# 5.1 何謂封裝？

在第 4 章稍微介紹了如何定義類別，有個觀念必須先釐清，定義類別並不等於作好了物件導向中**封裝（Encapsulation）**的概念，那麼到底什麼才有封裝的意涵？你必須以物件的角度來思考問題。

本節著重在封裝的觀念，並一併說明如何以 Java 語法來實作，有一些內容會略為與第 4 章重複，這是為了介紹上的完整性，在瞭解封裝基本概念之後，下一節會進入 Java 的語法細節。

## 5.1.1 封裝物件初始流程

假設你要寫個可以管理儲值卡的應用程式，首先得定義儲值卡會記錄哪些資料，像是儲值卡號碼、餘額、紅利點數，在 Java 中可使用 **class** 關鍵字進行定義：

```
package cc.openhome;
class CashCard {
    String number;
    int balance;
    int bonus;
}
```

假設你將這個類別是定義在 cc.openhome 套件，使用 CashCard.java 儲存，並編譯為 CashCard.class，並將這個位元碼給朋友使用，你的朋友要建立 5 張儲值卡的資料：

```
CashCard card1 = new CashCard();
card1.number = "A001";
card1.balance = 500;
card1.bonus = 0;

CashCard card2 = new CashCard();
card2.number = "A002";
card2.balance = 300;
card2.bonus = 0;

CashCard card3 = new CashCard();
card3.number = "A003";
card3.balance = 1000;
card3.bonus = 1;  // 單次儲值 1000 元可獲得紅利一點
...
```

在這邊可以看到，如果想存取物件的資料成員，可以透過「.」運算子加上資料

成員名稱。

你發現到每次他在建立儲值卡物件時，都會作相同的初始動作，也就是指定卡號、餘額與紅利點數，這個流程是重複的，更多的 CashCard 物件建立會帶來更多的程式碼重複，在程式中出現重複的流程，往往意謂著有改進的空間，在 4.1.1 中談過，Java 中可以定義**建構式（Constructor）**來改進這個問題：

Encapsulation1 CashCard.java

```
package cc.openhome;
class CashCard {
    String number;
    int balance;
    int bonus;
    CashCard(String number, int balance, int bonus) {
        this.number = number;
        this.balance = balance;
        this.bonus = bonus;
    }
}
```

正如 4.1.1 談過的，建構式是與類別名稱同名的**方法（Method）**，不用宣告傳回型態，在這個例子中，建構式上的 number、balance 與 bonus 參數，與類別的 number、balance、bonus 資料成員同名了，為了區別，在物件資料成員前加上 **this** 關鍵字，表示將 number、balance 與 bonus 參數的值，指定給這個物件的 number、balance、bonus 資料成員。

在你重新編譯 CashCard.java 為 CashCard.class 之後，交給你的朋友，同樣是建立五個 CashCard 物件，現在他只要這麼寫：

```
CashCard card1 = new CashCard("A001", 500, 0);
CashCard card2 = new CashCard("A002", 300, 0);
CashCard card3 = new CashCard("A003", 1000, 1);
...
```

比較看看，他應該會想寫這個程式片段，還是剛剛那個程式片段？那麼你封裝了什麼？你用了 Java 的建構式語法，**實現物件初始化流程的封裝**。封裝物件初始化流程有什麼好處？拿到 CashCard 類別的使用者，不用重複撰寫物件初始化流程，事實上，他也不用知道物件如何初始化，就算你修改了建構式的內容，重新編譯並給予位元碼檔案之後，CashCard 類別的使用者也無需修改程式。

實際上，如果你的類別使用者想建立 5 個 CashCard 物件，並將資料顯示出來，可以用陣列，而無需個別宣告參考名稱。例如：

Encapsulation1 CashApp.java

```
package cc.openhome;
```

```
public class CardApp {
    public static void main(String[] args) {
        CashCard[] cards = {
            new CashCard("A001", 500, 0),
            new CashCard("A002", 300, 0),
            new CashCard("A003", 1000, 1),
            new CashCard("A004", 2000, 2),
            new CashCard("A005", 3000, 3)
        };

        for(CashCard card : cards) {
            System.out.printf("(%s, %d, %d)%n",
                    card.number, card.balance, card.bonus);
        }
    }
}
```

執行結果如下所示：

```
(A001, 500, 0)
(A002, 300, 0)
(A003, 1000, 1)
(A004, 2000, 2)
(A005, 3000, 3)
```

| 提示 | 接下來說明範例時，都會假設有兩個以上的開發者。記得！如果物件導向或設計上的議題對你來說太抽象，請用兩人或多人共同開發的角度來想想看，這樣的觀念與設計對大家合作有沒有好處。 |
|---|---|

