第九章 介面

課前指引

介面(interface) 是一種特殊的類別，它只有方法宣告而沒有任何的實作（Java SE8允許方法有預設的實作，稱之為預設方法），與上一章所介紹的抽象類別功能類似，但用法與限制有很大的不同。並且它也是Java 對於『多型』的重要支援技術。此外，Java不支援一般類別的多重繼承，但可以透過介面模擬多重繼承的功能。

章節大綱

備註：可依進度點選小節

9.1 介面的定義

- 介面也可以稱為介面類別，它在編譯後也會產生「介面名稱.class」檔，但一般都只將之稱為**介面(interface)**
- 它的定義語法與類別定義語法很類似，以下是在Java中定義介面的語法：
- 【定義介面語法】

3

9.1 介面的定義

```
[封裝等級] interface 介面名稱 [extends 父介面]
{
   [public] [static final] 資料型態 field 名稱1= 初值; // 宣告field
   [public] [static final] 資料型態 field 名稱2= 初值; // 宣告field
            :
            :
   [public] [abstract] 回傳值型態 method 名稱1( 參數串宣告); // 宣告method
   [public] [abstract] 回傳值型態 method 名稱2( 參數串宣告); // 宣告method
            :
            :
            :
            :
   [public] default 回傳值型態 預設method 名稱A( 參數串宣告) // 宣告預設method
   {
        // 預設methodA 的實作區
   }
   [public] default 回傳值型態 預設method 名稱B( 參數串宣告) // 宣告預設method
   {
       // 預設methodB 的實作區
   }
            :
}
```

可繼承多個父介面，以,加以區隔即可

必須給予初值

雖然沒有method的實作部分，但不需要使用abstract來描述，編譯器會自動視為abstract

SE8新增的語法，可允許宣告預設的method，預設method可包含實作

4

9.1 介面的定義

- 語法說明：
  - (1)介面的宣告是用interface，並且沒有修飾字。
  - (2)介面沒有建構子。也不包含static初始區塊。
  - (3)介面的成員變數在宣告時就**必須給定初值**，並且繼承後無法改變。
    - 關於介面的成員變數將於9.1.1節中詳述。
  - (4)在SE7之前，介面的method只有宣告，沒有實作區（和抽象方法類似，但不需要加上abstract修飾字）。
    - 關於介面的method將於9.1.2 節中詳述。
  - (5)介面不能實作某一個介面，故沒有implements…。
  - (6)SE8允許介面宣告預設的method並且實作它，預設method的封裝等級為public（可省略不寫）。SE8也允許宣告靜態method並實作它。
  - (7)介面也可以繼承其他介面，並且可以有一個以上的父介面（非只有一個）。關於介面的繼承將於9.2節中詳述。父介面可以不只一個，我們會在下一節改寫語法格式。
  - (8)介面名稱在本書中將以I開頭。

5

9.1 介面的定義

- 實作介面語法：

```
[封裝等級] [修飾字] class 類別名稱 [extends 父類別] implements 介面名稱
{
   [封裝等級] [修飾字]   資料型態   field名稱1;  //宣告field
   [封裝等級] [修飾字]   資料型態   field名稱2;  //宣告field
                  :
                  :
   [封裝等級] [修飾字]  回傳值型態  method名稱1(參數串宣告)   //定義method
   {
         method的內容
   }
   [封裝等級] [修飾字]  回傳值型態  method名稱2(參數串宣告)   //定義method
   {
         method的內容
   }
                  :
                  :
}
```

可實作多個父介面，以,加以區隔即可

6

9.1 介面的定義

- 語法說明：
  - 類別要實作介面，必須以「implements 介面名稱」來宣告。並且實作介面與繼承父類別可以同時存在。實作介面可以不只一個，我們會在下一節改寫語法格式。
  - 【觀念範例9-1】：定義介面與實作介面。
    - **範例9-1：**ch9_01.java（隨書光碟 myJava\ch09\ch9_01.java）

```
1   /*   檔名:ch9_01.java          功能:定義介面與實作介面   */
6   public class ch9_01          //主類別
7   {
8       public static void main(String args[])
9       {
10          CCircle obj = new CCircle(5.0);
11          obj.computeArea();
12          obj.show();
13      }
14  }
```

7

9.1 介面的定義

必須給予初值

```
15
16  interface ICircle          //定義介面
17  {
18      double pi=3.14;        //宣告field
19      void show();           //宣告method
20      void computeArea();    //宣告method
21  }
22
23  class CCircle implements ICircle    //實作介面
24  {
25     protected double area;
26     protected double radius;
27     public CCircle(double i){ radius = i; }
28     @Override public void show()
29     {
30          System.out.println("area=" + area);
31     }
32     @Override public void computeArea()
33     {
34          area = radius * radius * pi;
35     }
36  }
```

定義ICircle介面

實作ICircle介面

改寫

改寫

執行結果

area=78.5

8

9.1 介面的定義

- 範例說明：
  - (1)第16~21行是ICircle介面的定義，宣告field時，必須給定初值。宣告method時，不會有實作。事實上，field與method還有其他的規定，我們將於後面兩小節中以本範例來說明。
  - (2)第23~36行，類別CCircle實作了ICircle介面，並且新增了area,radius欄位，以及實作了所有的methods。請注意，除非實作了所有的method，否則無法通過編譯
  - (3)雖然在ICircle中宣告了computeArea()，但為何不將area變數也宣告在介面中呢？
    - 這是因為我們希望area的值能夠改變，如果宣告在介面中，則只能設定初值而不能再被改變了。
  - 但相對地，pi宣告在ICircle中就比較適合，因為它具有常數性質。

9

9.1 介面的定義

- 範例說明：
  - (4)您可以把implements看作是另類的繼承，則本範例可將類別的關係繪製如下圖：

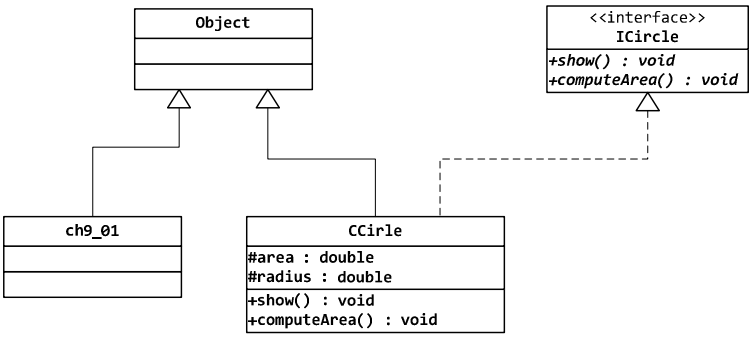

圖9-1 範例9-1的UML圖形

10

9.1 介面的定義

- 我們可以同時繼承類別並實作介面，請見以下範例：
  - 【觀念範例9-2】：同時繼承類別並實作介面。
    - **範例9-2：**ch9_02.java（隨書光碟 myJava\ch09\ch9_02.java）

```
1   /*  檔名:ch9_02.java          功能:同時繼承類別並實作介面  */
6   public class ch9_02          //主類別
7   {
8       public static void main(String args[])
9       {
10          CCircle obj = new CCircle(5.0);
11          obj.computeArea();
12          obj.show();
13      }
14  }
15
```

11

9.1 介面的定義

```
16  interface ICircle          //定義介面
17  {
18      double pi=3.14;      //宣告field
19      public void show(); //宣告method
20      void computeArea(); //宣告method
21  }
22
23  class CShape
24  {
25      protected double area;
26      public void show()
27      {
28          System.out.println("area=" + area);
29      }
30  }
31
32  class CCircle extends CShape implements ICircle    //實作介面
33  {
34     protected double radius;
35     public CCircle(double i){ radius = i; }
36     @Override public void computeArea()
37     {
38          area = radius * radius * pi;
39     }
40  }
```

執行結果

```
area=78.5
```

這裡不能加上@Override來做註記，因為編譯器會直接尋找CShape的父類別Object，但找不到show()

