Table of Content

物件與類別

物件(Object)

- 物件是一種電腦程式中自我存在的元素，又稱之為生存物件(instance)，代表各種相關連的型態群組，用以完成整個程式設計的任務。
  - 物件使用變數儲存狀態稱之為屬性(Property)或實例變數(Instance Variable)，而各種行為的程序和函數則稱之為方法(Method)。
  - 物件的屬性或成員以句點"."描述之。
  - Java 的物件是以類別來建立，是一種可由使用者自行定義的資料型態。

類別(Class)

- 類別是一種用來建立具有多個物件型態的樣版(template)。
  - 同一支程式裡可以有不同的類別，但只能有一個類別被宣告為 public，此即主體類別，表示可以由外部程式呼叫，未宣告為 public 的類別則僅能由程式內部類別呼叫，程式之主體類別名稱必須和其儲存檔案之主檔名相同。
  - 類別的宣告是物件的原型宣告，可分為兩部分：
    - 成員資料(Data Member)：物件的資料部分，屬於基本資料型態的變數、常數或其他物件的成員變數(Member Variables)。
    - 成員方法(Method Member)：物件的操作部分，即程式的函數。
  - 類別在宣告後得以建立物件，語法如下：

```
class 類別名稱
{
    [建構子]
    存取敘述修飾子 資料型態 成員資料 ;
    ……
    存取敘述修飾子 傳回值型態 成員方法 ( 參數列 )
    {
        程式敘述 ;
    }
    ……
}
```

  - 如類別中存在主程式，則主程式起始位置之字串 public static void main(String[] args)為保留型態，不得任意更改。
  - 建構子為其他程式要使用此類別宣告物件變數時首先使用到的成員資料，必須為 public 支存取修飾子，如省略的話，則會預設為空的建構子，如 public class_name() { }

public、private 與 protected

- 存取敘述修飾子 public, private, protected 等，定義類別中各成員資料或方法的生命週期，
  - public 表示為公用變數，其他類別中宣告為此類別的物件變數都可以使用；
  - private 表示為私用變數，僅提供此類別內之成員變數呼叫使用；
  - protected 表示屬於同一套件(package)的類別間可相互呼叫，在不同 package 則不得使用；
  - 若未定義，則僅限於類別內部使用，否則要加上 static 成為靜態資料或方法。

類別的組成－建構子、動態實例方法與靜態類別方法

建構子(Constructors)

- 讓類別在被呼叫時可以傳入參數
  - 建構子之名稱必須與所屬類別之名稱相同;
  - 建構子無回傳資料(內容無 return)，宣告中亦不得前置 void;
  - 建構子主要功能是在所屬類別產生新物件時作初始化動作。
  - 建構子的數目可以不只一個，亦即可利用相同名稱的建構子賦予不同的傳入引數，讓類別所提供的方法可以更有彈性的使用不同的參數進行各種資料運算。
  - 建構子的宣告方式如下，其中參數列可以無任何引數。

```
public 類別名稱 (參數列)
{
    程式敘述 ;
}
```

實例方法程序(Instance Methods)

- 實例方法程序屬於動態方法，當方法程序所屬之物件尚未產生之前，該方法程序無法執行，故須先作物件的宣告方可使用。
  - 例如宣告陣列時必須利用 new 來產生一個新的陣列物件。
  - 敘述方法如下：

```
存取敘述 傳回值型態 方法名稱 (參數列)
{
    程式敘述 ;
}
```

    - 存取敘述－public, private, protected 等，定義該方法的生命週期。
    - 若有無傳回值時，僅是利用方法程序進行某種運算，則傳回值型態為 **void**。
    - 若是有傳回值時，傳回值型態使用 int 或 char 等數值或字串之類的資料型態，此時的方法便稱為**函數(Functions)**，方法的程式區塊內部要利用 **return**

指令傳回一個符合方法資料型態的傳回值。

類別方法程序(Class Methods)

- 類別方法程序或稱為靜態方法(Static Method)，宣告時須前置 **static**－利用此修飾子，表示靜置於記憶體中，此方法便屬於此類別本身，不必另作物件的宣告即可直接使用。
  - 例如在前述之主程式中呼叫的類別方法或副程式，或是 Math 類別物件所提供的數學函數，或是利用 System 類別的 out.println 方法列印資料，均不需另外宣告便可直接呼叫使用。
  - 敘述語法如下(除了 static 外，其餘均和 Instance Method 一樣)：

```
存取敘述 static 傳回值型態 方法名稱 (參數列)
{
    程式敘述 ;
}
```

- **this**－this 在程式應用上、帶來許多方便。其意義為所屬類別之代名詞
  - this 可用於建構子間之呼叫：
    - 使用 this 呼叫建構子時，僅可在一建構子中、呼叫另一建構子；
    - this 之 statement 必須置於第一行、否則為錯誤編輯。
  - this 可用於成員方法中區別與引數同名的成員資料。
    - 假設成員資料中已經存在一個變數名稱 name，而成員方法或建構子中所傳進來的引數也叫作 name，此時 this.name 就代表類別中的成員 name，以和引數 name 做區隔。
- 建立類別範例－提供二手車的引擎號碼、顏色代號、製造商、型號及里程數

```
class Car {
// 定義類別內在成員資料
    private long engineNum ;
    private int color ;
    private String model ;
    private double mileage ;
    private String brand ;
// 預設空建構子，不需傳入引數
    public Car() {
    }
// 需要傳入引數的建構子
    public Car(int engineNum, int color, String model, int miles, String brand) {
// 利用 this 賦予和引數同名的內在成員資料初始值
        this.engineNum = engineNum ;
        this.color = color ;
```

```
            this.model = model ;
            this.mileage = miles ;
            this.brand = brand ;
        }
// 實例方法：設定引擎號碼
        public void setEngineNum (long num) {
            engineNum = num ;
        }
// 實例方法：設定車款
        public void setModel(String id) {
            model = id ;
        }
// 實例方法：設定車殼顏色代碼
        public void setColor (int color) {
            this.color = color ;
        }
// 實例方法：設定里程數
        public void setMileage(double miles) {
            mileage = miles ;
        }
// 實例方法：設定製造商
        public void setBrand(String name) {
            brand = name ;
        }
// 實例方法：取得引擎號碼
        public long getEngineNum() {
            return engineNum;
        }
// 實例方法：取得顏色代碼
        public int getColor() {
            return color;
        }
// 實例方法：取得車型款式
        public String getModel() {
            return model;
        }
// 實例方法：取得里程數
        public double getMileage() {
```

```
            return mileage;
        }
// 實例方法：取得製造商
        public String getBrand() {
            return brand;
        }
// 類別方法：傳回出產國
        public static String getNation() {
            String nation = "Made in Japan" ;
            return nation ;
        }
// 實例方法：列印二手車資訊
        public void printVehicle () {
            System.out.println("The Manufactor is " + getBrand()) ;
            System.out.println("The Model is " + model) ;
            System.out.println("Color Number is " + color) ;
            System.out.println("Engine Number is " + getEngineNum()) ;
            System.out.println("Mileage is " + mileage) ;
        }
}
```

- 利用主程式呼叫類別及執行結果

```
public class ClassTest {
        public static void main(String[] args) {
// 取得靜態類別方法
            System.out.println(Car.getNation()) ;
// 宣告類別物件並傳入資料
            Car car = new Car(82250080, 3, "Wish", 75008, "Toyota") ;
            car.printVehicle() ;      // 取得二手車資訊
// 設定新的資訊
            car.setBrand("Honda") ;
            car.setModel("Accord") ;
            car.setMileage(59225) ;
            car.setEngineNum(72320415) ;
            car.setColor(0) ;
// 取得靜態類別資訊 - 不需宣告類別物件
            System.out.println(Car.getNation()) ;
            car.printVehicle() ;      // 取得新的二手資訊
// 定義新的類別物件
```

```
        Car suv = new Car() ;
        suv.setBrand("Nissan") ;
        suv.setModel("X-Trail") ;
        System.out.println(suv.getNation()) ; //  以類別物件取得靜態類別方法資訊
        suv.printVehicle() ;
    }
}
-------------------------------------執行結果--------------------------
Made in Japan                   <-  靜態類別資料
The Manufactor is Toyota        |
The Model is Wish               |
Color Number is 3               |->  第一次傳入的資訊
Engine Number is 82250080       |
Mileage is 75008.0              |
Made in Japan                   <-  靜態類別資料
The Manufactor is Honda         |
The Model is Accord             |
Color Number is 0               |->  第二次傳入的資訊
Engine Number is 72320415       |
Mileage is 59225.0              |
Made in Japan                   <-  靜態類別資料
The Manufactor is Nissan        |->  新定義的類別物件
The Model is X-Trail            |
Color Number is 0               |->  未傳入引數者會傳回空值或 0
Engine Number is 0              |
Mileage is 0.0                  |
```

方法程序的重載(Overload)

