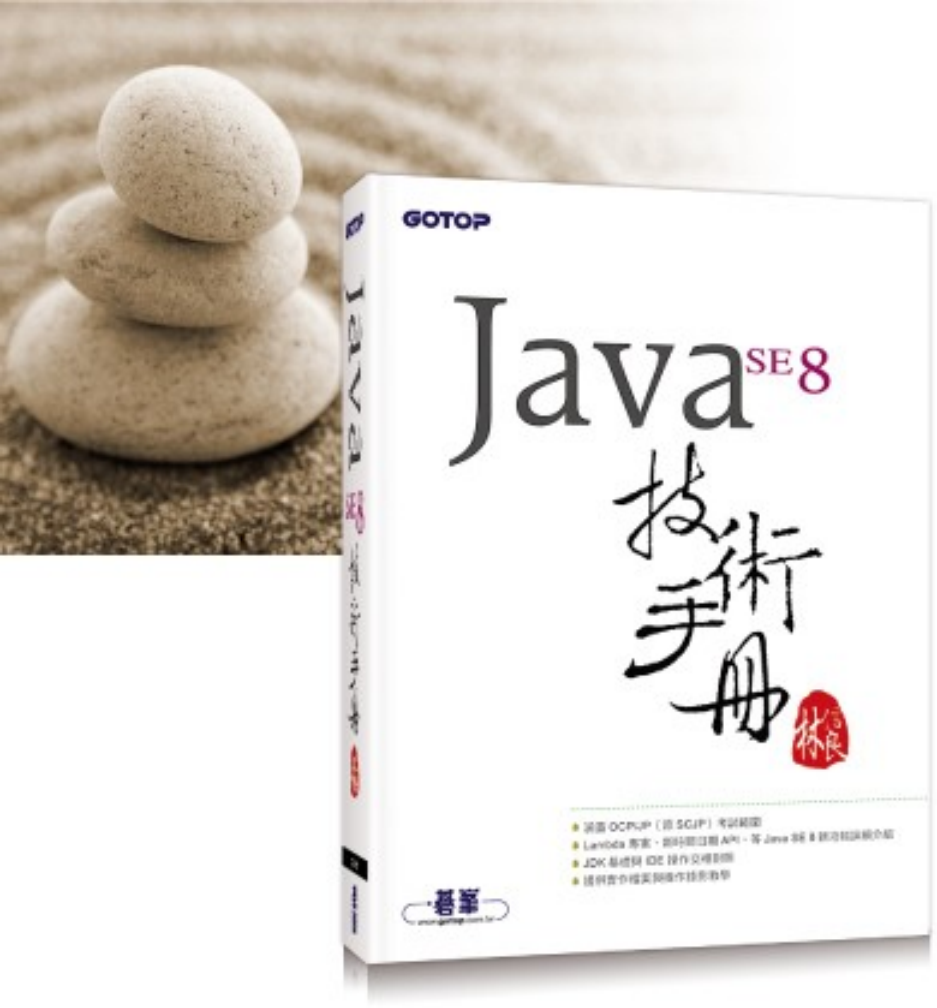


Java SE 8 技術手冊

林信良

- 涵蓋 OCP/JP（原 SCJP）考試範圍
- Lambda 專案、新時間日期 API、等 Java SE 8 新功能詳細介紹
- JDK 基礎與 IDE 操作交相對照
- 提供實作檔案與操作錄影教學

碁峰資訊

CHAPTER

自訂泛型、列舉與標註

學習目標

- 進階自訂泛型
- 進階自訂列舉
- 使用標準標註
- 自訂與讀取標註

使用 extends 與？

- 在定義泛型時，可以定義型態的邊界

```
class Animal {}
class Human extends Animal {}
class Toy {}
class Duck<T extends Animal> {}
public class BoundDemo {
    public static void main(String[] args) {
        Duck<Animal> ad = new Duck<Animal>();
        Duck<Human> hd = new Duck<Human>();
        Duck<Toy> hd = new Duck<Toy>();  // 編譯錯誤
    }
}
```

```
public class Sort<T extends Comparable<T>> {
    public void quick(T[] array) {
        sort(array, 0, array.length-1);
    }

    private void sort(T[] array, int left, int right) {
        if(left < right) {
            int q = partition(array, left, right);
            sort(array, left, q-1);
            sort(array, q+1, right);
        }

    }

    private int partition(T[] array, int left, int right) {
        int i = left - 1;
        for(int j = left; j < right; j++) {
            if(array[j].compareTo(array[right]) <= 0) {
                i++;
                swap(array, i, j);
            }
        }
        swap(array, i+1, right);
        return i + 1;
    }

    private void swap(T[] array, int i, int j) {
        T t = array[i];
        array[i] = array[j];
        array[j] = t;
    }
}
```

