序

圖靈獎得主 Dijkstra 在 2001 年寫信[1]給德州大學預算委員會，希望大學的程式設計入門課程中，不要使用命令式的 Java 取代函數式的 Haskell；2001 年前後是 Java 2 的年代，Java 1.3 推出沒多久，無怪乎 Dijkstra 會覺得 Java 看來就像個商業宣傳，完全可以想像 Dijkstra 為何對此感到不安。

然而從 Java 8 開始，Java 持續、穩建地納入、增強函數式概念的語法、API 等元素，跟上了其他具有一級函式特性語言的腳步，有了一些高階流程抽象可以使用，甚至在 Java 16、17，開始納入模式比對、`record`、`sealed` 類別，這些特性其實對應的是函數式中更為基礎的元素「代數資料型態（Algebraic data type）」，這讓開發者除了命令式的選擇之外，能更便利地基於函數式典範來思考與實作。

我本身也是函數式典範的愛好者，就我而言，命令式與函數式就是不同的思考方式，只不過人很容易受到第一次接觸的東西影響，甚至養成習慣，初入程式設計領域之人，若一開始是接受命令式的訓練，日後就會習慣用命令式來解決問題，若一開始是接受函數式的訓練，看到命令式 x = x + 1，往往也會難以接受。

這也是 Dijkstra 在信中談到的，我們會被使用的工具形塑，就程式語言這工具來說，影響更為深遠，因為它們形塑的是思考習慣！

就現實而言，大部分開發者確實是從命令式的訓練開始，這一方面是因為許多主流語言是命令式，另一方面，也代表著許多需求適合使用命令式解決；然而命令式與函數式各有其應用的情境，現今 Java 納入了越來越多的函數式元素，其實也是代表著今日應用程式想解決的問題領域，越來越多適合使用函數式來解決。

---

[1] To the members of the Budget Council：bit.ly/3HQhU29

如方才所言，命令式與函數式就是不同思考方式，身為一名開發者，其實可使用的思考方式越多，面對問題時可用的工具就越多，也就越有辦法從中選擇適用的方案。

當然，學習需要付出成本，不過我更傾向於認為這是一種投資，身為開發者在可用的思考等方面投資越多，能解決的問題就越廣越深，個人積累也益發深厚，估且不要說什麼提高自身價值或不可取代性，就根本而言，這是在尊重自己從事的領域。

畢竟程式設計領域，本身就是個需要不斷培養思考方式的領域，本來就是個需要不斷積累的領域，如果開發者吝於在學習上投資，懶得做好積累的功夫，不就是在貶低自身從事的工作？那麼又何必進入這個領域呢？

2022.04

導讀

這份導讀可讓你更了解如何使用本書。

新舊版差異

介紹 JDK15 至 17 的新特性，當然是本書的改版重點之一。4.4.3 介紹了 Java 15 的文字區塊（Text block），6.2.5 在談 `instanceof` 時也談到了 Java 16 的模式比對（Pattern matching），9.1.3 與 18.3.1 說明 Java 16 的 `record` 類別，18.3.2 討論 Java 17 的 `sealed` 類別。

改版的重點之二是除舊，一些過時或不需要再詳談的內容經過簡化或移除，像是舊版中第 1 章的 JDK 歷史、第 5 章的傳值呼叫、第 11 章的 `ForkJoinPool`、第 13 章的 `Data` 與 `Calendar` 說明、第 15 章的國際化、第 16 章的 `RowSet` 等。

除舊的另一目的是瘦身，讓新特性有足夠的篇幅說明，另外，15.2 增加了 Java 11 的 HTTP Client API 說明，11.1.5 增加了 `java.util.concurrent.atomic` 的說明。

至於書本的內容，照慣例整本都重新審閱了一次，在範例的部分也會適當地使用新特性，例如可以使用文字區塊簡化字串模版的部分就會使用，若類別實際上是資料載體的概念，就採用 `record` 類別等。

自 Java SE 12 開始，有些 Java 新功能未正式定案前，為了取得開發者的意見回饋，會以預覽形式發佈；由於預覽功能未來仍可能變動規格，本書不會說明預覽功能。