- 如同建構子可以相同名稱傳入不同型態的變數，方法程序亦具備這樣的功能，即為重載。重載之定義為，在同一類別內有兩個(含)以上之方法程序、具有相同之名稱，但其宣告之類型或參數個數卻不相同，*但不可以出現有相同名稱與引數卻有不同類型宣告的方法程序(如此的覆載邏輯將使電腦無法判別該使用哪一個方法)*。
- Example－

```
// 類別部份
class MeanValue {
    public MeanValue() {
    }
```

```
    public double mean(double a, double b) {      // 1.
        double c = Math.sqrt(a*a + b*b) ;
        return c ;
    }
    public double mean(double[] a) {              // 1.
        double c = 0.0 ;
        for(int i=0; i<a.length; i++) {
            c = c + a[i]*a[i] ;
        }
        return Math.sqrt(c) ;
    }
    public int mean(double[] a, double[] b) {     // 1.
        double c = 0.0 ;
        for(int i=0; i<a.length; i++) {
            c = c + a[i]*b[i] ;
        }
        return (int)Math.sqrt(c) ;
    }
}
// 測試主程式部份
public class OverLoadTest {
    public static void main(String[] args) {      // 2.
        MeanValue mv = new MeanValue() ;
        double a = 1.5, b = 2.3 ;
        double[] c = {1.1, 2.2, 3.3, 4.4, 5.5} ;
        double[] d = {6.6, 7.7, 8.8, 9.9, 11.11} ;
        System.out.println(mv.mean(a, b)) ;
        System.out.println(mv.mean(c)) ;
        System.out.println(mv.mean(c, d)) ;
    }
}
--------------------------------執行結果--------------------------------
Method 1: 2.745906043549196
Method 2: 8.15781833580523
Method 3: 12
```

註解：

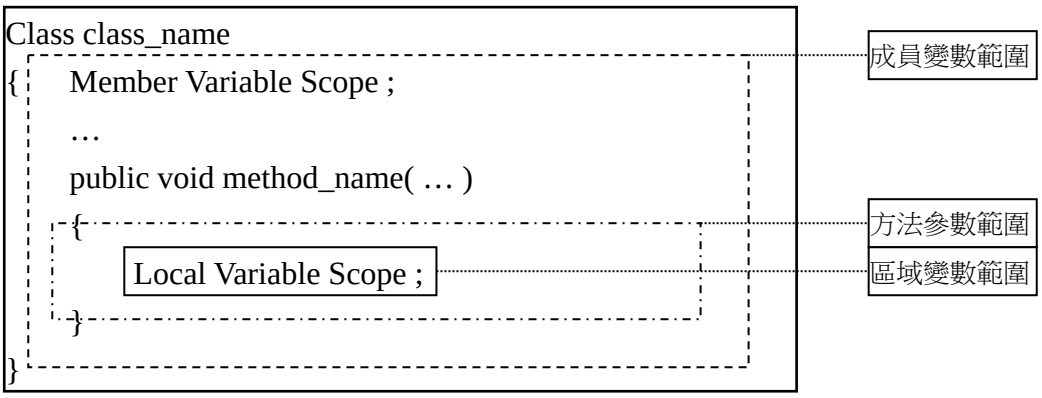
1. 類別中計算均值的三個 Method 具有相同之名稱 mean，但其宣告之資料型態不盡相同，有 int，也有 double；且輸入的引數也都不一樣，有純量，也有陣列。三個

Method 分別做不一樣的運算，且傳回不一樣的值。

2. 當主程式呼叫此 Method 時，則視引數型態來決定使用哪一個 mean 方法。

參數傳遞與變數範圍

- 區塊內的變數範圍(Scope)，亦可分為類別的成員變數、方法參數和區域變數，變數範圍會影響其變數值的存取。

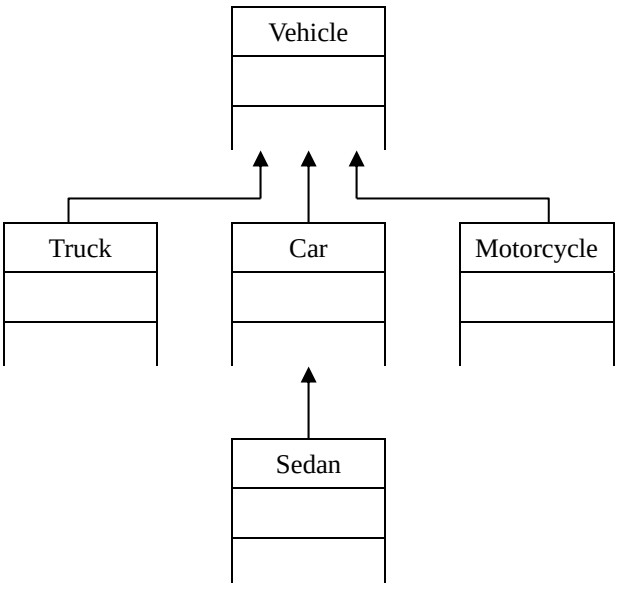


- **區域變數範圍(Local Variable Scope)**－在方法內宣告的變數，變數只能在宣告程式碼後的程式碼使用(不包括在宣告前)，在方法外的程式碼無法存取此變數。
- **方法參數範圍(Method Parameter Scope)**－這是傳入方法的參數列，其範圍是整個方法的程式區塊，在方法外的程式碼無法存取。
- **成員變數範圍(Member Variable Scope)**－不論是 static 宣告的類別變數或沒有宣告 static(物件的實例變數)，在整個類別的程式碼都可以存取此範圍的變數。

繼承(Inheritance)與覆寫(Override)

繼承(Inheritance)

- 繼承(Inheritance)讓類別得以包含現存類別的部分或全部成員資料和成原方法，且可新增成員或方法，甚至覆載或隱藏繼承類別的方法或變數。Java 的類別都是繼承 Object 這個宗祖類別，意即 Java 所宣告的類別都是屬於 Object 類別的子類別。
  - 子類別(Subclass)或延伸類別(Derived Class)：繼承其他的類別
  - 父類別(Superclass)或基礎類別(Base Class)：被繼承的別
  - 兄弟類別(Sibling Class)：彼此共同繼承一個父類別的子類別



- 以上圖之車系為例，交通工具父類別擁有卡車(Truck)、汽車(Car)、機車(Motorcycle)等三個子類別，而汽車對轎車(Sedan)而言則亦是父類別，故在 Car 類別中可定義如色彩(color)、引擎號碼(engineNum)等成員資料或方法，則在 Sedan 子類別中，可利用 **extends** 語法繼承 Car 父類別所定義的各種屬性：

```
class 子類別名稱 extends 父類別名稱
{
    ..... // 額外的成員資料和方法
}
```

  - 例如，Sedan 子類別的成員除了繼承 Car 類別所定義的各種資料，亦可增加顯示各種房車售價(頂級、中級或入門級價位)與車門樣式(兩門、三門、四門、五門)資訊的成員，

```
class Car {
    private long engineNum ;
    private int color ;
    public void setEngineNum(long num) { engineNum = num ;}
    public void setColor(int color) { this.color = color ;}
    .....
    public void printVehicle() { ..... }
}
class Sedan extends Car {
    private int door ;
    private double price ;
    public void setDoor(int door) {this.door = door ;}
    public void setPrice(double price) {this.price = price ;}
    ...
}
```

- 繼承的限制條件
  - 子類別不能存取父類別宣告成 private 的成員資料和方法。
  - 父類別的建構子不屬於子類別，所以子類別不可以繼承父類別的建構子，但可以呼叫父類別的建構子。
  - 每個子類別只能繼承一個父類別。

複寫(Override)和隱藏(Hide)

- 當父類別所定義的方法不適用於子類別時，可利用覆寫(Override)的概念在*子類別宣告同名、同參數及同傳回值但方法流程不同的方法*來取代；然而物件的實例方法(Instance Method)並不能取代宣告成 static 的類別方法(Class Method)，因此如欲取代父類別的類別方法時，子類別需要宣告同樣的類別方法來取代，即隱藏(Hide)。
  - 下例中，Car 父類別中，getNation()為 Class Method，printVehicle()為動態的

實例方法(Instance Method)，因此當 Vehicle 子類別繼承 Car 父類別的成員後想要對這些方法作修改，前者必須用 Hide 的方式(即成員方法中也要加上 static 宣告)，而後者則用 Override 的方式，此時若在主程式碼中宣告變數為此子類別的物件時，所使用的成員方法就不再是原父類別的方法了。

- 此外，父類別的成員資料也可以在子類別中做隱藏的動作，例如例中 Car 父類別中的 color 成員在 Sedan 子類別中被隱藏原來的屬性而變成 public String。
- 但是，父類別內的動態實例方法雖然可以被覆寫，卻不能像成員資料或靜態方法那樣被隱藏，*亦即要改變動態實例方法的資料型態，必須在同一個類別內的使用覆載(Overload)概念來處理。*

```
class Car
{     ……
      public static String getNation() {…運算 A…}
      public void printVehicle() {…運算 AA…}
      private int color ;
      ……
}
class Sedan extends Car
{     ……
      public static String getNation() {…運算 B…}
      public void printVehicle() {…運算 BB…}
      public String color ;
      ……
}
```