使用 extends 與？

- 可以如下使用 quick() 方法：

```
Sort<String> sort = new Sort<>();
String[] strs = {"3", "2", "5", "1"};
sort.quick(strs);
for(String s : strs) {
    System.out.println(s);
}
```

使用 extends 與？

- 若 extends 之後指定了類別或介面後，想再指定其它介面，可以使用 & 連接

```
public class Some<T extends Iterable<T> & Comparable<T>> {
    ...
}
```

使用 extends 與？

- 來看看在泛型中的型態通配字元？

```
public class Node<T> {
    public T value;
    public Node<T> next;

    public Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
```

使用 extends 與？

```
class Fruit {
    int price;
    int weight;
    Fruit() {}
    Fruit(int price, int weight) {
        this.price = price;
        this.weight = weight;
    }
}
```

```
class Apple extends Fruit {
    Apple() {}
    Apple(int price, int weight) {
        super(price, weight);
    }
    @Override
    public String toString() {
        return "Apple";
    }
}
```

```
class Banana extends Fruit {
    Banana() {}
    Banana(int price, int weight) {
        super(price, weight);
    }
    @Override
    public String toString() {
        return "Banana";
    }
}
```

使用 extends 與？

- 如果有以下程式片段，則會發生編譯錯誤：

```
Node<Apple> apple = new Node<>(new Apple(), null);
Node<Fruit> fruit = apple;  // 編譯錯誤，incompatible types
```

- Node<Apple> 是一種 Node<Fruit> 嗎？

使用 extends 與？

- 如果 B 是 A 的子類別，而 Node<B> 可視為一種 Node<A>，則稱 Node 具有共變性（Covariance）或有彈性的（flexible）
- Java 的泛型並不具有共變性，不過可以使用型態通配字元？與 extends 來宣告變數，使其達到類似共變性

```
Node<Apple> apple = new Node<>(new Apple(), null);
Node<? extends Fruit> fruit = apple; // 類似共變性效果
```

使用 extends 與？

```
public class CovarianceDemo {
    public static void main(String[] args) {
        Node<Apple> apple1 = new Node<>(new Apple(), null);
        Node<Apple> apple2 = new Node<>(new Apple(), apple1);
        Node<Apple> apple3 = new Node<>(new Apple(), apple2);

        Node<Banana> banana1 = new Node<>(new Banana(), null);
        Node<Banana> banana2 = new Node<>(new Banana(), banana1);

        show(apple3);
        show(banana2);
    }

    public static void show(Node<? extends Fruit> n) {
        Node<? extends Fruit> node = n;
        do {
            System.out.println(node.value);
            node = node.next;
        } while(node != null);
    }
}
```

使用 extends 與 ?

- 若宣告 ? 不搭配 extends，則預設為 ? extends Object

```
Node<?> node; // 相當於 Node<? extends Object>
```

- 這與宣告為 Node<Object> 不同，如果 node 宣告為 Node<Object>，那就真的只能參考至 Node<Object> 實例了

使用 extends 與 ?

- 以下會編譯錯誤：

```
Node<Object> node = new Node<Integer>(1, null);
```

- 以下會編譯成功：

```
Node<?> node = new Node<Integer>(1, null);
```

使用 extends 與 ?