12

9.1 介面的定義

- 範例說明：
  - (1)這個範例改寫自範例9-1，圖形CShape類別包含area變數看起來蠻合理的，但不適合將radius變數也移到CShape類別中，因為它不一定是圓形。
  - (2)第32~40行，類別CCircle同時繼承了CShape也實作了ICircle介面。您可以注意到，由於CShape已經實作了show()，所以類別CCircle也包含了show()的實作，故介面的methods，仍然都已經完全實作了。
  - (3)本範例既繼承又實作，那麼在CCircle類別內若使用super，會去取用父類別的成員還是介面的成員？
    - Java採用的是類別優先原則，因此答案是父類別的成員。

13

9.1 介面的定義

- (4)本範例可將類別的關係繪製如下圖：

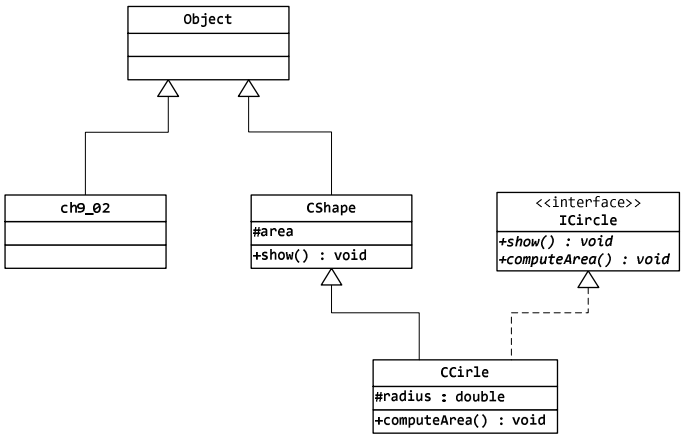

圖9-2 範例9-2的UML圖形

14

9.1.1 介面的欄位

- 在範例9-1中
  - 我們在介面內定義了成員變數（欄位）如下：
    - double pi=3.14;　　//宣告成員變數
  - 編譯器在處理介面的成員變數（欄位）時，會自動將封裝等級設為public，並且加上修飾字static與final，也就是和下列語法相同：
    - **public static final** double pi=3.14;
  - 我們之後會採用第二種語法

15

9.1.1 介面的欄位

- 既然編譯器會將成員變數的宣告視之為「public static final」，所以我們不可以將成員變數的封裝等級設定為「private」或「protected」，而如果未註明封裝等級，則也不是package等級，而是public等級。所以定義介面欄位的語法可以整理如下：

```
public static final 欄位名稱=初值;  //宣告欄位
```

16

9.1.1 介面的欄位

- 【語法說明】：
  - 當某一個類別實作介面時，介面的欄位也會被「繼承」
    - （請將實作看作是另類的繼承，但繼承時條件比較多）
  - 由於是public，所以外部程式可以取用
  - 而由於是static，所以類別的所有物件都將共用一份欄位實體
  - 若類別的物件要存取該欄位，則可以使用「介面名稱.欄位」來取得。

17

9.1.2 介面的成員函式

- 在範例9-1中
  - 我們在介面內宣告了成員函式如下：
    - `void show();` //宣告method
    - `void computeArea();` //宣告method
  - 事實上，編譯器在處理介面的成員函式時，會自動將封裝等級設為public，並且加上修飾字abstract，也就是和下列語法相同：
    - `public abstract void show();` //宣告method
    - `public abstract void computeArea();` //宣告method
  - 我們之後會採用第二種語法

18

9.1.2 介面的成員函式

- 既然編譯器會將介面方法的宣告視之為「public abstract」，所以我們同樣不可以將介面方法的封裝等級設定為「private」或「protected」，而如果未註明封裝等級，則也不是package等級，而是public等級。
- 由於abstract修飾字將與其他修飾字衝突，因此宣告介面方法的語法可以整理如下：

```
public abstract 回傳值資料型態 介面方法(參數串);  //宣告介面方法
```

Coding 注意事項

類別實作介面時，也不能將成員的封裝等級縮小。

19

9.1.3 UML的介面圖

- 由於接下來的程式碼會越來越複雜，搭配圖形來理解會比較容易，因此我們在此介紹UML的介面圖。
  - Java SE7之前的語法可以用UML充分來表達，但SE8新增了介面的預設方法與靜態方法，這部分則非所有用來畫UML圖形的軟體都支援。
- UML的介面圖和類別圖很類似
  - 但會以斜體來表示method，因為這些method都是抽象方法
  - 並且也會標註interface來表示介面。
  - 以範例9-1、9-2為例，其UML圖形如下：

20

9.1.3 UML的介面圖

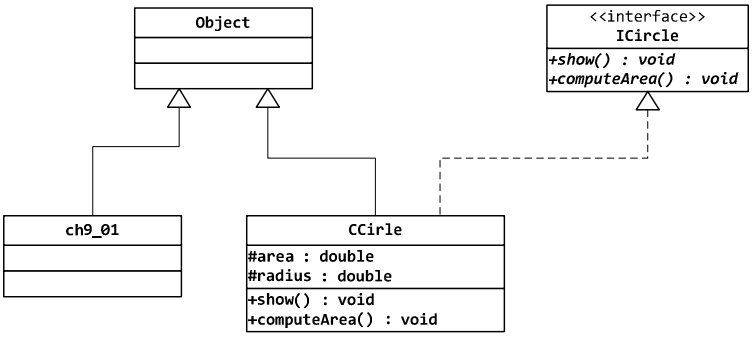

圖9-1 範例9-1的UML圖形

21

9.1.3 UML的介面圖

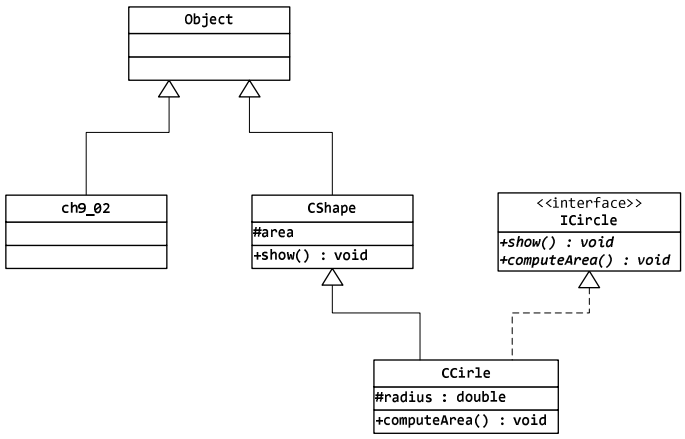

圖9-2 範例9-2的UML圖形

22

9.1.4 介面的預設方法

- Java SE8允許為介面提供預設方法，這破壞了Java語言原有的設計
  - 在SE7之前，Java和C++在這方面最大的差別在於C++只提供抽象類別機制而沒有介面，但C++允許多重繼承
    - （多重繼承容易產生所謂菱形繼承的問題）。
  - 介面
    - 而Java SE7之前為了提供類似多重繼承機制，所以設計出了介面這個語法，把完全沒有實作的類別歸類到介面當中，而有部分實作的類別歸類到抽象類別當中，類別可以實作多個介面，所以能夠達到類似多重繼承的效果。

23

9.1.4 介面的預設方法

- 原本完全沒有實作的就是「界面」，這樣的規劃非常好，可是SE8為何要允許提供可以有實作的預設方法呢？
  - 這是一個新功能與舊包袱所產生的問題。
    - 由於讀者對於Java的認識還不夠深（例如還未學習到何謂Collection介面與Lambda運算式），因此我們不再多做說明。
  - 讀者目前只需要理解，現在的Java已經允許宣告包含實作的預設方法。

24

9.1.4 介面的預設方法

- 問題是Java SE8有了可以包含實作的介面，那麼它和抽象類別又有何區別呢？
  - 類別可以實作多個介面，但無法繼承多個類別
  - 抽象類別可以宣告一般的成員變數，而介面只能宣告final的常數變數。
- 另外的問題是當介面的預設方法和繼承類別的方法產生衝突時，會是什麼樣的情況呢？
  - 答案是類別優先原則
  - 本身的類別最優先、父類別次之、介面最後。

