

Java程式設計進階

Lambda語法與函式介面

鄭安翔

ansel.cheng@hotmail.com

課程大綱

1) **Lambda 語法**

- **Lambda 語法**
- 函數式程式設計
- 常見函式介面**Lambda**實作

2) 內建函式介面

物件導向 vs. 函數式程式設計

- 物件導向程式設計 **Object Orientated Programming**
 - 物件為基本單元的程式設計方式
 - 物件包含資料狀態及邏輯操作
 - 物件互動造成資料狀態改變
- 函數式程式設計 **Functional Programming**
 - 以函式(方法)為基本單元來組成程式
 - 只包含傳入參數及傳回結果
 - 資料中避免包含可變的資料狀態，不要造成副作用
 - 常見函式導向語言：**Haskell、Lisp、R、Scala**

物件導向 vs. 函數式程式設計

- 物件導向程式設計寫分析工具
 - 撰寫分析工具類別，建立物件後呼叫分析方法
 - 定義分析介面及實作類別
 - 建立實作物件後呼叫分析方法
- 函數式程式設計
 - 定義分析函式(類別方法)
 - 傳入分析所需參數及分析規則
 - 傳回分析結果

撰寫分析工具類別

```
public class Shirt {  
    private int shirtID = 0;  
    private char colorCode = 'G';  
    private String size = "XL";  
    private double price = 299.00;  
    private String description = "Polo Shirt";  
  
    public Shirt(int i, char c, String s,  
                double p, String d){  
        shirtID = i;    colorCode = c;  
        size = s;    price = p;  
        description = d;  
    }  
  
    public int getShirtID () { ... }  
    public char getColorCode () { ... }  
    public String getSize () { ... }  
    public double getPrice () { ... }  
    public String getDescription () { ... }  
  
    public String toString() { ... }  
}
```

```
public class ShirtAnalyzerTest1 {  
    public static void main(String[] args){  
        Shirt[] shirts = { new Shirt(1, 'R', "S", 200, "T-Shirt"),  
                            new Shirt(2, 'G', "S", 200, "T-Shirt"),  
                            new Shirt(3, 'B', "M", 250, "T-Shirt"),  
                            new Shirt(4, 'R', "M", 500, "Polo Shirt"),  
                            new Shirt(5, 'G', "L", 600, "Polo Shirt"),  
                            new Shirt(6, 'B', "L", 600, "Polo Shirt") };  
  
        System.out.println("===Green Color Shirt===");  
        findShirtByColor(shirts, 'G');  
        System.out.println("===M Size Shirt===");  
        findShirtBySize(shirts, "M");  
    }  
    public static void findShirtByColor(Shirt[] shirts, char color){  
        for(Shirt currentShirt : shirts)  
            if (currentShirt.getColorCode()==color)  
                System.out.println("Match: " + currentShirt);  
    }  
    public static void findShirtBySize(Shirt[] shirts, String size){  
        for(Shirt currentShirt : shirts)  
            if (currentShirt.getSize().equals(size))  
                System.out.println("Match: " + currentShirt);  
    }  
}
```

Strategy Design Pattern

- Strategy Design Pattern 策略模式
 - 物件中某個行為，在不同的場景中，有不同的實現演算法
 - 定義一個策略介面
 - 通常為只有一個方法的函式介面
 - 實作函式介面
 - 類別實作函式介面
 - 匿名類別實作函式介面
 - Lambda Expression 實作函式介面 (函數式程式設計Functional Programming)
 - 可以靈活抽換不同的策略行為

定義策略函式介面

■ 定義策略介面

```
public interface ShirtAnalyzer {  
    public boolean analyze(Shirt shirt);  
}
```

- 只有 1 個 method
- 需要輸入 1 個衣服參數
- 回傳 boolean 值
- 保留未來擴充性

類別實作函式介面

```
@FunctionalInterface
public interface ShirtAnalyzer {
    public boolean analyze(Shirt shirt);
}
```

```
public class MSizeAnalyzer implements ShirtAnalyzer {
    @Override
    public boolean analyze(Shirt shirt){
        return shirt.getSize().equals("M");
    }
}
```

```
public class GreenColorAnalyzer implements ShirtAnalyzer {
    @Override
    public boolean analyze(Shirt shirt){
        return shirt.getColorCode()=='G';
    }
}
```

```
public class ShirtAnalyzerTest2 {
    public static void main(String[] args){
        Shirt[] shirts = { new Shirt(1, 'R', "S", 200, "T-Shirt"),
                           new Shirt(2, 'G', "S", 200, "T-Shirt"),
                           new Shirt(3, 'B', "M", 250, "T-Shirt"),
                           new Shirt(4, 'R', "M", 500, "Polo Shirt"),
                           new Shirt(5, 'G', "L", 600, "Polo Shirt"),
                           new Shirt(6, 'B', "L", 600, "Polo Shirt") };

        System.out.println("===Green Color Shirt===");
        findShirtByAnalyzer(shirts, new GreenColorAnalyzer());
        System.out.println("===M Size Shirt===");
        findShirtByAnalyzer(shirts, new MSizeAnalyzer());
    }

    public static void findShirtByAnalyzer(Shirt[] shirts,
                                           ShirtAnalyzer analyzer){
        for(Shirt currentShirt : shirts) {
            if (analyzer.analyze(currentShirt))
                System.out.println("Match: " + currentShirt);
        }
    }
}
```