- 使用通配字元 ? 與 extends 限制 T 的型態，就只能透過 T 宣告的名稱取得物件指定給 Object，或將 T 宣告的名稱指定為 null，除此之外不能進行其它指定動作

```
Node<? extends Fruit> node = new Node<>(new Apple(), null);
Object o = node.value;
node.value = null;
Apple apple = node.value;    // 編譯錯誤
node.value = new Apple();    // 編譯錯誤
```

使用 extends 與？

- Java 的泛型語法只用在編譯時期檢查，執行時期的型態資訊都是未知
  - 也就是執行時期實際上只會知道是 Object 型態（又稱為型態抹除）
- 由於無法在執行時期獲得型態資訊，編譯器只能就編譯時期看到的型態來作檢查

使用 super 與？

- 如果 **B** 是 **A** 的子類別，如果 **Node<A>** 視為一種 **Node<B>**，則稱 **Node** 具有逆變性（Contravariance）
- Java 泛型並不支援逆變性

```
Node<Fruit> fruit = new Node<>(new Fruit(), null);
Node<Banana> node = fruit;  // 實際上會編譯錯誤
```

使用 super 與 ?

- 可以使用型態通配字元 ? 與 super 來宣告，以達到類似逆變性的效果

```
Node<Fruit> fruit = new Node<>(new Fruit(), null);
Node<? super Banana> node = fruit;
Node<? super Apple> node = fruit;
```

使用 super 與？

- 你想設計了一個籃子，可以指定籃中置放的物品，放置的物品會是同一種類（例如都是一種 `Fruit`）
- 有一個排序方法，可指定 `java.util.Comparator` 針對籃中物品進行排序…

使用 super 與？

- 以下泛型未填寫部份該如何宣告？

```
public class Basket<T> {
    public T[] things;
    public Basket(T... things) {
        this.things = things;
    }
    public void sort(Comparator<__________> comparator) {
        // 作一些排序
    }
}
```

使用 super 與？

- 宣告為 <? extends T> 嗎？

```
Basket<Apple> apples = new Basket<>(
    new Apple(20, 100), new Apple(25, 150));
apples.sort(new Comparator<Apple>() {
    @Override
    public int compare(Apple apple1, Apple apple2) {
        return apple1.price - apple2.price;
    }
});

Basket<Banana> bananas = new Basket<>(
    new Banana(30, 200), new Banana(50, 300));
bananas.sort(new Comparator<Banana>() {
    @Override
    public int compare(Banana banana1, Banana banana2) {
        return banana1.price - banana2.price;
    }
});
```

使用 super 與？

- 你希望可以有以下的操作：

```
Comparator<Fruit> priceComparator =
                (fruit1, fruit2) -> fruit1.price - fruit2.price;
Basket<Apple> apples = new Basket<>(
    new Apple(20, 100), new Apple(25, 150));
apples.sort(priceComparator);

Basket<Banana> bananas = new Basket< >(
    new Banana(30, 200), new Banana(50, 300));
bananas.sort(priceComparator);
```

使用 super 與？

- 應該宣告為 <? super T>

```
public class Basket<T> {
    public T[] things;

    public Basket(T... things) {
        this.things = things;
    }

    public void sort(Comparator<? super T> comparator) {
        Arrays.sort(things, comparator);
    }
}
```

sort

```
public static <T> void sort(T[] a,
                            Comparator<? super T> c)
```

自訂列舉

- 在 7.2.3 中曾經簡介過列舉型態，請先瞭解該節內容 …

瞭解 java.lang.Enum 類別

- 在 7.2.3 節中使用 enum 定義過以下的 Action 列舉型態：

```
public enum Action {
    STOP, RIGHT, LEFT, UP, DOWN
}
```

瞭解 java.lang.Enum 類別

- enum 定義了特殊的類別，繼承自 **java.lang.Enum**

```
public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {
    ...
    public final int compareTo(E o) {
        Enum other = (Enum)o;
        Enum self = this;
        if (self.getClass() != other.getClass() && // optimization
            self.getDeclaringClass() != other.getDeclaringClass())
            throw new ClassCastException();
        return self.ordinal - other.ordinal;
    }
    ...
}
```

瞭解 java.lang.Enum 類別

```
...略
private final String name;
private final int ordinal;

protected Enum(String name, int ordinal) {
    this.name = name;
    this.ordinal = ordinal;
}

public final String name() {
    return name;
}
public String toString() {
    return name;
}
public final int ordinal() {
    return ordinal;
}
...略
```