25

9.1.4 介面的預設方法

- 【觀念範例9-3】：介面預設方法與繼承類別的衝突。
- **範例9-3：**ch9_03.java（隨書光碟myJava\ch09\ch9_03.java）

```
1   /*  檔名:ch9_03.java          功能:介面預設方法與繼承類別的衝突  */
6   public class ch9_03          //主類別
7   {
8       public static void main(String args[])
9       {
10          CCircle1 obj1 = new CCircle1(5.0);
11          CCircle2 obj2 = new CCircle2(5.0);
12          CCircle3 obj3 = new CCircle3(5.0);
13          obj1.computeArea();
14          obj1.show();
15          obj2.computeArea();
16          obj2.show();
17          obj3.computeArea();
18          obj3.show();
19      }
20  }
21
```

執行結果

```
pi=3.14
area=78.5
radius=5.0 area=78.5
```

26

9.1

```
22  interface ICircle        //定義介面
23  {
24      public static final double pi=3.14;  //宣告field
25      public abstract void computeArea();  //宣告method
26      public default void show()       //宣告預設method
27      {
28          System.out.println("pi=" + pi);
29      }
30  }
31
32  class CShape
33  {
34      protected double area;
35      public void show()
36      {
37          System.out.println("area=" + area);
38      }
39  }
40
41  class CCircle1 implements ICircle   //實作介面
42  {
43     protected double radius;
44     protected double area;
45     public CCircle1(double i){ radius = i; }
46     @Override public void computeArea()
47     {
48          area = radius * radius * pi;
49     }
50  }
51
```

預設方法show的實作

CShape也定義了一個show()

27

9.1.4 介面的預設方法

```
52  class CCircle2 extends CShape implements ICircle   //繼承並實作介面
53  {
54     protected double radius;
55     public CCircle2(double i){ radius = i; }
56     @Override public void computeArea()
57     {
58          area = radius * radius * pi;
59     }
60  }
61
62  class CCircle3 extends CShape implements ICircle   //繼承並實作介面
63  {
64     protected double radius;
65     public CCircle3(double i){ radius = i; }
66     @Override public void computeArea()
67     {
68          area = radius * radius * pi;
69     }
70     @Override public void show()
71     {
72          System.out.println("radius=" + radius + " area=" +area);
73     }
74  }
```

改寫了show

28

9.1.4 介面的預設方法

- 範例說明：
  - (1) CCircle1很單純，只實作了ICircle，沒有方法衝突的問題，所以執行show時，執行的是ICircle預設方法show的程式內容
  - (2) CCircle3也很單純，雖然它繼承了CShape並實作了ICircle，但show在CCircle3當中被改寫了，所以當然是執行改寫後的show的程式內容。
  - (3) CCircle2繼承了CShape並實作了ICircle，這個時候對於CCircle2而言，它的show方法是由繼承而來的，也就是CShape的show方法內容。
    - 這可以簡稱為類別優先原則。

29

9.1.5 介面的靜態方法

- 提供預設方法
  - 是因為想要讓實作它的類別更輕鬆，可以『不必』去改寫它就能產生物件
- 如果我們希望的不是減輕使用者的使用負擔，而是限制使用者『不可以』去改寫某一個介面的方法，那麼我們該怎麼做呢？
  - 這個時候，我們可以將之宣告為介面的一個靜態方法，如此所有實作它的類別都會共用這個方法
  - 而這也是SE8提供的新功能。

30

9.1.5 介面的靜態方法

- 請注意，雖然我們不可以改寫(override)介面的靜態方法，但我們在實作類別中，可以另外命名一個相同名稱的方法（甚至也可以令之為靜態方法）
  - 不過此時並非改寫
  - 因為在介面宣告的靜態方法(static method)，就是屬於介面的靜態方法，不是屬於類別，也不是屬於類別生成的物件所有。

- 既然如此，因而當我們想要執行介面的靜態方法時
  - 必須採用『介面名稱.靜態方法名稱(參數串)』這個語法來執行。

31

9.1.5 介面的靜態方法

- 【觀念範例9-4】：介面靜態方法的共用及不會被改寫。
- **範例9-4：**ch9_04.java（隨書光碟myJava\ch09\ch9_04.java）

```
1   /*  檔名:ch9_04.java          功能:介面靜態方法的共用及無法改寫  */
6   public class ch9_04          //主類別
7   {
8       public static void main(String args[])
9       {
10          CCircle1 obj1 = new CCircle1();
11          CCircle2 obj2 = new CCircle2();
12          obj1.show();             //執行的是實體方法show
13          ICircle.show();          //執行的是ICircle介面的靜態方法show
14          CCircle2.show();         //執行的是CCircle2類別的靜態方法show
15      }
16  }
17
```

執行結果

```
show實體方法
介面的靜態方法
CCircle2類別定義的show
```

32

9.1.5 介面的靜態方法

```
18  interface ICircle        //定義介面
19  {
20      public static void show()
21      {
22          System.out.println("介面的靜態方法");
23      }
24  }
25
26  class CCircle1 implements ICircle   //實作介面
27  {
28      public void show()
29      {
30          System.out.println("show實體方法");
31      }
32  }
33
34  class CCircle2 implements ICircle   //實作介面
35  {
36      public static void show()
37      {
38          System.out.println("CCircle2類別定義的show");
39      }
40  }
```

定義介面的靜態方法show()

這不是改寫， 也不可以加上@Override

屬於CCircle2的靜態方法

9.1.5 介面的靜態方法

- 範例說明：
  - (1) CCircle1和CCircle2都實作了ICircle介面，而ICircle介面裡已經定義一個靜態方法show及其內容。
  - (2) 第20~23行的靜態方法show屬於ICircle介面所有，要呼叫它必須如第13行的語法來呼叫。
  - (3) 在CCircle1當中，我們定義了一個show實體方法（第28行），這個方法不會去改寫ICircle介面的show靜態方法，所以不可以加上@Override，否則會產生編譯錯誤。並且呼叫時，必須透過物件來呼叫，如第12行的語法。
  - (4) 在CCircle2當中，我們定義了一個show靜態方法（第36行），這個方法是CCircle2類別的靜態方法（類別方法），所以可以透過類別來呼叫，如第14行，其實也可以改為透過物件obj2來呼叫。

9.1.5 介面的靜態方法

- 由上述的範例可知，介面的靜態方法（static method）專屬於介面所有
  - 任何實作它的類別都共用此方法
  - 並且無法改寫它。
- 至於介面的預設方法，其實是個實體方法(instance method)
  - 也就是該介面必須被實作後的類別生成物件之後，透過物件才能執行預設方法。

35

9.2 多重繼承

- 有些程式語言（例如C++）允許類別的繼承來源可以不只一個
  - 舉例來說，假設有一個計費類別CPay，它可能繼承電話類別CTelPhone與手機類別CCellPhone，取其各別的資料成員，以便製作公司的話費明細。
- 在Java中，一般類別(class)並不允許多重繼承，但當一般類別實作介面時，則允許同時實作多個介面，而在介面繼承時，也允許繼承多個父介面
  - 因此，我們可以利用這兩種機制來模擬多重繼承。

36

9.2.1實作多個介面

- 實作多個介面是在Java中模擬多重繼承的機制之一
  - Java的類別實作介面時，可以同時實作多個介面，因此，我們可以把類別實作介面宣告列改寫如下語法：
  - 【實作介面語法】：

```
[封裝等級] [修飾字]
                class 類別名稱 [extends 父類別] implements 介面1,介面2,...,介面n
{
   //類別實作區，需要實作所有的介面
}
```

實作多個介面

  - 【語法說明】：
    - (1)可以同時指定要實作的多個介面，中間以「,」隔開。
    - (2)所有介面內的抽象方法都必須實作（預設方法不屬於抽象方法），該類別才能夠正確編譯。

9.2.1實作多個介面

- 【觀念範例9-5】：同時實作多個介面。
- **範例9-5：**ch9_05.java（隨書光碟myJava\ch09\ch9_05.java）

```
1   /*   檔名:ch9_05.java          功能:實作多個介面  */
6   public class ch9_05          //主類別
7   {
8       public static void main(String args[])
9       {
10          CCircle obj = new CCircle(5.0);
11          obj.computeArea();
12          obj.show();
13          obj.draw(1);
14      }
15  }
16
```