- 利用 **super** 呼叫父類別的建構子－子類別不可以繼承父類別的建構子，需要使用時，用 super 指令來呼叫；同理，子類別內覆寫和隱藏的成員變數也可以使用 super 來呼叫。
- Example－建立子類別 Sedan 繼承並複寫父類別 Car

```
class Sedan extends Car {
      /** Creates a new instance of Sedan */
      public Sedan(){                          // 1.
            super() ;
      }
      public Sedan(int color) {                // 1.
            super() ;
            this.setColor(color) ;
      }
      private double price ;                   // 2.
      private int door ;                       // 2.
```

```
    public String color ;                       // 3.
    public static String getNation() {          // 3.
        String nation = "Assembled in Taiwan" ;
        return nation ;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public int getDoor() {
        return door;
    }
    public void setDoor(int door) {
        this.door = door;
    }
    public void setColor(int color) {           // 1.
        super.setColor(color) ;
        switch(super.getColor()) {
            case 1:
                this.color = "red" ;
                break ;
            case 2:
                this.color = "black" ;
                break ;
            case 3:
                this.color = "green" ;
                break ;
            case 4:
                this.color = "blue" ;
                break ;
            case 5:
                this.color = "silver" ;
                break ;
            case 6:
                this.color = "golden" ;
                break ;
```

```
                default:
                    this.color = "white" ;
            }
        }
        public void printVehicle() {                    // 4.
            System.out.println("The Manufactor is " + getBrand()) ;
            System.out.println("The Model is " + super.getModel()) ;
            System.out.println("The Door Number is " + door) ;
            System.out.println("The Color is " + color) ;
            System.out.println("Engine Number is " + super.getEngineNum()) ;
            System.out.println("Mileage is " + super.getMileage()) ;
            System.out.println("The Price is " + price) ;
        }
}
// 測試類別之主程式部份
public class InheritOverrideTest {
        public static void main(String[] args) {
            Car car = new Car() ;                       // 5.
            car.setBrand("Nissan") ;
            car.setModel("X-Trail") ;
            car.setColor(6) ;
            car.setEngineNum(22053366) ;
            car.setMileage(59550) ;
            System.out.println("Before inheriting and overriding....") ;
            System.out.println("The car is " + car.getNation()) ;
            car.printVehicle() ;
            Sedan nissan = new Sedan(5) ;      // 6.
            nissan.setEngineNum(22053366) ;
            nissan.setModel("X-Trail") ;
            nissan.setBrand("Nissan") ;
            nissan.setMileage(60000) ;
            nissan.setColor(6) ;                        // 6.
            nissan.setPrice(69900) ;
            nissan.setDoor(5) ;
            System.out.println("After inheriting and overriding....") ;
            System.out.println("The sedan is " + nissan.getNation()) ;
            nissan.printVehicle() ;
            nissan.color = "red" ;                      // 7.
```

```
        System.out.println("After modifing public variable....") ;
        nissan.printVehicle() ;
    }
}
```

註解：

1. 繼承類別 class Car 為設置父類別，其中含有 color 變數以供輸入顏色代號；
   class Sedan 為繼承 Car 的子類別，定義了兩種建構子：其中一種需要輸入 Car 類別所使用的顏色代碼，兩種建構子均利用 super()指令呼叫父類別的 Car 建構子，以期在子類別中可以使用父類別所提供的方法。並建立且覆寫 Car 類別之 setColor()方法，覆寫後的 setColor()中利用 switch 控制流程將顏色代碼 color 值轉為顏色字串，因此當 color 的整數代碼透過 super.setColor(color)在父類別被定義後，再透過 super.getColor()傳回子類別內，並將名稱相同型態不同的變數改為顏色的名稱。
   如使用有傳入顏色代碼的建構子，便直接執行此方法；如使用空建構子，則要另外執行此方法來傳回顏色名稱
2. Sedan 子類別自行定義的建構子中利用隱藏的方式使用與父類別相同名稱的類別變數 color，並改為 public 成員(可由呼叫的程式從外部設定其值)。
3. 在子類別中隱藏父類別的靜態方法 getNation()，將輸出的字串改變。
4. 在子類別中覆寫父類別的動態方法 printVehicle()，除印出父類別中既有的成員資料，再加印自己本身新增的資料：車門數目、顏色名稱及售價。
5. 主程式中首先宣告父類別 Car 的物件變數，賦予相關資料並印出。
6. 在主程式中宣告子類別 Sedan 的物件變數，延用繼承自父類別的方法輸入資料，再利用子類別新增的方法輸入資料。雖然呼叫建構子時有設定了顏色代碼為 5，但後續又呼叫了 setColor 方法將代碼改為 6，建構子的顏色輸入值將被新值覆蓋。
7. 有別於 setColor 方法，直接利用子類別重新定義的 color 變數給予新的顏色名稱，再印出車輛資料。

執行結果如下：

```
Before inheriting and overriding....
The car is Made in Japan ---------------------- 父類別的靜態方法輸出值
The Manufactor is Nissan
The Model is X-Trail
Color Number is 6
Engine Number is 22053366
Mileage is 59550.0
After inheriting and overriding....
The sedan is Assembled in Taiwan---------- 被子類別隱藏掉後的靜態方法輸出值
The Manufactor is Nissan
The Model is X-Trail
```

```
The Door Number is 5  ---------- 新增加的子類別資料
The Color is golden ---------- 對應顏色代碼後的顏色名稱
Engine Number is 22053366
Mileage is 60000.0
The Price is 69900.0---------- 新增加的子類別資料
After modifing public variable....
The Manufactor is Nissan
The Model is X-Trail
The Door Number is 5
The Color is red---------- 改變後的顏色名稱
Engine Number is 22053366
Mileage is 60000.0
The Price is 69900.0
```

抽象(Abstract)類別與多形(Polymorphism)

常數(final)

- **final**－在程式中宣告為 final 的變數視為不可被改變的固定值常數，只能取出來使用，但不能變更其值。
  - 基於保密和設計的原因，可將類別或其中的方法宣告成 final，防止被繼承、存取或覆寫原類別的操作。
  - 例如以下之子類別 Customer 可以繼承 Person 父類別，但因 Customer 被定義成 final，故不能再被任何子類別繼承；而其父類別中有幾個成員方法也被宣告成 final，故當 Customer 繼承 Person 時，並無法覆寫這些成員方法的使用權。

```
class Person
{     ……
      final String Lname = “Lin” ;
      final char getName() { return name ;}
      final int getAge() { return age;}
      final int setName(char c) { name = c ;}
      final void setAge(int age) { this.age = age ;}
}
final class Customer extends Person
{     ……
}
```

- 測試 final 範例

```
class parentClass
```

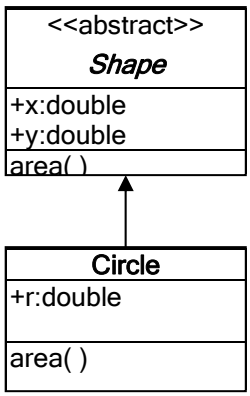
```
{
	int a ;
	public parentClass() {
		System.out.println("Parent: " + getInt()) ;
	}
	final int getInt() {
		setInt() ;
		return a ;
	}
	final int getInteger() {
		setInteger() ;
		return a ;
	}
	public void setInteger() {
		this.a = 5 ;
	}
//	private void setInt()	// Private member can only be used in parentClass
	public void setInt() {	// Public member can be overrided in childClass
		this.a = 10 ;
	}
}
class childClass extends parentClass
{
	public int b ;
	public childClass() {
		super() ;
		System.out.println("Child: " + this.a) ;
		System.out.println("Inherit: " + this.getInteger()) ;
		System.out.println("Override: " + this.getInt()) ;
		setInt() ;
		System.out.println("Child inherit: " + this.a) ;
		b = this.getInt() ;
		System.out.println("Child variable: " + b) ;
	}
	public void setInt() {
		this.a = 15 ;
	}
}
```

```
public class finalTest {
	public static void main(String args[]) {
		final float a = 2.5f ;
//		a = a + 1.5f ;  // Error: cannot assign a value to final variable a
		System.out.println("final variable:" + a) ;

		parentClass p = new parentClass() ;
		childClass c = new childClass() ;
		System.out.println("Parent Class: " + p.getInt()) ;
		System.out.println("Child Class: " + c.getInt()) ;
	}
}

/* ------------------Results for public void setInt() in parentClass:-------------
final variable:2.5
Parent: 10
Parent: 15
Child: 15
Inherit: 5
Override: 15
Child inherit: 15
Child variable: 15
Parent Class: 10
Child Class: 15
*/
/* ---------------- Results for private void setInt() in parentClass: -----------------
final variable:2.5
Parent: 10
Parent: 10
Child: 10
Inherit: 5
Override: 10
Child inherit: 15
Child variable: 10
Parent Class: 10
Child Class: 10
*/
```