瞭解 java.lang.Enum 類別

- 7.2.3 中 Action.class 反編譯後的內容 … .

```
public final class Action extends Enum {
    public static Action[] values() {
        return (Action[])$VALUES.clone();
    }

    public static Action valueOf(String s) {
        return (Action)Enum.valueOf(Action, s);
    }

    private Action(String s, int i) {
        super(s, i);
    }
    public static final Action STOP;
    public static final Action RIGHT;
    public static final Action LEFT;
    public static final Action UP;
    public static final Action DOWN;
    略...
    static {
        STOP = new Action("STOP", 0);
        RIGHT = new Action("RIGHT", 1);
        LEFT = new Action("LEFT", 2);
        UP = new Action("UP", 3);
        DOWN = new Action("DOWN", 4);
        略...
    }
}
```

瞭解 java.lang.Enum 類別

- 可以透過 Enum 定義的 **name()** 方法取得列舉成員名稱字串，這適用於需要使用字串代表列舉值的場合，相當於 toString() 的作用，事實上 toString() 也只是傳回 name 成員的值
- 可透過 **ordinal()** 取得列舉 int 值，這適用於需要使用 int 代表列舉值的場合

瞭解 java.lang.Enum 類別

- 例如 7.2.1 的 Game 類別，可以如下操作

```
public class GameDemo {
    public static void main(String[] args) {
        Game.play(Action2.DOWN.ordinal());
        Game.play(Action2.RIGHT.ordinal());
    }
}

public enum Action2 {
    STOP, RIGHT, LEFT, UP, DOWN
}
```

瞭解 java.lang.Enum 類別

- Enum 的 **valueOf()** 方法，可以傳入字串與 Enum 實例，它會傳回對應的列舉實例

```
Action2 action = Enum.valueOf(Action2.class, "UP");
out.println(Action2.UP == action);
```

- 通常會透過 Enum 子類別的 valueOf() 方法，其內部就使用了 Enum.valueOf() （可觀察先前反編譯 Action 列舉的程式碼）

```
Action2 action = Action2.valueOf("UP");
System.out.println(Action2.UP == action);
```

瞭解 java.lang.Enum 類別

- Enum 的 equals() 與 hashCode() 基本上繼承了 Object 的行為，但被標示為 final。

```
...略
    public final boolean equals(Object other) {
        return this==other;
    }

    public final int hashCode() {
        return super.hashCode();
    }
...略
```

進階 enum 運用

- **values()** 方法會將內部維護 Action 列舉實例的陣列複製後傳回
- 由於是複製品，因此改變傳回的陣列，並不會影響 Action 內部所維護的陣列

進階 enum 運用

- 可以自行定義建構式，條件是不得為公開（`public`）建構式，也不可以於建構式中呼叫 `super()`

進階 enum 運用

- 例如原本有個 interface 定義的列舉常數

```
public interface Priority {
    int MAX = 10;
    int NORM = 5;
    int MIN = 1;
}
```

進階 enum 運用

```
public enum Priority {
    MAX(10), NORM(5), MIN(1);          ← ❶ 呼叫 enum 建構式

    private int value;
    private Priority(int value) {      ← ❷ 不為 public 的建構式
        this.value = value;
    }

    public int value() {               ← ❸ 自定義方法
        return value;
    }

    public static void main(String[] args) {
        for(Priority priority : Priority.values()) {
            System.out.printf("Priority(%s, %d)%n",
                    priority, priority.value());
        }
    }
}
```

```
public final class Priority extends Enum {
    ...略
    private Priority(String s, int i, int value) {
        super(s, i);
        this.value = value;
    }
    public int value() {
        return value;
    }
    ...略
    public static final Priority MAX;
    public static final Priority NORM;
    public static final Priority MIN;
    private int value;
    private static final Priority $VALUES[];

    static
    {
        MAX = new Priority("MAX", 0, 10);
        NORM = new Priority("NORM", 1, 5);
        MIN = new Priority("MIN", 2, 1);
        $VALUES = (new Priority[] {
            MAX, NORM, MIN
        });
    }
}
```

進階 enum 運用

- 定義列舉時還可以實作介面，例如有個介面定義如下：

```
public interface Command {
    void execute();
}
```