執行結果

```
area=78.5
圖形為紅色圓形
```

9.2.1實作多個介面

```
17  interface ICircle        //定義介面
18  {
19      public static final double pi=3.14;
20      public abstract void show();
21      public abstract void computeArea();
22  }
23
24  interface IColor         //定義介面
25  {
26      public abstract void draw(int i);
27  }
28
29  class CShape
30  {
31      protected double area;
32      public void show()
33      {
34          System.out.println("area=" + area);
35      }
36  }
37
```


9.2.1實作多個介面

```
38  class CCircle extends CShape implements ICircle,IColor
39  {
40     protected double radius;
41     public CCircle(double i){ radius = i; }
42     @Override public void computeArea()
43     {
44          area = radius * radius * pi;
45     }
46     @Override public void draw(int i)
47     {
48          if(i==0)      System.out.println("圖形為藍色圓形");
49          else if(i==1) System.out.println("圖形為紅色圓形");
50          else if(i==2) System.out.println("圖形為綠色圓形");
51          else          System.out.println("顏色錯誤");
52     }
53  }
```

實作兩個介面

- 範例說明：
  - 這個範例改寫自範例9-2，這次CCircle類別不但繼承了CShape類別，也實作了ICircle與IColor兩個介面。本範例可將類別的關係繪製如下圖：

9.2.1實作多個介面

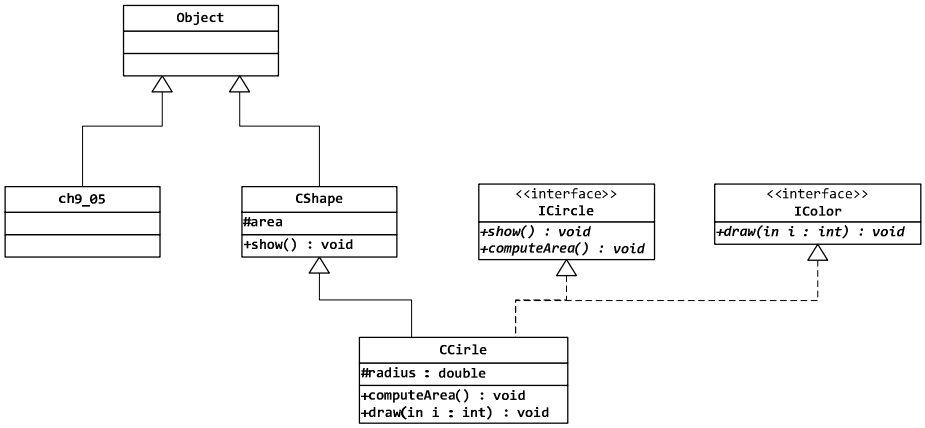

圖9-3 範例9-5的UML圖形

41

9.2.2 實作多個介面的預設方法衝突

- 萬一類別的眾多實作介面中，擁有完全同署名的預設方法，會執行哪一個呢？
  - 由於編譯器不知道要執行哪一個方法，所以會顯示編譯錯誤。
  - 解決之道是，由使用者在類別中改寫該方法。
    - 如果想要執行某個同署名的預設方法，可以在該改寫方法內，透過介面名稱與super關鍵字指定要執行哪一個同署名的預設方法。
  - 即便這些同署名的方法中，只有一個是預設方法，其餘皆為抽象方法，仍要採取上述的解決之道

42

9.2.2 實作多個介面的預設方法衝突

- 【觀念範例9-6】：解決多個介面衝突的預設方法。
  - **範例9-6：**ch9_06.java（隨書光碟 myJava\ch09\ch9_06.java）

```
1   /*   檔名:ch9_06.java          功能:解決多個介面衝突的預設方法  */
6   public class ch9_06          //主類別
7   {
8       public static void main(String args[])
9       {
10          CA obj = new CA();
11          obj.show();
12      }
13  }
14
15  interface IX        //定義介面
16  {
17      public default void show()
18      {
19          System.out.println("IX default");
20      }
21  }
22
```

執行結果

```
IY default
```


9.2.2 實作多個介面的預設方法衝突

```
23  interface IY        //定義介面
24  {
25      public default void show()
26      {
27          System.out.println("IY default");
28      }
29  }
30
31  class CA implements IX,IY   //實作兩個介面
32  {
33     public void show()       //一定要改寫show()方法
34     {
35          IY.super.show();
36     }
37  }
```

呼叫IY的show預設方法

- 範例說明：
  - IX與IY介面都有一個show()預設方法，如果我們實作這兩個介面卻不在實作類別中改寫show()方法，就會無法通過編譯。
  - 第35行的語法是要求執行IY介面的預設show()方法。由於super通常是代表父類別，而IY.super則是代表IY介面這個父類別
    - 這說明了Java把介面看作是父類別，並且想要透過介面來解決多重繼承的問題。

9.2.3 介面的繼承

- 一般類別不能有多個父類別，但介面可以有多個父介面
  - 這也是在Java中模擬多重繼承的機制之一，因此，我們可以將介面定義的語法修改如下：
  - 【定義介面語法】：

```
[public] interface 介面名稱 extends 父介面1,父介面2,...,父介面n
{
    //介面定義
}
```

繼承多個介面

  - 語法說明：
    - (1)在前面我們已經說明過封裝等級的限制，故此處僅列出[public]。
    - (2)介面繼承父介面，同樣使用extends來表示，並且可以有多個父介面，中間以「,」隔開。

45

9.2.3 介面的繼承

- (3)子介面可以重新宣告與父介面的同名方法。
- 如果方法的署名與回傳值型態完全相同，則重新宣告毫無意義，但編譯不會出現錯誤，若署名相同但回傳值型態不同，則編譯會出現錯誤。
- 如果方法的署名不同（僅方法名稱相同），則代表著將來實作的類別將會發生多載的現象。
- (4)多個父介面間可能會出現相同名稱的欄位，而實作介面時則必須避免以變數名稱直接存取該欄位，否則編譯器會無法辨識要存取哪一個欄位，而發生編譯錯誤。但若以「介面名稱.欄位」方式存取欄位，就不會出錯。

46

9.2.3 介面的繼承

- (5)多個父介面間若出現同署名但回傳資料型態不同的方法，則繼承多個介面將會發生編譯錯誤，理由同(3)。多個父介面間若出現名稱相同但署名不同的方法，則代表實作時將會發生多載的現象，理由同(3)。

- (6)如果沒有預設方法，則介面的繼承不會發生改寫(override)的現象，因為即使子介面宣告一個與父介面完全相同的方法，它也沒有實作，故沒有改寫的現象。只有在類別實作介面時，改寫才會發生，且必然發生，否則介面的方法會因為實作不完全，而產生編譯的錯誤。

47

9.2.3 介面的繼承

- 【觀念範例9-7】：同時繼承多個父介面。
  - **範例9-7：**ch9_07.java（隨書光碟 myJava\ch09\ch9_07.java）

```
1   /*  檔名:ch9_07.java          功能:繼承多個父介面  */
6   public class ch9_07          //主類別
7   {
8       public static void main(String args[])
9       {
10          CCircle obj = new CCircle(5.0);
11          obj.computeArea();
12          obj.show();
13          obj.draw(2,1);
14      }
15  }
16
```

執行結果

```
area=78.5
圖形為綠色圓形
圖形為實心
```

48

9.2.3 介面的繼承

```
17  interface ICircle        //定義介面
18  {
19      public static final double pi=3.14;
20      public abstract void computeArea();
21  }
22
23  interface IColor         //定義介面
24  {
25      public abstract void draw(int i);
26  }
27
28  interface IColorCircle extends IColor,ICircle  //繼承兩個父介面
29  {
30      public abstract void show();
31      public abstract void draw(int i);
32      //public abstract int draw(int i);
33      public abstract void draw(int i,int j);  //新增的方法宣告
34  }
35
36  class CShape
37  {
38      protected double area;
39      public void show()
40      {
41          System.out.println("area=" + area);
42      }
43  }
```

