

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 5

• 物件封裝



學習目標

- 瞭解封裝觀念與實現
- 定義類別、建構式與方法
- 使用方法重載與不定長度引數
- 瞭解static成員

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 ...

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

封裝物件初始流程

- 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  
...
```

封裝物件初始流程

- 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 ...
- 在程式中出現重複的流程，往往意謂著有改進的空間 ...

封裝物件初始流程

- 可以定義建構式（Constructor）來改進這個問題：

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

封裝物件初始流程

- 在你重新編譯CashCard.java為CashCard.class之後，交給你的朋友 ...

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

封裝物件初始流程

- 他應該會想寫哪個程式片段？
- 你封裝了什麼？
 - 你用了建構式語法，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

封裝物件操作流程

- 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
...
```

封裝物件操作流程

- 那些儲值的流程重複了
- 儲值應該是CashCard物件自己處理
- 可以定義方法（Method）來解決這個問題

```
void store(int money) { // 儲值時呼叫的方法 ← ❶ 不會傳回值
    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？");
    }
}
```

❷ 封裝儲值流程

```
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 ← ❸ 會傳回 int 型態
    if(bonus > 0) {
        this.bonus -= bonus;
    }
    return this.bonus;
}
```

封裝物件操作流程

- 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());
```

封裝物件操作流程

- 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。

封裝物件內部資料

- 你「希望」使用者如下撰寫程式 ...

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

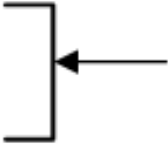
- 你的希望完全就是一廂情願，因為
CashCard使用者還是可以如下撰寫程式，
跳過你的相關條件檢查：

```
CashCard card1 = new CashCard("A001", 500, 0);  
card1.balance += scanner.nextInt();  
card1.bonus += 100;
```

封裝物件內部資料

- 你沒有封裝CashCard中不想讓使用者直接存取的私有資料，
- 如果有些資料是類別所私有，可以使用**private**關鍵字定義

```
class CashCard {  
    private String number;  
    private int balance;  
    private int bonus;  
    ...略
```



① 使用 private 定義私有

封裝物件內部資料

```
void store(int money) { ← ❷ 要修改 balance，得透過 store() 定義的流程
    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？");
    }
}
```

```
int getBalance() {
    return balance;
}
```

```
int getBonus() {
    return bonus;
}
```

← ❸ 提供取值方法成員

封裝物件內部資料

- 編譯器會讓使用者在直接存取number、balance與bonus時編譯失敗：

```
Scanner sc  
CashCard c  
card1.balance += scanner.nextInt();  
card1.bonus += 100;
```

balance has private access in cc.openhome.CashCard

(Alt-Enter shows hints)

封裝物件內部資料

- 如果沒辦法直接取得number、balance與bonus，那這段程式碼怎麼辦？

```
System.out.printf("明細 (Alt-Enter shows hints)
    card1.number] card1.balance, card1.bonus);
System.out.printf("明細 (%s, %d, %d)%n",
    card2.number, card2.balance, card2.bonus);
System.out.printf("明細 (%s, %d, %d)%n",
    card3.number, card3.balance, card3.bonus);
```

number has private access in cc.openhome.CashCard

封裝物件內部資料

- 基於你的意願，CashCard類別上定義了 `getNumber()`、`getBalance()` 與 `getBonus()` 等取值方法 ❸

```
System.out.printf("明細 (%s, %d, %d)%n",  
    card1.getNumber(), card1.getBalance(), card1.getBonus());  
System.out.printf("明細 (%s, %d, %d)%n",  
    card2.getNumber(), card2.getBalance(), card2.getBonus());  
System.out.printf("明細 (%s, %d, %d)%n",  
    card3.getNumber(), card3.getBalance(), card3.getBonus());
```

封裝物件內部資料

- 你封裝了什麼？
 - 封裝了類別私有資料，讓使用者無法直接存取，而必須透過你提供的操作方法，經過你定義的流程才有可能存取私有資料
 - 事實上，使用者也無從得知你的類別中有哪些私有資料，使用者不會知道物件的內部細節。

何謂封裝？

- 封裝目的主要就是隱藏物件細節，將物件當作黑箱進行操作。
 - 使用者會呼叫建構式，但不知道建構式的細節
 - 使用者會呼叫方法，但不知道方法的流程
 - 使用者也不會知道有哪些私有資料
 - 要操作物件，一律得透過你提供的方法呼叫

public 權限修飾

- 假設現在為了管理需求，要將CashCard類別定義至cc.openhome.virtual套件中
- 除了原始碼與位元碼的資料夾需求必須符合套件階層之外，原始碼內容也得作些修改：

