第六章

Annotation

最近這些日子被炒作到最熱門的程式設計術語之一就是「metadata」。metadata 其實只是關於資訊的資訊。它出現在介於 Java 光譜上，給編譯器用的未處理原始程式碼資訊，與純粹說明文件的 Javadoc 之間。metadata 通常是對在某時點上會被解譯的程式碼由程式碼或資料分析工具作出說明。

你第一個想到的可能會是，“呃，Javadoc 不就是在處理這個嗎？”。考慮一下這一點－有多少的方法可以說明「這個變數不應該是 null」。在 Javadoc 中，可能會用“non-null”，可能會是“這個變數不應為 null”，或許是“別指派 null 給它”。以上的講法在以文件說明上來看都是有效的，但並沒有全面的一致。並沒有任何工具可以來分析與歸納這個簡單的條件可以被陳述的各種變化。Tiger 中新出現的 annotation 是藉由使用已經定義好的 metadata 機制來尋求這個問題的解決方案。

簡單講，annotation 是可施用在 package 與 type declaration、constructor、method、field、parameter、以及甚至是 variable 的修飾字（modifier）。它是採用 name=value 成對的形式，並且你可以使用 Java 的內建型別，或者你自行定義的自訂 annotation 型別。

在本章中：

- 使用標準的 Annotation 型別
- Annotating 一個被 Override 的 Method
- Annotating 一個被 Deprecated 的 Method
- 抑止警告
- 建構自訂的 Annotation 型別
- 對 Annotation 作 Annotate
- 定義 Annotation 型別的 Target
- 設定 Annotation 型別的 Retention
- 製作 Annotation 型別的說明文件
- 設定 Annotation 的繼承
- Annotation 的 Reflecting

使用標準的 Annotation 型別

「Standard annotation types」是在 Tiger 中是以「隨拆即用（out of the box）」的形式提供的。他們有三種，這三種都定義在 java.lang 這個 package 中。這些 annotation 型別可在你自有的程式中豪不費力的運用。

我該怎麼做？

下面列出的是在 Tiger 中已經事先定義好的三種標準 annotation 型別，有簡短的說明，並會在稍後的分節中有更完整的討論：

這些都不需要被 import，因為它們全都在 "javalang" 這個 package 裡面，且是自動取得的。

Override

java.lang.Override 是用來指示有一個 method 它 override 掉它的 superclass 的 method。

Deprecated

java.lang.Deprecated 指示某一個 method 或是 element 型別的使用是被阻止的。

SupressWarnings

java.lang.SupressWarnings 會關掉 class、method、或 field 與 variable 初始化的編譯器警告。

下面是使用 Override 這個 annotation 型別的例子：

```
@Override
public String toString( ) {
  return super.toString( ) + " [modified by subclass]";
}
```

而這是使用 Deprecated 的例子：

```
@Deprecated public class Betamax { ... }
```

最後，這是 SuppressWarnings 的運用：

```
@SuppressWarnings("unchecked")
public void nastyMethod( ) {
  // 內容省略
}
```

我知道我還沒有告訴你這些是如何運作的－這是故意的。現在你看到了每個 annotation 的使用，你應該會發現每一個都有完全不同的語法。要能夠掌握到 annotation，我們得對理論作些許的探討，然後就會由各分節來對各個標準的 annotation 進行細節研究。忍住一下，我們先由一些技術細節著手，然後你就會準備好迎接更實際的指引。

發生了什麼事？

首先，你必須要了解「annotation」與「annotation 型別」的差異。先從後者說起，一個 annotation 型別是一個 annotation 的特有名稱，加上所有的預設值與相關資訊。你剛剛已經看過了三種 annotation 型別：Override、

Deprecated、與 SuppressWarnings。然後 annotation 使用一個 annotation 型別來將一些資訊與一個 Java 的程式元素（method、class、variable 等等）作關聯。所以你的程式碼可能只會用到一種 annotation 型別，像是 Override，但會有十或十五個的 annotation（如果它使用 Override 十或十五次）。

annotation 型別也可以有值－注意到 SuppressWarnings 傳入了一個“unchecked”的值，它會被 annotation 型別用來處理或儲存資訊。此傳入的值，加上所有的預設值，組合起來成為該 annotation 的 member。這就是我在這一節稍早所提過的 name=value 語法的由來－它讓 annotation 的 member 能夠被設定。

在這三種標準的 annotation 型別之外，還有三類的 annotation：

Marker annotation

marker annotation 是給沒有定義 member 的 annotation 型別使用。它們只是提供包含在 annotation 本身名稱中的資訊。一個所有 member 都有預設值的 annotation 同樣也可當作 marker 來使用，因為沒有資訊需要被傳入。marker 的語法就只是像下面這樣：

```
@MarkerAnnotation
```

Single-value annotation

single-value annotation 只有一個 member，稱為 value。single-value annotation 的格式是：

```
@SingleValueAnnotation("some value")
```

annotation 程式行的後面沒有分號。

Full annotation

full annotation 其實算不上是一類，它最多就是一個使用全方位 annotation 語法的 annotation 型別。跟在 annotation 名稱後面的是個括號，且所有的 member 都有被指派值：

```
@Reviews({  // 花括號用來表示值的 array
  @Review(grade=Review.Grade.EXCELLENT, reviewer="df"),
  @Review(grade=Review.Grade.UNSATISFACTORY, reviewer="eg",
          comment="This method needs an @Override annotation")
})
```

對每一種類都有稍稍不同的語法－這也就是為何每個標準的 annotation 型別使用上有些不同。接下來的三個分節個別都有其中一個標準 annotation 型別的細節，說明其所屬的種類，並給你使用 annotation 型別的進一步細節。

有關於 ...