無意義的宣告

錯誤的宣告,因為署名相同
但回傳值不同

49

9.2.3 介面的繼承

```
44
45  class CCircle extends CShape implements IColorCircle   //實作介面
46  {
47     protected double radius;
48     public CCircle(double i){ radius = i; }
49     @Override public void computeArea()
50     {
51          area = radius * radius * pi;
52     }
53     @Override public void draw(int i)
54     {
55          if(i==0)      System.out.println("圖形為藍色圓形");
56          else if(i==1) System.out.println("圖形為紅色圓形");
57          else if(i==2) System.out.println("圖形為綠色圓形");
58          else          System.out.println("顏色錯誤");
59     }
60     @Override public void draw(int i,int j)
61     {
62          draw(i);
63          if(j==0)      System.out.println("圖形為空心");
64          else if(j==1) System.out.println("圖形為實心");
65          else          System.out.println("填入錯誤");
66     }
67  }
```

50

9.2.3 介面的繼承

- 範例說明：
  - (1)這個範例改寫自範例9-5，我們將show()方法移到了IColorCircle介面中宣告。IColorCircle介面同時繼承了ICircle與IColor兩個介面。
  - 而CCircle類別則只註明實作IColorCircle介面，這樣子，CCircle類別就必須實作這三個介面的方法。
  - 事實上，如果您將第45行的implements IColorCircle改為implements IColorCircle,ICircle,IColor也是相同的效果。
  - (2)第28~34行是IColorCircle介面的定義，它除了繼承兩個介面外，還新增了show()、draw(int i)與draw(int i,int j)等方法
  - 其中，新增draw(int i)是毫無意義的，因為與父介面完全相同。
  - 至於新增的draw(int i,int j)，則在CCircle類別實作draw方法時，必定需要發生多載。
  - (3)第32行是不合法的宣告，因為我們既然從IColor介面繼承了void draw(int i)，就等於有了第31行的效果，而第31行與第32行是同署名但不同回傳值資料型態，故無法通過編譯。
  - (4)本範例可將類別的關係繪製如下圖（介面的繼承使用的也是實線）：

51

9.2.3 介面的繼承

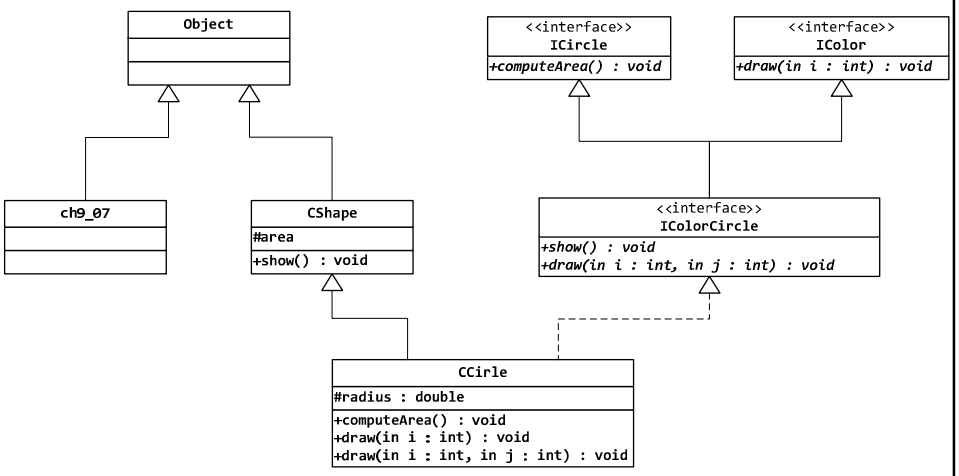

圖9-4 範例9-7的UML圖形

52

9.2.4 繼承多個介面的預設方法衝突

- 繼承介面時，如果父界面有預設方法
  - 子介面可以將之繼承
  - 也可以將之重新宣告為抽象方法。
- 而當繼承多個父介面時，若父界面當中出現了同署名的預設方法
  - 則子介面必須以預設方法的語法將之改寫（override）或將之重新宣告為抽象方法，否則會產生編譯的錯誤
    - 改寫時，可以透過介面名稱.super來指定要執行哪一個父介面的預設方法，請見範例9-8。

9.2.4 繼承多個介面的預設方法衝突

- 【觀念範例9-8】：繼承多個介面改寫衝突的預設方法。
  - **範例9-8：**ch9_08.java（隨書光碟 myJava\ch09\ch9_08.java）

```
1   /*   檔名:ch9_08.java          功能:繼承多個介面改寫衝突的預設方法  */
6   public class ch9_08          //主類別
7   {
8       public static void main(String args[])
9       {
10          CA obj = new CA();
11          obj.show();
12      }
13  }
14
15  interface IX        //定義介面
16  {
17      public default void show()
18      {
19          System.out.println("IX default");
20      }
21  }
22
```

執行結果

```
IY default
```

9.2.4 繼承多個介面的預設方法衝突

```
23  interface IY        //定義介面
24  {
25      public default void show()
26      {
27          System.out.println("IY default");
28      }
29  }
30
31  interface IZ extends IX,IY
32  {
33     public default void show()
34     {
35          IY.super.show();
36     }
37  }
38
39  class CA implements IZ  //實作IZ
40  {
41  }
```

一定要改寫show()方法，否則就要將之再宣告為抽象方法

- 範例說明：
  - 這個範例改寫自範例9-6。IZ 繼承了IX與IY介面，而父介面都有一個show()預設方法，所以除非在IZ介面中重新將之宣告為抽象方法，否則就必須用預設方法的語法來改寫它，如第33~36行。

55

9.2.5 多重繼承的變化

- 多重繼承的變化
  - 前面描述了兩種模擬多重繼承的方式，我們可以選擇以介面繼承多個父介面來達成繼承多個介面的需求；也可以選擇以類別同時實作多個介面來達成繼承多個介面的需求。
  - 或者，我們也可以將兩者合併同時使用。
  - 【觀念範例9-9】：同時繼承多個父介面與實作多個介面。
    - **範例9-9：**ch9_09.java（隨書光碟myJava\ch09\ch9_09.java）

56

9.2.5 多重繼承的變化

```
1  /*  檔名:ch9_09.java          功能:介面的多重繼承組合  */
6  public class ch9_09           //主類別
7  {
8      public static void main(String args[])
9      {
10         CCircle obj = new CCircle(5.0);
11         obj.computeArea();
12         obj.draw(2,1);
13     }
14 }
15
16 interface ICircle          //定義介面
17 {
18     public static final double pi=3.14;
19     public abstract void computeArea();
20 }
21
22 interface IColor          //定義介面
23 {
24     public abstract void draw(int i);
25 }
26
```

執行結果

```
area=78.5
圖形為綠色圓形
圖形為實心
```


9.2.5 多重繼承的變化

```
27 interface IColorCircle extends IColor,ICircle  //繼承兩個父介面
28 {
29     public abstract void show();
30     public abstract void draw(int i);   //無意義的宣告
31     //public abstract int draw(int i);  //錯誤,因為署名相同但回傳值不同
32     public abstract void draw(int i,int j);  //新增的方法宣告
33 }
34
35 interface IShape          //定義介面
36 {
37     public abstract void show();
38 }
39
40 class CCircle implements IShape,IColorCircle   //實作介面
41 {
42    protected double area;
43    protected double radius;
44    public CCircle(double i){ radius = i; }
45    @Override public void computeArea()
46    {
47         area = radius * radius * pi;
48    }
```

9.2.5 多重繼承的變化

```
49      @Override public void draw(int i)
50      {
51           show();
52           if(i==0)      System.out.println("圖形為藍色圓形");
53           else if(i==1) System.out.println("圖形為紅色圓形");
54           else if(i==2) System.out.println("圖形為綠色圓形");
55           else          System.out.println("顏色錯誤");
56      }
57      @Override public void draw(int i,int j)
58      {
59           draw(i);
60           if(j==0)      System.out.println("圖形為空心");
61           else if(j==1) System.out.println("圖形為實心");
62           else          System.out.println("填入錯誤");
63      }
64      @Override public void show()
65      {
66           System.out.println("area=" + area);
67      }
68  }
```

