

# Java<sup>SE 7</sup>

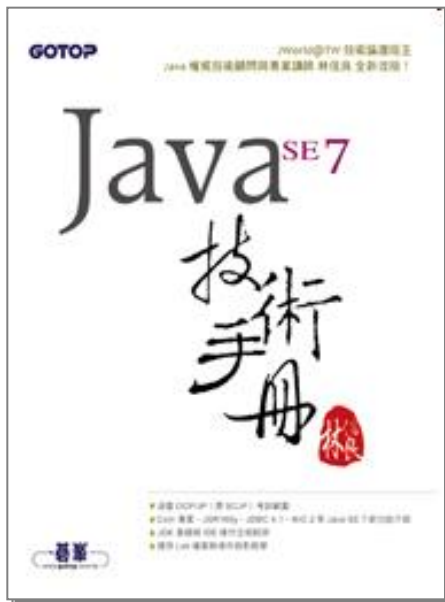
## 技術手冊

林信良

- 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 6

## • 繼承與多型



### 學習目標

- 瞭解繼承目的
- 瞭解繼承與多型的關係
- 知道如何重新定義方法
- 認識 `java.lang.Object`
- 簡介垃圾收集機制

# 繼承共同行為

- 假設你在正開發一款RPG ( Role-playing game ) 遊戲，一開始設定的角色有劍士與魔法師...

```
public class SwordsMan {
    private String name;    // 角色名稱
    private int level;      // 角色等級
    private int blood;      // 角色血量

    public void fight() {
        System.out.println("揮劍攻擊");
    }

    public int getBlood() {
        return blood;
    }

    public void setBlood(int blood) {
        this.blood = blood;
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
public class Magician {
    private String name;    // 角色名稱
    private int level;      // 角色等級
    private int blood;      // 角色血量

    public void fight() {
        System.out.println("魔法攻擊");
    }

    public void cure() {
        System.out.println("魔法治療");
    }

    public int getBlood() {
        return blood;
    }

    public void setBlood(int blood) {
        this.blood = blood;
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

## 繼承共同行為

- 如果要改進，可以把相同的程式碼提昇（ Pull up ）為父類別...

```
public class Role {
    private String name;
    private int level;
    private int blood;

    public int getBlood() {
        return blood;
    }

    public void setBlood(int blood) {
        this.blood = blood;
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

# 繼承共同行為

- 接著SwordsMan可以如下繼承Role：

```
public class SwordsMan extends Role {  
    public void fight() {  
        System.out.println("揮劍攻擊");  
    }  
}
```

## 繼承共同行為

- Magician也可以如下定義繼承Role類別：

```
public class Magician extends Role {  
    public void fight() {  
        System.out.println("魔法攻擊");  
    }  
  
    public void cure() {  
        System.out.println("魔法治療");  
    }  
}
```



# 繼承共同行為

- 如何看出確實有繼承了呢？

```
SwordsMan swordsMan = new SwordsMan();  
swordsMan.setName("Justin");  
swordsMan.setLevel(1);  
swordsMan.setBlood(200);  
System.out.printf("劍士：(%s, %d, %d)%n", swordsMan.getName(),  
    swordsMan.getLevel(), swordsMan.getBlood());
```

```
Magician magician = new Magician();  
magician.setName("Monica");  
magician.setLevel(1);  
magician.setBlood(100);  
System.out.printf("魔法師：(%s, %d, %d)%n", magician.getName(),  
    magician.getLevel(), magician.getBlood());
```

## 多型與is-a

- 子類別只能繼承一個父類別
- 繼承可避免類別間重複的行為定義
- 子類別與父類別間會有is-a的關係
  - **SwordsMan**是一種**Role** ( **SwordsMan** is a **Role** )
  - **Magician**是一種**Role** ( **Magician** is a **Role** )

## 多型與is-a

- 要開始理解多型（ Polymorphism ），必須先知道你操作的物件是「哪一種」東西
- 可以通過編譯：

```
SwordsMan swordsMan = new SwordsMan();  
Magician magician = new Magician();
```

## 多型與is-a

- 以下的程式片段也可以通過編譯？

```
Role role1 = new SwordsMan();  
Role role2 = new Magician();
```

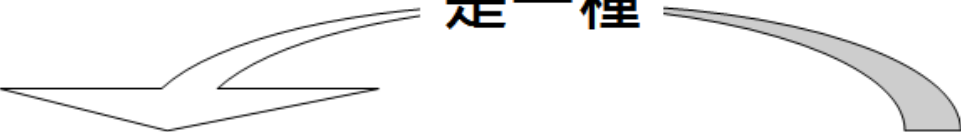
- 以下的程式片段為何無法通過編譯呢？

```
SwordsMan swordsMan = new Role();  
Magician magician = new Role();
```

## 多型與is-a

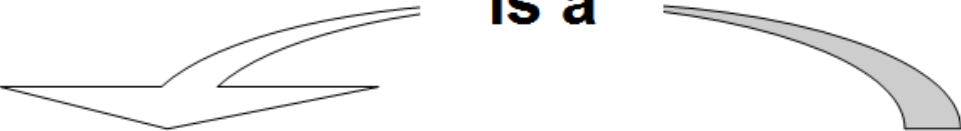
- 將自己當作編譯器，從=號右邊往左讀：右邊是不是一種左邊呢（右邊型態是不是左邊型態的子類別）？

是一種



```
Role role1 = new SwordsMan();
```

is a



```
Role role2 = new Magician();
```

## 多型與is-a

- 以下編譯失敗：

```
SwordsMan swordsMan = new Role(); // Role 是不是一種 SwordsMan?  
Magician magician = new Role(); // Role 是不是一種 Magician?
```