... 像是 XDoclet（*http://xdoclet.sourceforge.net*）這樣的工具已經處理到此類的分析，特別是企業等級的程式設計。XDoclet 已經提供了許多同樣的資訊，從原始程式碼與「受管理的相依性（managing dependence）」（特別是在 EJB 所需要的眾多原始檔）中讀出 metadata，而後全部經由 Javadoc 解析與某些很棒的類別產生工具完成。

這些工具確實是為何需要 annotation 的最佳範例。當 XDoclet 解析 Javadoc 時，它無法指出 Javadoc 的語法是否有錯－如果它發現了相符的 tag，它會使用這些 tag，而拼字的錯誤會被忽略而沒有任何的提示。然而，annotation 會被 Java 的編譯器所檢查－這代表著你的程式碼中不會有這樣的情況：

```
@Overridde  // 注意到字拼錯了
```

編譯器會抱怨這個錯字，意味著你不只是在程式碼的層次上被保護著，保護也會發生在 annotation 層級。現在可以採用這額外的檢查，搭配上 XDoclet 所提供的關聯、分析、與產生程式碼功能。一下子，一個很好的工具就變得更好，還有更好的使用者友善性，還有更為標準 Java 工具套件的組件。簡單的說，就是 annotation 會產生讓 XDoclet 這樣的工具成為每個程式設計者必備組件的真正突破。

Annotating 一個被 Override 的 Method

Override 這個 annotation 型別是個 marker 的 interface，沒有初始化的 member。它是用來指示某一 method 是 overriding 它的 superclass 中的 method 而來的。它是用來幫你保證確實會有 override 某個 method－利用檢查拼字錯誤或弄混的 method 名稱與參數 list 來達成。在這種例子中的錯誤，能夠由編譯器來捕捉問題的所在並予以回報。

我該怎麼做？

大多因為巧合的緣故，此念作 "at" 的 @ 符號用在 annotation 型別上很順口。

正因為 Override 是個 marker 的 interface，所以你不需要賦予它值。只要前置 annotation 語法的標記就好，那就是一個 at 的符號（@），然後鍵入 “Override”。它應該是獨立的一行，放在你要宣告為一個 overriding 的 method 的宣告之前，如同範例 6-1 所示。

範例 6-1. 使用 Override 的 annotation 型別

```
package com.oreilly.tiger.ch06;

public class OverrideTester {
```

```
    public OverrideTester( ) { }

    @Override
    public String toString( ) {
      return super.toString( ) + " [OverrideTester Implementation]";
    }

    @Override
    public int hashCode( ) {
      return toString( ).hashCode( );
    }
  }
```

這並不特別的迷人或有魅力，但它編譯時不會有任何的問題。它會在你沒有作出你想做的事情時變的很有用處。假設把 hashCode() 這個 method 改成這樣：

```
    @Override
    public int hasCode( ) {
      return toString( ).hashCode( );
    }
```

這個 method 有出現在本書的 Override-Tester 範例原始程式碼中，但被加以註解掉。

此處 hashCode() 拼錯了，且此 method 不再有 overriding 任何事物，這會讓 annotation 跳出來說話。編譯這個 class 你會得到下面的錯誤：

```
    [javac] src\ch06\OverrideTester.java:1:
            method does not override a method from its superclass
    [javac]   @Override
    [javac]    ^
    [javac] 1 error
```

突然間，這小小的 annotation 變成有大大的作用，能夠在編譯時期捕捉錯誤。它又很容易就能整合進你的程式設計過程中，因此建議你應該要經常使用。

有關於 ...

... 需要被 overridden 的 method？Override 標示出施以 overriding 的 method，而不是被 overridden 的 method－這是個很重要的分野，且你應該要分的很清楚。Java 本來就有方法能夠指示某個 method 應該要被 overridden－明確的說，就是把 method 宣告為 abstract。

Annotating 一個被 Deprecated 的 Method

如同 Override，java.lang.Deprecated 也是個 marker 的 annotation 型別。它在 Javadoc 中有著一個同類，就是 @deprecated 這個 tag。雖然由不同的工具使用（見《有關於 ...》小節），但兩者都指示相同的事物。任何時候你想要確保 class 會對某特定 method 作 overriding 時發出警告就使用 Deprecated。

我該怎麼做？

Deprecated 是個 marker 的 interface，使用時沒有括號或者是 member 的值，跟 Override 是一樣的。然而，它是要置於與被禁止的宣告同一行的位置，而 Override 的 annotation 是置於前一行的位置。範例 6-2 是個使用 Deprecated 的簡單例子。

範例 6-2. 使用 Deprecated 的 annotation 型別

```
package com.oreilly.tiger.ch06;

public class DeprecatedClass {

  /**
   * 此 method 被禁止作出 doSomethingElse( ) 所容許的行為
   * 可以 doSomethingElse( ) 來替代
   */
  @Deprecated public void doSomething( ) {
    // 作些事情 ...
  }

  public void doSomethingElse( ) {
    // 作其它的事情（可能會更好）
  }
}
```

對它本身來說，此 annotation 沒有任何作用。然而，它會在其它的 class 要 override 此被禁止的 method 時起作用，如同範例 6-3 中的 class 一樣。

範例 6-3. overriding 一個被標示為禁止的 method

```
package com.oreilly.tiger.ch06;

public class DeprecatedTester extends DeprecatedClass {

  public void doSomething( ) {
    // Override 被禁止的 method
  }
}
```

使用 "-Xlint:deprecation" 旗標打開 deprecation 的檢查。

如果你編譯這些 class，並將編譯器的 deprecation 檢查打開，你會得到這樣的警告：

```
[javac] src\ch06\DeprecatedTester.java:5: warning:
        [deprecation] doSomething( ) in
          com.oreilly.tiger.ch06.DeprecatedClass has been deprecated
[javac]   public void doSomething( ) {
[javac]               ^
```

再一次地，這裡並沒有談到革命性的新功能，但卻能夠為 XDoclet 這樣的自我檢查工具提供一些幫助。

有關於 ...

... 那個稱為 @deprecated 的 Javadoc 標籤？首先，要知道它不會完全的被 Deprecated 這種 annotation 型別給廢棄掉。Javadoc 的 comments 是由 Javadoc 此工具來運用，且是所有 class 說明文件的重要部分。而 Deprecated 這個 annotation 型別是由編譯器用來確保你的程式碼符合說明文件所指示的－method 或 class 是確實的被禁止。這是很重要的差別，相當值得記住。事實上，你應該總是併用兩者，一個給說明文件用，另一個給編譯用。此外，編譯器還是會讀取與處理 @deprecated 標籤以備後向的相容性。

Javadoc 的 tag 是小寫的，annotation 型別是大寫的。

還有，你可能會想到同樣可以傳遞給 javac 的 -deprecation 旗標。如果你加上 -deprecation 旗標來編譯，但是沒有使用 -Xlint:deprecation，你會得到與使用 -Xlint:deprecation 相同的結果。事實上，從我對 JDK 的試驗上來看，這兩個旗標的功能在 Tiger 上是完全相同的。