- 範例說明：
  - 這個範例改寫自範例9-7，我們將CShape類別改IShape介面。本範例的類別關係如下圖：

59

9.2.5 多重繼承的變化

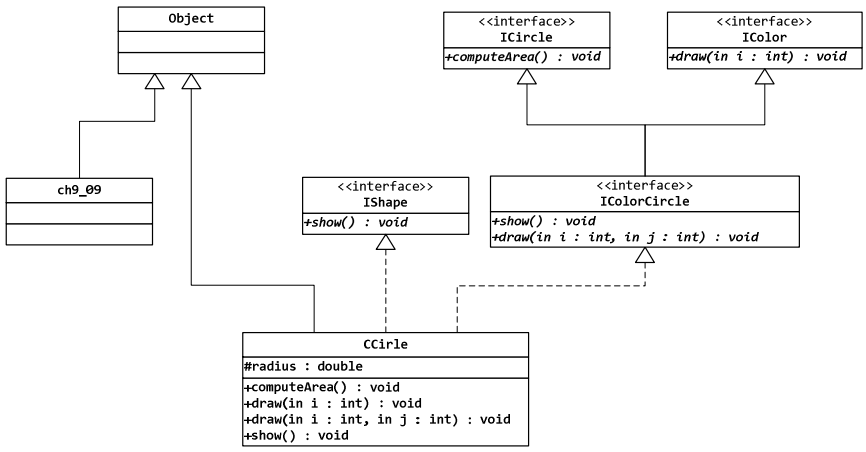

圖9-5 範例9-9的UML圖形

60

9.2.6 多重繼承的注意事項

- 即便不考慮SE8提供的介面預設方法及介面靜態方法，多重繼承的組合仍非常複雜，同時平行的介面與類別可能在定義欄位與方法時，會出現同名的問題
- 在撰寫多重繼承的程式時，需要特別注意下列事項：
  - (1)類別必須實作所有層級的父介面方法，包含父介面繼承的介面方法。
  - (2)實作父介面的方法時，只能宣告為public等級
  - (3)讀取介面欄位時，使用「介面名稱.欄位名稱」方式讀取。
  - (4)父類別與父介面可能出現「相同署名」的方法，若相同署名但回傳值型態不同，則無法多重繼承它們。

61

9.3 多型：介面型態變數與實作之類別物件

- 介面無法產生物件實體，但介面可以當作資料型態來宣告變數（之前所提的抽象類別亦如此），而使用介面資料型態宣告的變數可以指向實作它的類別物件實體以及其他相容類別的物件。
  - 所以我們可以宣告一個物件變數，宣告型態時採用介面型態，但卻將它指向實作類別的物件實體。如下範例：

```
interface IA{.....}
class CA implements IA{.....}
IA obj= new CA();
```

介面型態變數參考實作類別物件實體

  - 當介面型態的物件變數指向實作子類別的物件實體後，我們可以透過該變數執行實作類別中改寫(override)的method，但不能執行實作類別的新增method。由於欄位必定宣告為public等級，故也可以存取介面定義的欄位，但不可存取實作類別的新增成員變數。

62

9.3 多型：介面型態變數與實作之類別物件

- 【觀念範例9-10】：介面型態變數參考實作類別物件實體。
  - **範例9-10：**ch9_10.java（隨書光碟 myJava\ch09\ch9_10.java）

```
1   /*   檔名:ch9_10.java          功能:介面型態變數參考實作類別物件實體   */
6   public class ch9_10   //主類別
7   {
8       public static void main(String args[])
9       {
10          int i;
11          IA obj =  new CA();
12          obj.show1();          //可執行介面宣告的方法(由子類別改寫的method)
13          i = obj.varA;         //可存取介面定義的欄位
14          //obj.varB = 10;
15          //obj.show2();
16      }
17  }
18
19  interface IA              //定義介面
20  {
21      public static final int varA=5;
22      public abstract void show1();
23  }
24
```

執行結果

實作類別改寫的show1()執行中

介面型態變數參考實作類別物件**實體**

不可存取新增的成員變數

不可執行新增的method

9.3 多型：介面型態變數與實作之類別物件

```
25  class CA implements IA   // 類別CB實作界面IA
26  {
27      public int varB;
28      @Override public void show1()
29      {
30          System.out.println("實作類別改寫的show1()執行中");
31      }
32      public void show2()
33      {
34          System.out.println("實作類別新增的show2()執行中");
35      }
36  }
```

- 範例說明：
  - (1)第11行，宣告變數型態為介面IA，但new產生實體時則為實作類別CA，因此，介面變數obj將指向實作類別物件實體。
  - (2)第12行，obj可以執行介面宣告的方法。
  - 第13行，obj可以讀取（唯讀）介面定義的欄位。
  - 第14行，obj不能存取實作類別新增的成員變數。
  - 第15行，obj不能執行實作類別新增的method。

9.4 多型與動態繫結

- 介面無法產生物件實體，它最主要是用來製作一個共同的介面，若程式需要使用此介面時，則將之實作即可。
  - 抽象類別也具有這樣的性質，但介面除了預設方法及靜態方法外，完全沒有實作部份，並且允許同時實作多個介面，因此，比抽象類別更具有抽象性。
- 介面與抽象類別都是實現物件導向重要的「多型」機制。而多型又與動態繫結有關，在本節中，我們將對於動態繫結進行介紹。

65

9.4 多型與動態繫結

- 在物件導向程式中，為了實現多型，通常採用動態繫結來完成方法呼叫。動態繫結與靜態繫結的涵義如下：
  - 靜態繫結(static binding)：
    - 方法呼叫在編譯程式時就決定，「編譯時期」根據物件變數的型態，決定要執行哪一個類別的方法內容。
  - 動態繫結(dynamic binding)：
    - 方法呼叫在執行程式時才決定，「執行時期」根據物件變數的型態，決定要執行哪一個類別的方法內容，該方法的內容一定是最新改寫的內容。

66

9.4 多型與動態繫結

- Java支援靜態繫結與動態繫結
  - 並且只有被宣告為final的方法會使用靜態繫結，其餘方法都採用動態繫結。
  - 並且Java的動態繫結稱為動態方法分派(dynamic method dispatch)。
    - 對於可以在繼承後改寫的方法，父類別變數可以指向本身物件實體，也可以指向子孫類別的物件實體，而執行該方法呼叫時，必定是執行最新改寫的方法實作，故採用動態繫結。
    - 至於被宣告為final的方法，由於不可以被改寫(override)，因此，不必理會該類別變數究竟是指向本身物件實體還是子孫類別的物件實體，在方法呼叫時，必定是執行本身的方法實作，故在編譯時期就可以完成方法呼叫所繫結的方法實作，而這將使得執行效率較佳。

67

9.4多型與動態繫結

- 多型中的改寫(override)由於採用了動態繫結，因此可以使得執行方法呼叫必定能夠執行到最新改寫的方法實作。
  - 而改寫在抽象類別的abstract method中必定會發生，在介面中也必定會發生，故兩者皆為Java對於多型的設計
  - 如果是一般的類別，則必須子孫類別改寫了method的實作，才會發生多型的現象。
  - 【觀念範例9-11】：動態繫結與靜態繫結。
    - **範例9-11：**ch9_11.java（隨書光碟 myJava\ch09\ch9_11.java）

68

9.4多型與動態繫結

```
1   /*   檔名:ch9_11.java          功能:動態繫結與靜態繫結   */
6   public class ch9_11          //主類別
7   {
8       public static void main(String args[])
9       {
10          CA obj;             //宣告父類別型態變數
11          obj = new CB();     //obj指向子類別CB物件實體
12          //obj.show1();        //編譯時期即可知無法執行，因為CA未宣告此method
13          obj.show2();        //被宣告為final的方法,為靜態繫結
14          obj.show3();        //因為是抽象方法，故必發生多型之改寫(動態繫結)
15          obj.show4();        //使用動態繫結以正確執行方法
16          System.out.println("--------------------");
17          obj = new CC();     //obj指向子類別CC物件實體
18          //obj.show1();        //編譯時期即可知無法執行，因為CA未宣告此method
19          obj.show2();        //被宣告為final的方法,為靜態繫結
20          obj.show3();        //因為是抽象方法，故必發生多型之改寫(動態繫結)
21          obj.show4();        //使用動態繫結以正確執行方法
22      }
23  }
24
```