由於 Java SE 17 開始，LTS 的釋出週期，加速為兩年一次，未來書籍的改版時機，也將基於 LTS 的版本。

對話框

本書會出現以下的對話框：

提示 »» 針對課程中提到的觀念，提供一些額外資源或思考方向，暫時忽略這些提示對課程進行沒有影響，然而有時間的話，針對這些提示多做思考或討論是有幫助的。

注意 »» 針對課程中提到的觀念，以對話框方式特別呈現出必須注意的使用方式、陷阱或避開問題的方法，看到這個對話框時請集中精神閱讀。

附錄

範例檔案中包括本書中全部範例，提供 Eclipse 範例專案，附錄 A 說明如何使用這些範例專案。

聯繫作者

若有堪誤回報等相關書籍問題，可透過網站與作者聯繫：

- openhome.cc

繼承與多型

6

CHAPTER

學習目標

- 認識繼承目的
- 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

6.1 何謂繼承？

物件導向中，子類別繼承（Inherit）父類別，可避免重複定義行為與實作，然而並非想避免重複定義行為與實作時就使用繼承，濫用繼承而導致程式維護上的問題時有所聞，如何正確判斷繼承的時機，以及繼承後如何活用多型，才是學習繼承時的重點。

6.1.1 繼承共同行為與實作

多個類別間若重複定義了相同的行為與實作，可運用繼承來重構。以實際範例說明比較清楚，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。首先你定義了劍士類別：

```
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;
    }
}
```

接著為魔法師定義類別：

```
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;
    }
}
```

你注意到什麼呢？只要是遊戲中的角色，都會具有角色名稱、等級與血量，類別中也為名稱、等級與血量定義了取值方法與設值方法，Magician 中粗體字部分與 SwordsMan 中對應的程式碼重複了。**重複在程式設計上，就是不好的訊號**。舉例來說，若要將 name、level、blood 改為其他名稱，就要修改 SwordsMan 與 Magician 兩個類別，如果有更多類別具有重複的程式碼，就要修改更多類別，造成維護上的不便。

如果要改進，可以把相同的程式碼提昇（Pull up）為父類別：

Game1 Role.java

```
package cc.openhome;

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;
    }
}
```

這個類別在定義上沒什麼特別的新語法，只不過是將 SwordsMan 與 Magician 中重複的程式碼複製過來。接著 SwordsMan 可以如下繼承 Role：

Game1_SwordsMan.java

```
package cc.openhome;

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

在這邊看到了新的關鍵字 **extends**，這表示 SwordsMan 會擴充 Role 的行為與實作，也就是繼承 Role 的行為與實作，再擴充 Role 原本沒有的 fight()行為與實作。程式面上來說，Role 中有定義的程式碼，SwordsMan 都繼承而擁有了，並再定義了 fight()方法的程式碼。類似地，Magician 也可以如下定義繼承 Role 類別：

Game1_Magician.java

```
package cc.openhome;

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");
    }
}
```

Magician 繼承 Role 的行為與實作，再擴充了 Role 原本沒有的 fight()與 cure()行為與實作。

如何看出確實有繼承了呢？以下簡單的程式可以看出：

Game1_RPG.java

```
package cc.openhome;

public class RPG {
    public static void main(String[] args) {
        demoSwordsMan();
        demoMagician();
```

```
    }

    static void demoSwordsMan() {
        var 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());
    }

    static void demoMagician() {
        var 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());
    }
}
```

雖然 SwordsMan 與 Magician 沒有定義 getName()、getLevel()與 getBlood() 等方法，但從 Role 繼承了這些方法，也就如範例中可以直接使用，執行的結果如下：