抑止警告

隨著 Tiger 的降臨，在 Tiger 之前的程式碼會有一如往常運作，卻產生警告的時候。這通常會發生在 collection 上，因為 Tiger 容許更強的 typing，並且也逼著你要這麼寫程式。然而，一個完整測試過的程式應該不會發出或者產生警告，因此這就有點自相矛盾了（catch 22）。你的程式碼運作正常，但卻在 Tiger 上產生警告－另外一方面，忽略程式的所有警告也不是個好主意。你要怎麼解決這個兩難的問題？

使用 "-Xlint" 開關可以讓你看到 Java 編譯器產生的警告。

答案是使用另一個標準的 annotation 型別，SuppressWarnings，來讓你關閉特定 class、method、或 field/variable 初始化的警告。在此同時，對於其它程式段的警告依然完好無缺，這是理所當然的。

舉例來說，沒有在 "switch" 述句中處理到所有的 enumerated 型別是一個應該被修正的警告。

我該怎麼做？

SuppressWarnings 並不像 Deprecated 與 Override 是個 marker 的 interface，但是有一個稱為 value 的 member。value 是個 String 的 array（String[]），而此 member 的值是要被抑止的這些警告型別所組成的 array。所以，你可以使用下面的 annotation 來抑止 unchecked 的警告：

這段程式碼在 com.oreilly.tiger.ch06.SuppressWarningsTester。

```
/**
 * Tiger 之前的普通 method
 */
@SuppressWarnings(value={"unchecked"})
public void nonGenericsMethod(  ) {
  List wordList = new ArrayList(  );

  wordList.add("foo");
}
```

如果你已經熟悉 Tiger 程式碼，你會發現有些事情不對勁－因為沒有用到 generic，這會因為 wordlist 沒有被指定型別而引發一個 unchecked 的警告（在沒有使用 SuppressWarnings annotation 的情況下）。然而，這個警告會因使用 annotation 而消失－這在你於 Tiger 下編譯給 Tiger 之前的平台時特別有用。

如果你把 annotation 給移除掉，你會看到這個程式碼產生下面這樣的警告：

```
[javac] src\ch06\SuppressWarningsTester.java:15:
  warning: [unchecked] unchecked call to add(E) as a member of the
           raw type java.util.List
[javac]     wordList.add("foo");
[javac]             ^
```

當然，如果你真的只有打算把部署在 Tiger 之前的平台上這就不是問題，此時 SuppressWarnings 就真的發揮了作用。你同時也可以指定要忽略多個警告：

```
/**
 * Tiger 之前的普通 method
 */
@SuppressWarnings(value={"unchecked", "fallthrough"})
public void nonGenericsMethod(  ) {
  List wordList = new ArrayList(  );

  wordList.add("foo");
}
```

事實上，還有一種註解的方式可以採用－僅有一個 member 的 annotation 型別會自動的傳入所有的值給該 member，只要此 member 的名稱是 value 就可以。因此，你可以省略掉宣告中的 value= 這一部份：

花括號用在 member 是值的 array 的時候。

```
/**
 * Tiger 之前的普通 method
 */
@SuppressWarnings({"unchecked", "fallthrough"})
public void nonGenericsMethod(  ) {
  List wordList = new ArrayList(  );

  wordList.add("foo");
}
```

至少在寫書時，這可以讓我少打一些字。編譯器會夠聰明的知道要把這些值導給此 annotation 型別的 value 這個唯一的 member。

建構自訂的 Annotation 型別

你所看過的三個 annotation 型別是很有用的，但卻很難就這樣涵蓋到所有你想要在你自己的原始程式碼中使用的 annotation 類型。事實上，你可能會因為涉入 annotation 甚深而想要自己定義一些。幸好，Tiger 能夠讓你使用 @interface 關鍵字，以及幾個在語言中古怪的新語法結構來這麼做。

我該怎麼做？

annotation 型別在最基礎的層次上，是個 Java 的 interface。因此，它們看起來很類似於一般 Java interface 的定義，但你是用 @interface 關鍵字來取代 interface，這會告訴編譯器你是在撰寫一個 annotation 型別而不是一般的 interface。範例 6-4 是個很簡單的 marker interface。

範例 6-4. 簡單的 marker 類 annotation 型別

```
package com.oreilly.tiger.ch06;

/**
 * 指示 method 或 class 仍然在發展中的 marker 類 annotation
 */
public @interface InProgress { }
```

你可以對任何你想要的 method 或 class 使用它：

```
  @com.oreilly.tiger.ch06.InProgress
  public void calculateInterest(float amount, float rate) {
    // 稍後要把這個 method 寫完
  }
```

在這個範例中，你不需要對 "InProgress" 的 package 下 prefix，但在此處有下的原因是為了展現出它們是如同其它 Java 的 class 一樣的被使用。

你可以一樣容易的定義一個有 member 的 annotation 型別，如範例 6-5 所示。

範例 6-5. 有一個 member 的 annotation

```
package com.oreilly.tiger.ch06;

/**
 * 指示有工作需要完成的 annotation 型別
 */
public @interface TODO {
  String value( );
}
```

很奇怪的，這些搞 Java 的人們沒有沿用 JavaBean 的「速食（fried-and-true）」命名傳統來作出像是 setXXX() 與 getXXX() 的 method。

member 的宣告是相當的簡單，但後面的括號看起來有點奇怪－因為那不只是個 member 的宣告，同時也是 method 的宣告。你實際上定義了一個稱為 value() 的 method，然後編譯器自動的使用相同名稱建立一個 member 變數。在同樣的規則下，因為 annotation 型別的功能像是個 interface，所以這些 method 都是隱含的 abstract，且沒有程式本體。

稱為 value 的變數讓被參考此型別的原始程式碼能夠使用 @TODO(stringValue) 的縮寫來使用。這是個相當好的功能，你應該要盡可能的加以使用。你會在下面看到它的運用：

```
@com.oreilly.tiger.ch06.InProgress
@TODO("Figure out the amount of interest per month")
public void calculateInterest(float amount, float rate) {
  // 稍後要把這個 method 寫完
}
```

加入額外（或者兩個）的 member 也是相當的簡單，如範例 6-6 所展示出的。

這其實是範例 6-5 中 "TODO" 的加強版。

範例 6-6. 在 annotation 型別中的多個 member

```
package com.oreilly.tiger.ch06;

public @interface GroupTODO {

  public enum Severity { CRITICAL, IMPORTANT, TRIVIAL, DOCUMENTATION };

  Severity severity(  );
  String item(  );
  String assignedTo(  );
}
```