進階 enum 運用

```
public enum Action3 implements Command {
    STOP, RIGHT, LEFT, UP, DOWN;
    public void execute() {
        switch(this) {
            case STOP:
                out.println("播放停止動畫");
                break;
            case RIGHT:
                out.println("播放向右動畫");
                break;
            case LEFT:
                out.println("播放向左動畫");
                break;
            case UP:
                out.println("播放向上動畫");
                break;
            case DOWN:
                out.println("播放向下動畫");
                break;
        }
    }
}
```

進階 enum 運用

- 可以如下執行程式：

```
public class Game3 {
    public static void play(Action3 action) {
        action.execute();
    }

    public static void main(String[] args) {
        Game3.play(Action3.RIGHT);
        Game3.play(Action3.DOWN);
    }
}
```

進階 enum 運用

- 定義 enum 時有個特定值類別本體（Value-Specific Class Bodies）語法

```
public enum Action3 implements Command {
    STOP {
        public void execute() {
            out.println("播放停止動畫");
        }
    },
    RIGHT {
        public void execute() {
            out.println("播放右轉動畫");
        }
    },
    LEFT {
        public void execute() {
            out.println("播放左轉動畫");
        }
    },
```

進階 enum 運用

```
        UP {
            public void execute() {
                out.println("播放向上動畫");
            }
        },
        DOWN {
            public void execute() {
                out.println("播放向下動畫");
            }
        };
    }
```

進階 enum 運用

- 實際上，編譯器會將 Action3 標示為抽象類別：

```
public abstract class Action3 extends Enum implements Command {
    ...
}
```

- 並為每個列舉成員後的 { } 語法，產生匿名內部類別，這個匿名內部類別繼承了 Action3，實作了 execute() 方法 …

進階 enum 運用

```
...略
    static
    {
        STOP = new Action3("STOP", 0) {
            public void execute() {
                System.out.println("\u64AD\u653E\u505C\u6B62\u52D5\u756B");
            }
        };
        RIGHT = new Action3("STOP", 0) {
            public void execute() {
                System.out.println("\u64AD\u653E\u505C\u6B62\u52D5\u756B");
            }
        };
         ...略
    }
...略
```

進階 enum 運用

- 以先前 Priority 為例，可改寫為以下：

```
public enum Priority2 {
    MAX(10) {
        public String toString() {
            return String.format("(%2d) - 最大權限", value);
        }
    },
    NORM(5) {
        public String toString() {
            return String.format("(%2d) - 普通權限", value);
        }
    },

    MIN(1) {
        public String toString() {
            return String.format("(%2d) - 最小權限", value);
        }
    };
```

進階 enum 運用

```
    protected int value;
    private Priority2(int value) {
        this.value = value;
    }
    public int value() {
        return value;
    }

    public static void main(String[] args) {
        for(Priority2 priority : Priority2.values()) {
            System.out.println(priority);
        }
    }
}
```

常用標準標註

```
public class WorkerThread extends Thread {

        method does not override or implement a method from a supertype
        ----
        (Alt-Enter shows hints)

    @Override
    public void Run() {
        //...
    }
}
```

常用標準標註

- 如果某個方法原先存在於 API 中，後來不建議再使用，可於該方法上標註

```
public class Some {
    @Deprecated
    public void doSome() {
        ...
    }
}
```

常用標準標註

- 若有使用者後續又想呼叫或重新定義這個方法，編譯器會提出警訊 ..

```
Note: XXX.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

常用標準標註

- 在 JDK5 之後加入泛型支援，對於支援泛型的 API，建議明確指定泛型真正型態，如果沒有指定的話，編譯器會提出警訊

```
public void doSome() {
     List list = new ArrayList();
     list.add("Some");
}
```

```
Note: xxx.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

常用標準標註

- 如果不想看到這個警訊，可以使用 **@SuppressWarnings** 指定抑制 unckecked 的警訊產生：

```
@SuppressWarnings(value={"unchecked"})
public void doSome() {
     List list = new ArrayList();
     list.add("Some");
}
```

常用標準標註

- **@SuppressWarnings** 的 **value** 可以指定要抑制的警訊種類。例如你真的想呼叫 @Deprecated 標示過的方法，又不想看到警訊，可以如下：

```
@SuppressWarnings(value={"deprecation"})
```