匿名類別實作函式介面

@FunctionalInterface

```
public interface ShirtAnalyzer {  
    public boolean analyze(Shirt shirt);  
}
```

```
public class ShirtAnalyzerTest3 {  
    public static void main(String[] args){  
        Shirt[] shirts = { new Shirt(1, 'R', "S", 200, "T-Shirt"),  
                            new Shirt(2, 'G', "S", 200, "T-Shirt"),  
                            new Shirt(3, 'B', "M", 250, "T-Shirt"),  
                            new Shirt(4, 'R', "M", 500, "Polo Shirt"),  
                            new Shirt(5, 'G', "L", 600, "Polo Shirt"),  
                            new Shirt(6, 'B', "L", 600, "Polo Shirt") };  
  
        System.out.println("===Green Color Shirt===");  
        findShirtByAnalyzer(shirts, new ShirtAnalyzer(){  
            public boolean analyze(Shirt shirt) {  
                return shirt.getColorCode()=='G';  
            }  
        });  
  
        System.out.println("===M Size Shirt===");  
        findShirtByAnalyzer(shirts, new ShirtAnalyzer(){  
            public boolean analyze(Shirt shirt) {  
                return shirt.getSize().equals("M");  
            }  
        });  
    }  
  
    public static void findShirtByAnalyzer (Shirt[] shirts,  
                                           ShirtAnalyzer analyzer){  
        for(Shirt currentShirt : shirts) {  
            if (analyzer.analyze(currentShirt))  
                System.out.println("Match: " + currentShirt);  
        }  
    }  
}
```

實作分析工具比較

- 撰寫一般類別
 - 若分析的需求樣式很多，類別必須增加其他方法
- 一般類別實作介面interface
 - 以實作類別擴充/增加分析樣式
 - 符合 OCP (open for extension, close for modification) 法則
- 使用匿名類別實作介面interface
 - 類別不需命名
 - 寫法複雜
 - 類別編譯後會加上數字編號，辨識來源困難
- 使用Lambda Expression實作介面interface

Lambda 表示式

■ Lambda 表示式

- Lambda (λ) 是一個希臘字母
- 用來宣告匿名 (沒有名稱) 函式
 - 通常只包含一個運算式
 - 可以有一個傳回值，即運算式的運算結果
- 搭配 **Functional Interface** 使用
 - 將 **Lambda Expression** 指定到一個函式介面的變數
 - 使用方法定義來取代匿名類別定義

Lambda 表示式語法

■ Lambda 表示式語法

(arguments) -> { body }

方法參數 箭頭符號 方法內容

□ 傳入參數：(type1 arg1, type2 arg2, ...)

- 方法不帶參數以 () 表示
- 只有1個參數時，可省略()
- 參數型態可自動推斷時可省略型態宣告

□ 方法內容：{ }

- 方法傳回值可為void或任意型態資料
- 一般(多行)表達：需有大括號{ }，程式碼用分號(;)區隔，傳回值前加return
- 省略(單行)表達：大括號{ }可省略，程式碼後不需分號;，傳回值不加return
- 方法可宣告區域變數，但不可與傳入參數同名

範例	結果	說明
<code>(int x, int y) -> {return x + y;}</code>	OK	Body回傳int
<code>(int x, int y) -> x + y</code>	OK	Body只有1行時，可以去除 { }; return
<code>(x, y) -> x + y</code>	OK	Java自動推斷參數型態，可以省略型態宣告
<code>(String s) -> { return s.contains("word"); }</code>	OK	Body回傳boolean
<code>(String s) -> s.contains("word")</code>	OK	Body只有1行時，可以去除 { }; return
<code>s -> s.contains("word")</code>	OK	只有1個參數時，可以去除()
<code>() -> Math.random()</code>	OK	方法不帶參數
<code>(x, y) -> System.out.println(x + y)</code>	OK	Body無傳回值
<code>(x, y) -> { System.out.println(x + y); return x+y; }</code>	OK	Body超過1行時，程式碼用分號(;)區隔，傳回值前加return，不可去除 { }
<code>a, b -> a.startsWith(b)</code>	NG	2個以上參數需有()
<code>a -> { return a.startsWith("test") }</code>	NG	方法內容加上{}後，每行程式碼都要有分號(;)區隔
<code>(a, b) -> { int a = 0; return 5; }</code>	NG	方法內容裡的區域變數名稱不能和參數相同

Lambda 宣告匿名函式

■ Lambda 宣告匿名函式

- 沒有方法名稱，類別中方法的定義可以被移除
- 若類別中只有這個方法，連類別的定義都可以省下來！
- 語法簡潔適用多重實作
 - 不需另外撰寫實作類別
 - 直接在方法呼叫時以Lambda Expression建立實作的物件
- 抽換實作內容
 - 建立一個介面型態的變數，以Lambda Expression建立實作物件儲存於變數中
 - 不需修改呼叫的方法

Lambda 表示式取代類別定義

```
@FunctionalInterface  
public interface ShirtAnalyzer {  
    public boolean analyze(Shirt shirt);  
}
```

傳統類別實作

Lambda表示式

```
shirt.getColor() == 'G'
```

類別只使用一次,不需命名
介面 **ShirtAnalyzer** 只有一個方法
方法的名稱,參數型態,個數及
回傳型態皆可推論。
簡化變數名稱：
shirt => s

```
ShirtAnalyzer analyzer1 =  
    new GreenColorAnalyzer();
```