```
package cc.openhome.virtual;  
class CashCard {  
    ...  
}
```

public 權限修飾

- 你發現使用CashCard的程式碼都出錯了：

```
import cc.openhome.virtual.CashCard;
```

```
cc.openhome.virtual.CashCard is not public in cc.openhome.virtual; cannot be accessed from outside package
```

```
----
```

```
(Alt-Enter shows hints)
```

```
public static void main(String[] args) {  
    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());  
}
```


public 權限修飾

- 如果沒有宣告權限修飾的成員，只有在相同套件的類別程式碼中，才可以直接存取，也就是「套件範圍權限」
- 如果不同套件的類別程式碼中，想要直接存取，就會出現圖5.4的錯誤訊息
- 如果想在其它套件的類別程式碼中存取某套件的類別或物件成員，則該類別或物件成員必須是公開成員，要使用**public**加以宣告

public 權限修飾

```
public class CashCard { ← ❶ 這是個公開類別  
    ...略
```

```
    public CashCard(String number, int balance, int bonus) { ← ❷ 這是個公開建構式  
        ...略  
    }
```

```
    public void store(int money) {  
        ...略  
    }
```

```
    public void charge(int money) {  
        ...略  
    }
```

```
    public int exchange(int bonus) {  
        ...略  
    }
```

```
    public int getBalance() {  
        return balance;  
    }
```

```
    public int getBonus() {
```

← ❸ 這些是公開方法

關於建構式

- 在定義類別時，可以使用建構式定義物件建立的初始流程
- 建構式是與類別名稱同名，無需宣告傳回型態的方法

```
public class Some {  
    private int a = 10;    // 指定初始值  
    private String text;   // 預設值 null  
    public Some(int a, String text) {  
        this.a = a;  
        this.text = text;  
    }  
    ...  
}
```

關於建構式

- 如果你如下建立Some物件，成員a與text會初始兩次：

```
Some some = new Some(10, "some text");
```

資料型態	初始值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	\u0000
boolean	false
類別	null

關於建構式

- 如果定義類別時，沒有撰寫任何建構式，編譯器會自動加入無參數、內容為空的建構式

```
public class Some {  
}
```

- 反組譯後你會看到：

```
public class Some {  
    public Some() {  
    }  
}
```

關於建構式

- 編譯器會在你沒有撰寫任何建構式時，自動加入預設建構式（ Default constructor ）
- 沒有撰寫任何建構式時，也可以如下以無引數方式呼叫建構式：

```
Some some = new Some ();
```

關於建構式

- 如果自行撰寫了建構式，編譯器就不會自動建立預設建構式

```
public class Some {  
    public Some(int a) {  
    }  
}
```

- 就只有一個具int參數的建構式，所以就不可以new Some()來建構物件，而必須使用new Some(1)的形式來建構物件

建構式與方法重載

- 建構物件時也許希望有對應的初始流程，可以定義多個建構式，只要參數型態或個數不同，這稱之為重載（Overload）建構式

- 建構時有兩種選擇，一是使用new Some (100) 的方式，另一個是使用new Some (100, "some text") 的方式

```
public class Some {  
    private int a = 10;  
    private String text = "n.a.";  
    public Some(int a) {  
        if(a > 0) {  
            this.a = a;  
        }  
    }  
    public Some(int a, String text) {  
        if(a > 0) {  
            this.a = a;  
        }  
        if(text != null) {  
            this.text = text;  
        }  
    }  
    ...  
}
```

建構式與方法重載

- 定義方法時也可以進行重載，可為類似功能的方法提供統一名稱，但根據參數型態或個數的不同呼叫對應的方法