抽象類別(abstract)



- 宣告為 abstract 的類別沒有建構子，因此不能用來建立物件，只能用來讓子類別繼承，就像是有身體卻沒有頭部的抽象形體，當類別擁有抽象方法，表示這一定是抽象類別。
  - 以右圖所示之 UML 類別圖，抽象類別 Shape 定義了(x,y)點座標並提供了計算面積的 area()抽象方法計算面積，但此方法**不存在程式碼區塊**，而利用子類別 Circle 繼承 Shape 來實作此抽象方法以計算圓面積。
  - 例中，Circle 除繼承了 Shape 的 x, y 作為圓心座標，亦新增半徑 r。(註：類別圖中，上層表類別名稱，且以<<abstract>>註明為抽象類別，第二層為成員變數，第三層為成員方法，並分別利用+, -, # 來代表 public, private 及 protected)
- 抽象類別雖然不能建立物件，但是可以宣告成參考子類別實例的物件變數，如
  Shape c2 = new Circle(10.0, 10.0, 7.0) ;
  - 因為 Shape 類別宣告的物件變數 c2 可以參考子類別的實例，所以 Java 程式可以使用 instanceof 運算子來判斷物件變數參考的是哪一個物件，如
    if (c2 instanceof Circle) { ..... }
    如果上式之 c2 參考了 Circle 物件，則會執行區塊語句。
  - 由於宣告成 abstract 的物件只可以參考，並無法呼叫或存取子類別新增的實例變數和方法，故需要先經過型態轉換的動作，例如
    Circle c ;
    c = (**Circle**) c2 ;
    亦即將原宣告為抽象類別 Shape 之物件變數 c2 轉為 Circle 子類別的物件變數，即可進行存取。
- Example

```
abstract class Shape {   // Shape 抽象類別宣告
    public double x;     // X 座標
    public double y;     // y 座標
    //  抽象方法: 計算面積
    abstract double area();                    // 1.
}
class Circle extends Shape {                   // 2. Circle 類別宣告
    public double r;     //  半徑
    public Circle(double x, double y, double r) {     //  建構子
        this.x = x;
        this.y = y;
        this.r = r;
    }
```

```
    public double area() {    // 成員方法: 實作抽象方法 area()
        return 3.1416*r*r ;
    }
}
// 主程式類別
public class AbstractTest {
    public static void main(String[] args) {     // 主程式
        Circle c;   // Circle 類別的物件變數
        // 宣告 Circle 類別型態的變數, 並且建立物件
        Circle c1 = new Circle(5.0, 10.0, 4.0);             // 3.
        Shape c2 = new Circle(10.0, 10.0, 7.0);             // 4.
        // 顯示圓形 c1 的資料
        System.out.println("圓形 c1 的資料 =====");
        System.out.println("X 座標: " + c1.x);
        System.out.println("Y 座標: " + c1.y);
        System.out.println("半徑: " + c1.r);
        // 呼叫物件的實例方法
        System.out.println("面積: " + c1.area());
        // 顯示圓形 c2 的資料, 檢查是否為 Circle 的實例
        if ( c2 instanceof Circle)                          // 5.
            System.out.println("----->c2 是 Circle 物件");
        System.out.println("圓形 c2 的資料 =====");
        System.out.println("X 座標: " + c2.x);
        System.out.println("Y 座標: " + c2.y);
        c = (Circle) c2;                                    // 6. 型別轉換
        System.out.println("半徑: " + c.r);
        System.out.println("面積: " + c2.area());    // 呼叫物件的實例方法
    }
}
```

註解：

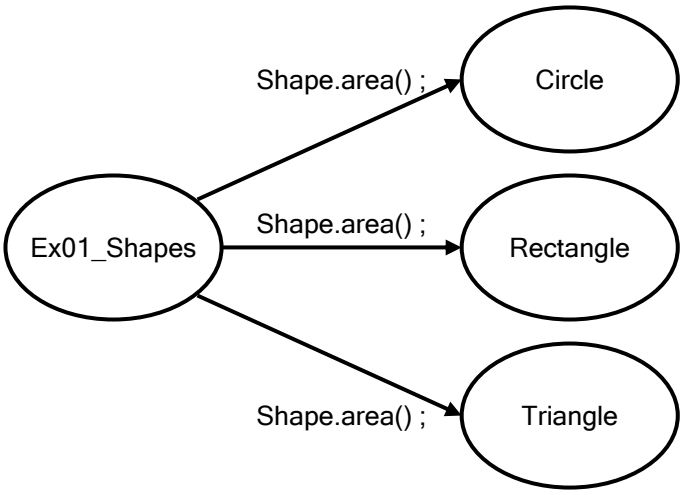
1. 抽象類別的宣告，內含抽象方法 area()
2. 繼承 Shape 類別的 Circle 子類別，新增變數 r，並實作抽象方法 area()；由於繼承了抽象類別 Shape，於是擁有了 Shape 內的公開成員變數 x 和 y 的使用權，因此可使用 this.x 與 this.y 取得 Circle 建構子傳來的引數。
3. 使用 Circle 類別宣告物件變數 c1，然後使用 new 運算子建立 Circle 物件
4. 使用 Shape 類別宣告物件變數 c2，然後使用 new 運算子建立 Circle 物件
5. if 條件檢查 Shape 類別宣告的物件變數是否是參考 Circle 物件
6. 型態轉換 c2 成為 Circle 類別宣告的物件變數，然後顯示 c.r 實例變數的值

多形(Polymorphism)

- Polymorphism 的目的在讓應用程式更容易擴充，不需要針對不同的資料型態分別建立類別，而是繼承一個基礎類別來建立同名方法，以處理不同的資料型態，如果有新的資料型態，則只須新增繼承的子類別。
- 多型的抽象類別宣告：利用上例的觀念，建立了 Shape 抽象類別定義了一個點座標及計算面積用的抽象方法 area()，而以 Circle 子類別繼承之，可以共用此方法計算圓面積，同理，也可同時建立 Triangle、Rectangle 子類別來計算三角形與矩形的面積。

```
abstract class Shape {   // Shape 抽象類別宣告
    public double x;          // x 座標
    public double y;          // y 座標
    abstract void area();    //  抽象方法: 計算面積
}
class Circle extends Shape       // Circle 類別宣告
{    ……
    public void area() { …… };    //  成員方法: 實作抽象方法 area()計算圓面積
}
class Triangle extends Shape          // Triangle 類別宣告
{    ……
    public void area() { …… };    //  成員方法: 實作抽象方法 area()計算三角形面積
}
class Rectangle extends Shape       // Rectangle 類別宣告
{    ……
    public void area() { …… };    //  成員方法: 實作抽象方法 area()計算矩形面積
}
```

- 同名異式的多型方法－在主程式中要使用這些實作抽象方法計算各種圖形的面積，可宣告 Shape 類別的物件變數 s，接著分別宣告各圖形 Circle, Rectangle, Triangle 之類別的物件變數 s1, s2, s3，此時，便可以共同的抽象方法名稱 area()計算各種圖形的面積。其中，area()方法就是多形，一個多形方法在執行時會依照實際參考的物件型態(Circle.area(), Rectangle.area(), Triangle.area())執行正確的實例方法。



➢ Example

```
abstract class Shape {   // Shape 抽象類別宣告
    public double x;     // 成員資料：X 座標
    public double y;     // 成員資料：y 座標
    // 抽象方法: 計算面積
    abstract double area();
}
class Circle extends Shape {    // Circle 類別宣告
    private double r;        //成員資料：半徑
    public Circle(double x, double y, double r) {     // 建構子
        this.x = x;
        this.y = y;
        this.r = r;
    }
    public double area() { // 成員方法: 實作抽象方法 area()
        return 3.1416*r*r ;
    }
}
class Rectangle extends Shape {    // Rectangle 類別宣告
    private double x1;        //成員資料：X 座標
    private double y1;        //成員資料：Y 座標
    public Rectangle(double x, double y, double x1,double y1) { // 建構子
        this.x = x;
        this.y = y;
        this.x1 = x1;
        this.y1 = y1;
    }
    public double area() { // 成員方法: 實作抽象方法 area()
        return (y1-y)*(x1-x) ;
    }
}
class Triangle extends Shape {       // Triangle 類別宣告
    private double x1;        // X 座標
    private double y1;        // Y 座標
    private double l;         // 三角形底長
    public Triangle(double x, double y, double x1, double y1, double l) { // 建構子
        this.x = x;
        this.y = y;
```

```
        this.x1 = x1;
        this.y1 = y1;
        this.l = l;
    }
    public double area() { // 成員方法: 實作抽象方法 area()
       return (y1-y)*l/2.0) ;
    }
}
// 主程式類別
public class PolymorphismTest {
     public static void main(String[] args) {     // 主程式
          Shape s;       // 抽象類別的物件變數
          // 宣告類別型態的變數, 並且建立物件
          Circle s1 = new Circle(5.0, 10.0, 4.0);
          Rectangle s2 = new Rectangle(10.0, 10.0, 20.0, 20.0);
          Triangle s3 = new Triangle(10.0, 10.0, 5.0, 25.0, 5.0);
          // 呼叫抽象類型物件的抽象方法 area()
          s = s1;        // 圓形
          System.out.println("圓面積" + s.area());              // 1.
          s = s2;        // 長方形
          System.out.println("長方形面積" + s.area());        // 2.
          s= s3;         // 三角形
          System.out.println("三角形面積" + s.area());        // 3.
     }
}
```