```
劍士：(Justin, 1, 200)
魔法師：(Monica, 1, 100)
```

若要將 name、level、blood 改為其他名稱，就只要修改 Role.java，只要是繼承 Role 的子類別都無需修改。

注意 »»» 有的書籍或文件會說，private 成員無法繼承，那是錯的！如果 private 成員無法繼承，那為什麼上面的範例 name、level、blood 記錄的值會顯示出來呢？private 成員會被繼承，只不過子類別無法直接存取，必須透過父類別提供的方法來存取（如果父類別願意提供存取方法的話）。

6.1.2　多型與 is-a

在 Java 中，**子類別只能繼承一個父類別**，繼承除了可避免類別間重複的行為與實作定義外，還有個重要的關係，**子類別與父類別間會有 is-a 的關係，中文稱為「是一種」的關係**，這是什麼意思？以先前範例來說，SwordsMan 繼承了 Role，

SwordsMan 是一種 Role（SwordsMan is a **Role）**，Magician 繼承了 Role，**Magician 是一種 Role（Magician** is a **Role）**。

為何要知道繼承時，父類別與子類別間會有「是一種」的關係？因為要開始理解多型（Polymorphism），必須先知道操作的物件是「哪一種」東西！

來看實際的例子，以下的程式碼片段可以通過編譯：

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

那你知道以下的程式片段也可以通過編譯嗎？

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

那為何以下的程式片段為何無法通過編譯呢？

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

編譯器就是語法檢查器，要知道以上程式片段為何可以通過或無法編譯，就是將自己當作編譯器，檢查語法的邏輯是否正確，**方式是從=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）？

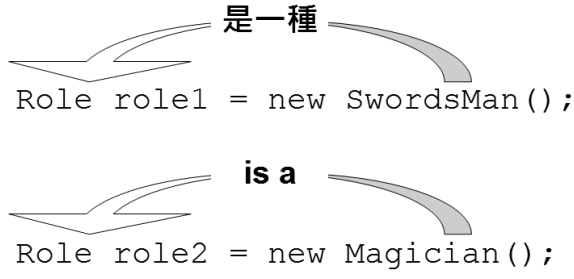

圖 6.1　運用 is a 關係判斷語法正確性

從右往左讀，SwordsMan 是不是一種 Role 呢？是的！因此編譯通過。Magician 是不是一種 Role 呢？是的！因此編譯通過。同樣的判斷方式，可以知道為何以下編譯失敗：

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

編譯器認為第一行，Role 不一定是一種 SwrodsMan，編譯失敗，對於第二行，編譯器認為 Role 不一定是一種 Magician，編譯失敗。繼續把自己當成編譯器，再來看看以下的程式片段是否可以通過編譯：

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

這個程式片段最後會編譯失敗，先從第一行看，SwordsMan 是一種 Role，該行可以通過編譯。編譯器檢查這類語法，一次只看一行，就第二行而言，編譯器看到 role1 為 Role 宣告的名稱，於是檢查 Role 是不是一種 SwordsMan，答案是不一定，第二行編譯失敗！

編譯器會檢查父子類別間的「是一種」關係，如果不要編譯器囉嗦，可以叫它住嘴：

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

對於第二行，原本編譯器想囉嗦地告訴你，Role 不一定是一種 SwordsMan，但你加上了(SwordsMan)讓它住嘴了，因為這表示，你就是要讓 Role **扮演（CAST）** SwrodsMan，**既然都明確要求編譯器別囉嗦了，編譯器就讓這段程式碼通過編譯了，不過後果得自行負責！**

以上面這個程式片段來說，role1 確實參考至 SwordsMan 實例，在第二行讓 SwordsMan 實例**扮演** SwordsMan 沒有什麼問題，執行時期不會出錯。

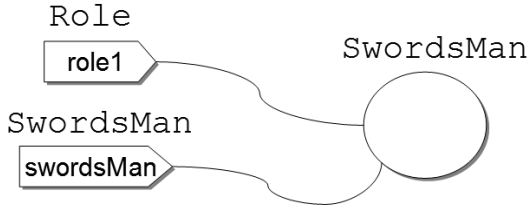

圖 6.2　判斷是否可扮演（CAST）成功

以下的程式片段，編譯可以成功，但是執行時期會出錯：

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

對於第一行，Magician 是一種 Role，可以通過編譯，對於第二行，role2 為 Role 型態，編譯器原本認定 Role 不一定是一種 SwordsMan 而想要囉嗦，但是你明確告訴編譯器，就是要讓 Role 扮演為 SwordsMan，編譯器通過編譯了，不過後果自負，實際上，role2 參考的是 Magician，你要讓魔法師假扮為劍士？這在執行時期會發生錯誤而拋出 **java.lang.ClassCastException**。

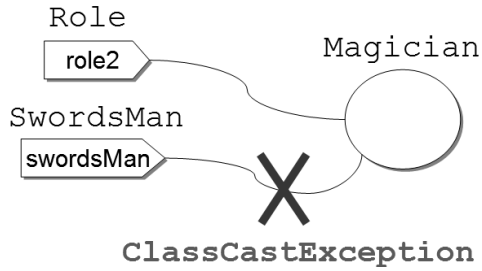

圖 6.3 扮演（CAST）失敗，執行時拋出 ClassCastException

使用是一種（is-a）原則，就可以判斷，編譯是成功或失敗；將扮演（CAST）看作是叫編譯器住嘴語法，並留意參考的物件實際型態，就可以判斷何時扮演成功，何時會拋出 ClassCastException。例如以下編譯成功，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 swordsMan2 = role;    // Role 不一定是一種 SwordsMan，編譯失敗
```