```
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(char[] data)
public static String valueOf(char[] data, int offset, int count)
public static String valueOf(double d)
public static String valueOf(float f)
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(Object obj)
```

建構式與方法重載

- 方法重載讓程式設計人員不用苦惱方法名稱的設計，用一致名稱來呼叫類似功能的方法
- 方法重載可根據傳遞引數的型態不同，也可根據參數列個數的不同來設計方法重載

```
public class SomeClass {  
    public void someMethod() {  
    }  
    public void someMethod(int i) {  
    }  
    public void someMethod(float f) {  
    }  
    public void someMethod(int i, float f) {  
    }  
}
```

建構式與方法重載

- 返回值型態不可作為方法重載依據，以下方法重載並不正確：

```
public class Some {  
    public int someMethod(int i) {  
        return 0;  
    }  
    public double someMethod(int i) {  
        return 0.0;  
    }  
}
```

建構式與方法重載

- 在JDK5之後使用方法重載時，要注意自動裝箱、拆箱問題

```
class Some {  
    void someMethod(int i) {  
        System.out.println("int 版本被呼叫");  
    }  
    void someMethod(Integer integer) {  
        System.out.println("Integer 版本被呼叫");  
    }  
}  
  
public class OverloadBoxing {  
    public static void main(String[] args) {  
        Some s = new Some();  
        s.someMethod(1);  
    }  
}
```

建構式與方法重載

- 編譯器在處理重載方法時，會依以下順序來處理：
 - 還沒有裝箱動作前可符合引數個數與型態的方法。
 - 裝箱動作後可符合引數個數與型態的方法。
 - 嘗試有不定長度引數（稍後說明）並可符合引數型態的方法。
 - 找不到合適的方法，編譯錯誤。

使用 `this`

- 除了被宣告為 `static` 的地方外，`this` 關鍵字可以出現在類別中任何地方

使用this

- 在建構式參數與物件資料成員同名時，可用this加以區別

```
public class CashCard {  
    private String number;  
    private int balance;  
    private int bonus;  
    public CashCard(String number, int balance, int bonus) {  
        this.number = number;    // 參數 number 指定給這個物件的 number  
        this.balance = balance;  // 參數 balance 指定給這個物件的 balance  
        this.bonus = bonus;      // 參數 bonus 指定給這個物件的 bonus  
    }  
    ...  
}
```


使用this

- 在5.2.3看到過這個程式片段：

```
public class Some {  
    private int a = 10;  
    private String text = "n.a.";  
    public Some(int a) {  
        if(a > 0) {  
            this.a = a;  
        }  
    }  
    public Some(int a, String text) {  
        this(a);  
        if(text != null) {  
            this.text = text;  
        }  
    }  
    ...  
}
```

使用this

- 可以在建構式中呼叫另一個已定義的建構式

```
public class Some {  
    private int a = 10;  
    private String text = "n.a.";  
    public Some(int a) {  
        if(a > 0) {  
            this.a = a;  
        }  
    }  
    public Some(int a, String text) {  
        this(a);  
        if(text != null) {  
            this.text = text;  
        }  
    }  
    ...  
}
```

- 在建構物件之後、呼叫建構式之前，若有想執行的流程，可以使用{ } 定義

```
class Other {  
    {  
        System.out.println("物件初始區塊");  
    }  
  
    Other() {  
        System.out.println("Other() 建構式");  
    }  
  
    Other(int o) {  
        this();  
        System.out.println("Other(int o) 建構式");  
    }  
}  
  
public class ObjectInitialBlock {  
    public static void main(String[] args) {  
        new Other(1);  
    }  
}
```

使用this

- 如果區域變數宣告了`final`，表示設值後就不能再變動
- 物件資料成員上也可以宣告**`final`**