- 以下的程式片段是否可以通過編譯：

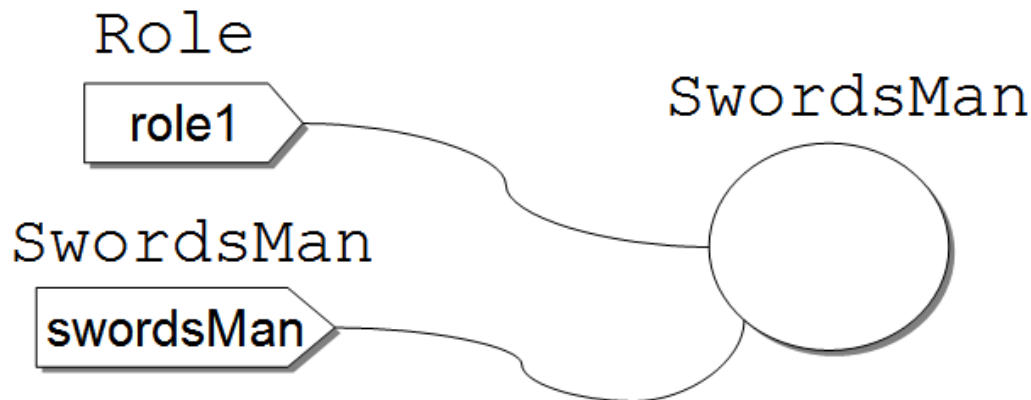
```
Role role1 = new SwordsMan();  
SwordsMan swordsMan = role1;
```

## 多型與is-a

- 如果你不想要編譯器囉嗦，可以叫它住嘴：

```
Role role1 = new SwordsMan();  
SwordsMan swordsMan = (SwordsMan) role1;
```

- 執行時期並不會出錯

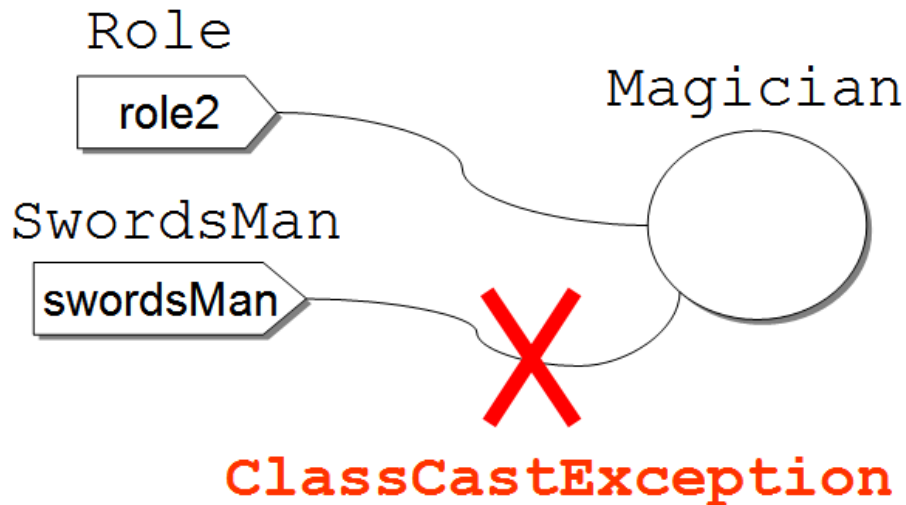


## 多型與is-a

- 以下的程式片段編譯可以成功：

```
Role role2 = new Magician();  
SwordsMan swordsMan = (SwordsMan) role2;
```

- 執行時期發生錯誤





## 多型與is-a

- 以下編譯成功，執行也沒問題：

```
SwordsMan swordsMan = new SwordsMan();  
Role role = swordsMan;    // SwordsMan 是一種 Role
```

- 以下程式片段會編譯失敗：

```
SwordsMan swordsMan = new SwordsMan();  
Role role = swordsMan;    // SwordsMan 是一種 Role，這行通過編譯  
SwordsMan swordsMan = role;    // Role 不一定是一種 SwordsMan，編譯失敗
```

## 多型與is-a

- 以下程式片段編譯成功，執行時也沒問題：

```
SwordsMan swordsMan = new SwordsMan();  
Role role = swordsMan;          // SwordsMan 是一種 Role，這行通過編譯  
// 你告訴編譯器要讓 Role 扮演 SwordsMan，以下這行通過編譯  
SwordsMan swordsMan = (SwordsMan) role; // role 參考 SwordsMan 實例，執行成功
```

- 以下程式片段編譯成功，但執行時拋出  
ClassCastException：

```
SwordsMan swordsMan = new SwordsMan();  
Role role = swordsMan;          // SwordsMan 是一種 Role，這行通過編譯  
// 你告訴編譯器要讓 Role 扮演 Magician，以下這行通過編譯  
Magician magician = (Magician) role; // role 參考 SwordsMan 實例，執行失敗
```

## 多型與is-a

- 好像只是在玩弄語法？
- 設計static方法，顯示所有角色的血量 ...

```
public static void showBlood(SwordsMan swordsMan) {  
    System.out.printf("%s 血量 %d%n",  
        swordsMan.getName(), swordsMan.getBlood());  
}  
  
public static void showBlood(Magician magician) {  
    System.out.printf("%s 血量 %d%n",  
        magician.getName(), magician.getBlood());  
}
```

# 多型與is-a

- 重載方法的運用

```
showBlood(swordsMan);    // swordsMan 是 SwordsMan 型態  
showBlood(magician);     // magician 是 Magician 型態
```