69

9.4多型與動態繫結

```
25  interface IA                //定義介面
26  {
27      public abstract void show1();
28  }
29
30  abstract class CA           //定義父類別
31  {
32      public final void show2()
33      {
34          System.out.println("This is CA's show2()");
35      }
36      public abstract void show3();
37      public void show4()
38      {
39          System.out.println("This is CA's show4()");
40      }
41
42  }
43
```

70

9.4多型與動態繫結

```
44  class CB extends CA implements IA   // 類別CB實作界面IA
45  {
46      public void show1()
47      {
48          System.out.println("This is CB's show1()");
49      }
50      public void show3()
51      {
52          System.out.println("This is CB's show3()");
53      }
54  }
55
56  class CC extends CA implements IA   // 類別CC實作界面IA
57  {
58      public void show1()
59      {
60          System.out.println("This is CC's show1()");
61      }
62      public void show3()
63      {
64          System.out.println("This is CC's show3()");
65      }
66      public void show4()
67      {
68          System.out.println("This is CC's show4()");
69      }
70  }
```


9.4多型與動態繫結

- 執行結果

```
This is CA's show2()
This is CB's show3()
This is CA's show4()
--------------------
This is CA's show2()
This is CC's show3()
This is CC's show4()
```

- 範例說明：
  - (1)第10行，宣告obj為父類別CA變數型態，它可以指向父類別或子孫類別的物件實體。
  - (2)第11行，obj指向子類別CB物件實體。第17行，又將obj指向子類別CC物件實體。
  - 由於父類別未宣告show1()方法，故第12、18行不可執行，這在編譯時期就可以決定（因為obj為父類別型態變數，而只要檢查父類別是否宣告過該method即可）。

9.4多型與動態繫結

- (3)第13、19行，呼叫final show2()，不論該實體為CB類別的物件或CC類別的物件，都必定是執行CA類別的method實作，因此，可於編譯時期決定，故為靜態繫結。而其他的show3(), show4()函式呼叫，則為動態繫結，以保證它必定能夠執行到最新被改寫的method實作。

延伸學習：**反射機制**

Java透過了反射機制使得可以在執行期間載入與使用編譯時期未知的類別，關於Java的反射機制已經超出本書範圍，有興趣的讀者，可以參考下列網址的說明。
http://jjhou.csdn.net/javatwo-2004-reflection.pdf

延伸學習：**多型與彈性設計**

事實上，多型可以使軟體設計更具有彈性，對於未來功能的擴展與變化更容易維護，有許多的設計模式採用了多型來設計，請初學者在學習程式語言、UML 之後，一定要去學習一下設計模式(Design Pattern)。

73

9.5變數的物件型態轉換

- 父類別型態的變數雖然可以指向子類別物件實體，但卻不能執行子類別新增的method，例如範例9-11的第12行
  - 如果想要執行子類別新增的method，我們可以透過強制型態轉換來達成，它與原始資料型態的轉型類似，語法如下

```
(類別型態)  物件參考變數;
```

  - 【觀念範例9-12】：類別型態的轉型。
    - **範例9-12：**ch9_12.java（隨書光碟 myJava\ch09\ch9_12.java）

74

9.5 變數的物件型態轉換

```
1   /*   檔名:ch9_12.java          功能:類別型態的轉型  */
6   public class ch9_12          //主類別
7   {
8       public static void main(String args[])
9       {
10          CA obj1;             //宣告父類別型態變數
11
12          obj1 = new CB();     //obj1指向子類別CB物件實體
13          // obj1.show1();     //未轉型不能執行父類別未宣告的show1()
14          ((CB)obj1).show1();  //轉型後可執行show1()
15          obj1.show2();        //可執行父類別宣告的show2()
16          //obj1.show3();      //未轉型不能執行父類別未宣告的show3()
17          ((CB)obj1).show3();  //轉型後可執行與show3()
18          System.out.println("--------------------");
19          CB obj2;             //宣告子類別型態變數
20          obj2 = (CB) obj1;
21          obj2.show1();
22          obj2.show2();
23          obj2.show3();
24      }
25  }
```

obj2指向obj1所指物件(需要轉型，不能直接指定)

75

9.5 變數的物件型態轉換

```
26
27  interface IA             //定義介面
28  {
29      public abstract void show1();
30  }
31
32  abstract class CA        //定義父類別
33  {
34      public void show2()
35      {
36          System.out.println("This is CA's show2()");
37      }
38  }
39
40  class CB extends CA implements IA   // 類別CB實作界面IA
41  {
42      @Override public void show1()
43      {
44          System.out.println("This is CB's show1()");
45      }
46      public void show3()
47      {
48          System.out.println("This is CB's show3()");
49      }
50  }
```

76

9.5 變數的物件型態轉換

- 執行結果：

```
This is CB's show1()
This is CA's show2()
This is CB's show3()
--------------------
This is CB's show1()
This is CA's show2()
This is CB's show3()
```

- 範例說明：
  - (1)第10行，宣告父類別CA型態的變數obj1。但在第12行，它將指向子類別CB的物件實體。
  - (2)第13、16行無法執行父類別未宣告的函式。但第14、17行，透過轉型就能夠執行show1()及show3()。
  - (3)第19行，宣告子類別CB型態的變數obj2。我們欲將obj2與obj1指向同一個物件實體，故在平常狀況時，會採用obj2=obj1來設定，但由於兩者資料型態不同，所以需要經過轉型，故應撰寫為第20行格式。obj2可以執行所有CB類別定義的public method，因為它本來就是CB型態的變數。

77

9.5 變數的物件型態轉換

- 物件參考之間的轉型注意事項
  - 物件參考之間的轉型，並非毫無限制，我們只能在有直系繼承關係的類別中進行轉換，才能通過編譯，但仍有可能發生執行時期的錯誤。如下所述：
    - (1)欲將父類別型態變數指向子類別物件實體，不會發生問題，因為子類別必定包含父類別的副本。因此透過轉型可達到目的。
    - (2)欲將子類別型態變數指向父類別物件實體，會發生問題，因為父類別的物件實體沒有子類別的副本。即使透過轉型可使之通過編譯器的檢查，但它會產生執行時期的錯誤。範例如下：

78

9.5 變數的物件型態轉換

```
public class ch9_13        //主類別
{
   public static void main(String args[])
   {
            CA obj1 = new CA();  //宣告父類別型態變數並產生物件實體
            CB obj2 = (CB) obj1;
   }
}
class CA        //定義父類別
{
   public void show1()   { … }
}

class CB extends CA
{
   public void show2()    {  …  }
}
```

執行時期會發生錯誤

9.5 變數的物件型態轉換

- 除此之外，介面變數也可以與一般類別變數進行型別轉換，如果將實作(implements)視為繼承，則只有要直系繼承關係，就可以進行轉型。
- 介面型態的變數可以和Object類別型態的變數進行轉型，因為介面一定必須由某個類別來實作，此時，該類別與介面有直系繼承關係，而任何類別又與Object類別有直系繼承關係，故兩者可以進行轉換。

9.5 變數的物件型態轉換

- 介面引數與介面回傳值
  - 在Java內建類別的method中，常常可以看到參數或回傳值的型態被宣告為介面。
  - 既然介面可以當作型態宣告變數，當然也可以用來宣告method的參數與回傳值之型態。
  - 不過由於傳遞時，必須傳遞一個真實的物件參考或null，因此，一般我們會將物件當作傳遞引數或用來接收回傳值，這中間牽扯到是否需要進行轉型的問題
  - 【觀念範例9-13】：介面參數與介面回傳值。
    - 範例9-13：ch9_13.java（隨書光碟 myJava\ch09\ch9_13.java）

81

9.5 變數的物件型態轉換

```
1   /*   檔名:ch9_13.java     功能:介面參數與介面回傳值   */
6   public class ch9_13          //主類別
7   {
8       public static void main(String args[])
9       {
10          CA objCA = new CA();        //objCA為CA類別的物件
11          CB objCB = new CB();        //objCB為CB類別的物件
12          //-----------------------------------------------------
13          objCB.show3(objCA);         //把CA類別的物件當作引數傳入
14          //-----------------------------------------------------
15          objCA=(CA)objCB.set(10); //由於回傳值被宣告為介面，因此需要轉型
16          System.out.println("objCA的欄位varB=" + objCA.varB);
17      }
18  }
19
20  interface IA                //定義介面
21  {
22      public static final int varA=5;
23      public abstract void show1();
24  }
25
```