```
class Something {  
    final int x = 10;  
    ...  
}
```

- 程式中其它地方不能再有對`x`設值的動作，否則會編譯錯誤

使用this

- 那以下的程式片段呢？

```
public class Something {  
    final int x;  
    ...  
}
```

- 如果物件資料成員被宣告為**final**，但沒有明確使用=指定值，那表示延遲物件成員值的指定

使用this

- 在建構式執行流程中，一定要有對該資料成員指定值的動作，否則編譯錯誤

```
class Something
```

```
    final int x;
```

```
    Something() {
```

```
    }
```

```
    Something(int x) {
```

```
        this.x = x;
```

```
    }
```

```
}
```

variable x might not have been initialized

variable x might not have been initialized

(Alt-Enter shows hints)

使用this

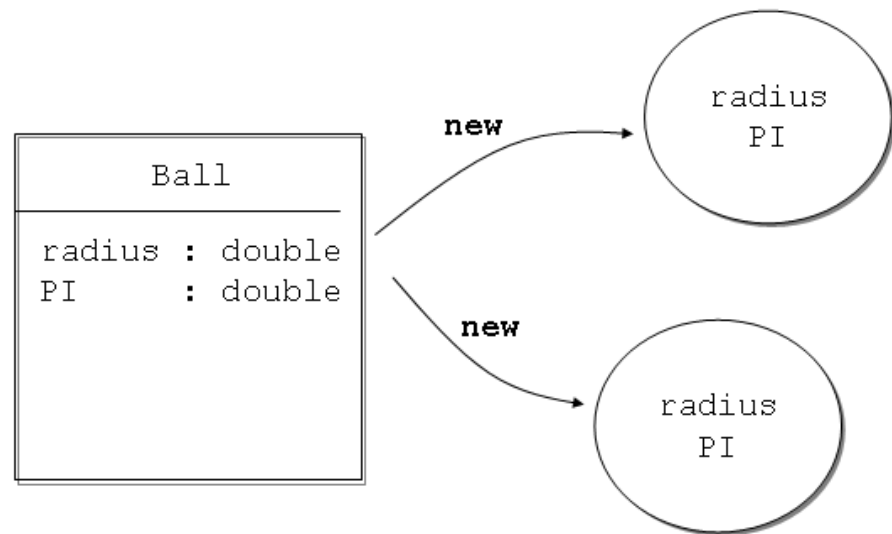
- 如果改為以下就可以通過編譯：

```
class Something {  
    final int x;  
    Something() {  
        this(10);  
    }  
    Something(int x) {  
        this.x = x;  
    }  
}
```

static類別成員

- 建立了多個Ball物件，那每個Ball物件都會有自己的radius與PI成員

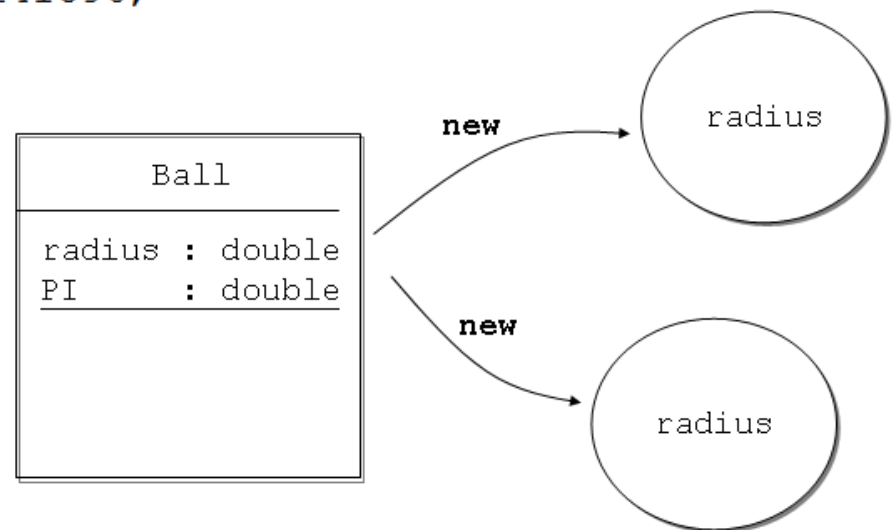
```
class Ball {  
    double radius;  
    final double PI = 3.14159;  
    ...  
}
```



static類別成員

- 圓周率其實是個固定的常數，不用每個物件各自擁有，你可以在PI上宣告**static**，表示它屬於類別：