以方法定義，
取代類別定義

```
ShirtAnalyzer analyzer1 =  
    s -> s.getColor() == 'G'
```

Lambda 宣告匿名函式

@FunctionalInterface

```
public interface ShirtAnalyzer {  
    public boolean analyze(Shirt shirt);  
}
```

findShirtByAnalyzer() 方法可判斷
第 2 個參數為ShirtAnalyzer型態

ShirtAnalyzer 為Functional介面
只有analyze()方法

analyze()方法的傳入參數為String
簡化變數名稱

```
findShirtByAnalyzer(shirts, new ShirtAnalyzer(){  
    public boolean analyze(Shirt shirt) {  
        return shirt.getColorCode() == 'G';  
    }  
});
```



```
findShirtByAnalyzer(shirts,  
    public boolean analyze(Shirt shirt) {  
        return shirt.getColorCode() == 'G';  
    }  
);
```



```
findShirtByAnalyzer(shirts,  
    (Shirt shirt) -> { return shirt.getColorCode() == 'G'; }  
);
```



```
findShirtByAnalyzer(shirts, s -> s.getColorCode() == 'G' );
```


使用Lambda Expression實作函式介面

```
@FunctionalInterface
public interface ShirtAnalyzer {
    public boolean analyze(Shirt shirt);
}
```

```
public class ShirtAnalyzerTest4 {
    public static void main(String[] args){
        Shirt[] shirts = { new Shirt(1, 'R', "S", 200, "T-Shirt"),
                           new Shirt(2, 'G', "S", 200, "T-Shirt"),
                           new Shirt(3, 'B', "M", 250, "T-Shirt"),
                           new Shirt(4, 'R', "M", 500, "Polo Shirt"),
                           new Shirt(5, 'G', "L", 600, "Polo Shirt"),
                           new Shirt(6, 'B', "L", 600, "Polo Shirt") };

        System.out.println("===Green Color Shirt===");
        findShirtByAnalyzer(shirts, s -> s.getColorCode()=='G' );

        System.out.println("===M Size Shirt===");
        findShirtByAnalyzer(shirts, s -> s.getSize().equals("M") );
    }

    public static void findShirtByAnalyzer(Shirt[] shirts,
                                           ShirtAnalyzer analyzer){
        for(Shirt currentShirt : shirts) {
            if (analyzer.analyze(currentShirt))
                System.out.println("Match: " + currentShirt);
        }
    }
}
```

Functional Interface

- Functional Interface 函式介面
 - 介面中只有一個抽象方法
 - Java 8 新增
 - @FunctionalInterface 標註
 - 可檢查介面是否符合Functional Interface
 - 不加標註不影響是否為函式介面
 - 支援 Functional Programming 函式程式設計
 - 只定義方法的使用，不在意物件建構及屬性狀態
 - 物件導向語言的物件，包含物件的屬性狀態和方法

Functional Interface

- 函式介面除了一個抽象方法之外
 - 可以有其他 **static** 方法
 - 可以有其他 **default** 方法
 - 可以有其他繼承自 **Object**類別的方法
 - `boolean equals(Object obj)`
 - `int hashCode()`
 - `String toString()`

JDK 常見函式介面

■ Java 8之前稱為Single Abstract Method (SAM)

- ❑ java.lang.Comparable
- ❑ java.lang.Runnable
- ❑ java.util.concurrent.Callable

```
package java.lang;  
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

```
package java.lang;  
@FunctionalInterface  
public interface Comparable<T> {  
    public abstract int compareTo(T o);  
}
```

```
package java.util.concurrent;  
@FunctionalInterface  
public interface Callable<T> {  
    public abstract T call() throws Exception;  
}
```

Lambda運算式取代匿名/內部類別

■ java.lang.Runnable 介面

方法名稱	傳回值	說明
<i>run()</i>	void	定義多執行緒的執行內容

■ java.util.Comparator<T> 介面

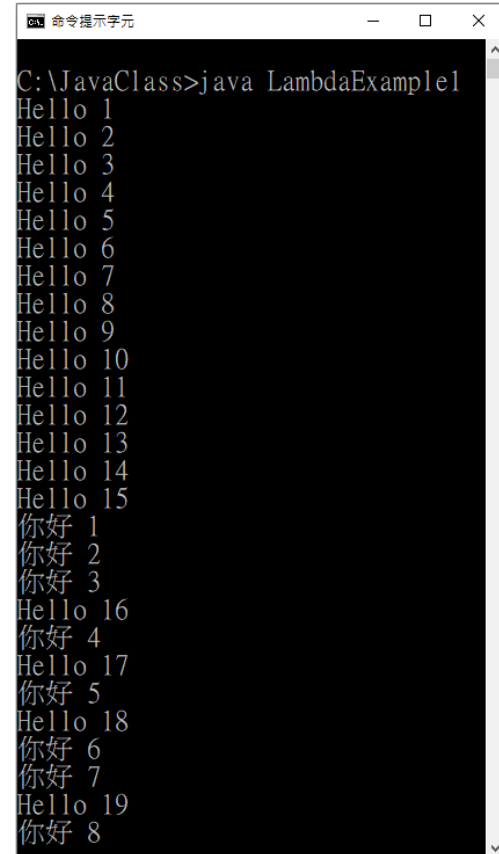
方法名稱	傳回值	說明
<i>compare(T o1, T o2)</i>	int	自訂兩個傳入物件的排序規則

Lambda運算式取代匿名/內部類別

```
public class LambdaExample1 {  
    public static void main(String[] args) {  
        Runnable r1 = new Runnable(){  
            public void run(){  
                for (int i=1; i<=100; i++)  
                    System.out.println("Hello "+i);  
            }  
        };  
        new Thread(r1).start();  
    }  
}
```

Lambda Expression 定義多執行緒內容