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 swordsMan2 = (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
```

就上例來說，宣告 swordsMan 時可以使用 var，而 role 指定給 swordsMan2 時，已經明確告知編譯器要轉為 SwordsMan 型態，宣告 swordsMan2 時就可以使用 var：

```
var swordsMan = new SwordsMan();
Role role = swordsMan;
var swordsMan2 = (SwordsMan) role;
```

6.1.3　重新定義實作

現在有個需求，請設計 static 方法，可以播放角色攻擊動畫，你也許會這麼想，學剛剛多型的寫法，設計個 drawFight() 方法如何？

```
static void drawFight(Role role) {
    System.out.println("攻擊");
    role.fight();
}
```

The method fight() is undefined for the type Role
2 quick fixes available:
Create method 'fight()' in type 'Role'
Add cast to 'role'
Press 'F2' for focus

圖 6.4　Role 沒有定義 fight() 方法

對 drawFight() 方法而言，只知道傳進來的會是一種 Role 物件，編譯器也只能檢查呼叫的方法，在 Role 上是否有定義，顯然地，Role 沒有定義 fight() 方法，因此編譯錯誤。

然而仔細觀察一下 SwordsMan 與 Magician 的 fight() 方法，**方法簽署（method signature）**都是：

```
public void fight()
```

也就是說，操作介面是相同的，只是方法實作內容不同，可以將 fight() 方法提昇至 Role 類別定義：

Game3 Role.java

```
package cc.openhome;

public class Role {
    ...略
    public void fight() {
```

```
        // 子類別要重新定義 fight()的實作
    }
}
```

在 Role 類別定義了 fight()方法，實際上角色如何攻擊，只有子類別才知道，因此這邊的 fight()方法內容為空，沒有任何程式碼實作。SwordsMan 繼承 Role 之後，再對 fight()的實作進行定義：

Game3_SwordsMan.java

```
package cc.openhome;

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

在繼承父類別之後，定義與父類別中相同的方法簽署，但實作內容不同，這稱為**重新定義（Override）**。Magician 繼承 Role 之後，也重新定義了 fight()的實作：

Game3_Magician.java

```
package cc.openhome;

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

Role 現在定義了 fight()方法（雖然方法區塊中沒有程式碼），編譯器能找到 Role 的 fight()了，因此可以如下撰寫：

Game3_RPG.java

```
package cc.openhome;

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

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

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

    static void drawFight(Role role) {  ←— ❸宣告爲 Role 型態
        System.out.print(role.getName());
        role.fight();
    }
}
```

在 drawFight()方法宣告了 Role 型態的參數❸，那方法中呼叫的，到底是 Role 定義的 fight()，還是個別子類別定義的 fight()呢？如果傳入 drawFight()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：

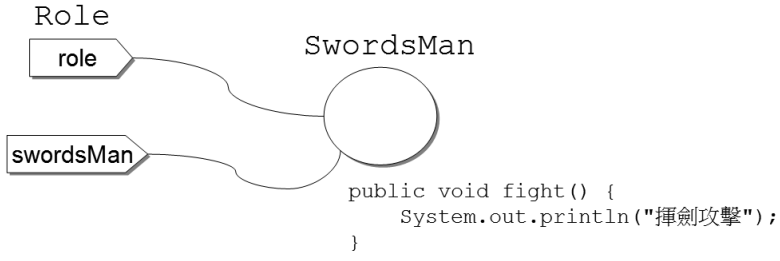

圖 6.5　role 牌子掛在 SwordsMan 實例

這就好比 role 牌子掛在 SwordsMan 實例身上，你要求有 role 牌子的物件攻擊，發動攻擊的物件就是 SwordsMan 實例。同樣地，如果傳入 drawFight()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：

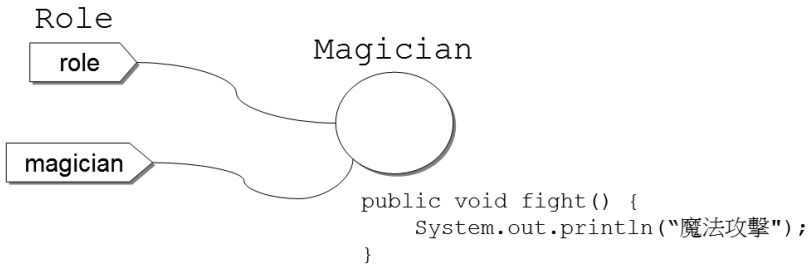

圖 6.6 role 牌子掛在 Magician 實例

因此範例最後的執行結果是：