- 也可以一次指定抑制多項警訊：

```
@SuppressWarnings(value={"unchecked", "deprecation"})
```

常用標準標註

- 有沒有可能建立 `List<String>[]` 陣列實例？答案是不行！
- 宣告 `List<String>[] lists` 是可以的，只是實際上不會有人這麼做
- 可以宣告 `List<String>[] lists` 是為了支援可變長度引數

```
public class Util {
    public static <T> void doSome(List<String>... varargs) {
        ...略
    }
}
```

常用標準標註

- 在 JDK6 中這個程式碼可以順利編譯，也不會有任何警訊
- 如果你這麼使用：

```
List<String> list1 = new ArrayList<>();
List<String> list2 = new ArrayList<>();
Util.doSome(list1, list2);
```

```
>javac -Xlint:unchecked Main.java
Main.java:7: warning: [unchecked] unchecked generic array creation for varargs
parameter of type List<String>[]
        Util.doSome(list1, list2);
                    ^
1 warning
```

常用標準標註

- Java 泛型語法是提供編譯器資訊，使其可在編譯時期檢查型態錯誤
- 編譯器只能就 `List<String>` 的型態資訊，在編譯時期幫你檢查呼叫 `doSome()` 時，傳入的 `list1` 與 `list2` 是否為 `List<String>` 型態

常用標準標註

- 設計 doSome() 的人在實作流程時，是有可能發生編譯器無法檢查出來的執行時期型態錯誤
- 這類問題稱為 Heap pollution

```
public static <T> void doSome(List<String>... varargs) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList;                    // 語意不對，不過編譯器不會有警訊
    String s = stringLists[0].get(0); // 執行時期發生 ClassCastException
}
```

常用標準標註

- 即使編譯器提醒身為 `doSome()` 的客戶端可能會有這類問題發生又如何？
- 在 JDK7 中，同樣的 `Util` 類別編譯時，會發生以下警訊：

```
Util.java:3: warning: [unchecked] Possible heap pollution from parameterized vararg
type List<String>
    public static <T> void doSome(List<String>... varargs) {
                                                 ^
1 warning
```

常用標準標註

- 如果開發人員確定避免了這個問題，則可以使用 `@SafeVarargs` 加以標註

```
public class Util {
    @SafeVarargs
    public static <T> void doSome(List<String>... varargs) {
        ...略
    }
}
```

常用標準標註

- 如下呼叫 `Util.doSome()` 不會再發生警訊

```
List<String> list1 = new ArrayList<>();
List<String> list2 = new ArrayList<>();
Util.doSome(list1, list2);
```

自訂標註型態

- 所有標註型態其實都是 **java.lang.annotation.Annotation** 子介面
  - @Override 型態 **java.lang.Override**
  - @Deprecated 型態 **java.lang.Deprecated**
  - …

自訂標註型態

- 要定義一個標註可以使用 `@interface`

Annotation Test.java

```
package cc.openhome;
public @interface Test {}
```

```
public class SomeTestCase {
    @Test
    public void testDoSome() {
        ...略
    }
}
```

自訂標註型態

- 設定單值標註（Single-value Annotation）

```
package cc.openhome;
public @interface Test2 {
    int timeout();
}

@Test2(timeout = 10)
public void testDoSome2() {
    ...
}
```

自訂標註型態

- 標註屬性也可以用陣列形式指定

```
package cc.openhome;
public @interface Test3 {
    String[] args();
}

@Test3(args = {"arg1", "arg2"})
public void testDoSome3() {
    ...
}
```

自訂標註型態

- 在定義標註屬性時，如果屬性名稱為 **value**，則可以省略屬性名稱，直接指定值

```
public @interface Ignore {
    String value();
}
```

- 這個標註可以使用 `@Ignore(value = "message")` 指定，也可以使用 `@Ignore("message")` 指定

自訂標註型態

- 以下這個標註：

```
public @interface TestClass {
    Class[] value();
}
```

- 可以使用 `@TestClass(value = {Some.class, Other.class})` 指定，也可以使用 `@TestClass({Some.class, Other.class})` 指定

自訂標註型態

- 使用 **default** 關鍵字可以對成員設定預設值

```
public @interface Test4 {
    int timeout() default 0;
    String message default "";
}
```