```
Runnable r2 = () -> {  
    for(int i=1; i<=100; i++)  
        System.out.println("你好 "+i);  
};  
new Thread(r2).start();
```



```
C:\JavaClass>java LambdaExample1  
Hello 1  
Hello 2  
Hello 3  
Hello 4  
Hello 5  
Hello 6  
Hello 7  
Hello 8  
Hello 9  
Hello 10  
Hello 11  
Hello 12  
Hello 13  
Hello 14  
Hello 15  
你好 1  
你好 2  
你好 3  
Hello 16  
你好 4  
Hello 17  
你好 5  
Hello 18  
你好 6  
你好 7  
Hello 19  
你好 8
```

Lambda 運算式範例

```
public class Student implements Comparable {
    String firstName, lastName;
    int studentID=0;
    double GPA = 0.0;
    public Student(String first, String last, int ID, double gpa){
        this.firstName = first;
        this.lastName = last;
        this.studentID = ID;
        this.GPA = gpa;
    }
    public String firstName() { return firstName; }
    public String lastName() { return lastName; }
    public int studentID() { return studentID; }
    public double GPA() { return GPA; }
    public int compareTo(Object o){
        double f = GPA - ((Student)o).GPA();
        if ((f==0))
            return 0; //0 表示相等
        else if(f < 0)
            return -1; //-1表示小於或排序在前
        else
            return 1; //+1表示大於或排序在後
    }
}
```

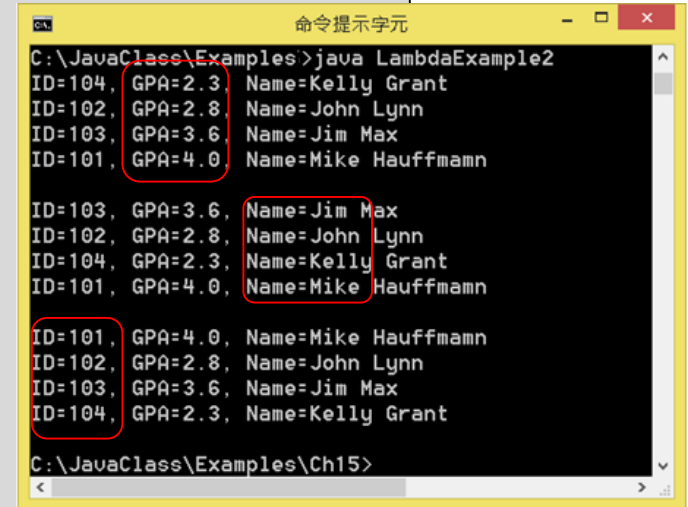
```
import java.util.*;
public class NameComp implements Comparator {
    public int compare(Object o1, Object o2){
        return ((Student)o1).firstName.compareTo(((Student)o2).firstName);
    }
}
```

Lambda 運算式範例

```
import java.util.*;
public class LambdaExample2 {
    public static void printList(List studentList){
        for(Object obj : studentList){
            Student s = (Student) obj;
            System.out.println( "ID="+s.studentID() + ", GPA="+s.GPA() +
                                ", Name=" + s.firstName() + " " + s.lastName());
        }
    }
    public static void main(String[] args) {
        List studentList = new ArrayList();
        studentList.add(new Student("Mike", "Hauffmamn", 101, 4.0));
        studentList.add(new Student("John", "Lynn", 102, 2.8));
        studentList.add(new Student("Jim", "Max", 103, 3.6));
        studentList.add(new Student("Kelly", "Grant", 104, 2.3));
        Collections.sort(studentList);
        printList(studentList);

        Comparator c = new NameComp();
        Collections.sort(studentList, c);
        printList(studentList);

        Collections.sort(studentList, (s1, s2) -> ((Student)s1).studentID()-((Student)s2).studentID() );
        printList(studentList);
    }
}
```



```
命令提示字元
C:\JavaClass\Examples>java LambdaExample2
ID=104, GPA=2.3, Name=Kelly Grant
ID=102, GPA=2.8, Name=John Lynn
ID=103, GPA=3.6, Name=Jim Max
ID=101, GPA=4.0, Name=Mike Hauffmamn

ID=103, GPA=3.6, Name=Jim Max
ID=102, GPA=2.8, Name=John Lynn
ID=104, GPA=2.3, Name=Kelly Grant
ID=101, GPA=4.0, Name=Mike Hauffmamn

ID=101, GPA=4.0, Name=Mike Hauffmamn
ID=102, GPA=2.8, Name=John Lynn
ID=103, GPA=3.6, Name=Jim Max
ID=104, GPA=2.3, Name=Kelly Grant