## 5.1.2 封裝物件操作流程

假設現在你的朋友使用 CashCard 建立 3 個物件，並要再對所有物件進行儲值的動作：

```
Scanner scanner = new Scanner(System.in);
CashCard card1 = new CashCard("A001", 500, 0);
int money = scanner.nextInt();
if(money > 0) {
    card1.balance += money;
    if(money >= 1000) {
        card1.bonus++;
```

```
    }
}
else {
    System.out.println("儲值是負的？你是來亂的嗎？");
}

CashCard card2 = new CashCard("A002", 300, 0);
money = scanner.nextInt();
if(money > 0) {
    card2.balance += money;
    if(money >= 1000) {
        card2.bonus++;
    }
}
else {
    System.out.println("儲值是負的？你是來亂的嗎？");
}

CashCard card3 = new CashCard("A003", 1000, 1);
// 還是那些 if..else 的重複流程
...
```

Lab

你的朋友作了簡單的檢查，就是儲值不能是負的，而儲值大於 1000 的話，就給予紅利一點，很容易就可以發現，那些儲值的流程重複了。你想了一下，儲值這個動作應該是 CashCard 物件自己處理！在 Java 中，你可以定義**方法（Method）**來解決這個問題：

Encapsulation2　CashCard.java

```
package cc.openhome;
class CashCard {
    String number;
    int balance;
    int bonus;
    CashCard(String number, int balance, int bonus) {
        this.number = number;
        this.balance = balance;
        this.bonus = bonus;
    }
```

❶不會傳回值

```
    void store(int money) {  // 儲值時呼叫的方法
```

```
        if(money > 0) {
            this.balance += money;
            if(money >= 1000) {
                this.bonus++;
            }
        }
        else {
            System.out.println("儲值是負的？你是來亂的嗎？");
        }
    }

    void charge(int money) { // 扣款時呼叫的方法
        if(money > 0) {
            if(money <= this.balance) {
                this.balance -= money;
            }
            else {
                System.out.println("錢不夠啦！");
            }
        }
        else {
            System.out.println("扣負數？這不是叫我儲值嗎？");
        }
    }
    int exchange(int bonus) {  // 兌換紅利點數時呼叫的方法
        if(bonus > 0) {
            this.bonus -= bonus;
        }
        return this.bonus;
    }
}
```

❷封裝儲值流程

❸會傳回 int 型態

在 CashCard 類別中，除了定義儲值用的 store() 方法之外，你還考慮到扣款用的 charge() 方法，以及兌換紅利點數的 exchange() 方法。在類別中定義方法，如果不用傳回值，方法名稱前可以宣告 **void**❶。

先前看到的儲值重複流程，現在都封裝到 store() 方法中❷，這麼作的好處是使用 CashCard 的使用者，現在可以這麼撰寫了：

```
Scanner scanner = new Scanner(System.in);
CashCard card1 = new CashCard("A001", 500, 0);
card1.store(scanner.nextInt());

CashCard card2 = new CashCard("A002", 300, 0);
card2.store(scanner.nextInt());

CashCard card3 = new CashCard("A003", 1000, 1);
card3.store(scanner.nextInt());
```

好處是什麼顯而易見，相較於先前得撰寫重複流程，CashCard 使用者應該會比較想寫這個吧！你封裝了什麼呢？**你封裝了儲值的流程**。哪天你也許考慮每加值 1000 元就增加一點紅利，而不像現在就算加值 5000 元也只有一點紅利，就算改變了 store()的流程，CashCard 使用者也無需修改程式。

同樣地，charge()與 exchange()方法也分別封裝了扣款以及兌換紅利點數的流程。為了知道兌換紅利點數後，剩餘的點數還有多少，exchange()必須傳回剩餘的點數值，方法若會傳回值，必須於方法前宣告傳回值的型態❸。

| 提示 | 在 Java 命名慣例中，方法名稱首字是小寫。 |
|---|---|

Lab

其實如果是直接建立三個 CashCard 物件，而後進行儲值並顯示明細，可以如下使用陣列，讓程式更簡潔：

Encapsulation2 CashApp.java

```
package cc.openhome;

import java.util.Scanner;

public class CardApp {
    public static void main(String[] args) {
        CashCard[] cards = {
            new CashCard("A001", 500, 0),
            new CashCard("A002", 300, 0),
            new CashCard("A003", 1000, 1)
        };

        Scanner scanner = new Scanner(System.in);
        for(CashCard card : cards) {
            System.out.printf("為 (%s, %d, %d) 儲值：",
                    card.number, card.balance, card.bonus);
            card.store(scanner.nextInt());
            System.out.printf("明細 (%s, %d, %d)%n",
```