- 如果有一百個角色呢？重載出一百個方法？

# 多型與is-a

- 這些角色都是一種Role

```
public static void showBlood(Role role) { ← ❶ 宣告為 Role 型態
    System.out.printf("%s 血量 %d\n",
        role.getName(), role.getBlood());
}
```

```
public static void main(String[] args) {
    SwordsMan swordsMan = new SwordsMan();
    swordsMan.setName("Justin");
    swordsMan.setLevel(1);
    swordsMan.setBlood(200);

    Magician magician = new Magician();
    magician.setName("Monica");
    magician.setLevel(1);
    magician.setBlood(100);

    showBlood(swordsMan); ← ❷ SwordsMan 是一種 Role
    showBlood(magician); ← ❸ magician 是一種 Role
}
```

## 多型與is-a

- 什麼叫多型？以抽象講法解釋，就是使用單一介面操作多種型態的物件！
- 若用以上的範例來理解，在showBlood()方法中，既可以透過Role型態操作SwordsMan物件，也可以透過Role型態操作Magician物件。

## 重新定義行為

- 請設計static方法，可以播放角色攻擊動畫 ...

```
public static void drawFight(Role role) {  
    cannot find symbol  
      symbol: method fight()  
    location: variable role of type cc.openhome.Role  
    ----  
System. (Alt-Enter shows hints)  
    role.fight();  
}
```

## 重新定義行為

- 對drawFight()方法而言，只知道傳進來的會是一種Role物件，所以編譯器也只能檢查你呼叫的方法，Role是不是有定義
- 仔細觀察一下SwordsMan與Magician的fight()方法的方法簽署（**method signature**）...

```
public void fight()
```



# 重新定義行為

- 將 `fight()` 方法提昇至 `Role` 類別中定義：

```
public class Role {  
    ...略  
    public void fight() {  
        // 子類別要重新定義 fight() 的實際行為  
    }  
}  
  
public class SwordsMan extends Role {  
    public void fight() {  
        System.out.println("揮劍攻擊");  
    }  
}  
  
public class Magician extends Role {  
    public void fight() {  
        System.out.println("魔法攻擊");  
    }  
    ...略  
}
```

# 重新定義行為

```
public static void drawFight(Role role) { ← ❶ 宣告為 Role 型態
    System.out.print(role.getName());
    role.fight();
}

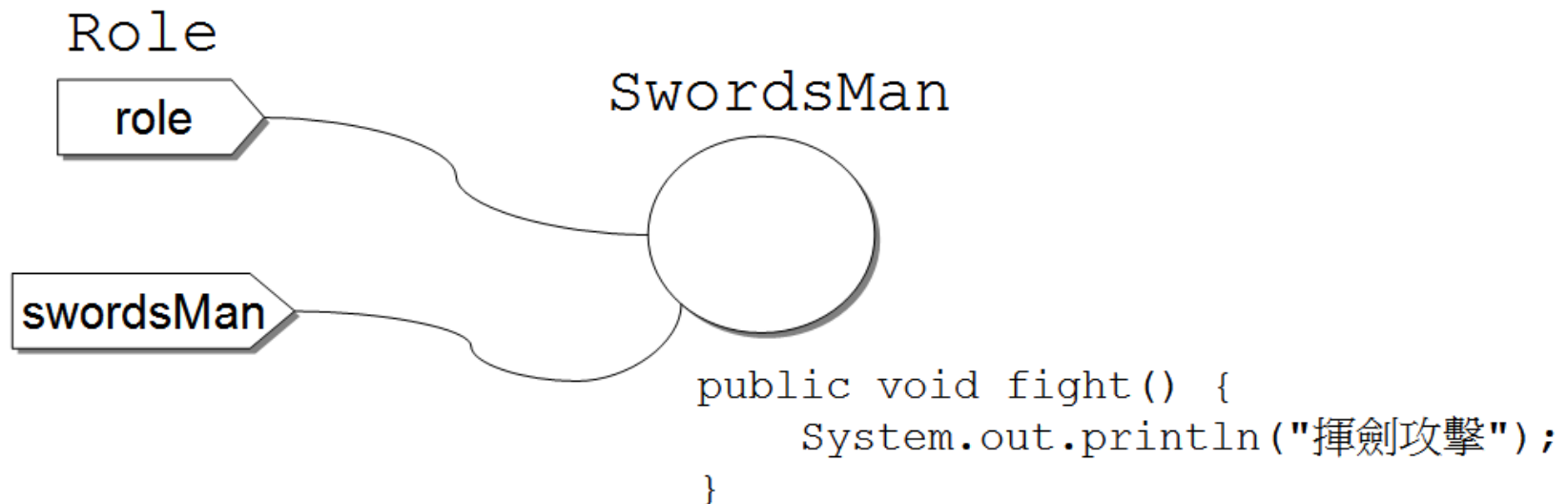
public static void main(String[] args) {
    SwordsMan swordsMan = new SwordsMan();
    swordsMan.setName("Justin");
    swordsMan.setLevel(1);
    swordsMan.setBlood(200);

    Magician magician = new Magician();
    magician.setName("Monica");
    magician.setLevel(1);
    magician.setBlood(100);

    drawFight(swordsMan); ← ❷ 實際操作的是 SwordsMan 實例
    drawFight(magician); ← ❸ 實際操作的是 Magician 實例
}
```