C:\JavaClass\Examples\Ch15>
```


Lambda運算式取代匿名/內部類別

■ java.util.Collection 介面 (ArrayList 的父介面)

方法名稱	傳回值	說明
removeIf(Predicate<? super E> filter)	default boolean	刪除集合中滿足指定判定條件的元素
replaceAll(UnaryOperator<E> operator)	void	將集合元素用指定運算的結果替換

■ java.util.function.Predicate<T> 介面

方法名稱	傳回值	說明
<i>test</i> (T t)	<i>boolean</i>	判定：依傳入參數依判定布林結果傳回

■ java.util.function.UnaryOperator<T> 介面

方法名稱	傳回值	說明
<i>apply</i> (T t)	<i>T</i>	函式：將傳入參數依指定函式運算後傳回相同型態運算結果

Wildcard in Generics

- generics (泛型) 中使用Wildcards (萬用字元)
 - ? **super** T
 - 型別必須是 T 或 T的 父類別
 - ? **extends** T
 - 型別必須是 T 或 T的 子類別

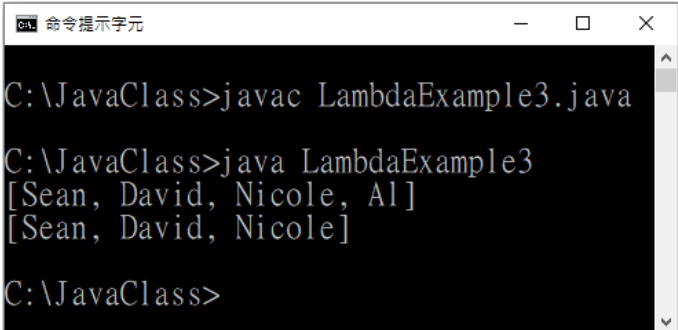
Lambda 運算式範例

```
import java.util.ArrayList;

public class LambdaExample3{
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<>();
        myList.add("Sean");
        myList.add("David");
        myList.add("Amy");
        myList.add("Nicole");
        myList.add("Al");

        myList.removeIf(s->s.equals("Amy"));
        System.out.println(myList.toString());

        myList.removeIf(s->s.length()<3);
        System.out.println(myList.toString());
    }
}
```



```
C:\JavaClass>javac LambdaExample3.java

C:\JavaClass>java LambdaExample3
[Sean, David, Nicole, Al]
[Sean, David, Nicole]

C:\JavaClass>
```

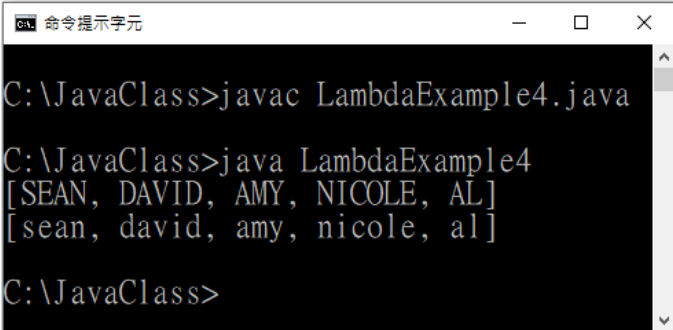
Lambda 運算式範例

```
import java.util.ArrayList;

public class LambdaExample4{
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<>();
        myList.add("Sean");
        myList.add("David");
        myList.add("Amy");
        myList.add("Nicole");
        myList.add("Al");

        myList.replaceAll(s->s.toUpperCase());
        System.out.println(myList.toString());

        myList.replaceAll(s->s.toLowerCase());
        System.out.println(myList.toString());
    }
}
```



A screenshot of a Windows command prompt window titled "命令提示字元". The window shows the following commands and output:

```
C:\JavaClass>javac LambdaExample4.java

C:\JavaClass>java LambdaExample4
[SEAN, DAVID, AMY, NICOLE, AL]
[sean, david, amy, nicole, al]

C:\JavaClass>
```

課程大綱

- 1) Lambda 語法
- 2) 內建函式介面
 - 基本函式介面
 - 變形函式介面

內建 Function Interfaces

■ Java 8 新增 function interfaces

□ java.util.function 套件

□ 將常用傳入參數及傳回值組合定義為函式介面

- Function(函式)：將傳入參數 T ，轉換為傳回值 R
- Predicate(判定)：依傳入參數 T ，判定傳回 **boolean** 結果
- Consumer(消費)：使用傳入參數 T 運算，無傳回值 **void**
- Supplier(提供)：提供 T 型別的物件實體傳回

□ 使用Lambda Expression 定義函式介面

內建 Function Interfaces 範例

```
import java.util.*;
public class Person {
    private String name, email;    private int age;
    public Person() {}
    public Person(String name, String email, int age) {
        this.name = name;    this.age = age;    this.email = email;
    }
    public String getName() {    return name;    }
    public int getAge() {    return age;    }
    public String getEmail() {    return email;    }
    @Override
    public String toString() {        return "Name=" + name + ", Age=" + age + ", email=" + email;    }
    public void printPerson() {    System.out.println(this);    }
    public static List<Person> createList() {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Bob", "bob@gmail.com", 21));
        people.add(new Person("Jane", "jane@gmail.com", 34));
        people.add(new Person("John", "johnx@gmail.com", 25));
        people.add(new Person("Phil", "phil@gmail.com", 65));
        people.add(new Person("Betty", "betty@gmail.com", 55));
        return people;
    }
}
```

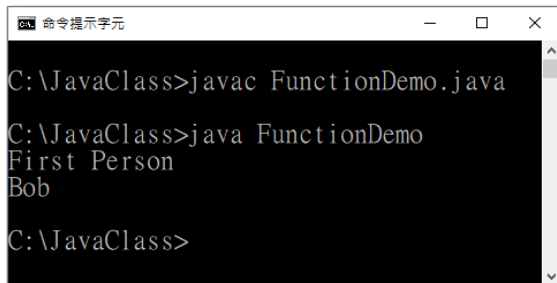
java.util.function.Function<T,R>

方法名稱	傳回值	說明
<code>apply(T t)</code>	<code>R</code>	函式：將傳入參數依指定函式運算後傳回
<code>andThen(Function<? super R, ? extends V> after)</code>	default <V> <code>Function<T,V></code>	傳回一個組合函式，先將此函式應用於其輸入，然後將結果作為輸入應用於 after 函式
<code>compose(Function<? super V, ? extends T> before)</code>	default <V> <code>Function<V,R></code>	傳回一個組合函式，先將輸入應用於 before 函式，然後將其結果作為輸入應用於此函式。
<code>identity()</code>	static <T> <code>Function<T,T></code>	傳回一個傳回其輸入參數的函式。

Function<T,R>