在此處，enumerated 型別被 severity 這個 member 用來「加料」。item 還是用來保存需要被處理的項目，而 assignedTo 提供了 TODO 項目被指派的個人：

enumerated 型別在第三章有說明。

```
@com.oreilly.tiger.ch06.InProgress
@GroupTODO(
  severity=GroupTODO.Severity.CRITICAL,
  item="Figure out the amount of interest per month",
  assignedTo="Brett McLaughlin"
)
public void calculateInterest(float amount, float rate) {
  // 稍後要把這個 method 寫完
}
```

最後一項，你可以設定 member 的值使用預設值，雖然說這樣的語法會更奇怪；範例 6-7 是範例 6-6 的更新版本。

範例 6-7. 在 annotation 型別中的預設值

```
package com.oreilly.tiger.ch06;

import java.util.Date;

public @interface GroupTODO {

  public enum Severity { CRITICAL, IMPORTANT, TRIVIAL, DOCUMENTATION };

  Severity severity(  ) default Severity.IMPORTANT;
  String item(  );
  String assignedTo(  );
  String dateAssigned(  );
}
```

這或許比讓我輸入新的 generic 語法還要更奇怪，所以可能得要花點時間才能適應過來。然而，能夠設定預設值還是很好的，不管它是否在語法上有點怪異。

有關於 ...

...extending 其它的 interface，甚至是 annotation 型別？不可以。@interface 關鍵字隱含著指示出一個 java.lang.annotation.Annotation 的延伸—你無法編譯一個明確嚐試要 extend 任何其它東西的 annotation 型別。然而，你可以 extend 並實作 annotation 型別，雖然這樣子的延伸與實作不會被當作 annotation 型別來對待。

對 Annotation 作 Annotate

如同你可以 annotate 類別，你同樣也可以 annotate 自訂的 annotation。乍看之下這似乎有點蠢，但是如果你開始建構一個大型的自訂註記庫，這就會變得相當的重要。就好像 Javadoc 與 comments 對一個正在研究已經寫好的 class 的程式設計師來說是非常有用的，meta-annotation，或者說是 annotation 的 annotation，對要辨識其他人自訂 annotation 的意圖是不可或缺的。

我該怎麼做？

有四種標準的 meta-annotation，全部都定義在 java.lang.annotation 這個 package 中：

Target

這個 meta-annotation 指定哪個程式元素可以有其所定義的 annotation。

Retention

此 meta-annotation 指示是否一個 annotation 要被編譯器給丟掉，或者是保留在編譯過的 class 檔案中。在 annotation 被保留的情況下，它也指定是否它會在 Java virtual machine 載入 class 時讀取該 annotation。

Documented

這個 meta-annotation 指示被定義的 annotation 應該被視為所註記之程式元素的公開 API 之一。

Inherited

此 meta-annotation 應用於目標為 class 的 annotation 型別上，指示出此被 annotated 的型別是繼承下來的。

以上都有某種程度上的自我說明性質，所以只要幾個範例就可以讓這些 meta-annotation 很容易的上手。每一個都會在接下來的幾個分節中討論到細節。

定義 Annotation 型別的 Target

第一個要討論的 meta-annotation 是 Target，它是用來指定所定義的 annotation 可以用在哪些程式元素上。這可以防止 annotation 的誤用，所以強力推薦用來作為你自訂 annotation 的健康檢查。

我該怎麼做？

Target 應該直接用在 annotation 定義的行之前：

```
@Target({ElementType.TYPE,
         ElementType.METHOD,
         ElementType.CONSTRUCTOR,
         ElementType.ANNOTATION_TYPE})
public @interface TODO {
```

在 Target 中真正的參數型別是 ElementType[]。

Target 採用單一的一個 member，型態是值的 array，每一個值都應該是來自 java.lang.annotation.ElementType 這個 enum 的 enumerated 值。

此 enum 定義了 annotation 型別中各種允許作為目標的程式元素，展示於範例 6-8。

範例 6-8. ElementType 的 enum

```
package java.lang.annotation;

public enum ElementType {
  TYPE,             // Class, interface, or enum ( 但不是 annotation)
```

```
  FIELD,            // Field ( 包括 enumerated values)
  METHOD,           // Method ( 不包括 constructors)
  PARAMETER,        // Method parameter
  CONSTRUCTOR,      // Constructor
  LOCAL_VARIABLE,   // Local variable 或 catch 述句
  ANNOTATION_TYPE,  // Annotation Types (meta-annotations)
  PACKAGE           // Java Package
}
```

你會記得要 import 進 Target 與 ElementType 到你的程式碼中。範例 6-9 展示更新過的 TODO 這個 annotation 型別，它最開始是在範例 6-5 定義的。

範例 6-9. 對 annotation 作 annotate

```
package com.oreilly.tiger.ch06;

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

/**
 * 指示還需要完成的工作之 annotation 型別
 */
@Target({ElementType.TYPE,
         ElementType.METHOD,
         ElementType.CONSTRUCTOR,
         ElementType.ANNOTATION_TYPE})
public @interface TODO {
  String value(  );
}
```

值得一提的是 Target 這個 meta-annotation 被用在自己身上的情形（見範例 6-10），指示它僅能作為 meta-annotation。

範例 6-10. Target annotation 型別的程式碼

```
package java.lang.annotation;

@Documented
@Retention(RetentionPolicy.RUNTIME);
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
  ElementType[] value(  );
}
```

有關於 ...

... 所有程式元素都適用的 annotation 型別？在這種情況下，你就不需要使用 Target 了。

有些人為了說明文件的緣故建議用 Target 來指示所有可用的 ElementTypes。老實說，我認為這樣子做很蠢。

設定 Annotation 型別的 Retention

Retention 這個 meta-annotation 定義 Java 的編譯器應該如何對待 annotation。annotation 可被編譯器排除於編譯過的 class 檔案之外，或者保留在 class 檔案中。此外，Java virtual machine 可以忽略 annotation（當它們被保留在 class 檔案時），或者在 class 第一次被載入時讀取那些 annotation。所有這些選項全部都由 Retention 來定義。

我該怎麼做？

如同 Target，你在 annotation 的定義之前（public @interface 這一行）指定此 annotation 型別的 retention。還有，也如同 Target，Retention 的參數必須來自支援 enum 之 class 的值－在此例中為 java.lang.annotation.RetentionPolicy。此 enum 顯示在範例 6-11。