```
class Ball {  
    double radius;  
    static final double PI = 3.141596;  
    ...  
}
```



static類別成員

- 被宣告為static的成員，是將類別名稱作為名稱空間

```
System.out.println(Ball.PI);
```

- 也可以將宣告方法為static成員

```
class Ball {  
    double radius;  
    static final double PI = 3.141596;  
    static double toRadians(double angdeg) { // 角度轉徑度  
        return angdeg * (Ball.PI / 180);  
    }  
}
```

static類別成員

- 被宣告為static的方法，也是將類別名稱作為名稱空間

```
System.out.println(Ball.toRadians(100));
```

- 雖然語法上，也是可以透過參考名稱存取static成員，但非常不建議如此撰寫：

```
Ball ball = new Ball();  
System.out.println(ball.PI); // 極度不建議  
System.out.println(ball.toRadians(100)); // 極度不建議
```

static類別成員

- Java程式設計領域，早就有許多良好命名慣例，沒有遵守慣例並不是錯，但會造成溝通與維護的麻煩
- 以類別命名實例來說，首字是大寫，以static使用慣例來說，是透過類別名稱與運算子來存取

static類別成員

- 在大家都遵守命名慣例的情況下，看到首字大寫就知道它是類別，透過類別名稱與.運算子來存取，就會知道它是static成員
- 一直在用的System.out、System.in呢？

Fields

Modifier and Type	Field and Description
-------------------	-----------------------

static <code>PrintStream</code>	<code>err</code>
---------------------------------	------------------

	The "s
--	--------

`out`

static <code>InputStream</code>	<code>in</code>
---------------------------------	-----------------

	The "s
--	--------

`public static final PrintStream out`

static <code>PrintStream</code>	<code>out</code>
---------------------------------	------------------

	The "standard" output stream.
--	-------------------------------

static類別成員

- 先前遇過的還有`Integer.parseInt()`、`Long.parseLong()`等剖析方法
- `static`成員屬於類別所擁有，將類別名稱當作是名稱空間是其最常使用之方式
- 在Java SE API中，只要想到與數學相關的功能，就會想到`java.lang.Math`，因為有許多以`Math`類別為名稱空間的常數與公用方法

static 類別成員

The screenshot shows the Java Platform SE 7 API documentation for the `Math` class. The browser window is titled "Math (Java Platform SE 7)" and the address bar shows the URL `http://download.oracle.com/javase/7/docs/api/java/lang/Math.html`. The page is divided into two main sections: "Fields" and "Method Summary".

Fields

Modifier and Type	Field and Description
static double	E The double value that is closer than any other to <i>e</i> , the base of the natural logarithms.
static double	PI The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter.

Method Summary

Methods

Modifier and Type	Method and Description
static double	abs (double <i>a</i>) Returns the absolute value of a double value.
static float	abs (float <i>a</i>) Returns the absolute value of a float value.
static int	abs (int <i>a</i>) Returns the absolute value of an int value.
static long	abs (long <i>a</i>) Returns the absolute value of a long value.
static double	acos (double <i>a</i>) Returns the arc cosine of a value; the returned angle is in the range 0.0 through <i>pi</i> .
static double	asin (double <i>a</i>) Returns the arc sine of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$.
static double	atan (double <i>a</i>) Returns the arc tangent of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$.
static double	atan2 (double <i>y</i> , double <i>x</i>) Returns the angle <i>theta</i> from the conversion of rectangular coordinates (<i>x</i> , <i>y</i>) to polar coordinates (<i>r</i> , <i>theta</i>).
static double	cbrt (double <i>a</i>) Returns the cube root of a double value.

static 類別成員

- 因為都是static成員，所以你就可以這麼使用：