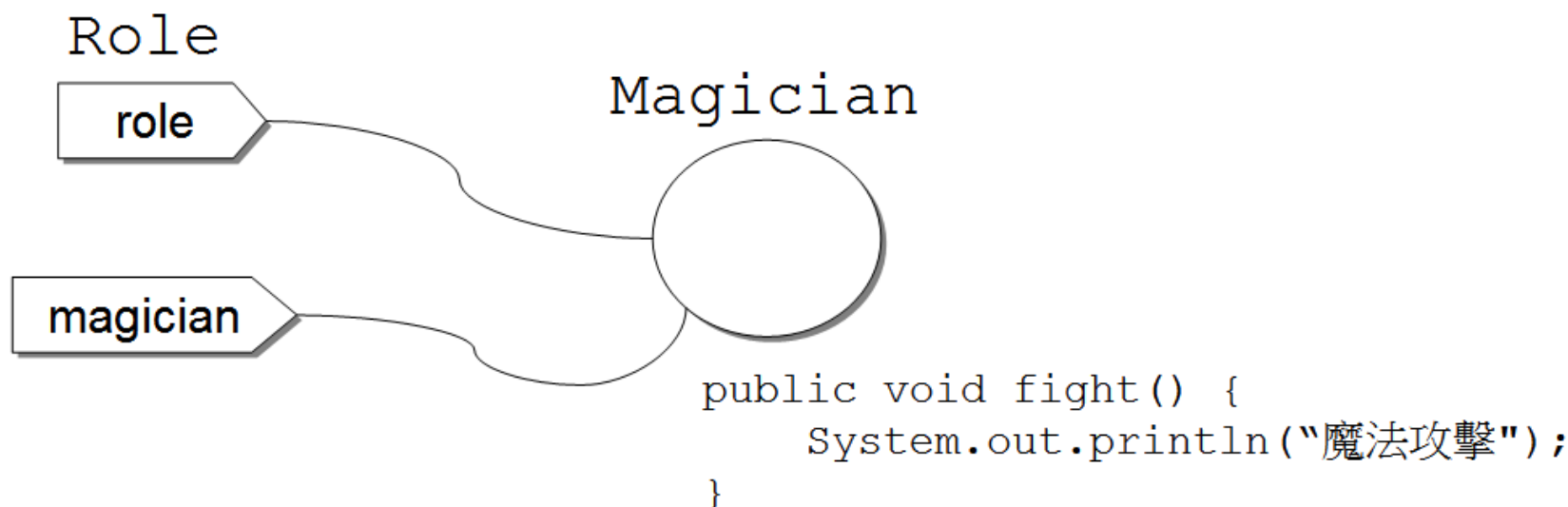
# 重新定義行為

- 如果傳入 `fight()` 的是 `SwordsMan`，`role` 參數參考的就是 `SwordsMan` 實例，操作的就是 `SwordsMan` 上的方法定義：



## 重新定義行為

- 如果傳入 `fight()` 的是 `Magician`，`role` 參數參考的就是 `Magician` 實例，操作的就是 `Magician` 上的方法定義：



# 重新定義行為

- 在重新定義父類別中某個方法時，子類別必須撰寫與父類別方法中相同的簽署
- 如果疏忽打錯字了...

```
public class SwordsMan extends Role {  
    public void Fight() {  
        System.out.println("揮劍攻擊");  
    }  
}
```

# 重新定義行為

- 在JDK5之後支援標註（ Annotation ）
- `@Override`要求編譯器檢查，該方法是不是真的重新定義了父類別中某個方法

```
public class SwordsMan extends Role {
```

method does not override or implement a method from a supertype

----

(Alt-Enter shows hints)

```
@Override
```

```
public void Fight() {
```

```
    System.out.println("揮劍攻擊");
```

```
}
```

```
}
```

# 抽象方法、抽象類別

- 上一個範例中Role類別的定義中，fight()方法區塊中實際上沒有撰寫任何程式碼
- 沒有任何方式強迫或提示子類別一定要實作fight()方法

# 抽象方法、抽象類別

- 如果某方法區塊中真的沒有任何程式碼實作，可以使用**abstract**標示該方法為抽象方法（Abstract method）

```
public abstract class Role {  
    ...略  
    public abstract void fight();  
}
```

- 內含抽象方法的類別，一定要在**class**前標示**abstract**，如上例所示，這表示這是一個定義不完整的抽象類別（Abstract class）



# 抽象方法、抽象類別

- 如果嘗試用抽象類別建構實例，就會引發編譯錯誤：

```
cc.openhome.Role is abstract; cannot be instantiated
----
(Alt-Enter shows hints)
```

```
Role role = new Role();
```

# 抽象方法、抽象類別

- 子類別如果繼承抽象類別，對於抽象方法有兩種作法
  - 繼續標示該方法為`abstract`（該子類別因此也是個抽象類別，必須在`class`前標示`abstract`）
  - 實作抽象方法
- 兩個作法都沒實施，就會引發編譯錯誤：

```
public class SwordsMan extends Role {
```

```
cc.openhome.SwordsMan is not abstract and does not override abstract method fight() in
cc.openhome.Role
----
(Alt-Enter shows hints)
```

# protected成員

- 上一節的RPG遊戲來說，如果建立了一個角色，想顯示角色的細節，必須如下撰寫：

```
SwordsMan swordsMan = new SwordsMan();  
...略  
System.out.printf("劍士 (%s, %d, %d)%n", swordsMan.getName(),  
    swordsMan.getLevel(), swordsMan.getBlood());  
Magician magician = new Magician();  
...略  
System.out.printf("魔法師 (%s, %d, %d)%n", magician.getName(),  
    magician.getLevel(), magician.getBlood());
```

# protected成員

- 可以在SwordsMan或Magician上定義個toString()方法，傳回角色的字串描述：

```
public class SwordsMan extends Role {
    ...略
    public String toString() {
        return String.format("劍士 (%s, %d, %d)", this.getName(),
            this.getLevel(), this.getBlood());
    }
}

public class Magician extends Role {
    ...略
    public String toString() {
        return String.format("魔法師 (%s, %d, %d)", this.getName(),
            this.getLevel(), this.getBlood());
    }
}
```

# protected成員

- 客戶端就可以如下撰寫：

```
SwordsMan swordsMan = new SwordsMan();  
...略  
System.out.println(swordsMan.toString());  
Magician magician = new Magician();  
...略  
System.out.printf(magician.toString());
```