```
package java.util.function

public interface Function<T, R> {
    public R apply(T t);
}
```



```
命令提示字元
C:\JavaClass>javac FunctionDemo.java
C:\JavaClass>java FunctionDemo
First Person
Bob
C:\JavaClass>
```

```
public class FunctionDemo {
    public static void main(String[] args) {
        List<Person> pl = Person.createList();
        Person first = pl.get(0);
        // 使用Lambda Expression
        Function<Person, String> getNameFromPerson
            = p -> p.getName();

        System.out.println("First Person");
        System.out.println(getNameFromPerson.apply(first));
    }
}
```

java.util.function.Predicate<T>

方法名稱	傳回值	說明
<i>test</i> (T t)	<i>boolean</i>	判定：依傳入參數依判定布林結果傳回
and(Predicate<? super T> other)	default Predicate<T>	傳回一個組合判定，將此判定結果與other判定的結果作 AND 捷徑邏輯運算
or(Predicate<? super T> other)	default Predicate<T>	傳回一個組合判定，將此判定結果與other判定的結果作 OR 捷徑邏輯運算
negate()	default Predicate<T>	將此判定結果做NOT運算後傳回
isEqual(Object targetRef)	static <T> Predicate<T>	傳回一個測試物件是否相等的判定

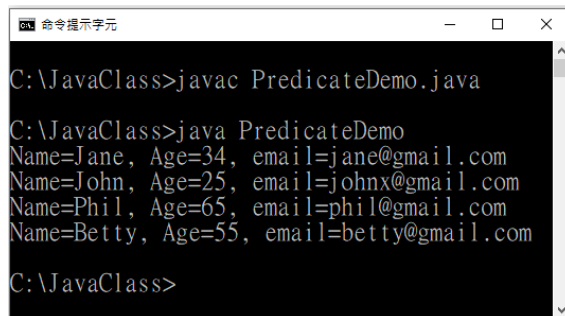
Predicate<T>

```
package java.util.function;

public interface Predicate<T> {

    public boolean test(T t);

}
```



```
命令提示字元
C:\JavaClass>javac PredicateDemo.java
C:\JavaClass>java PredicateDemo
Name=Jane, Age=34, email=jane@gmail.com
Name=John, Age=25, email=johnx@gmail.com
Name=Phil, Age=65, email=phil@gmail.com
Name=Betty, Age=55, email=betty@gmail.com
C:\JavaClass>
```

```
public class PredicateDemo {
    public static void main(String[] args) {
        // 使用Lambda Expression
        Predicate<Person> olderThan23 = p -> p.getAge() >= 23;

        for (Person p : Person.createList()) {
            if (olderThan23.test(p)) {
                System.out.println(p);
            }
        }
    }
}
```

java.util.function.Consumer<T>

方法名稱	傳回值	說明
<i>accept</i> (<i>T t</i>)	<i>void</i>	消費：將傳入參數依指定邏輯執行，無傳回值
<i>andThen</i> (<i>Consumer<? super T> after</i>)	default <i>Consumer<T></i>	傳回一個組合消費，先將此消費應用於其輸入，然後將結果作為輸入應用於 after 消費

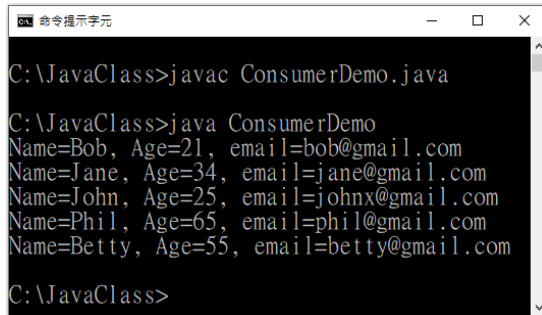
Consumer<T>

```
package java.util.function;

public interface Consumer<T> {

    public void accept(T t);

}
```



```
C:\JavaClass>javac ConsumerDemo.java

C:\JavaClass>java ConsumerDemo
Name=Bob, Age=21, email=bob@gmail.com
Name=Jane, Age=34, email=jane@gmail.com
Name=John, Age=25, email=johnx@gmail.com
Name=Phil, Age=65, email=phil@gmail.com
Name=Betty, Age=55, email=betty@gmail.com

C:\JavaClass>
```

```
public class ConsumerDemo {
    public static void main(String[] args) {
        // 使用Lambda Expression
        Consumer<Person> printPerson = p -> p.printPerson();

        for (Person p: Person.createList()) {
            printPerson.accept(p);
        }
    }
}
```

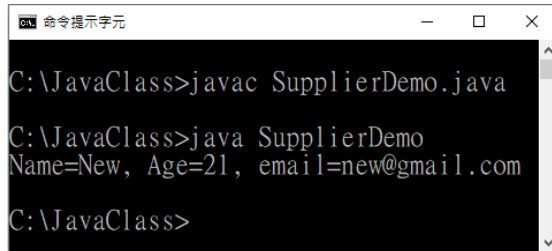
java.util.function.Supplier<T>

方法名稱	傳回值	說明
<i>get()</i>	<i>T</i>	提供：取得所提供之物件

Supplier<T>