自訂標註型態

- 如果是 Class 設定的屬性比較特別，必須自訂一個類別作為預設值

```
public @interface Test5 {
    Class expected() default Default.class;
    class Default {}
}
```

自訂標註型態

- 如果要設定陣列預設值的話，可以在 `default` 之後加上 `{}`

```
public @interface Test6 {
    String[] args() default {};
}

public @interface Test7 {
    String[] args() default {"arg1", "arg2"};
}
```

自訂標註型態

- 可使用 **java.lang.annotation.Target** 限定標註使用位置，限定時可指定 **java.lang.annotation.ElementType** 的列舉值

```
package java.lang.annotation;
public enum ElementType {
    TYPE,                       // 可標註於類別、介面、列舉等
    FIELD,                      // 可標註於資料成員
    METHOD,                     // 可標註於方法
    PARAMETER,                  // 可標註於方法上的參數
    CONSTRUCTOR,                // 可標註於建構式
    LOCAL_VARIABLE,             // 可標註於區域變數
    ANNOTATION_TYPE,            // 可標註於標註型態
    PACKAGE                     // 可標註於套件
    TYPE_PARAMETER,             // 可標註於型態參數，JDK8 新增
    TYPE_USE                    // 可標註於各式型態，JDK8 新增
}
```

自訂標註型態

- 想將 `@Test8` 限定只能用於方法：

```
@Target({ElementType.METHOD})
public @interface Test8 {}
```

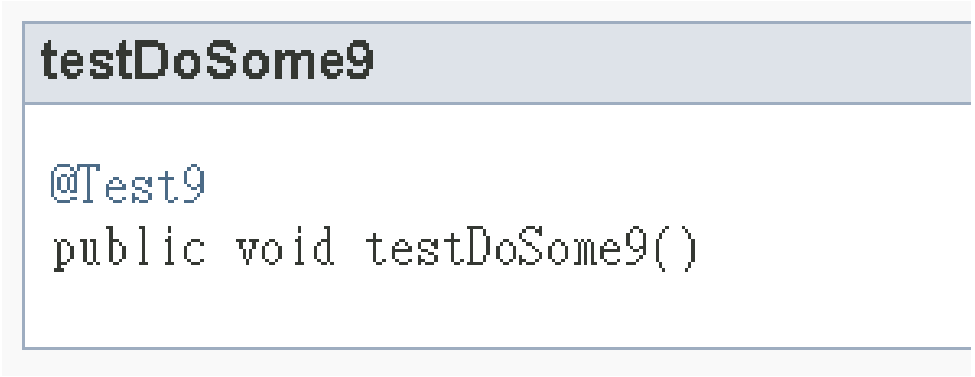
```
annotation type not applicable to this kind of declaration
---
(Alt-Enter shows hints)
```

```
@Test8
public class SomeTestCase {
    --
```

自訂標註型態

- 想要將標註資料加入文件，可以使用 **java.lang.annotation.Documented**

```
@Documented
public @interface Test9 {}
```



自訂標註型態

- 在定義標註時設定 **java.lang.annotation.Inherited** 標註，就可以讓標註被子類別繼承

```
@Inherited
public @interface Test10 {}
```

JDK8 標註增強

- 在 JDK8 出現之前，`ElementType` 的列舉成員，是用來限定哪個宣告位置可以進行標註
- JDK8 的 `ElementType` 多了兩個列舉成員 `TYPE_PARAMETER`、`TYPE_USE`，它們是用來限定哪個型態可以進行標註

```
public class MailBox<@Email T> {
    …
}
```

JDK8 標註增強

- 在定義 `@Email` 時，必須在 `@Target` 設定 `ElementType.TYPE_PARAMETER`，表示這個標註可用來標註型態參數

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE_PARAMETER)
public @interface Email {}
```

JDK8 標註增強

- 一個標註如果被設定為 `ElementType.TYPE_USE`，只要是型態名稱，都可以進行標註

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE_USE)
public @interface Test11 {}
```

JDK8 標註增強

- 以下幾個標註範例都是可以的：

```
List<@Test11 Comparable> list1 = new ArrayList<>();
List<? extends Comparable> list2 = new ArrayList<@Test11 Comparable>();
@Test11 String text;
text = (@Test11 String) new Object();
java.util. @Test11 Scanner console;
console = new java. util. @Test11 Scanner(System.in);
```