```
System.out.println(Math.PI);
```

```
System.out.println(Math.toRadians(100));
```


static類別成員

- 由於static成員是屬於類別，而非個別物件，所以在static成員中使用this，會是一種語意上的錯誤
- 在**static**方法或區塊（稍後說明）中不能出現**this**關鍵字

```
class Ball {  
    double radius;  
  
    static void dos {  
        double r = this.radius;  
    }  
}
```

non-static variable this cannot be referenced from a static context

(Alt-Enter shows hints)

static類別成員

- 如果你在程式碼中撰寫了某個物件資料成員，雖然沒有撰寫`this`，但也隱含了這個物件某成員的意思

```
class Ball {  
    double radius;  
  
    static void dos {  
        double r = radius;  
    }  
}
```

non-static variable radius cannot be referenced from a static context

(Alt-Enter shows hints)

static類別成員

- static方法或區塊中，也不能呼叫非static方法或區塊

```
class Ball {  
    double radius;
```

```
    void doOther() {  
    }
```

```
    static void d  
        doOther();
```

```
}
```

non-static method doOther() cannot be referenced from a static context

(Alt-Enter shows hints)

static類別成員

- static方法或區塊中，可以使用static資料成員或方法成員

```
class Ball {  
    static final double PI = 3.141596;  
    static void doOther() {  
        double o = 2 * PI;  
    }  
  
    static void doSome() {  
        doOther();  
    }  
    ...  
}
```

static類別成員

- 如果你有些動作，想在位元碼載入後執行，則可以定義static區塊

```
class Ball {  
    static {  
        System.out.println("位元碼載入後就會被執行");  
    }  
}
```

static類別成員

- 在JDK5之後，新增了import static語法

```
import java.util.Scanner;
import static java.lang.System.in;
import static java.lang.System.out;

public class ImportStatic {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(in);
        out.print("請輸入姓名：");
        out.printf("%s 你好！%n", scanner.nextLine());
    }
}
```

static類別成員

- 如果一個類別中有多個static成員想偷懶，也可以使用*

```
import static java.lang.System.*;
```

- 名稱衝突編譯器可透過以下順序來解析：
 - 區域變數覆蓋
 - 成員覆蓋
 - 重載方法比對

static類別成員

- 如果編譯器無法判斷，則會回報錯誤

```
import static java.util.Arrays.*;  
import static cc.openhome.Util.*;
```

reference to sort is ambiguous, both method sort(int[]) in cc.openhome.Util and
method sort(int[]) in java.util.Arrays match

(Alt-Enter shows hints)

```
    sort(new int[] {3, 1, 5});  
}  
}
```


不定長度引數

- 若方法的引數個數事先無法決定該如何處理？

```
System.out.printf("%d", 10);  
System.out.printf("%d %d", 10, 20);  
System.out.printf("%d %d %d", 10, 20, 30);
```

不定長度引數

- 在JDK5之後支援不定長度引數 (Variable-length Argument)

```
public class MathTool {  
    public static int sum(int... numbers) {  
        int sum = 0;  
        for(int number : numbers) {  
            sum += number;  
        }  
        return sum;  
    }  
}
```

```
System.out.println(MathTool.sum(1, 2));  
System.out.println(MathTool.sum(1, 2, 3));  
System.out.println(MathTool.sum(1, 2, 3, 4));
```

不定長度引數

- 實際上不定長度引數是編譯器蜜糖

```
public static transient int sum(int ai[]) {  
    int i = 0;  
    int ail[] = ai;  
    int j = ail.length;  
    for(int k = 0; k < j; k++) {  
        int l = ail[k];  
        i += l;  
    }  
    return i;  
}
```

不定長度引數

- `System.out.println(MathTool.sum(1, 2, 3))`，展開後也是變為陣列：

```
System.out.println(  
    sum(new int[] {1, 2, 3})  
);
```

不定長度引數

- 使用不定長度引數時，方法上宣告的不定長度參數必須是參數列最後一個
- 使用兩個以上不定長度引數也是不合法的

內部類別

- 可以在類別中再定義類別，稱之為內部類別（Inner class）