```
package java.util.function;

public interface Supplier<T> {
    public T get();
}
```



命令提示字元

```
C:\JavaClass>javac SupplierDemo.java
C:\JavaClass>java SupplierDemo
Name=New, Age=21, email=new@gmail.com
C:\JavaClass>
```

```
public class SupplierDemo {
    public static void main(String[] args) {
        // 使用Lambda Expression
        Supplier<Person> personSupplier =
            () -> new Person("New", "new@gmail.com", 21);

        System.out.println(personSupplier.get());
    }
}
```

變形 functional interfaces

- 將四大類函式介面延伸為進階的函式介面
 - **Primitive (基本型別) :**
 - 傳入或回傳的物件為基本型別的 `wrapper class` 物件
 - 如 `DoubleFunction`、`ToDoubleFunction`
 - **Binary (二運算元) :**
 - 2個傳入的參數
 - 如 `BiPredicate`
 - **Unary (單一運算元) :**
 - 傳入參數與傳回值型別一致
 - 如 `UnaryOperator`

基本型別變形函式介面

- 基本型別**Primitive** 變形函式介面
 - ❑ 輸入基本型別，如 **XXXFunction**
 - ❑ 回傳基本型別，如 **ToXXXFunction**
 - ❑ 減少 **auto-boxing** 和 **unboxing** 運算
 - 提升執行效能

java.util.function	
Interfaces	
BiConsumer	
BiFunction	
BinaryOperator	
BiPredicate	
BooleanSupplier	
Consumer	
DoubleBinaryOperator	
DoubleConsumer	
DoubleFunction	
DoublePredicate	
DoubleSupplier	
DoubleToIntFunction	
DoubleToLongFunction	
DoubleUnaryOperator	
Function	
IntBinaryOperator	
IntConsumer	
IntFunction	
IntPredicate	
IntSupplier	
IntToDoubleFunction	
IntToLongFunction	
IntUnaryOperator	
LongBinaryOperator	
LongConsumer	
LongFunction	
LongPredicate	
LongSupplier	
LongToDoubleFunction	
LongToIntFunction	
LongUnaryOperator	
ObjDoubleConsumer	
ObjIntConsumer	
ObjLongConsumer	
Predicate	
Supplier	
ToDoubleBiFunction	
ToDoubleFunction	
ToIntBiFunction	
ToIntFunction	
ToLongBiFunction	
ToLongFunction	
UnaryOperator	

輸入基本型別

java.util.function.DoubleFunction<R>

方法名稱	傳回值	說明
<i>apply(double value)</i>	<i>R</i>	函式：將傳入的double參數依指定函式運算後傳回

java.util.function.DoublePredicate

方法名稱	傳回值	說明
<i>test(double value)</i>	<i>boolean</i>	判定：依傳入的double參數判定布林結果傳回

java.util.function.DoubleConsumer

方法名稱	傳回值	說明
<i>accept(double value)</i>	<i>void</i>	消費：將傳入的double參數依指定邏輯執行，無傳回值

java.util.function.DoubleSupplier

方法名稱	傳回值	說明
<i>getAsDouble()</i>	<i>double</i>	提供：取得double資料

java.util.function.DoubleFunction<R>

```
import java.util.function.DoubleFunction;

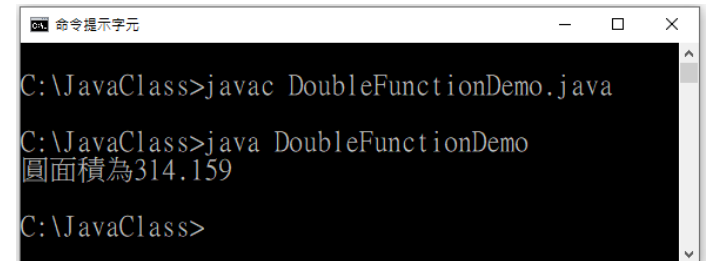
public class DoubleFunctionDemo {
    public static void main(String[] args) {

        // 使用Lambda Expression
        DoubleFunction<String> area = v -> "圓面積為" + v * v * 3.14159;

        String result = area.apply(10);
        System.out.println(result);
    }
}
```

```
package java.util.function;

public interface DoubleFunction<R> {
    public R apply(double v);
}
```



```
命令提示字元
C:\JavaClass>javac DoubleFunctionDemo.java
C:\JavaClass>java DoubleFunctionDemo
圓面積為314.159
C:\JavaClass>
```

輸出基本型別

java.util.function.ToDoubleFunction<T>

方法名稱	傳回值	說明
<i>applyAsDouble(T value)</i>	<i>double</i>	函式：將傳入參數依指定函式運算後為double傳回

java.util.function.IntToDoubleFunction

方法名稱	傳回值	說明
<i>applyAsDouble(int value)</i>	<i>double</i>	函式：將傳入int參數依指定函式運算後為double傳回

java.util.function.LongToDoubleFunction

方法名稱	傳回值	說明
<i>applyAsDouble(long value)</i>	<i>double</i>	函式：將傳入int參數依指定函式運算後為double傳回

java.util.function.ToDoubleFunction<T>

```
import java.util.List;
import java.util.function.ToDoubleFunction;