註解：

1. 相當於執行 Circle.area() ;
2. 相當於執行 Rectangle.area() ;
3. 相當於執行 Triangle.area() ;

介面(Interface)與套件(Package)

介面(Interface)

- Java 不像 C++一樣能支援一個類別繼承多個父類別的多重繼承(Multiple Inheritance)，但是可利用介面的實作來提供相同目的的多重繼承。
  - 宣告介面－介面是在類別繼承架構中定義類別行為，內含常數和方法宣告，但無實作的程式碼，其宣告的方式與抽象類別的差別如下：

```
public interface 介面名稱
{
```

```
    final 資料型態 常數 = 值 ;
    ……
    傳回值型態 介面方法 ( 參數列 ) ;
    ……
}
```

- 介面內的數值只能定義常數，不能定義沒有值的變數。
- 抽象類別的方法可能只有宣告，但是仍然可以擁有一般方法，*介面的方法都只有宣告*，而且一定沒有實作的程式碼。
- 與類別類似，同一支程式裡面可以有好幾個介面，但這些介面不能宣告成為 public，宣告成 public 的介面也必須以程式名稱命名。
- 介面不屬於類別的繼承架構，一個介面可以繼承多個介面，而毫無關係的類別也一樣可以實作同一個介面。
- *類別只能繼承一個抽象類別，但是可以同時實作多個介面，而實作介面中所宣告的每一個方法都必須予以處理。*

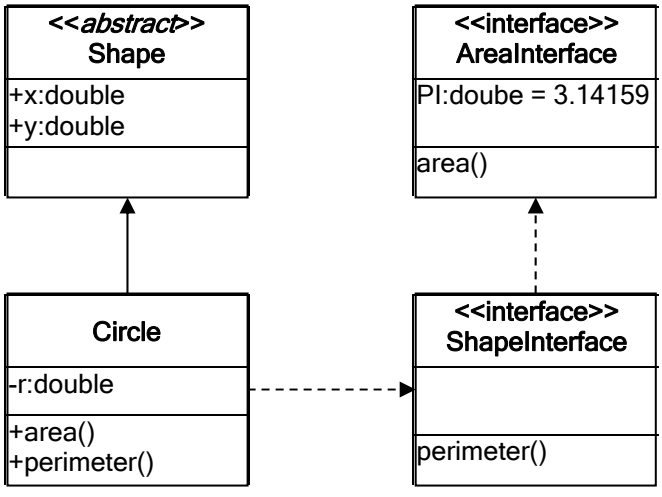
➢ 實作介面

```
class 類別名稱 implements 介面名稱 1, 介面名稱 2, …
{
    …… // 實作介面的方法
}
```

➢ 繼承介面

```
interface 介面名稱 extends 繼承的介面 1, 繼承的介面 2
{
    …… // 額外的常數和方法
}
```

- 利用前述實作抽象類別的例子，將原先在抽象類別 Shape 中的 area()方法在介面 AreaInterface 中宣告，接著新增 ShapeInterface 介面繼承 AreaInterface，內含計算周長的方法 perimeter()。
- Circle 類別在此繼承 Shape 的 x,y 座標，在植入了介面 ShapeInterface 時獲得方法 perimeter()，並透過其繼承了 AreaInterface 所定義的常數 PI 及方法 area()，於是新增一個變數 r(圓半徑)之後，可計算圓面積及圓周長。



➢ 實作介面基本範例

```
// ShapeInterface 介面宣告
interface ShapeInterface {
    final double PI = 3.1415926;                // 常數的宣告
    void show();                                // 介面方法: 印出相關參數
}
// Circle 類別宣告
class Circle implements ShapeInterface {
    private double r;                           // 成員資料：半徑
    public Circle(double r) {                   // 建構子
        this.r = r;
    }
    // 實作 ShapeInterface 中的 show 方法
     public void show() {
           System.out.println("PI=" + this.PI) ;
     }
    // 利用 getR()方法傳回私有變數 r 的值
     public double getR() {
           return r ;
     }
}
// 主程式類別
public class InterfaceTest {
    public static void main(String[] args) {// 主程式
          // 宣告 Circle 類別型態的變數，並且建立物件
          Circle c = new Circle(5.5);
          System.out.println("利用類別方法顯示相關參數") ;
          c.show() ;
          // 顯示介面的常數值
          System.out.println("利用介面取得 PI 常數: " + ShapeInterface.PI);
          System.out.println("利用物件變數取得圓周率: " + c.PI) ;
          System.out.println("利用物件變數取得圓半徑: " + c.getR()) ;
    }
}
------------------------------執行結果----------------------
利用類別方法顯示相關參數
PI=3.1415926
利用介面取得 PI 常數: 3.1415926
利用物件變數取得圓周率: 3.1415926
```

利用物件變數取得圓半徑: 5.5

➢ 介面及抽象類別實作範例

```
abstract class Shape {     // Shape 抽象類別宣告
    public double x;     // X 座標
    public double y;     // Y 座標
}
interface AreaInterface {    // AreaInterface 介面宣告
    double area();                              //  介面方法: 計算面積
}
Interface CircumInterface {    // CircumInterface 介面宣告
    double perimeter();                         //  介面方法: 計算周長
}
// ShapeInterface 介面宣告, 繼承 AreaInterface 與 CircumInterface
interface ShapeInterface extends AreaInterface, CircumInterface {
    final double PI = 3.1415926;                //  常數的宣告
    void show();                                //  介面方法: 印出相關參數
}
// Circle 類別宣告
class Circle extends Shape implements ShapeInterface {
    private double r;        //  成員資料：半徑
    public Circle(double x, double y, double r) {  //  建構子
        this.x = x;
        this.y = y;
        this.r = r;
    }
    //  實作 ShapeInterface 所繼承之 AreaInterface 介面的方法 area()
    public double area() {
            return this.PI * this.r * this.r ;
    }
    //  實作 ShapeInterface 所繼承之 CircumInterface 介面的方法 perimeter()
    public double perimeter() {
            return 2.0 * this.PI * this.r ;
     }
    //  實作 ShapeInterface 中的 show 方法
     public void show() {
            System.out.println("PI=" + this.PI) ;
            System.out.println("r=" + this.r) ;
            System.out.println("x=" + this.x) ;
```

```
            System.out.println("y=" + this.y) ;
        }
      // 利用 getR()方法傳回私有變數 r 的值
        public double getR() {
            return r ;
        }
}
// 主程式類別
public class InterfaceTest {
    public static void main(String[] args) { // 主程式
            // 宣告 Circle 類別型態的變數，並且建立物件
            Circle c = new Circle(10.0, 7.0, 5.5);
            // 顯示介面的常數值
            System.out.println("利用類別方法顯示相關參數") ;
            c.show() ;
            System.out.println("利用介面取得 PI 常數: " + ShapeInterface.PI);
            System.out.println("利用物件變數取得圓周率: " + c.PI) ;
            System.out.println("圓半徑: " + c.getR()) ;
            System.out.println("圓心: (" + c.x + " , " + c.y + ")") ;
            // 呼叫物件的介面方法 area()
            System.out.println("圓面積: " + c.area()) ;
            // 呼叫物件的介面方法 perimeter()
            System.out.println("圓周長: " + c.perimeter()) ;
    }
}
------------------------------執行結果----------------------
利用類別方法顯示相關參數
PI=3.1415926
r=5.5
x=10.0
y=7.0
利用介面取得 PI 常數: 3.1415926
利用物件變數取得圓周率: 3.1415926
圓面積: 95.03317615
圓周長: 34.5575186
圓半徑: 5.5
圓心: (10.0 , 7.0)
```

套件(Package)

- **套件(Package)**－將相關的類別和介面集合起來，可編譯成套件，如一般程式所使用的函式庫，利用 import 指令可把套件匯入程式中，便可使用套件中所有提供的物件。而 Java 標準的 API 即是一個名為 java 的大套件，並擁有數個子套件如 lang(所有語言基礎)、awt(執行 GUI 元件)、io(支援輸出與輸入)等。
- 類別與介面的管理－將原本獨立的 Java 程式宣告成 public 的類別或介面，並將主程式分離出來，利用 javac 編譯後，便可形成獨立的物件檔，此時只要單獨編譯主程式檔，仍可以達成相同的功能。
- 以下範例實作中建立幾個獨立的類別與介面：以 ShapeInterface 父介面宣常數及計算面積與周長的方法，讓 Shape 抽象類別繼承並增加參考點座標變數；而類別 Circle 及 Rectangle 分別實作介面提供面積與周長的計算，並利用主程式呼叫執行：