```
class Some {  
    class Other {  
    }  
}
```

```
class Some {  
    private class Other {  
    }  
}
```

```
class Some {  
    static class Other {  
    }  
}
```

內部類別

- 一個被宣告為`static`的內部類別，通常是將外部類別當作名稱空間

```
Some.Other o = new Some.Other();
```

- 可以存取外部類別`static`成員，但不可存取外部類別非`static`成員

```
class Some {  
    static int x;  
    int y;  
    static class Other {  
        void doOther() {  
            System.out.println(x);  
            System.out.println(y);  
        }  
    }  
}
```

non-static variable y cannot be referenced from a static context

(Alt-Enter shows hints)

內部類別

- 方法中也可以宣告類別

```
class Some {  
    public void doSome () {  
        class Other {  
        }  
    }  
}
```


內部類別

- 實務上比較少看到在方法中定義具名的內部類別，倒很常看到方法中定義匿名內部類別（ Anonymous inner class ）並直接實例化

```
Object o = new Object() {  
    public String toString() {  
        return "無聊的語法示範而已";  
    }  
};
```

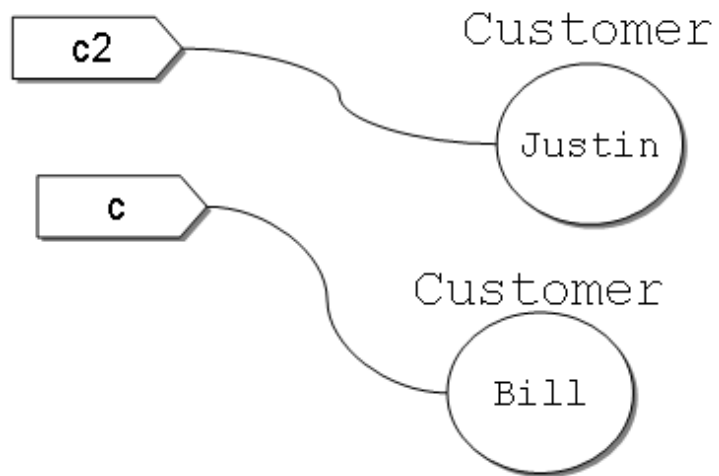
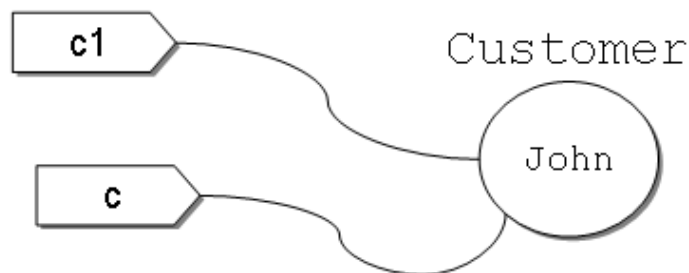
傳值呼叫

```
class Customer {  
    String name;  
    Customer(String name) {  
        this.name = name;  
    }  
}
```

```
public class CallByValue {  
    public static void some(Customer c) {  
        c.name = "John"; ← ❶ 改變了哪個物件？  
    }
```

```
    public static void other(Customer c) {  
        c = new Customer("Bill"); ← ❷ c 參考了哪個物件？  
    }
```

```
    public static void main(String[] args) {  
        Customer c1 = new Customer("Justin");  
        some(c1); ← ❸ c1 參考的物件會被改變嗎？  
        System.out.println(c1.name);  
  
        Customer c2 = new Customer("Justin");  
        other(c2); ← ❹ c2 參考的物件不會被改變嗎？  
        System.out.println(c2.name);  
    }  
}
```



傳值呼叫

- 如果由方法中傳回物件，並指定給變數，也是這種行為

```
public Customer create (String name) {  
    Customer c = new Customer(name);  
    ...  
    return c;  
}
```

```
public void doService() {  
    Customer customer = create("Irene");  
    ...  
}
```