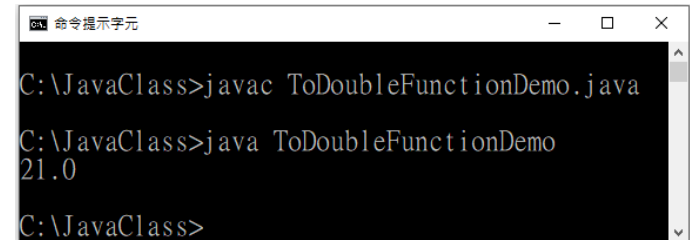
public class ToDoubleFunctionDemo {
    public static void main(String[] args) {
        List<Person> pl = Person.createList();
        Person first = pl.get(0);

        // 使用Lambda Expression
        ToDoubleFunction<Person> toDoubleAge = p -> p.getAge();

        System.out.println(toDoubleAge.applyAsDouble(first));
    }
}
```

```
package java.util.function;

public interface ToDoubleFunction<T> {
    public double applyAsDouble(T t);
}
```



```
命令提示字元
C:\JavaClass>javac ToDoubleFunctionDemo.java
C:\JavaClass>java ToDoubleFunctionDemo
21.0
C:\JavaClass>
```

二運算元

java.util.function.BiFunction<T,U,R>

方法名稱	傳回值	說明
<i>apply(T t, U u)</i>	<i>R</i>	函式：將傳入的兩個參數依指定函式運算後傳回

java.util.function.BiPredicate<T,U>

方法名稱	傳回值	說明
<i>test(T t, U u)</i>	<i>boolean</i>	判定：依傳入的兩個參數判定布林結果傳回

java.util.function.BiConsumer<T,U>

方法名稱	傳回值	說明
<i>accept(T t, U u)</i>	<i>void</i>	消費：將傳入的兩個參數依指定邏輯執行，無傳回值

java.util.function.ToDoubleBiFunction<T,U>

方法名稱	傳回值	說明
<i>applyAsDouble(T t, U u)</i>	<i>double</i>	函式：將傳入的兩個參數依指定函式運算後為double傳回

java.util.function.BiPredicate<T>

```
import java.util.List;
import java.util.function.BiPredicate;

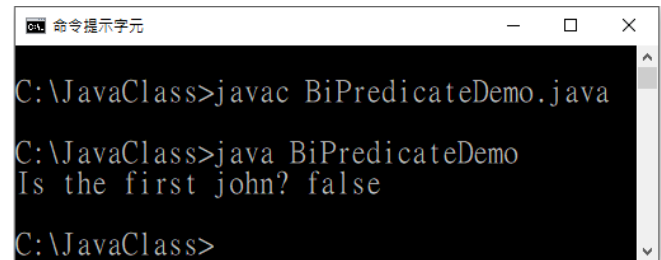
public class BiPredicateDemo {
    public static void main(String[] args) {
        List<Person> pl = Person.createList();
        Person first = pl.get(0);
        String testName = "john";

        // 使用Lambda Expression
        BiPredicate<Person, String> nameBiPred =
            (p, s) -> p.getName().equalsIgnoreCase(s);

        System.out.println("Is the first john? " +
            nameBiPred.test(first, testName));
    }
}
```

```
package java.util.function;

public interface BiPredicate<T, U> {
    public boolean test(T t, U u);
}
```



```
命令提示字元
C:\JavaClass>javac BiPredicateDemo.java
C:\JavaClass>java BiPredicateDemo
Is the first john? false
C:\JavaClass>
```

單一運算元

java.util.function.UnaryOperator<T>

方法名稱	傳回值	說明
<i>apply(T t)</i>	<i>T</i>	繼承自 <code>Function<T,T></code> 函式：將傳入參數依指定函式運算後傳回相同型態傳回值。

java.util.function.DoubleUnaryOperator

方法名稱	傳回值	說明
<i>applyAsDouble(double operand)</i>	<i>double</i>	函式：將傳入double參數依指定函式運算後傳回double值

java.util.function.IntUnaryOperator

方法名稱	傳回值	說明
<i>applyAsInt(int operand)</i>	<i>int</i>	函式：將傳入int參數依指定函式運算後傳回int

java.util.function.LongUnaryOperator

方法名稱	傳回值	說明
<i>applyAsLong(long operand)</i>	<i>long</i>	函式：將傳入long參數依指定函式運算後傳回long

java.util.function.UnaryOperator<T>

```
import java.util.List;
import java.util.function.UnaryOperator;

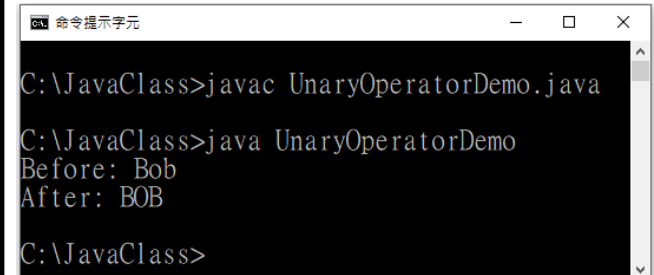
public class UnaryOperatorDemo {
    public static void main(String[] args) {
        List<Person> pl = Person.createList();
        Person first = pl.get(0);

        // 使用Lambda Expression
        UnaryOperator<String> upperStr = (s) -> s.toUpperCase();

        System.out.println("Before: " + first.getName());
        System.out.println("After: " + upperStr.apply(first.getName()));
    }
}
```

```
package java.util.function;

public interface UnaryOperator<T>
    extends Function<T,T> {
    public T apply(T t);
}
```



```
命令提示字元
C:\JavaClass>javac UnaryOperatorDemo.java
C:\JavaClass>java UnaryOperatorDemo
Before: Bob
After: BOB
C:\JavaClass>
```