- 以下的標註就不合法：

```
@Test11 java.lang.String text;
```

JDK8 標註增強

```
public @interface Filter {
    String[] value();
}
```

- 這可以讓你如下進行標註：

```
@Filter({"/admin", "/manager"})
public interface SecurityFilter {
    ...
}
```

JDK8 標註增強

- JDK8 新增了個 @Repeatable

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Filters.class)
public @interface Filter {
    String value();
}

@Retention(RetentionPolicy.RUNTIME)
@interface Filters {
    Filter[] value();
}
```

```
@Filter("/admin")
@Filter("/manager")
public interface SecurityFilter {}
```

執行時期讀取標註資訊

- 如果希望於執行時期讀取標註資訊，可以於自訂標註時使用 **java.lang.annotation.Retention** 搭配 **java.lang.annotation.RetentionPolicy** 列舉指定…

```
package java.lang.annotation;
public enum RetentionPolicy {
    SOURCE,   // 標註資訊留在原始碼（不會儲存至.class 檔案）
    CLASS,    // 標註資訊會儲存至.class 檔案，但執行時期無法讀取
    RUNTIME   //標註資訊會儲存至.class 檔案，但執行時期可以讀取
}
```

執行時期讀取標註資訊

- 可使用 `java.lang.reflect.AnnotatedElement` 介面實作物件取得標註資訊

Modifier and Type	Method and Description
`<T extends Annotation> T`	`getAnnotation(Class<T> annotationClass)` Returns this element's annotation for the specified type if such an annotatio
`Annotation[]`	`getAnnotations()` Returns annotations that are *present* on this element.
`default <T extends Annotation> T[]`	`getAnnotationsByType(Class<T> annotationClass)` Returns annotations that are *associated* with this element.
`default <T extends Annotation> T`	`getDeclaredAnnotation(Class<T> annotationClass)` Returns this element's annotation for the specified type if such an annotatio
`Annotation[]`	`getDeclaredAnnotations()` Returns annotations that are *directly present* on this element.
`default <T extends Annotation> T[]`	`getDeclaredAnnotationsByType(Class<T> annotationClass)` Returns this element's annotation(s) for the specified type if such annotation *indirectly present*.
`default boolean`	`isAnnotationPresent(Class<? extends Annotation> annotationClass)` Returns true if an annotation for the specified type is *present* on this elemer

執行時期讀取標註資訊

- Class、Constructor、Field、Method、Package 等類別，都實作了 AnnotatedElement 介面
- 如果標註在定義時的 RetentionPolicy 指定為 RUNTIME，就可以用 Class、Constructor、Field、Method、Package 等類別的實例，取得設定的標註資訊

執行時期讀取標註資訊

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Debug {
    String name();
    String value();
}

public class Other {
    @Debug(name = "caterpillar", value = "2011/10/10")
    public void doOther() {
        ...略
    }
}
```

```
public static void main(String[] args) throws NoSuchMethodException {
    Class<Other> c = Other.class;
    Method method = c.getMethod("doOther");
    if(method.isAnnotationPresent(Debug.class)) {
        out.println("已設定 @Debug 標註");
        showDebugAnnotation(method);
    } else {
        out.println("沒有設定 @Debug 標註");
    }
    showAllAnnotations(method);
}

private static void showDebugAnnotation(Method method) {
    // 取得 @Debug 實例
    Debug debug = method.getAnnotation(Debug.class);
    out.printf("value: %s%n", debug.value());
    out.printf("name : %s%n", debug.name());
}

private static void showAllAnnotations(Method method) {
    Annotation[] annotations = method.getAnnotations();
    for(Annotation annotation : annotations) {
        out.println(annotation.annotationType().getName());
    }
}
```

執行時期讀取標註資訊

- JDK8 新增了 `getDeclaredAnnotation()`、`getDeclaredAnnotationsByType()`、`getAnnotationsByType()`

```
Filter[] filters = SecurityFilter.class.
        getAnnotationsByType(Filter.class);
for(Filter filter : filters) {
    out.println(filter.value());
}

out.println(SecurityFilter.class.getAnnotation(Filter.class));
```