1. 建立介面 ShapeInterface.java

```
public interface ShapeInterface {     // ShapeInterface 介面宣告
    final double PI = 3.1415926;          // 常數的宣告
    void area();                          // 介面方法: 計算面積
    void perimeter();                     // 介面方法: 計算周長
}
```

2. 建立抽象類別 Shape.java

```
public abstract class Shape {   // Shape 類別宣告
    public double x;     //成員資料：X 座標
    public double y;     //成員資料：Y 座標
}
```

3. 建立一般類別 Circle.java

```
public class Circle extends Shape implements ShapeInterface {
    private double r;        //成員資料：半徑
    public Circle(double x, double y, double r) { //  建構子
        this.x = x;
        this.y = y;
        this.r = r;
    }
    public void area() { //  實作 ShapeInterface 介面的方法 area()
        System.out.println("圓面積: " + PI*r*r);
    }
    public void perimeter() { //  實作 ShapeInterface 介面的方法 perimeter()
        System.out.println("圓周長: " + 2.0*PI*r);
    }
}
```

4. 建立 Rectangle.java

```
public class Rectangle extends Shape implements ShapeInterface {
    private double x1;        //成員資料：X 座標
    private double y1;        //成員資料：Y 座標
    public Rectangle(double x, double y, double x1, double y1) {// 建構子
        this.x = x;
        this.y = y;
        this.x1 = x1;
        this.y1 = y1;
    }
    public void area() { // 實作 ShapeInterface 介面的方法 area()
        System.out.println("長方形面積: " + (y1-y)*(x1-x));
    }
    public void perimeter() { // 實作 ShapeInterface 介面的方法 perimeter()
        System.out.println("長方形周長: " + 2.0*((y1-y)+(x1-x)));
    }
}
```

5. 建立含主程式之類別 ShapePackage.java

```
public class ShapePackage
{   // 主程式
    public static void main(String[] args)
    {   // 宣告 Circle 類別型態的變數，並且建立物件
        Circle c = new Circle(5.0, 10.0, 4.0);
        Rectangle r = new Rectangle(5.0,5.0,10.0,10.0);
        // 呼叫物件的介面方法 area()
        c.area();
        r.area();
        // 呼叫物件的介面方法 perimeter()
        c.perimeter();
        r.perimeter();
    }
}
```

6. 執行過程

```
D:\Course_Data\Example\PackageTest>javac Shape.java
D:\Course_Data\Example\PackageTest>javac ShapeInterface.java
D:\Course_Data\Example\PackageTest>javac -cp .;D: Circle.java
D:\Course_Data\Example\PackageTest>javac -cp .;D: Rectangle.java

D:\Course_Data\Example\PackageTest>javac -cp .;D: ShapePackage.java
```

```
D:\Course_Data\Example\PackageTest>java ShapePackage
圓面積: 50.2654816
長方形面積: 25.0
圓周長: 25.1327408
長方形周長: 20.0
```

- 由於預設的 classpath 只有 Java 原始套件，故自訂類別時有繼承其他自定的類別或介面的情況下，編譯器會找不到這些類別的位置，此時可利用-cp 的編譯選項將所使用類別之路徑加上去。
- 如果在執行 javac 時沒有使用-cp 參數設定路徑，也可能通過編譯(即如例中所有的程式均在同一個路徑下)，但在以 java 執行程式時，就必須加上-cp 的參數設定路徑，方可讓主程式在執行時找到相關的類別檔。
- 其中，在路徑之前加上句點 .(或%classpath%)的目的是保留原來的已經設定在環境變數下的類別路徑，而在此處附加新的類別路徑上去。
- 或者可在進行編譯前利用 set classpath 的指令將新的類別路徑加上去，例如，假設程式及類別檔所在路徑為 D:\Course_Data\Example\PackageTest
  set CLASSPATH = ".;D:\ Course _Data\Example\PackageTest"
  設定完後就不需要在編譯時加上路徑了。

➢ 建立套件－利用 **package** 指令敘述，在每個獨立出來的 Java 類別程式前面加上套件名稱，即可將這些類別包含到該套件裡面。以上例中之 Shape.java 為例，假如需要包含至套件 Shapes 中，則為

```
package Shapes ;
public abstract class Shape
{
    ……
}
```

- 編譯套件的類程式時，將這些程式集合在同一個目錄底下，而該目錄名稱即是套件名稱，套件名稱與路徑和檔案所在目錄是彼此有關聯的！
- 套件中的子類別，如有繼承父類別或介面的情形，除非在實體目錄中也是對應存在於父類別所在之套件目錄下的自類別中，在編譯時要利用 **javac -classpath** 指令來指定套件父類別所在位置。

➢ 匯入套件－利用 **import** 指令敘述，將所需用到套件中的類別匯入執行的程式中。例如沿用前例，套件名稱為 Shapes，內含被獨立出來的 Circle 及 Rectangle 兩個類別，在主程式欲將此套件類別匯入，則有

```
import Shapes.Circle ;
import Shapes.Rectangle ;
public class ShapePackage{
    public static void main(String[] args) { //主程式
……
```

- 其中，亦可利用 import Shapes.* ;，一次把所有的套件類別全部匯入。

➢ Example－將前例中的類別製作成套件並編譯執行

1. 製作套件 Shapes：令套件中各類別所存在目錄全名為
   D:\Course_Data\Example\Shapes

/* 程式範例: Shape.java */

```
package Shapes;
public abstract class Shape     // Shape 類別宣告
{   // 成員資料
    public double x;     // X 座標
    public double y;     // y 座標
}
```

/* 程式範例: ShapeInterface.java */

```
package Shapes;
public interface ShapeInterface     // ShapeInterface 介面宣告
{   // 常數的宣告
    final double PI = 3.1415926;
    // 介面方法: 計算面積
    void area();
    // 介面方法: 計算周長
    void perimeter();
}
```

/* 程式範例: Circle.java */

```
package Shapes;
// Circle 類別宣告
public class Circle extends Shape implements ShapeInterface
{   // 成員資料
    private double r;        // 半徑
    // 建構子
    public Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    // 實作 ShapeInterface 介面的方法 area()
    public void area() {
        System.out.println("圓面積: " + PI*r*r);
    }
    // 實作 ShapeInterface 介面的方法 perimeter()
```

```
    public void perimeter() {
        System.out.println("圓周長: " + 2.0*PI*r);
    }
}
```

/* 程式範例: Rectangle.java */

```
package Shapes;
// Rectangle 類別宣告
public class Rectangle extends Shape implements ShapeInterface
{   // 成員資料
    private double x1;        // X 座標
    private double y1;        // Y 座標
    // 建構子
    public Rectangle(double x, double y, double x1, double y1) {
        this.x = x;
        this.y = y;
        this.x1 = x1;
        this.y1 = y1;
    }
    // 實作 ShapeInterface 介面的方法 area()
    public void area() {
        System.out.println("長方形面積: " + (y1-y)*(x1-x));
    }
    // 實作 ShapeInterface 介面的方法 perimeter()
    public void perimeter() {
        System.out.println("長方形周長: " + 2.0*((y1-y)+(x1-x)));
    }
}
```

2. 編譯套件各類別建立套件：其中 Circle 和 Rectangle 兩個子類別繼承了 Shape 抽象類別的變數和 ShapeInterface 介面的方法，故須以-classpath 或-cp 指定其套件位置。

```
D:\Course_Data\Example\Shapes>javac Shape.java
D:\Course_Data\Example\Shapes>javac ShapeInterface.java
D:\Course_Data\Example\Shapes>javac -cp .;D:\Course_Data\Example Circle.java
D:\Course_Data\Example\Shapes>javac -cp .;D:\Course_Data\Example Rectangle.java
```

3. 建立匯入套件類別之主程式

```
import Shapes.* ;
// import Shapes.Circle ;
// import Shapes.Rectangle ;
public class ShapePackage
```

```
{   // 主程式
    public static void main(String[] args)
    {   // 宣告 Circle 類別型態的變數，並且建立物件
        Circle c = new Circle(5.0, 10.0, 4.0);
        Rectangle r = new Rectangle(5.0,5.0,10.0,10.0);
        // 呼叫物件的介面方法 area()
        c.area();
        r.area();
        // 呼叫物件的介面方法 perimeter()
        c.perimeter();
        r.perimeter();
    }
}
```

4. (a)編譯並執行主程式－如主程式恰好在套件目錄(即 Shapes)上層

```
D:\Course_Data\Example>javac -cp .;D:\Course_Data\Example ShapePackage.java
D:\Course_Data\Example>java ShapePackage
圓面積: 50.2654816
長方形面積: 25.0
圓周長: 25.1327408
長方形周長: 20.0
```