82

9.5 變數的物件型態轉換

```
26  class CA implements IA   // 類別CA實作界面IA
27  {
28      public int varB;
29      public CA()      { varB = 0; } //建構子
30      public CA(int i) { varB = i; } //建構子
31      @Override public void show1()
32      {
33          System.out.println("This is CA's show1()");
34      }
35      public void show2()
36      {
37          System.out.println("This is CA's show2()");
38      }
39  }
40
41  class CB
42  {
43      public void show3(IA obj)      //參數被宣告為介面型態
44      {
45          System.out.println("傳入之物件的varA欄位為" + obj.varA);
46          obj.show1();
47          //obj.show2();
48      }
49      public IA set(int i)      //回傳值被宣告為介面型態IA
50      {
51          CA obj = new CA(i);
52          return obj;
53      }
54  }
```

此行不合法，obj被宣告為IA介面，不可執行介面未宣告的method

回傳的是CA類別的物件，而CA實作了IA，故不須轉型

9.5 變數的物件型態轉換

- 執行結果：

```
傳入之物件的varA欄位為5
This is CA's show1()
objCA的欄位varB=10
```

- 範例說明：
  - (1)第10~11行，objCA是CA類別的物件，objCB是CB類別的物件。
  - (2)IA為介面，類別CA實作介面IA。
  - (3)CB類別的show3()宣告參數為介面型態。set()宣告回傳值為介面型態。
  - (4)這個程式分為兩大部分來看，第一部分為第13行，objCB執行show3()時，將物件變數objCA傳入method中，由於CA與IA有直系繼承關係，故不須轉型，obj也可以指向objCA物件，但不可執行show2()（第47行），因為obj為介面變數，無法執行非介面宣告的method。
  - (5)第二部分為第15行，呼叫set()，並以objCA接收這個回傳值。而第51行將obj宣告為CA物件，回傳CA物件時與宣告的介面型態回傳值由於有直系繼承關係，故不必轉型，但接收端需要進行轉型才能接收介面變數（第15行）

9.5 變數的物件型態轉換

- 陣列與Object型態的型態轉換
  - 在介紹陣列時曾經提及，Java的陣列可以視為一種特殊的物件，它也繼承了內建的Object類別，因此，陣列可以和Object類別型態的變數進行型態轉換，下面這個範例簡單示範了兩者之轉換方式。
  - 【觀念範例9-14】：陣列與Object型態的型態轉換。
    - 範例9-14：ch9_14.java（隨書光碟 myJava\ch09\ch9_14.java）

9.5 變數的物件型態轉換

```
1   /*   檔名:ch9_14.java          功能:陣列與Object型態的型態轉換   */
6   public class ch9_14          //主類別
7   {
8       public static void main(String args[])
9       {
10          int[][] ary1 = new int[][]{{1,2,3},{4,5,6}};//二維陣列實體
11          Object obj1;
12          obj1 = (Object)ary1;
13          System.out.println("ary1執行toString=" + ary1.toString());
14          System.out.println("obj1執行toString=" + obj1.toString());
15
16          int[][] ary2 = (int[][]) obj1;
17          System.out.println("ary1[1][1]=" + ary1[1][1]);
18          System.out.println("ary2[1][1]=" + ary2[1][1]);
19      }
20  }
```

型態轉換

宣告二維陣列參考並進行型態轉換

- 執行結果：

```
ary1執行toString=[[I@5fa6fb3e
obj1執行toString=[[I@5fa6fb3e
ary1[1][1]=5
ary2[1][1]=5
```

9.6 instanceof運算子

- 在前面的章節中，我們一直強調子類別之物件實體內擁有父類別的一份實體，因此，可以執行父類別的method及存取父類別的變數。
- 而當繼承很複雜時，在程式中要看出物件是否擁有某類別的實體有時並非易事，此時，可以藉由instanceof運算子來判斷。語法如下：

```
物件參考   instanceof   型態;
```

87

9.6 instanceof運算子

- 語法說明：
  - (1)雖然instanceof是很長的一個關鍵字，但您只要將其看作是運算子即可
    - 例如　obj 「instanceof」MyClass
    - 它的回傳值是一個布林值，就如同我們在使用a「>」b判斷式般使用即可。
  - (2)若「物件參考」所指實體為右方「型態」，則會回傳true。
  - (3)若「物件參考」所指實體為右方「型態」之子孫類別型態（含繼承與實作），則會回傳true。
  - (4)若非上述兩種狀況，則會回傳false。若「物件參考」為null（未指向任何實體），也會回傳false。
- 【觀念範例9-15】：instanceof運算子。
  - 範例9-15：ch9_15.java（隨書光碟 myJava\ch09\ch9_15.java）

88

9.6 instanceof運算子

```
1   /*   檔名:ch9_15.java          功能:instanceof運算子   */
6   public class ch9_15          //主類別
7   {
8       public static void main(String args[])
9       {
10          CA objA = new CA();        //obj1為CA類別的物件
11          CB objB = new CB();        //obj2為CB類別的物件
12          CC objC = new CC();        //obj2為CB類別的物件
13            //-------------------------------------------------------
14          if (objA instanceof CA)
15              System.out.println("objA擁有一份CA類別的實體");
16          if (objB instanceof CA)
17              System.out.println("objB擁有一份CA類別的實體");
18          if (objC instanceof CA)
19              System.out.println("objC擁有一份CA類別的實體");
20          if (objA instanceof CC)          結果為false
21              System.out.println("objA擁有一份CC類別的實體");
22          if (objB instanceof IA)
23              System.out.println("objB擁有一份IA介面的實作實體");
24      }
25  }
```


9.6 instanceof運算子

```
26
27  interface IA              //定義介面
28  {
29      public static final int var1=5;
30  }
31
32  class CA implements IA   // 類別CA實作界面IA
33  {
34      public int var2;
35  }
36
37  class CB extends CA   // 類別CB繼承CA
38  {
39      public int var3;
40  }
41
42  class CC extends CB   // 類別CC繼承CB
43  {
44      public int var4;
45  }
```

9.6 instanceof運算子

- 執行結果：

```
objA擁有一份CA類別的實體
objB擁有一份CA類別的實體
objC擁有一份CA類別的實體
objB擁有一份IA介面的實作實體
```

- 範例說明：
  - (1)第20行為false，因為objA並沒有類別CC的實體。
  - (2)第14、16、18行都為true，因為objA, objB, objC都擁有類別CA的一份實體（有些是因為繼承機制而來）。
  - (3)第22行也為true，因為實作介面，也會使得物件擁有介面宣告的成員。

91

9.7介面與抽象類別的異同

- 改寫是多型中最重要的機制，而介面與抽象類別都會使得改寫必定發生，兩者之間有很多類似處，但也有些不同，我們將之整理如下：
  - 介面與抽象類別的相似處
    - (1)只要包含抽象方法就一定要宣告為抽象類別或介面。抽象類別內可以包含抽象方法也可以包含非抽象方法，但介面除了靜態方法與預設方法外，全部都是抽象方法。
    - (2)介面與抽象類別都不能直接產生物件實體。必須透過繼承或實作，並改寫實作出方法的細節，才能夠產生物件實體。

92

9.7介面與抽象類別的異同

- 介面與抽象類別的相異處
  - (1)抽象類別有建構子；介面沒有建構子。
  - (2)在加入介面的預設方法機制之後，介面與抽象類別更像了，不過，抽象類別的成員變數並無特殊規定；但介面的成員變數一定是final變數。
  - (3)抽象類別的成員之封裝等級並無特殊規定；但介面的成員一定是public，並且不可以加上public以外的封裝修飾字。
  - (4)在UML圖中，介面並不會顯示其資料成員，而抽象類別會顯示其資料成員
    - 所以如果您只想要定義的這個東西該有哪些「操作」，那麼就應該宣告為「介面」。如果不是，那就宣告為類別，如果當中包含了部分暫時無法或可能會再細分為幾種狀況的操作時，就將之宣告為抽象方法（讓該類別變成抽象類別），留待繼承的類別來實作。。

93

本章結束

Q&A討論時間

94