範例 6-11. RetentionPolicy 這個 enum

```
package java.lang.annotation;

public enum RetentionPolicy {
  SOURCE,       // Annotation 會被編譯器給丟棄
  CLASS,        // Annotation 保留在 class 檔案中，但會被 VM 所忽略
  RUNTIME       // Annotation 保留在 class 檔案中且由 VM 讀取
}
```

對所有的 annotation 來說，預設的 retention 政策（policy）是 RetentionPolicy.CLASS。這會保留 annotation，但不需要 VM 於載入 class 時去讀取它們。

別忘了 import 進 java.lang.annotation.Retention 與 java.lang.annotation.RetentionPolicy。

一個使用 Retention 的好例子是發生在 SuppressWarnings 這個 annotation 上。因為該 annotation 型別純粹是用在編譯上（確保特定型態的警告會被抑止），它不需要保留在 class 的 bytecode 中：

```
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
```

製作 Annotation 型別的說明文件

如果你使用過新的 Ant 的 buildfile，可以用 "ant Javadoc" 來產生 Javadoc。

annotation 是個很棒的「添加物」，並且在你需要除錯、更新、或維護由他人撰寫的程式碼時更是個酷的不得了的功能。它同時也造就了在開放源碼專案上的「殺手級」程式碼階層管理系統。在《建構自訂的 Annotation 型別》一節中，我開發了一個非常簡單的 annotation 型別－InProgress、TODO、與 GroupTODO－在那種環境下是可以正常運作的。也許那些型別還算不錯，但如果你有完整的研究過程式碼，會發現他們在程式碼的 Javadoc 中是看不到的。這正是

Documented 這個 meta-annotation 發揮作用的地方。你可以用它來確保你的 annotation 會顯示在所產生的 Javadoc 之中。

我該怎麼做？

首先，要了解還有什麼是產生本書原始程式碼的 Javadoc 時沒有使用 Documented 而會失去的細節。打開那些 API 的文件，瀏覽到 com.oreilly.tiger.ch06 這個 package，然後看到 AnnotationTester 這個 class。如果有需要的話就向下捲動到 calculateInterest() 這個 method－你應該會看到類似圖 6-1 的說明。

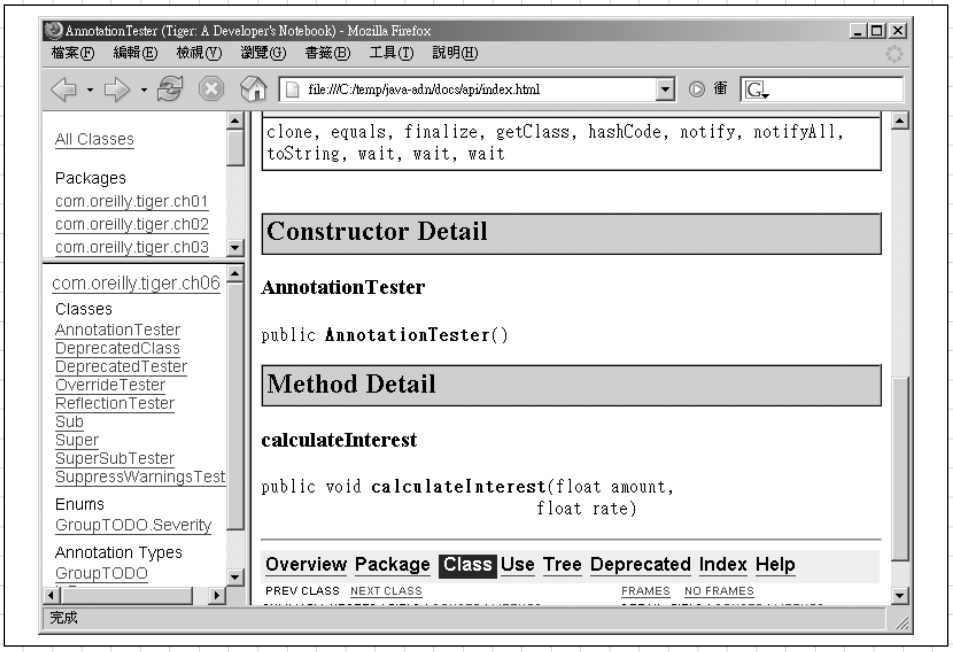

圖 6-1. 沒有 annotation 說明文件的 calculateInterest()

沒有什麼特別的，對吧？沒錯－但是有東西不見了。還記得這個 method 的程式碼：

```
@com.oreilly.tiger.ch06.InProgress
@GroupTODO(
  severity=GroupTODO.Severity.CRITICAL,
  item="Figure out the amount of interest per month",
  assignedTo="Brett McLaughlin",
  dateAssigned="04-26-2004"
)
public void calculateInterest(ßoat amount, ßoat rate) {
  // 稍後要把這個 method 寫完
}
```

在程式碼中的 InProgress 與 GroupTODO 這兩個 annotation 帶有相當重要的資訊，而這些資訊在此 Javadoc 中不見了。

要解決這個問題，你會需要加入 @Documented 這個 meta-annotation 到任何你想要在 Javadoc 中顯示的 annotation 型別中。在此例中，InProgress 與 GroupTODO、以及 TODO，可以使用這個外加項目。範例 6-12 展示更新過的 InProgress；你可以在其它的 annotation 型別中加入相同的程式行。

範例 6-12. 對 InProgress 加入說明文件

```
package com.oreilly.tiger.ch06;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/**
 * 指示 method 或 class 仍然在發展中的 marker 類 annotation
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress { }
```

我同時也加入了 Retention 這個 meta-annotation 到這個 class 中；任何時候你使用到 Documented 這個 annotation，你就應該成對的加入 RetentionPolicy.RUNTIME 這個 retention 政策。對定義在 com.oreilly.tiger.ch03 的其它 annotation 型別也做出同樣的修改。

"ant clean" 會刪除掉所有編譯過的 class 檔案與 Javadoc 檔案，能夠讓你產生乾淨的 Javadoc。

清除掉舊的 Javadoc 檔案。現在重新編譯你的 class 並再一次的執行 Javadoc 產生器。這一次，你會得到有一點不一樣的輸出－再看一下 com.oreilly.tiger.ch06，檢視 AnnotationTester，進到 calculateInterest() 這個 method。圖 6-2 顯示同樣的 method，但這一次該 annotation 出現在 Javadoc 中。