4. (b)編譯並執行主程式－如主程式不在套件目錄(即 Shapes)上層或任意目錄

```
D:\Course_Data\Example\Shapes>javac -cp .;D:\Course_Data\Example
ShapePackage.java
D:\Course_Data\Example\Shapes>java -cp .;D:\Course_Data\Example
ShapePackage
圓面積: 50.2654816
長方形面積: 25.0
圓周長: 25.1327408
長方形周長: 20.0
```

- **protected** － 宣告的成員方法或變數可以在同一類別、其子類別或同一套件存取，其存取權限介於 publuc 與 private 之間。
  - public：擁有全域變數，任何類別都可以存取，包含子類別。
  - private：只可以在同一類別中存取，不可以在子類別中存取。
  - 不使用修飾子：預設範圍是同一個類別和套件中存取，但不包含不同套件的子類別，其存取範圍比 protected 小，因為 protected 可包含不同套件的子類別。

執行緒概念(Threads)

- 執行緒(Threads)－可視為一種 Lightweight process (輕量行程)，

- 能單獨存在或獨立執行，必須隸屬一個程式
- 程式啟動執行緒後，可作為幕後處理(daemon)
- 將程式分割成多個同步執行緒一起執行，稱之為平行程式設計(parallel programming)
- 單一應用程式擁有多個執行流程，稱之為多執行緒(multithreads)，即多工處理

➢ 建立執行緒方式

- 實作 Runnable 介面 – 建立物件類別實作 Runnable 介面之 run()方法

```
class ComputeClass
{     ……
      //一般執行計算的物件類別
      public long sum() {//計算方法 … }
      ……
}
class ComputeTread extends ComputeClass implements Runnable
{     //建構子
      public ComputeThread(…) { … }
      //實作 run()方法執行執行緒呼叫 ComputeClass 的方法(如 sum)
      public void run() { … }
}
public class RunnableTest
{     //主程式
      public static void main(String[] args)
      { … //建立執行緒
          ComputeThread ct = new ComputeThread( … );
          Thread t = new Thread(ct);
          t.start() ; //啟動執行緒
        …
      }
}
```

- 繼承 Thread 類別 – 直接繼承 Thread 類別複寫 run()方法

```
class ComputeTread extends Thread
{     //建構子
      public ComputeThread(…) { … }
      //複寫 run()方法執行執行緒
      public void run() { … }
}
public class ThreadTest
{     //主程式
```

```
    public static void main(String[] args)
    { … //建立執行緒
        ComputeThread ct = new ComputeThread( … );
        ct.start() ; //啟動執行緒
      …
    }
}
```

- Thread 物件之建構子與方法

建構子

Thread()	建立 Thread 物件，參數 String 是執行緒名稱，Runnable 是實作 Runnable 介面的物件
Thread(String)	
Thread(Runnable)	
Thread(Runnable, String)	

方法

int activeCount()	取得目前正在執行中的執行緒數目
Thread currentThread()	取得目前的執行緒物件
void sleep(long)	讓執行緒暫時停止執行一段時間，參數 long 為停滯的毫秒數
boolean isAlive()	檢查目前執行緒是否在執行中，傳回值 true 為是，false 為否
void start()	啟動執行緒
void setName(String)	將執行緒指定為參數 String 字串的名稱
String getName()	取得執行緒的名稱字串
String toString()	取得執行緒名稱、優先權和群組名稱的字串，預設執行緒群組是 main

- 以下程式實作 Runnable 介面，利用兩個執行緒累計執行停滯時間

```
public class RunnableTest {
//主程式: 呼叫 SumClass 啟動執行緒
    public static void main(String[] args) {
        System.out.print("執行緒: ");
        System.out.println(Thread.currentThread());//輸出目前執行緒物件
        //建立執行緒物件
        SumRunner st1 = new SumRunner(15, 1);
        Thread t1 = new Thread(st1, "執行緒 A");//定義執行緒 1
        SumRunner st2 = new SumRunner(20, 2);
        Thread t2 = new Thread(st2, "執行緒 B");//定義執行緒 2
        //啟動執行緒
        t1.start();
        t2.start();
    }
```

```
}

class SumClass {
	private long length = 100;//進行加總延遲時間的迴圈數(預設為 100)
	//建構子: 宣告時輸入要執行加總的迴圈數
	public SumClass(long length) {
		this.length = length;
	}
	public long sum(int threadID) {
		long temp = 0;//停滯時間總和
		for(int i=1; i<=length; i++) {
			int sleepduration = (int)(Math.random()*10);//隨機產生停滯時間
			try {//Stop running for a short moment
				Thread.sleep(sleepduration);//時間暫停
			} catch (Exception e) {}
			temp += sleepduration;
			System.out.println(threadID + ":" + i);//印出迴圈數(因啟動執行緒平行運
算, 執行緒代號會交互被輸出)
		}
		return temp;
	}
}
class SumRunner extends SumClass implements Runnable {
	private int threadID;
	//建構子: 繼承 SumClass 中的成員變數(length)
	public SumRunner(long length, int threadID) {
		super(length);
		this.threadID = threadID;
	}
	//實作執行緒方法
	public void run() {
		System.out.println(Thread.currentThread() + "延遲時間總和 = " +
sum(this.threadID) + "毫秒");
	}
}
```

```
/******************執行結果****************/
執行緒: Thread[main,5,main]
```

```
2:1
1:1
1:2
2:2
:
1:14
2:17
1:15
Thread[執行緒 A,5,main]延遲時間總和 = 77 毫秒
2:18
2:19
2:20
Thread[執行緒 B,5,main]延遲時間總和 = 84 毫秒
```

- 以下程式實作 Thread 類別，利用兩個執行緒累計執行停滯時間(執行結果同上)

```
public class ThreadTest {
    public static void main(String[] args) {
        System.out.print("執行緒: ");
        System.out.println(Thread.currentThread());//輸出目前執行緒物件
        //建立執行緒物件
        SumThread st1 = new SumThread(15, 1, "執行緒 A");//定義執行緒 1
        SumThread st2 = new SumThread(20, 2, "執行緒 B");//定義執行緒 2
        //啟動執行緒
        st1.start();
        st2.start();
    }
}

class SumThread extends Thread {
    private long length = 100;//進行加總延遲時間的迴圈數(預設為 100)
    private int threadID = 0;//執行緒編號，預設為 0

    //建構子: 繼承 SumClass 中的成員變數(length)
    public SumThread(long length, int threadID, String name) {
        super(name);
        this.threadID = threadID;
        this.length = length;
    }
    @Override
```

```
    public void run() {
        long temp = 0;//停滯時間總和
        for (int i = 1; i <= length; i++) {
            int sleepduration = (int) (Math.random() * 10);//隨機產生停滯時間
            try {//Stop running for a short moment
                Thread.sleep(sleepduration);//時間暫停
            } catch (Exception e) {
            }
            temp += sleepduration;
            System.out.println(threadID + ":" + i);//印出迴圈數(因啟動執行緒平行運
算，執行緒代號會交互被輸出)
        }
        System.out.println(Thread.currentThread() + "延遲時間總和 = " + temp + "毫
秒");
    }
}
```

- 應用執行緒建立幕後執行模式
  - 自動啟動命令提示模式下執行的程式
  - 建立幕後程式，執行 指令如: java DaemonFiling 0 test.txt
  - 繼承 Thread 類別並在 run()方法中利用 Runtime 產生執行緒
  - 使用 Process 類別(exec()方法)代入幕後執行指令(cmd)

    RunTime.getRunTime().exec(cmd);

```
import java.io.IOException;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

public class DaemonTest extends Thread {
    private String cmd;
    public static void main(String[] args) {
        // 呼叫物件類別並輸入幕後執行指令(註)
        DaemonTest run = new DaemonTest("java DaemonFiling 0 test.txt");
        try {
            run.start();//Start the thread
            run.setDaemon(true);//Send to daemon
        } catch (Exception e) {
            System.out.println("Running is terminated because process nothing");
        } finally {
```

```
                Scanner sc = new Scanner(System.in);
                if (run.isAlive()) {
                    System.out.println("Running is alive");
                }
                if (run.isDaemon()) {
                    System.out.println("Running is in daemon");
                }
                if (run.isInterrupted()) {
                    System.out.println("Running is interrupted");
                }
                System.out.println("Do you want to stop the thread process(y/n)?");
                if(sc.next().toLowerCase().charAt(0) == 'y') {
                    run.stop();
                    System.out.println("Stop the thread process");
                }
            }
        }
// 建構子
        public DaemonTest(String cmd) {// 讀取幕後執行指令
            this.cmd = cmd;
            System.out.println("Command: " + cmd);
        }
        @Override
        public void run() {//複寫 run()方法
            try {// 進行幕後執行程序
                Process proc = Runtime.getRuntime().exec(cmd);
            } catch (IOException ex) {// 例外處理(程式碼為 Netbeans 工具自動產生)
                Logger.getLogger(DaemonTest.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }

}
```