- 不過因為Role中的name、level與blood被定義為private，所以無法直接於子類別中存取，只能透過getName()、getLevel()、getBlood()來取得

# protected成員

- 只想讓子類別可以直接存取name、level與blood的話，可以定義它們為**protected**：

```
public abstract class Role {
    protected String name;
    protected int level;
    protected int blood;
    ...略
}

public class SwordsMan extends Role {
    ...略
    public String toString() {
        return String.format("劍士 (%s, %d, %d)", this.name,
                               this.level, this.blood);
    }
}
```

# protected成員

關鍵字	類別內部	相同套件類別	不同套件類別
public	可存取	可存取	可存取
protected	可存取	可存取	子類別可存取
無	可存取	可存取	不可存取
private	可存取	不可存取	不可存取

## 重新定義的細節

- 有時候重新定義方法時，並非完全不滿意父類別中的方法，只是希望在執行父類別中方法的前、後作點加工

```
public abstract class Role {  
    ...略  
    public String toString() {  
        return String.format("(%s, %d, %d)", this.name,  
                                this.level, this.blood);  
    }  
}
```



## 重新定義的細節

- 如果想取得父類別中的方法定義，可以於呼叫方法前，加上**super**關鍵字

```
public class SwordsMan extends Role {  
    ...略  
    @Override  
    public String toString() {  
        return "劍士 " + super.toString();  
    }  
}
```

## 重新定義的細節

- 可以使用`super`關鍵字呼叫的父類別方法，不能定義為`private`
- 對於父類別中的方法權限，只能擴大但不能縮小
  - 若原來成員`public`，子類別中重新定義時不可為`private`或`protected`

```
public class SwordsMan extends Role {  
    protected void fight() {  
        System.out.println("fighting");  
    }  
}
```

fight() in cc.openhome.SwordsMan cannot override fight() in cc.openhome.Role attempting to assign weaker access privileges; was public  
----  
(Alt-Enter shows hints)

# 重新定義的細節

- 在JDK5之前...

```
public class Bird {  
    protected String name;  
    public Bird(String name) {  
        this.name = name;  
    }  
    public Bird copy() {  
        return new Bird(name);  
    }  
}
```

```
public class Chicken extends Bird {
```

```
    public Chicken copy() {
```

```
        Chicken
```

```
        chicken
```

```
        return
```

```
    }
```

```
}
```

copy() in cc.openhome.Chicken cannot override copy() in cc.openhome.Bird  
return type cc.openhome.Chicken is not compatible with cc.openhome.Bird  
----  
(Alt-Enter shows hints)

## 重新定義的細節

- 在JDK5之後，重新定義方法時，如果返回型態是父類別中方法返回型態的子類別，也是可以通过編譯的，圖6.11的例子，在JDK5中並不會出現編譯錯誤

## 再看建構式

- 在建構子類別實例後，會先進行父類別定義的初始流程，再進行子類別中定義的初始流程
- 也就是建構子類別實例後，會先執行父類別建構式定義的流程，再執行子類別建構式定義的流程

## 再看建構式

- 如果子類別建構式中沒有指定執行父類別中哪個建構式，預設會呼叫父類別中無參數建構式

```
class Some {  
    Some() {  
        System.out.println("呼叫 Some()");  
    }  
}  
  
class Other extends Some {  
    Other() {  
        System.out.println("呼叫 Other()");  
    }  
}
```

## 再看建構式

- 如果想執行父類別中某建構式，可以使用 **super()** 指定：

```
class Some {
    Some() {
        System.out.println("呼叫 Some()");
    }
    Some(int i) {
        System.out.println("呼叫 Some(int i)");
    }
}
class Other extends Some {
    Other() {
        super(10);
        System.out.println("呼叫 Other()");
    }
}
```

## 再看建構式

- 當你這麼撰寫時：

```
class Some {  
    Some() {  
        System.out.println("呼叫 Some()");  
    }  
}  
class Other extends Some {  
    Other() {  
        System.out.println("呼叫 Other()");  
    }  
}
```



## 再看建構式

- 等於你這麼撰寫：

```
class Some {  
    Some() {  
        System.out.println("呼叫 Some()");  
    }  
}  
class Other extends Some {  
    Other() {  
        super();  
        System.out.println("呼叫 Other()");  
    }  
}
```

## 再看建構式

- 知道以下為什麼會編譯錯誤嗎？

```
class Some {  
    Some(int i) {  
        System.out.println("呼叫Some(int 10)");  
    }  
}
```

constructor Some in class cc.openhome.Some cannot be applied to given types;  
required: int  
found: no arguments  
reason: actual and formal argument lists differ in length  
----

```
class Other {  
    Other() {  
        System.out.println("呼叫Other()");  
    }  
}
```

Other()

## 再看final關鍵字

- 如果在指定變數值之後，就不想再改變變數值，可以在宣告變數時加上final限定
- 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，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

# 再看final關鍵字

- 如果**class**前使用了**final**關鍵字定義，那麼表示這個類別是最後一個了，不會再有子類別，也就是不能被繼承
  - String在定義時就限定為final

## Class String

java.lang.Object  
    java.lang.String

### All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

---

```
public final class String
    extends Object
    implements Serializable, Comparable<String>, CharSequence
```

## 再看final關鍵字

- 打算繼承final類別，則會發生編譯錯誤：

```
cannot inherit from final java.lang.String
----
(Alt-Enter shows hints)
```

```
class IterableString extends String {
```

# 再看final關鍵字

- 定義方法時，也可以限定該方法為**final**，這表示最後一次定義方法了，也就是子類別不可以重新定義**final**方法
  - java.lang.Object上有幾個final方法

## notify

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.