設定 Annotation 的繼承

有一種狀況百出的典型情境來自於繼承。考慮到某一個 class 被設定為 deprecated，並且打算要逐漸地廢除掉。那是很有可能（並且相當容易的）會讓一個不進入狀況的程式設計師把這個 class 拿來 extend，用的到處都是，並且還規避可能會出現的 deprecated 警告，如果有使用到它的 superclass 的話。這個問題來自於一個被 deprecated 的 class 沒有把 deprecation 狀態傳遞給它的 subclass－因此在你的程式碼中建構出一個「很有前途」的問題。一下子你突然

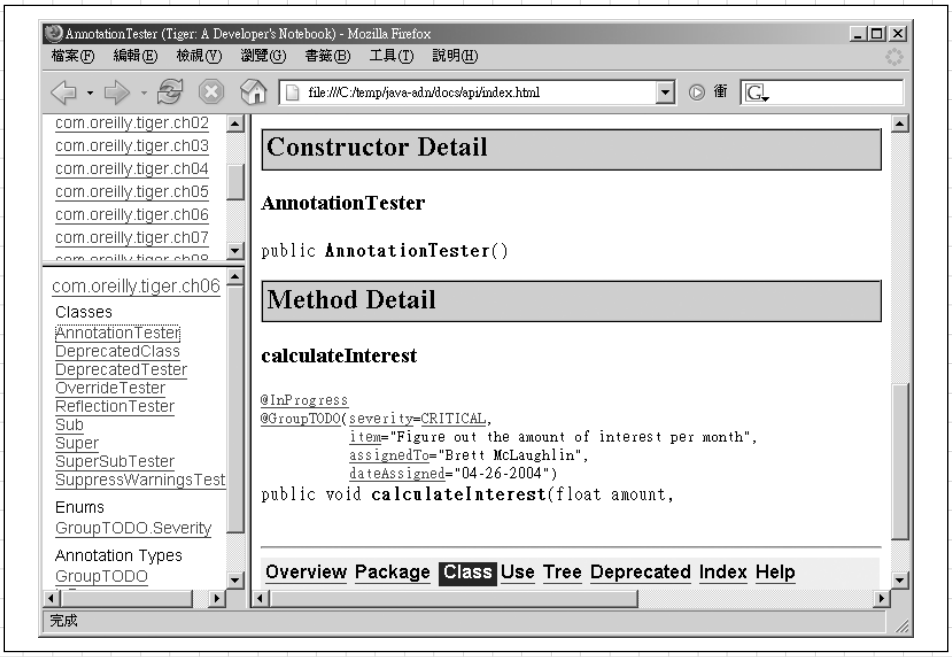

圖 6-2. 顯示 annotation 的 Javadoc

有了數以千計個對非 deprecated 之 class 的參考，但你又不能移除掉那個 deprecated 的 class，因為它是所有參考到的物件的 superclass！使用 Inherited 這個 meta-annotation 可以幫你脫困。

我該怎麼做？

讓我們來看另外一個用應該被繼承的 annotation 來表示狀態的範例：InProgress 的指示器。範例 6-13 展示一個使用這個 annotation 的簡單 class。

範例 6-13. 使用 InProgress 的 annotation

```
package com.oreilly.tiger.ch06;

import java.io.IOException;
import java.io.PrintStream;

@InProgress
public class Super {

  public void print(PrintStream out) throws IOException {
    out.println("Super printing...");
  }
}
```

範例 6-14 是另外一個 extend 了 Super 的簡單 class。

範例 6-14. Super 的 extending

```
package com.oreilly.tiger.ch06;

import java.io.IOException;
import java.io.PrintStream;

public class Sub extends Super {

  public void print(PrintStream out) throws IOException {
    out.println("Sub printing...");
  }
}
```

如果你對這些 class 產生 Javadoc，你會看到如圖 6-3 所示，Super 有被正確的標註為「正在開發中的」。

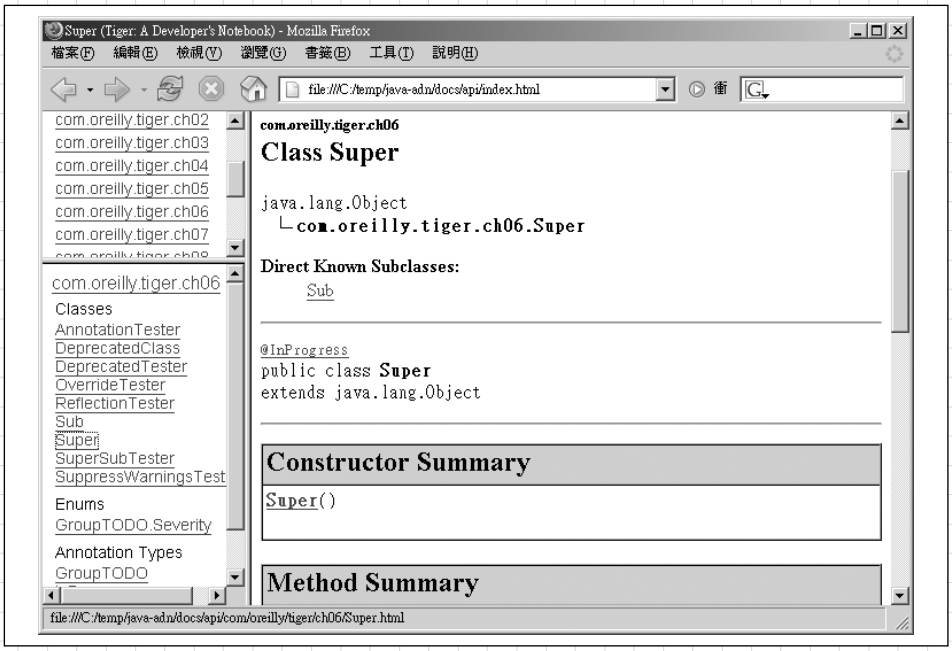

圖 6-3. Super，帶有 InProgress 的 annotation

問題在於 Sub，它是根據一個開發中的 class 所建構的，並沒有這樣的指示器（見圖 6-4）。

問題就在於 InProgress 這個 annotation 型別沒有被繼承下來；這個使用情境顯示出那或許會是個錯誤。要解決這個錯誤，可對 InProgress 做出以下的改變：

