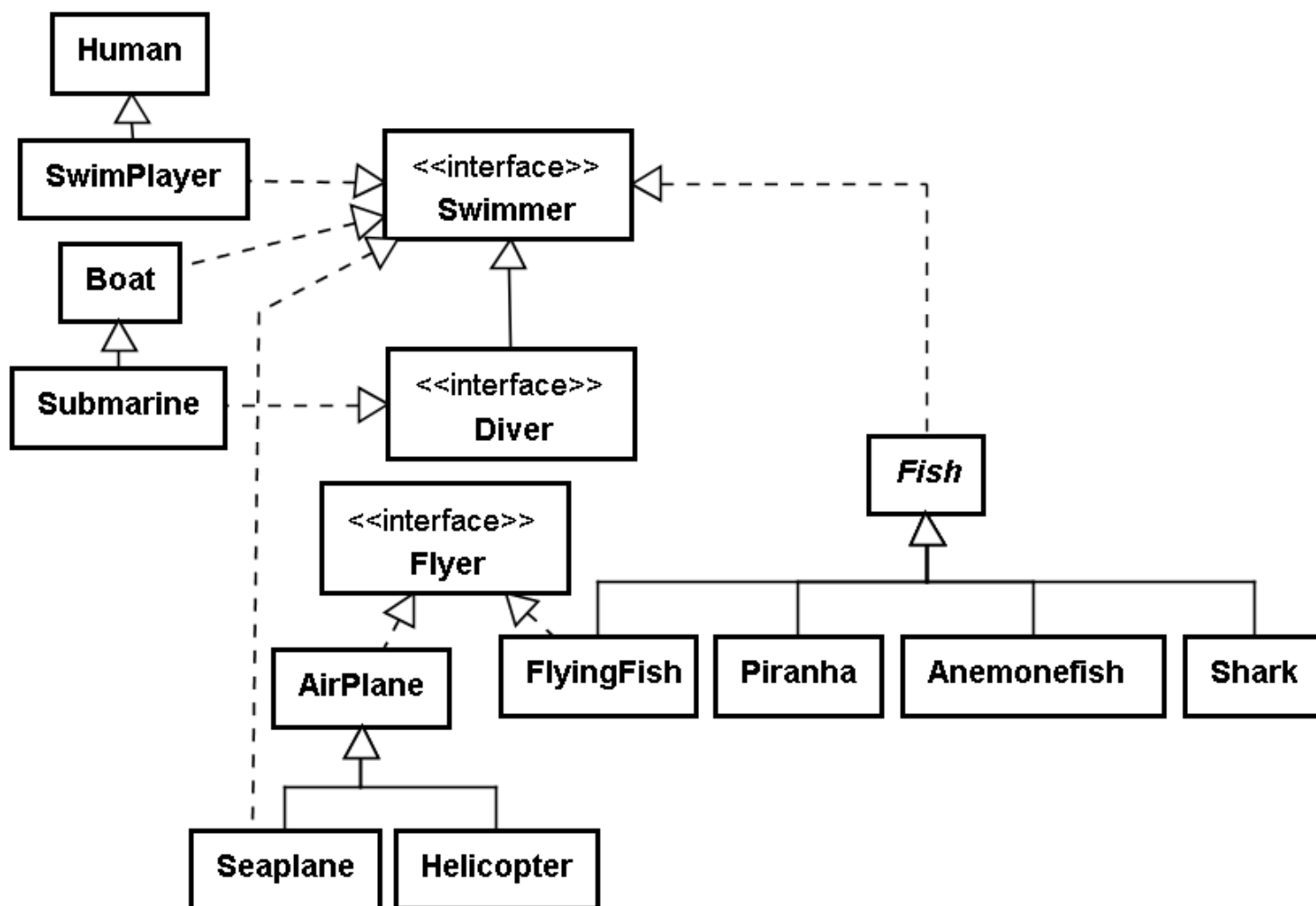
解決需求變化

```
public class Submarine extends Boat implements Diver {  
    public Submarine(String name) {  
        super(name);  
    }  
  
    @Override  
    public void dive() {  
        System.out.printf("潛水艇 %s 潛行%n", name);  
    }  
}
```


解決需求變化

- 需求不斷變化，架構也有可能因此而修改，好的架構在修改時，其實也不會全部的程式碼都被牽動，這就是設計的重要性

解決需求變化



解決需求變化

- 過像這位老闆無止境地在擴張需求，他說一個你改一個，也不是辦法，找個時間，好好跟老闆談談這個程式的需求邊界到底在哪吧！ ...

介面的預設

- 在Java中，使用interface來定義抽象的行為外觀，方法要宣告為**public abstract**，無需且不能有實作

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

```
public interface Swimmer {  
    void swim(); // 預設就是public abstract  
}
```

介面的預設

- 認證考試上經常會出這個題目：

```
interface Action {  
    void execute();  
}  
  
class Some implements Action {  
    void execute() {  
        System.out.println("作一些服務");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Action action = new Some();  
        action.execute();  
    }  
}
```

- 「請問你執行結果為何？」這個問題本身就是個陷阱，根本無法編譯成功 ...