```
Justin 揮劍攻擊
Monica 魔法攻擊
```

在重新定義父類別的方法時，方法簽署必須相同，若疏忽打錯字了：

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

以這邊的例子來說，父類別定義的是 `fight()`，但子類別定義了 `Fight()`，這就不是重新定義 `fight()`了，而是子類別新定義了一個 `Fight()`方法，這是合法的方法定義，編譯器不會發出任何錯誤訊息，你只會在運行範例時，狐疑為什麼 `SwordsMan` 完全沒有攻擊。

在重新定義方法時，可以在子類別的方法前標註`@Override`，這表示要求編譯器檢查，該方法是否重新定義了父類別的方法，若不是就會引發編譯錯誤。例如：

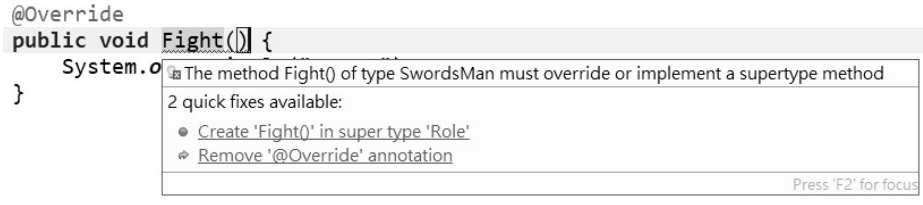

圖 6.7 編譯器檢查是否真的重新定義父類別某方法

除了`@Override` 之外，其他的標註，會在後續章節的適當時候介紹，而標註詳細語法會在第 19 章說明。

6.1.4　抽象方法、抽象類別

上一個範例的 Role 類別定義中，fight()方法中實際上沒有撰寫程式碼，雖然滿足了多型需求，但會引發的問題是，沒有任何方式強迫或提示子類別要實作 fight()方法，只能口頭或在文件上告知，若有人沒有傳達到、沒有看文件或文件看漏了呢？

可以使用 **abstract** 標示該方法為**抽象方法（Abstract method）**，該方法不用撰寫{}區塊，直接;結束即可。例如：

Game4_Role.java

```
package cc.openhome;

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

類別若有未實作的抽象方法，表示這個類別定義不完整，**定義不完整的類別不能用來生成實例**，這就好比設計圖不完整，不能用來生產成品一樣。**內含抽象方法的類別，一定要在 class 前標示 abstract**，表示這是一個定義不完整的**抽象類別（Abstract class）**。如果嘗試用抽象類別建構實例，就會引發編譯錯誤：

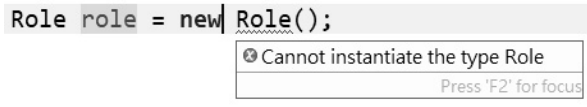

圖 6.8　不能實例化抽象類別

子類別若繼承抽象類別，對於抽象方法有兩種處理方式，其一是繼續標示該方法為 abstract（該子類別因此也是個抽象類別，必須在 class 前標示 abstract），另一個作法是實作抽象方法。如果兩個方式都沒實施，就會引發編譯錯誤：

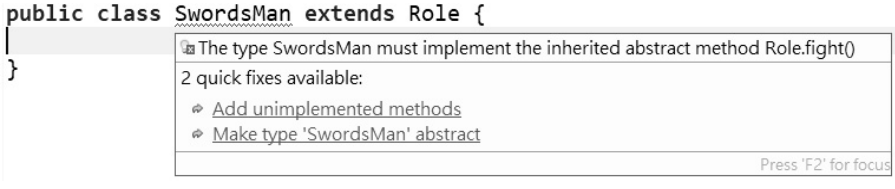

圖 6.9　沒有實作抽象方法

6.2 繼承語法細節

上一節介紹了繼承的基礎觀念與語法，然而結合 Java 的特性，繼承還有不少細節必須明瞭，像是哪些成員只限定在子類別中使用、哪些方法簽署算重新定義、Java 物件都是一種 java.lang.Object 等細節，這將於本節中詳細說明。

6.2.1 protected 成員

就上一節的 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：

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

這對使用 SwordsMan 或 Magician 的客戶端有點不方便，如果可以在 SwordsMan 或 Magician 定義 toString()方法，傳回角色的字串描述：