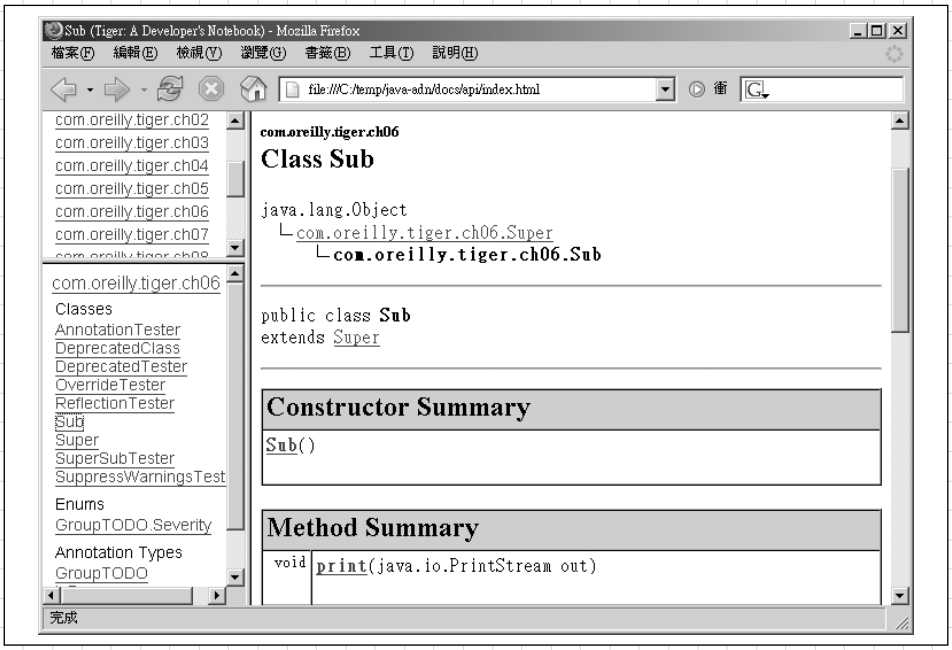

圖 6-4. Sub，並沒有 annotation

```
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress { }
```

重新編譯你的 class，並重新產生你的 Javadoc。我敢跟你賭你一定會預期看到另外一個圖上面 Sub 被展開且標示著開發中。呃，那也是我所預期的－但那還沒有好。在假設的情況中，這在最終一刻前會被修正過來，但希望你至少有關於應該要發生什麼事情的概念。

值得注意的還有說明文件的功能並沒有正確的運作，但是 reflection 有。你可以透過 reflection 確實的判別 Sub 還是在開發階段－但它沒有顯示在 Javadoc 中。想要了解 reflection 如何的運用 annotation，請見《Annotation 的 Reflecting》一節。

有關於 ...

...interface？被標示為 Inherited 的 annotation 僅能施用於 subclass，所以實作一個帶有 annotation 的 method 並不會產生出會被繼承的 annotation。

此外，如果你從 superclass 中 override 一個 method，你並沒有繼承下此 method 的 annotation。僅有在 superclass 本身中的 annotation 會被繼承下來，且是全部的被 subclass 給繼承下來。

TIP

是的，這是火星文。停下來，深呼吸三次，然後再讀一遍。我保證你會看得懂。

Annotation 的 Reflecting

到目前為止，所有對 annotation 的討論都圍繞著以肉眼的角度來看它們－不管是從程式碼或者 Javadoc。然而，現今已有許多值得一提的程式自我檢查工具會使用 reflection 來判斷某個 class（或 field、或 method）具有什麼樣的 annotation。java.lang.reflect 這個 package 具有數個添加物能讓這工作易如反掌。

我該怎麼做？

檢查 annotation 最簡單的方法是使用 isAnnotationPresent() 這個 method。它讓你指定要檢查的 annotation，然後傳回 true/false 的結果：

此程式碼是在 ReflectionTester 這個 class 中。

```
public void testAnnotationPresent(PrintStream out) throws IOException {
  Class c = Super.class;
  boolean inProgress = c.isAnnotationPresent(InProgress.class);
  if (inProgress) {
    out.println("Super is In Progress");
  } else {
    out.println("Super is not In Progress");
  }
}
```

執行上面的程式碼會產生下面的輸出：

```
run-ch06:
     [echo] Running Chapter 6 examples from Java Tiger:
            A Developer's Notebook
     [echo] Running ReflectionTester...
     [java] Super is In Progress
```

另外，採用這個方式可讓你利用到 Inherited 這個 annotation 的好處，其細節描述於《設定 Annotation 的繼承》一節：

這部分有假設你已經照著《設定 Annotation 的繼承》一節標示 "Super" 為開發中。

```
public void testInheritedAnnotation(PrintStream out) throws IOException {
  Class c = Sub.class;
  boolean inProgress = c.isAnnotationPresent(InProgress.class);
  if (inProgress) {
    out.println("Sub is In Progress");
```

```
    } else {
      out.println("Sub is not In Progress");
    }
  }
```

要記得雖然 Sub 沒有被標示為開發中，但它是繼承自開發中的 Super。此外，InProgress 這個 annotation 是被標示成 Inherited，所以「開發中」的這個指示器應該會被傳遞到 subclass。執行這個新的 method 會顯示出它確實有用：

```
run-ch06:
     [echo] Running Chapter 6 examples from Java Tiger: A Developer's Notebook
     [echo] Running VarargsTester...
     [java] Super is In Progress
     [java] Sub is In Progress
```

雖然這沒有被選入 Javadoc，但可以確定會出現在你的 reflection 導向程式碼中。

如果你沒有檢查到 marker 的 interface，你就必須要找到 isAnnotationPresent() 以外的方法—特別是當你需要從 annotation 取得值的時候。下面是個簡單的例子：

在 annotation 上的 reflection 僅在執行期有保留的 annotation 型別上有作用。

```
  public void testGetAnnotation(PrintStream out)
    throws IOException, NoSuchMethodException {

    Class c = AnnotationTester.class;
    MethodElement element = c.getMethod("calculateInterest",
                                ßoat.class, ßoat.class);

    GroupTODO groupTodo = element.getAnnotation(GroupTODO.class);
    String assignedTo = groupTodo.assignedTo(  );

    out.println("TODO Item on Annotation Tester is assigned to: '" +
        assignedTo + "'");
  }
```