介面的預設

- 在interface中，可以定義常數：

```
public interface Action {  
    public static final int STOP = 0;  
    public static final int RIGHT = 1;  
    public static final int LEFT = 2;  
    public static final int UP = 3;  
    public static final int DOWN = 4;  
}
```

```
public static void play(int action) {
    switch(action) {
        case Action.STOP:
            System.out.println("播放停止動畫");
            break;
        case Action.RIGHT:
            System.out.println("播放向右動畫");
            break;
        case Action.LEFT:
            System.out.println("播放向左動畫");
            break;
        case Action.UP:
            System.out.println("播放向上動畫");
            break;
        case Action.DOWN:
            System.out.println("播放向下動畫");
            break;
        default:
            System.out.println("不支援此動作");
    }
}

public static void main(String[] args) {
    play(Action.RIGHT);
    play(Action.UP);
}
```

介面的預設

- 想想看，如果將上面這個程式改為以下，哪個在維護程式時比較清楚呢？

```
public static void play(int action) {  
    switch(action) {  
        case 0:    // 數字比較清楚？還是列舉常數比較清楚？  
            System.out.println("播放停止動畫");  
            break;  
        case 1:  
            System.out.pr  
            break;  
        case 2:  
            System.out.println("播放向左動畫");  
            break;  
            略...  
        default:  
            System.out.println("不支援此動作");  
    }  
}
```

```
public static void main(String[] args) {  
    play(1);    // 數字比較清楚？還是列舉常數比較清楚？  
    play(3);  
}
```


介面的預設

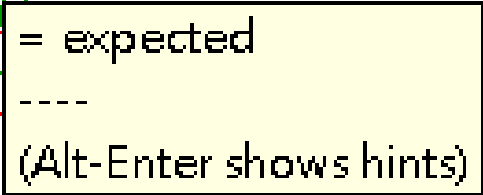
- 在interface中，也只能定義public static final的列舉常數

```
public interface Action {  
    int STOP = 0;  
    int RIGHT = 1;  
    int LEFT = 2;  
    int UP = 3;  
    int DOWN = 4;  
}
```

介面的預設

- 在介面中列舉常數，一定要使用=指定值，否則就會編譯錯誤：

```
public interface Action {  
    int STOP;  
    int RIGHT;  
    int LEFT;  
    int UP;  
    int DOWN;  
}
```



= expected

(Alt-Enter shows hints)

介面的預設

- 如果有兩個介面都定義了某方法，而實作兩個介面的類別會怎樣嗎？

```
interface Some {  
    void execute();  
    void doSome();  
}  
  
interface Other {  
    void execute();  
    void doOther();  
}
```

介面的預設

```
public class Service implements Some, Other {  
    @Override  
    public void execute() {  
        System.out.println("execute()");  
    }  
    @Override  
    public void doSome() {  
        System.out.println("doSome()");  
    }  
    @Override  
    public void doOther() {  
        System.out.println("doOther()");  
    }  
}
```

介面的預設

- 在設計上，你要思考一下：**Some**與**Other**定義的**execute()**是否表示不同的行為？
- 如果表示相同的行為，那可以定義一個父介面 ...

```
interface Action {  
    void execute();  
}  
interface Some extends Action {  
    void doSome();  
}  
interface Other extends Action {  
    void doOther();  
}
```

匿名內部類別

- 經常會有臨時繼承某個類別或實作某個介面並建立實例的需求，由於這類子類別或介面實作類別只使用一次，不需要為這些類別定義名稱 ...

```
new 父類別() | 介面() {  
    // 類別本體實作  
};
```

匿名內部類別

```
Object o = new Object() { // 繼承 Object 重新定義 toString() 並直接產生實例
    @Override
    public String toString() {
        return "無聊的語法示範而已";
    }
};
```

```
Some some = new Some() { // 實作 Some 介面並直接產生實例
    public void doService() {
        System.out.println("作一些事");
    }
};
```

匿名內部類別

- 假設你打算開發多人連線程式，對每個連線客戶端，都會建立Client物件封裝相關資訊：

```
public class Client {  
    public final String ip;  
    public final String name;  
    public Client(String ip, String name) {  
        this.ip = ip;  
        this.name = name;  
    }  
}
```


匿名內部類別

- 可以將Client加入或移除的資訊包裝為ClientEvent：

```
public class ClientEvent {  
    private Client client;  
    public ClientEvent(Client client) {  
        this.client = client;  
    }  
  
    public String getName() {  
        return client.name;  
    }  
  
    public String getIp() {  
        return client.ip;  
    }  
}
```