The awakened thread will not be able to proceed until the current thread relinquishes

## 再看final關鍵字

- 如果你嘗試在繼承父類別後，重新定義final方法，則會發生編譯錯誤：

```
class Some extends Object {
```

```
    notify() in cc.openhome.Some cannot override notify() in java.lang.Object  
    overridden method is final
```

```
----
```

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

```
public void notify() {  
    // ...  
}
```

# java.lang.Object

- 定義類別時沒有使用 `extends` 關鍵字指定繼承任何類別，則繼承 **java.lang.Object**

```
public class Some {  
    ...  
}
```

```
public class Some extends Object {  
    ...  
}
```



# java.lang.Object

- Java中所有物件，一定「是一種」Object

```
Object o1 = "Justin";  
Object o2 = new Date();
```

- 如果有個需求是使用陣列收集各種物件，那該宣告為什麼型態呢？答案是Object[]！

```
Object[] objs = {"Monica", new Date(), new SwordsMan()};  
String name = (String) objs[0];  
Date date = (Date) objs[1];  
SwordsMan swordsMan = (SwordsMan) objs[2];
```

# java.lang.Object

- 以下定義的ArrayList類別，可以不限長度地收集物件：

```
public class ArrayList {  
    private Object[] list; ← ❶ 使用 Object 陣列收集  
    private int next; ← ❷ 下一個可儲存物件的索引  
  
    public ArrayList(int capacity) { ← ❸ 指定初始容量  
        list = new Object[capacity];  
    }  
  
    public ArrayList() {  
        this(16); ← ❹ 初始容量預設為 16  
    }  
  
    public void add(Object o) { ← ❺ 收集物件方法  
        if(next == list.length) { ← ❻ 自動增長 Object 陣列長度  
            list = Arrays.copyOf(list, list.length * 2);  
        }  
        list[next++] = o;  
    }  
}
```

# java.lang.Object

```
public Object get(int index) { ← ⑦ 依索引取得收集之物件  
    return list[index];  
}
```

```
public int size() { ← ⑧ 已收集的物件個數  
    return next;  
}
```

```
}
```

# java.lang.Object

```
ArrayList list = new ArrayList();
Scanner scanner = new Scanner(System.in);
String name;
while(true) {
    System.out.print("訪客名稱：");
    name = scanner.nextLine();
    if(name.equals("quit")) {
        break;
    }
    list.add(name);
}
System.out.println("訪客名單：");
for(int i = 0; i < list.size(); i++) {
    String guest = (String) list.get(i);
    System.out.println(guest.toUpperCase());
}
```

# java.lang.Object

- java.lang.Object是所有類別的頂層父類別
- Object上定義的方法，所有物件都繼承下來了，只要不是被定義為final的方法，都可以重新定義

# java.lang.Object

## Methods

Modifier and Type	Method and Description
protected <b>Object</b>	<b>clone()</b> Creates and returns a copy of this object.
boolean	<b>equals(Object obj)</b> Indicates whether some other object is "equal to" this one.
protected void	<b>finalize()</b> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<b>Class</b> <?>	<b>getClass()</b> Returns the runtime class of this Object.
int	<b>hashCode()</b> Returns a hash code value for the object.
void	<b>notify()</b> Wakes up a single thread that is waiting on this object's monitor.
void	<b>notifyAll()</b> Wakes up all threads that are waiting on this object's monitor.
<b>String</b>	<b>toString()</b> Returns a string representation of the object.
void	<b>wait()</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object.
void	<b>wait(long timeout)</b> Causes the current thread to wait until either another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object, or a specified amount of time has elapsed.
void	<b>wait(long timeout, int nanos)</b> Causes the current thread to wait until another thread invokes the <b>notify()</b> method or the <b>notifyAll()</b> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

# java.lang.Object

- Object的toString() 預設定義為：

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

- 6.2.1的範例中，SwordsMan等類別，是重新定義了toString()
- 許多方法若傳入物件，預設都會呼叫toString()
  - 例如System.out.print() 等方法

# java.lang.Object

- 6.2.1的這個程式片段：

```
SwordsMan swordsMan = new SwordsMan();  
...略  
System.out.println(swordsMan.toString());  
Magician magician = new Magician();  
...略  
System.out.printf(magician.toString());  
  
SwordsMan swordsMan = new SwordsMan();  
...略  
System.out.println(swordsMan);  
Magician magician = new Magician();  
...略  
System.out.printf(magician);
```



# java.lang.Object

- 在Java中要比較兩個物件的實質相等性，並不是使用==，而是透過equals()方法
- equals()方法是Object類別就定義的方法

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

```
public class Cat {  
    ...  
    public boolean equals(Object other) {  
        // other 參考的就是這個物件，當然是同一物件  
        if (this == other) {  
            return true;  
        }  
        /* other 參考的物件是不是 Cat 建構出來的  
           例如若是 Dog 建構出來的當然就不用比了 */  
        if (!(other instanceof Cat)) {  
            return false;  
        }  
  
        Cat cat = (Cat) other;  
        // 定義如果名稱與生日，表示兩個物件實質上相等  
        if (!getName().equals(cat.getName())) {  
            return false;  
        }  
        if (!getBirthday().equals(cat.getBirthday())) {  
            return false;  
        }  
        return true;  
    }  
}
```

# java.lang.Object

- **instanceof** 運算子可以用來判斷物件是否由某個類別建構，左運算元是物件，右運算元是類別
- 編譯器會檢查左運算元型態是否在右運算元型態的繼承架構

```
inconvertible types  
  required: java.util.Date  
  found:   java.lang.String  
-----  
(Alt-Enter shows hints)
```

```
boolean isDate = ["Justin" instanceof java.util.Date;
```