一旦問題中的 method 被找到（在此例中為 AnnotationTester 類別的 calculateInterest()），就可以查詢該 method 的特定 annotation。在這個例子裡，程式碼找尋的是 GroupTODO 這個 annotation，並取出 assignedTo 的值。此 method 的輸出顯示在下面：

```
run-ch06:
     [echo] Running Chapter 6 examples from Java Tiger:
            A Developer's Notebook

     [echo] Running ReßectionTester...
     [java] Super is In Progress
     [java] Sub is In Progress
     [java] TODO Item on Annotation Tester is assigned to:
            'Brett McLaughlin'
```

要能夠使用這程式碼，很明顯的你必須要很清楚的知道你在找什麼－那是 getAnnotation() 的少數幾個缺點之一。

for/in 迴圈在第七章有詳述，而 printf() 與其它的格式化 method 則涵蓋於第九章。

最後，如果嚐試要找尋某程式元素的所有 annotation，或者需要 iterate 過所的 annotation 來找尋特定的目標，你可以使用 getAnnotations()。舉例來說，下面是個簡單的工具 method 能夠列出所指定之元素的所有 annotation：

```
public void printAnnotations(AnnotatedElement e, PrintStream out)
  throws IOException {

  out.printf("Printing annotations for '%s'%n%n", e.toString( ));

  Annotation[] annotations = e.getAnnotations( );
  for (Annotation a : annotations) {
    out.printf("    * Annotation '%s' found%n",
      a.annotationType( ).getName( ));
  }
}
```

如果你將 AnnotationTester 的 calculateInterest() 這個 method 餵進去，你會看到下面的輸出：

```
run-ch06:
     [echo] Running Chapter 6 examples from Java Tiger: A Developer's Notebook
     [echo] Running ReßectionTester...
     [java] Super is In Progress
     [java] Sub is In Progress
     [java] TODO Item on Annotation Tester is assigned to: 'Brett McLaughlin'
     [java] Printing annotations for 'public void com.oreilly.tiger.ch06.Annotat
ionTester.calculateInterest(¤oat,¤oat)'

     [java]     * Annotation 'com.oreilly.tiger.ch06.InProgress' found
     [java]     * Annotation 'com.oreilly.tiger.ch06.GroupTODO' found
```

這程式碼是相當的直接了當，所以我把細節探討的部分留給你自己做。範例 6-15 是 ReflectionTester 的完整程式碼列表，全部 reflection 導向的 annotation method 都在這裡。

範例 6-15. 測試 reflection 導向的 annotation method

如果你不想取出繼承的 annotation，你可以使用 getDeclaredAnnotations() 來代替 getAnnotations()。

```
package com.oreilly.tiger.ch06;

import java.io.IOException;
import java.io.PrintStream;
import java.lang.reßect.AnnotatedElement;
import java.lang.annotation.Annotation;

public class ReßectionTester {
```

```
public ReßectionTester(  ) {
}

public void testAnnotationPresent(PrintStream out) throws IOException {
  Class c = Super.class;
  boolean inProgress = c.isAnnotationPresent(InProgress.class);
  if (inProgress) {
    out.println("Super is In Progress");
  } else {
    out.println("Super is not In Progress");
  }
}

public void testInheritedAnnotation(PrintStream out) throws IOException {
  Class c = Sub.class;
  boolean inProgress = c.isAnnotationPresent(InProgress.class);
  if (inProgress) {
    out.println("Sub is In Progress");
  } else {
    out.println("Sub is not In Progress");
  }
}

public void testGetAnnotation(PrintStream out)
  throws IOException, NoSuchMethodException {

  Class c = AnnotationTester.class;
  AnnotatedElement element = c.getMethod("calculateInterest",
                                ßoat.class, ßoat.class);

  GroupTODO groupTodo = element.getAnnotation(GroupTODO.class);
  String assignedTo = groupTodo.assignedTo(  );

  out.println("TODO Item on Annotation Tester is assigned to: '" +
      assignedTo + "'");
}

public void printAnnotations(AnnotatedElement e, PrintStream out)
  throws IOException {

  out.printf("Printing annotations for '%s'%n%n", e.toString(  ));

  Annotation[] annotations = e.getAnnotations(  );
  for (Annotation a : annotations) {
    out.printf("    * Annotation '%s' found%n",
      a.annotationType(  ).getName(  ));
  }
}
```

AnnotatedElement 是 reflection 建構（如同 method 與 class）所實作的一個新的 interface。它能讓此程式碼存取使用到的新 annotation method。

範例 6-15. 測試 reflection 導向的 annotation method（續）

```
    public static void main(String[] args) {
      try {
        ReßectionTester tester = new ReßectionTester( );

        tester.testAnnotationPresent(System.out);
        tester.testInheritedAnnotation(System.out);

        tester.testGetAnnotation(System.out);

        Class c = AnnotationTester.class;
        AnnotatedElement element = c.getMethod("calculateInterest",
                                               ßoat.class, ßoat.class);
        tester.printAnnotations(element, System.out);
      } catch (Exception e) {
        e.printStackTrace( );
      }
    }
  }
```

發生了什麼事？

我已經把 generic 語法簡化以便讓它更清楚一點，檢視 Javadoc 上的 AnnotatedElement 可看到更詳細的 method 參數與回傳型別說明。

你剛看到的這些程式碼的最重大關鍵是 java.lang.reßect.AnnotatedElement 這個新的 interface。在 Tiger 中，核心的 reflection 建構都有實作這個 interface，包括：Class、Constructor、Field、Method、Package、與 AccessibleObject。這使得你剛看過的程式碼能夠對 annotation 作自我檢查－有實作 AnnotatedType 的結果就是這些元素型別都提供了下列的 method：

```
public Annotation getAnnotation(Class annotationType);

public Annotation[] getAnnotations( );

public Annotation[] getDeclaredAnnotations( );

public boolean isAnnotationPresent(Class annotationType);
```

因為所有的 Java 程式元素都能夠當作一個 AnnotatedType，你永遠都可以使用這些 method 來取得某個元素的 annotation。

有關於 ...

... 沒有被標記為執行期可見的 annotation？回憶到你必須明確的設定某個 annotation 的保留狀態為 RetentionpPolicy.RUNTIME 才能使其發生作用。就算 annotation 在編譯過程中被保留下來（預設的行為），如果 VM 沒有在載入期

間讀入被保留的訊息，則 reflection 無法讀出這 annotation。事實上，這是為什麼 Inherited 與 Documented 兩個 annotation 應該都要與下列的 annotation 成對的使用：

```
@Retention(RetentionPolicy.RUNTIME)
```

這會確保你的說明文件與 / 或繼承能夠確實的讓程式碼自我檢查工具讀取。