匿名內部類別

- 可以定義 `ClientListener` 介面，如果有物件對 `Client` 加入 `ClientQueue` 有興趣，可以實作這個介面：

```
public interface ClientListener {  
    void clientAdded(ClientEvent event);    // 新增 Client 會呼叫這個方法  
    void clientRemoved(ClientEvent event); // 移除 Client 會呼叫這個方法  
}
```

```
public class ClientQueue {  
    private ArrayList clients = new ArrayList(); ← ❶ 收集連線的 Client  
    private ArrayList listeners = new ArrayList(); ← ❷ 收集對 ClientQueue 有  
                                                    興趣的 ClientListener  
    public void addClientListener(ClientListener listener) { ← ❸ 註冊  
        listeners.add(listener); ClientListener  
    }  
  
    public void add(Client client) {  
        clients.add(client); ← ❹ 新增 Client  
        ClientEvent event = new ClientEvent(client); ← ❺ 通知資訊包裝為  
        for(int i = 0; i < listeners.size(); i++) { ClientEvent  
            ClientListener listener = (ClientListener) listeners.get(i);  
            listener.clientAdded(event); ← ❻ 逐一取出 ClientListener 通知  
        }  
    }  
  
    public void remove(Client client) {  
        clients.remove(client);  
        ClientEvent event = new ClientEvent(client);  
        for(int i = 0; i < listeners.size(); i++) {  
            ClientListener listener = (ClientListener) listeners.get(i);  
            listener.clientRemoved(event);  
        }  
    }  
}
```

匿名內部類別

```
ClientQueue queue = new ClientQueue();
queue.addClientListener(new ClientListener() {
    @Override
    public void clientAdded(ClientEvent event) {
        System.out.printf("%s 從 %s 連線%n",
            event.getName(), event.getIp());
    }

    @Override
    public void clientRemoved(ClientEvent event) {
        System.out.printf("%s 從 %s 離線%n",
            event.getName(), event.getIp());
    }
});

Client c1 = new Client("127.0.0.1", "Caterpillar");
Client c2 = new Client("192.168.0.2", "Justin");
queue.add(c1);
queue.add(c2);
queue.remove(c1);
queue.remove(c2);
```

匿名內部類別

- 如果要在匿名內部類別中存取區域變數，則該區域變數必須是**final**，否則會發生編譯錯誤：

```
int[] arrs = {10, 20};
```

```
Object o = new Object() {
```

```
    public String toString() {
```

```
        return "example: " + arrs[0];
```

```
    }
```

```
};
```

local variable arrs is accessed from within inner class; needs to be declared final

(Alt-Enter shows hints)

匿名內部類別

- 必須宣告 `arrs` 為 `final` 才可以通過編譯：

```
final int[] x = {10, 20};
Object o = new Object() {
    public String toString() {
        return "example: " + x[0];
    }
};
```

使用enum列舉常數

- 參數接受的是int型態，這表你可以傳入任何的int值，因此不得已地使用default，以處理執行時期傳入非定義範圍的int值

```
...
    public static void play(int action) {
        switch(action) {
            case Action.STOP:
                System.out.println("播放停止動畫");
                break;
            略...
            default:
                System.out.println("不支援此動作");
        }
    }
    ...
```

使用enum列舉常數

- 從JDK5之後新增了**enum**語法，可用於定義列舉常數：

```
public enum Action {  
    STOP, RIGHT, LEFT, UP, DOWN  
}
```


使用enum列舉常數

```
public final class Action extends Enum {  
    略...  
    private Action(String s, int i) {  
        super(s, i);  
    }  
    public static final Action STOP;  
    public static final Action RIGHT;  
    public static final Action LEFT;  
    public static final Action UP;  
    public static final Action DOWN;  
    略...  
    static {  
        STOP = new Action("STOP", 0);  
        RIGHT = new Action("RIGHT", 1);  
        LEFT = new Action("LEFT", 2);  
        UP = new Action("UP", 3);  
        DOWN = new Action("DOWN", 4);  
        略...  
    }  
}
```

使用enum列舉常數

```
public static void play(Action action) { ← ❶ 宣告為 Action 型態
    switch(action) {
        case STOP: // 也就是 Action.STOP ← ❷ 列舉 Action 實例
            System.out.println("播放停止動畫");
            break;
        case RIGHT: // 也就是 Action.RIGHT
            System.out.println("播放向右動畫");
            break;
        case LEFT: // 也就是 Action.LEFT
            System.out.println("播放向左動畫");
            break;
        case UP: // 也就是 Action.UP
            System.out.println("播放向上動畫");
            break;
        case DOWN: // 也就是 Action.DOWN
            System.out.println("播放向下動畫");
            break;
    }
}

public static void main(String[] args) {
    play(Action.RIGHT); ← ❸ 只能傳入 Action 實例
    play(Action.UP);
}
```