# java.lang.Object

- 執行時期，並非只有左運算元物件為右運算元類別直接實例化才傳回true，只要左運算元型態是右運算元型態的子類型，instanceof也是傳回true

# 關於垃圾收集

- JVM有垃圾收集（ Garbage Collection, GC ）機制，收集到的垃圾物件所佔據的記憶體空間，會被垃圾收集器釋放
- 執行流程中，無法透過變數參考的物件，就是GC認定的垃圾物件

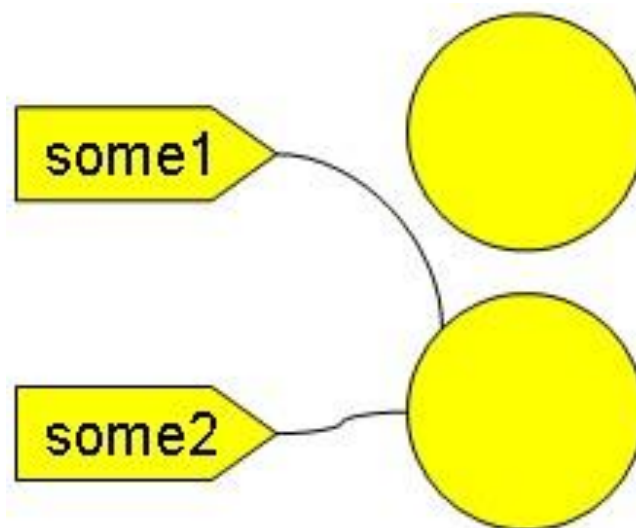
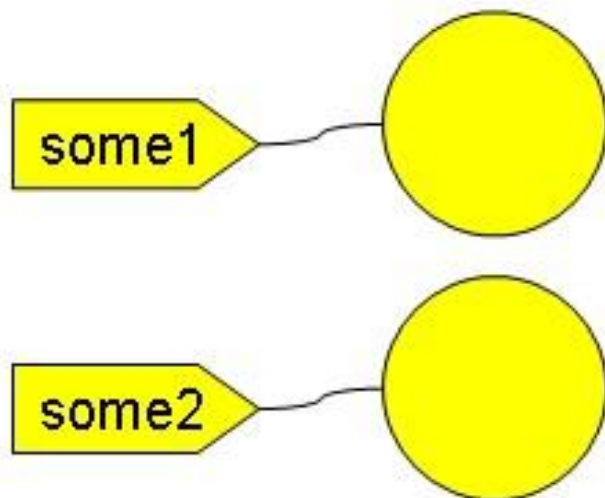
# 關於垃圾收集

- 假設你有一個類別：

```
public class Some {  
    Some next;  
}
```

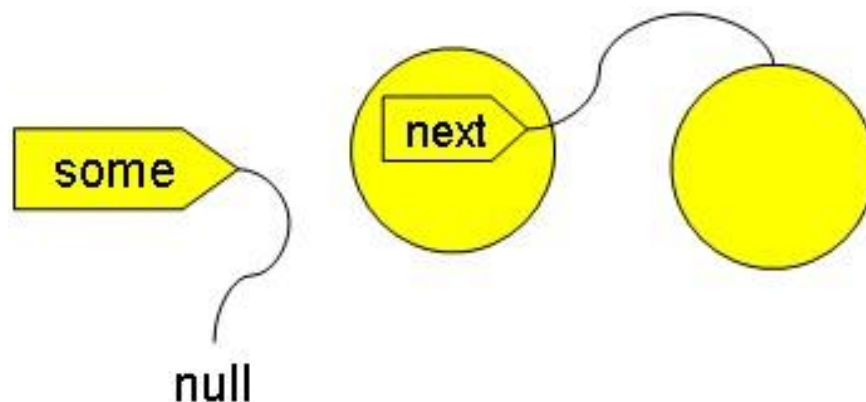
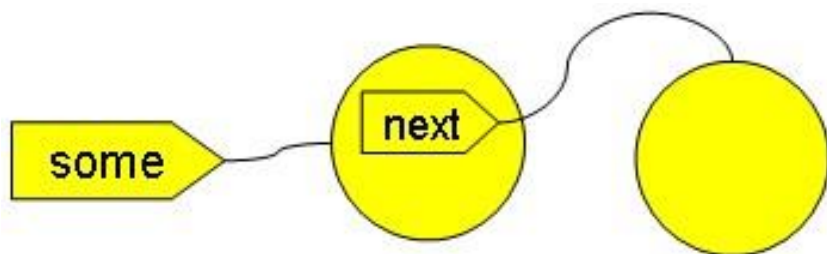
# 關於垃圾收集

```
Some some1 = new Some();  
Some some2 = new Some();  
Some some1 = some2;
```