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

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

客戶端就可以如下撰寫：

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

```
...略
out.printf(magician.toString());
```

看來客戶端簡潔許多。不過你定義的 toString()在取得名稱、等級與血量時不是很方便，因為 Role 的 name、level 與 blood 被定義為 private，無法直接於子類別中存取，只能透過 getName()、getLevel()、getBlood()來取得。

將 Role 的 name、level 與 blood 定義為 public？這又會完全開放 name、level 與 blood 存取權限，你並不想這麼做，只想讓子類別直接存取 name、level 與 blood 的話，可以定義它們為 **protected**：

Game5 Role.java

```
package cc.openhome;

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

被宣告為 protected 的成員，同一套件的其他類別可以直接存取，不同套件的類別，繼承後的子類別可以直接存取。現在 SwordsMan 可以如下定義 toString()：

Game5 SwordsMan.java

```
package cc.openhome;

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

Magician 也可以如下撰寫：

Game5 Magician.java

```
package cc.openhome;

public class Magician extends Role {
    ...略
```

```
    public String toString() {
        return "魔法師 (%s, %d, %d)".formatted(
            this.name, this.level, this.blood);
    }
}
```

提示 如果方法中沒有同名參數，this 可以省略，不過基於程式可讀性，多打個 this 會比較清楚。

到這邊為止，你已經看過三個權限關鍵字，也就是 public、protected 與 private，雖然只有三個權限關鍵字，但實際上有四個權限範圍，因為沒有定義權限關鍵字時，預設就是套件範圍，表 6.1 列出了權限關鍵字與權限範圍的關係：

表 6.1 權限關鍵字與範圍

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

提示 簡單來說，依權限小至大來區分，就是 private、無關鍵字、protected 與 public，設計時要使用哪個權限，是依經驗或團隊討論而定，如果一開始不知道使用哪個權限，就先使用 private，日後視需求再放開權限。

別忘了 Java SE 9 以後支援模組化，如果是採用模組方式設計，public、protected 還會受到模組描述檔中設定的權限限制。

6.2.2 重新定義的細節

在 6.1.3 看過如何重新定義方法，有時候重新定義方法，並非完全不滿意父類別中的方法，只是希望在執行父類別方法的前、後做點加工。例如，也許 Role 類別原本就定義了 toString()方法：

Game6_Role.java

```
package cc.openhome;

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

如果在 SwordsMan 子類別重新定義 toString()時，可以執行 Role 的 toString()方法取得字串結果，再串接"劍士"字樣，不就是想要的描述了嗎？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。例如：

Game6_SwordsMan.java

```
package cc.openhome;

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

類似地，Magician 在重新定義 toString()時，也可以如法泡製：

Game6_Magician.java

```
package cc.openhome;

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

可以使用 super 關鍵字呼叫的父類別方法，不能定義為 private（因為該方法限定只能父類別內使用）。

重新定義方法要注意，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，子類別重新定義時不可為 private 或 protected。例如：

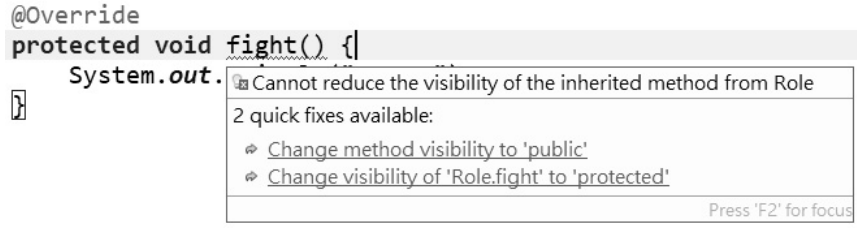

圖 6.10　重新定義時不能縮小方法權限

重新定義方法除了可以定義權限較大的關鍵字外，若返回型態是父類別中**方法返回型態的子類別**，也可以通過編譯。例如原先設計了 Bird 類別：

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

原先 copy()傳回了 Bird 型態，如果 Chicken 繼承 Bird，重新定義 copy()方法時可以傳回 Chicken，例如：

```
public class Chicken extends Bird {
    public Chicken(String name) {
        super(name);
    }
    public Chicken copy() {
        return new Chicken(name);
    }
}
```

注意》》 static 方法屬於類別擁有，若子類別中定義了相同簽署的 static 成員，該成員屬於子類別擁有，而非重新定義，static 方法也沒有多型，因為物件不會個別擁有 static 成員。