(註): DaemonFiling 類別可讀取或寫入一般文字資料檔，使用指令如下

java DaemonFiling 0 test.txt → 建立或更新 test.txt 文字檔，寫入現在時間

java DaemonFiling 1 test.txt → 刪除已存在之 test.txt 文字檔

```
import java.io.File;
import java.io.FileWriter;
```

```
import java.util.Date;

public class DaemonFiling {

	public static void main(String[] args) {
		// java DaemonFiling opt filename
		try {
			String opt = args[0];
			String filename = args[1];
			File f = new File(filename);
			if (opt.equals("0")) {//Create or Update file
				FileWriter fw;
				if (f.exists()) {
					fw = new FileWriter(f, true);
				} else {
					fw = new FileWriter(f);
				}
				Date date = new Date();
				fw.write(date.toString() + "\n");//Record current time
				fw.close();
				System.out.println("File is created or updated");
			} else {
				if(f.exists()) {//Delete file if file is existing
					f.delete();
				}
				System.out.println("File is deleted or absent");
			}
		} catch (Exception e) {//Throw the exception information
			System.out.println("0: Input filename to create or "
					+ "1: output filename to delete is required");
		}
	}

}
```

泛型類別(Generic Class)

泛型的概念

- 泛型類別使用時機：定義類別時，發現到好幾個類別的邏輯其實都相同，就只是當中所涉及的型態不一樣時，會增加不必要的檔案管理困擾。
- 可任意使用型態代號，如<E>，用來宣告一個型態持有者(Holder) E，之後可以用 E 作為型態代表來宣告變數名稱。
- 語法範例

```
public class GenericFoo<E> {
    private E foo;
    public void setFoo(E foo) {
        this.foo = foo;
    }
    public E getFoo() {
        return foo;
    }
}
```

  - 以上類別 GenericFoo 擁有泛型資料型態 E ，使用時可以以下方式宣告

```
GenericFoo<Boolean> foo1 = new GenericFoo<Boolean>();
GenericFoo<Integer> foo2 = new GenericFoo<Integer>();

foo1.setFoo(new Boolean(true));
Boolean b = foo1.getFoo();

foo2.setFoo(new Integer(10));
Integer i = foo2.getFoo();
```

  - 如果使用泛型類別，但宣告時不一併指定型態，則預設會使用 Object，此時要自己轉換物件的介面型態

泛型的應用

- 應用泛型類別置入不同資料型態的計算
  - 使用一般資料型態

```
public class GenericBasicTest {
    public GenericBasicTest() { }
    public static void main(String[] args) {
        //建立泛型物件，只允許輸入整數
        GenericFoo<Integer> foo_i = new GenericFoo<Integer>();
        //建立泛型物件，只允許輸入實數
```

```
            GenericFoo<Double> foo_d = new GenericFoo<Double>();
            foo_i.setFoo(100);
            foo_d.setFoo(100.0);
            Integer foo_i1 = foo_i.getFoo();
            Double foo_d1 = foo_d.getFoo();
            System.out.println(foo_i1 + " <-> " + foo_d1);
            foo_i.setIntegerFoo(100);
            int foo_i2 = foo_i.getFoo().intValue();//取得泛型物件計算後整數值
            System.out.println(foo_i2);
            foo_d.setDoubleFoo(100.0);
            double foo_i3 = foo_d.getFoo().doubleValue();//取得泛型物件計算後實數值
            System.out.println(foo_i3);
            int foo_i4 = foo_i.getIntegerFoo(100);
            System.out.println(foo_i4);
            double foo_i5 = foo_d.getDoubleFoo(100.0);
            System.out.println(foo_i5);
        }
}
//建立泛型物件
class GenericFoo<E> {
        private E foo;
        public void setFoo(E foo) {
            this.foo = foo;
        }
        public E getFoo() {
            return foo;
        }
        public void setIntegerFoo(E foo) {//以泛型計算整數資料
            this.foo = (E) (Object) (Integer.parseInt(foo.toString()) * 10);
        }
        public void setDoubleFoo(E foo) {//以泛型計算實數資料
            this.foo = (E) (Object) (Double.parseDouble(foo.toString()) * Math.PI);
        }
        public int getIntegerFoo(E foo) {//輸入泛型資料計算後傳回整數值
            return Integer.parseInt(foo.toString()) * 100;
        }
        public double getDoubleFoo(E foo) {//輸入泛型資料計算後傳回實數值
            return Double.parseDouble(foo.toString()) * Math.PI;
```

```
    }
}
```

- 應用於 List 抽象類別

```
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;
//建立泛型串列物件
public class GenericListTest {
//物件建構子
    public GenericListTest() {
    }
//物件主程式
    public static void main(String[] args) {
//宣告 List 抽象類別，只允許輸入字串 String 資料
//抽象類別必須透過實體類別實體化，在此例如 ArrayList, LinkedList, Vector 等等
        List<String> alist = new ArrayList<String>();
        alist.add("Snow");
        alist.add(0, "Ball");//將下一個字串放在前一個字串之前，字串依序後推一位
        Iterator it = alist.iterator();//宣告走訪串列物件
        while (it.hasNext()) {//走訪串列
            System.out.println(it.next());//輸出走訪資料
        }
        GenericListTest glt = new GenericListTest();//宣告泛型串列物件
        glt.setList(alist);
        int[] xx = {1, 2, 3, 4, 5};
        List blist = glt.getList(xx);
        it = blist.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
//物件方法：輸入任意型態的 List 建立串列
    public void setList(List<?> list) {//<?> 符號代表所有資料型態均可接受(可省略)
        Iterator it = list.iterator();
        while (it.hasNext()) {
            System.out.println("Set: " + it.next());
        }
    }
```

```
//物件方法：輸入整數陣列傳回整數型態串列
    public List<Integer> getList(int[] x) {
        List<Integer> list = new ArrayList<Integer>();//宣告只允許存放整數之串列
        for (int xi : x) {//利用 foreach 迴圈將整數置入串列中
            list.add(xi * 10);
        }
        return list;
    }
}
```

- 擴充泛型類別－繼承一泛型類別，保留其型態持有者，並新增型態持有者
  - 以下範例定義一泛型的父類別 MainGenericFoo，接著以子類別 SubGenericFoo 繼承

```
//建立父類別 MainGenericFoo
public class MainGenericFoo<T1, T2> {
    private T1 foo1;
    private T2 foo2;
    public void setFoo1(T1 foo1) {
        this.foo1 = foo1;
    }
    public T1 getFoo1() {
        return foo1;
    }
    public void setFoo2(T2 foo2) {
        this.foo2 = foo2;
    }
    public T2 getFoo2() {
        return foo2;
    }
}

//建立子類別 SubGenericFoo.java 擴充以上之父類別
public class SubGenericFoo<T1, T2, T3> extends GenericFoo<T1, T2> {
    private T3 foo3;
    public void setFoo3(T3 foo3) {
        this.foo3 = foo3;
    }
    public T3 getFoo3() {
```

```
            return foo3;
        }
}
//測試以上物件類別
public class GenericInheritTest {
        public static void main(String[] args) {
            SubGenericFoo<String, Integer, Double> sgf =
new SubGenericFoo<String, Integer, Double>();//宣告泛型類別，指定三種資料型態
            sgf.setFoo1("Hello Kitty");
            sgf.setFoo2(12);
            sgf.setFoo3(34.5);
            System.out.println(sgf.getFoo1() + " earns " + sgf.getFoo3() + " as " +
sgf.getFoo2() + " years old");
        }

}
```

- 實作泛型介面－實作一泛型介面，保留其型態持有者，並新增型態持有者
  - 以下範例定義一泛型的介面 GenericInterfaceFoo，接著以類別 GenericClassFoo 進行實作

```
//建立介面 GenericInterfaceFoo
interface GenericInterfaceFoo<T1, T2> {
        public void setFoo1(T1 foo1);
        public void setFoo2(T2 foo2);
        public T1 getFoo1();
        public T2 getFoo2();
}

//建立類別 GenericClassFoo 實作 GenericInterfaceFoo 介面
class GenericClassFoo<T1, T2> implements GenericInterfaceFoo<T1, T2> {
        private T1 foo1;
        private T2 foo2;
        public void setFoo1(T1 foo1) {
            this.foo1 = foo1;
        }
        public T1 getFoo1() {
            return foo1;
        }
```

```
        public void setFoo2(T2 foo2) {
            this.foo2 = foo2;
        }
        public T2 getFoo2() {
            return foo2;
        }
}

//測試以上物件類別與介面
public class GenericInterfaceTest {
        public static void main(String[] args) {
            GenericClassFoo<String, Boolean> gif =
new GenericClassFoo<String, Boolean>();
            gif.setFoo1("Steve Jobs");
            gif.setFoo2(false);
            System.out.println(gif.getFoo1() + " is alive? " + gif.getFoo2());
        }
}
```