# 關於垃圾收集

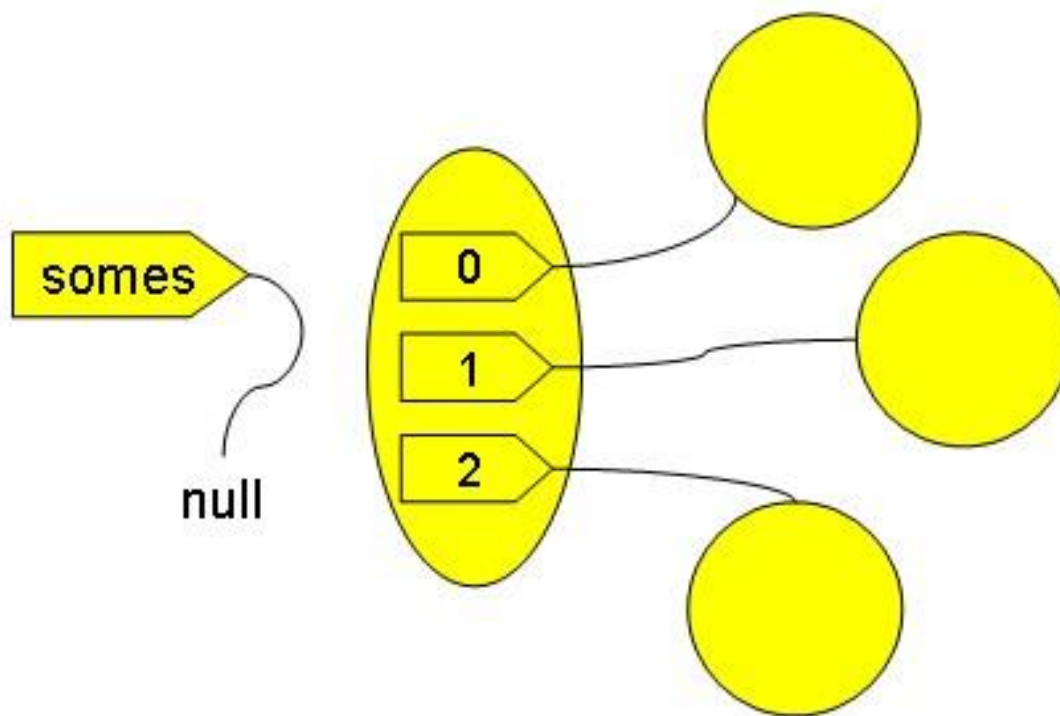
```
Some some = new Some();  
some.next = new Some();  
some = null;
```





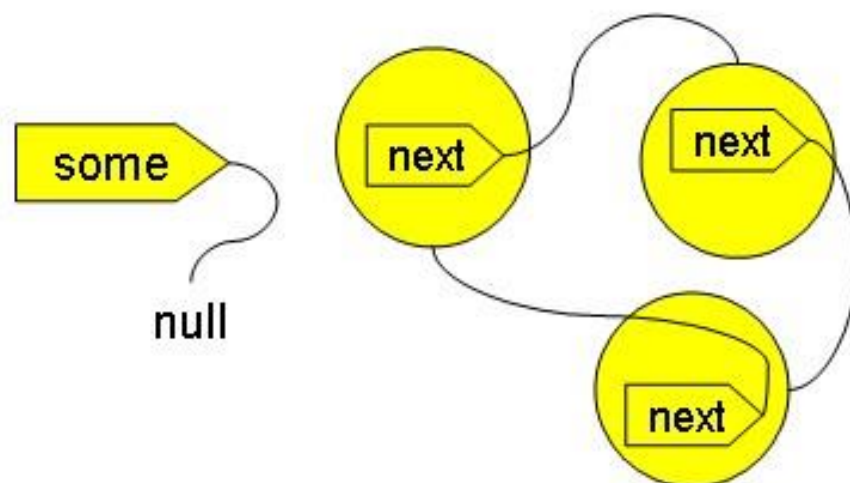
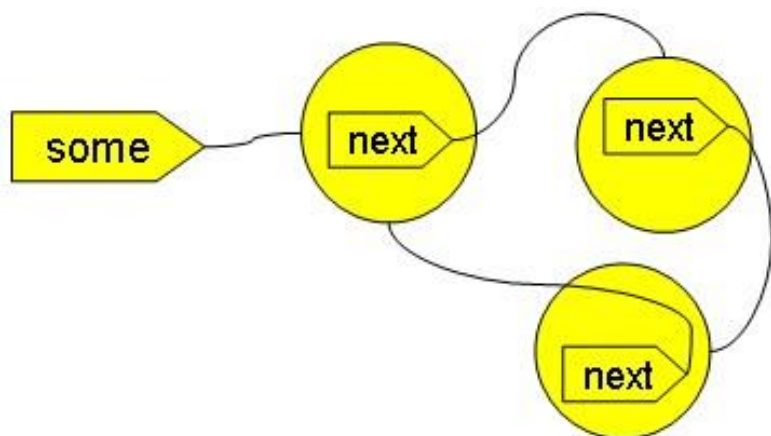
# 關於垃圾收集

```
Some[] somes = {new Some(), new Some(), new Some};  
somes = null;
```



# 關於垃圾收集

```
Some some = new Some();  
some.next = new Some();  
some.next.next = new Some();  
some.next.next.next = some;  
some = null;
```



## 再看抽象類別

- 開發一個猜數字遊戲 ...

```
public class Guess {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        int number = (int) (Math.random() * 10);  
        int guess;  
        do {  
            System.out.print("輸入數字：");  
            guess = scanner.nextInt();  
        } while(guess != number);  
        System.out.println("猜中了");  
    }  
}
```

- 老闆皺著眉頭說：「我有說要在文字模式下執行這個遊戲嗎？」

## 再看抽象類別

```
public abstract class GuessGame {  
    public void go() {  
        int number = (int) (Math.random() * 10);  
        int guess;  
        do {  
            print("輸入數字:");  
            guess = nextInt();  
        } while(guess != number);  
        println("猜中了");  
    }  
  
    public abstract void print(String text);  
    public abstract void println(String text);  
    public abstract int nextInt();  
}
```

## 再看抽象類別

```
public class ConsoleGame extends GuessGame {  
    private Scanner scanner = new Scanner(System.in);  
  
    @Override  
    public void print(String text) {  
        System.out.print(text);  
    }  
  
    @Override  
    public void println(String text) {  
        System.out.println(text);  
    }  
  
    @Override  
    public int nextInt() {  
        return scanner.nextInt();  
    }  
}
```

## 再看抽象類別

```
GuessGame game = new ConsoleGame();  
game.go();
```