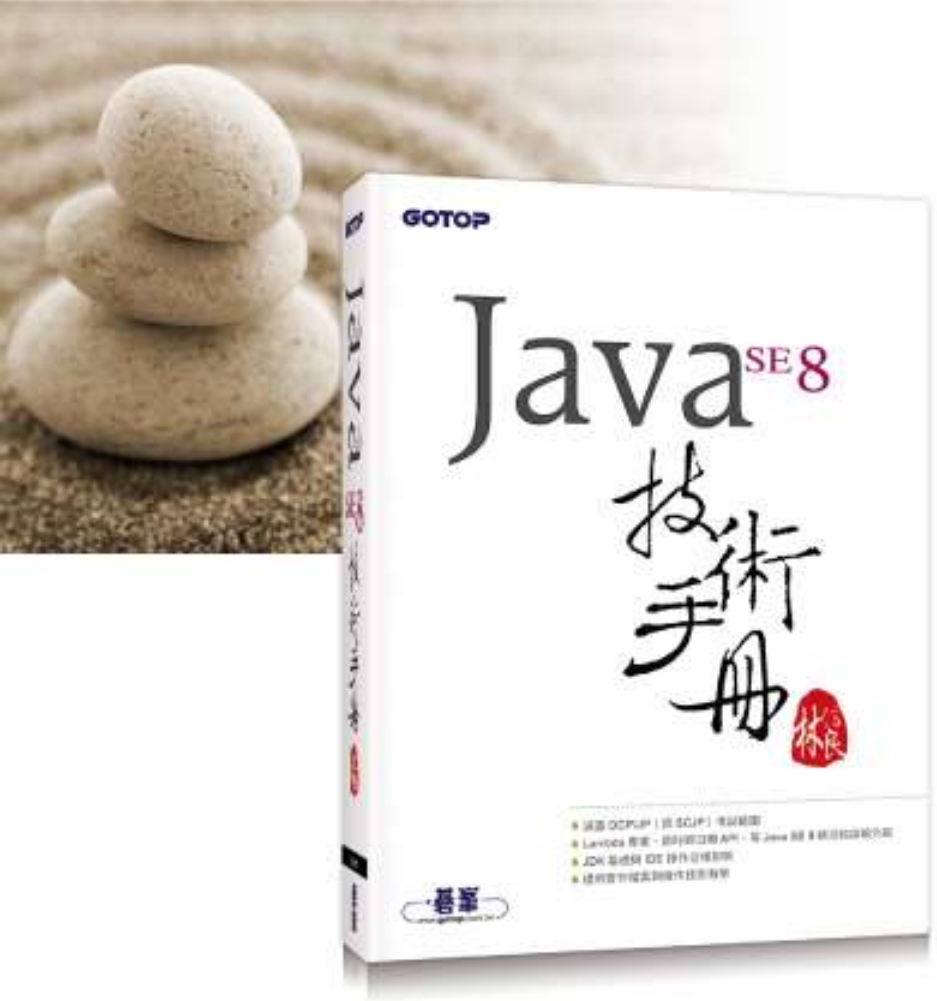


Java SE 8 技術手冊

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

例外處理

學習目標

- 使用try、catch處理例外
- 認識例外繼承架構
- 認識throw、throws的使用時機
- 運用finally關閉資源
- 使用自動關閉資源語法
- 認識AutoCloseable介面

使用try、catch

```
Scanner console = new Scanner(System.in);
double sum = 0;
int count = 0;
while(true) {
    int number = console.nextInt();
    if(number == 0) {
        break;
    }
    sum += number;
    count++;
}
System.out.printf("平均 %.2f%n", sum / count);
```

```
10 20 3o 40 0
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at cc.openhome.Average.main(Average.java:11)
Java Result: 1
```

使用try、catch

- 所有錯誤都會被包裹為物件，如果你願意，可以**嘗試（try）**執行程式並**捕捉（catch）**代表錯誤的物件後作一些處理

使用try、catch

```
try {
    Scanner console = new Scanner(System.in);
    double sum = 0;
    int count = 0;
    while (true) {
        int number = console.nextInt();
        if (number == 0) {
            break;
        }
        sum += number;
        count++;
    }
    System.out.printf("平均 %.2f%n", sum / count);
} catch (InputMismatchException ex) {
    System.out.println("必須輸入整數");
}
```

```
10 20 3o 40 0
必須輸入整數
```

使用try、catch

- 有時錯誤可以在捕捉處理之後，繼續程式正常執行流程

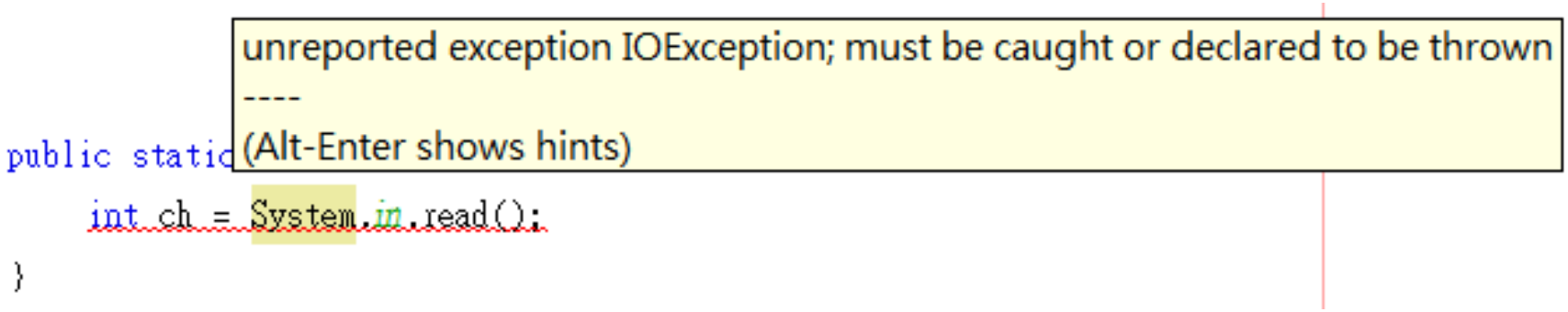
使用try、catch

```
Scanner console = new Scanner(System.in);
double sum = 0;
int count = 0;
while (true) {
    try {
        int number = console.nextInt();
        if (number == 0) {
            break;
        }
        sum += number;
        count++;
    } catch (InputMismatchException ex) {
        System.out.printf("略過非整數輸入：%s%n", console.next());
    }
}
System.out.printf("平均 %.2f%n", sum / count);
```

```
10 20 3o 40 0
略過非整數輸入：3o
平均 23.33
```

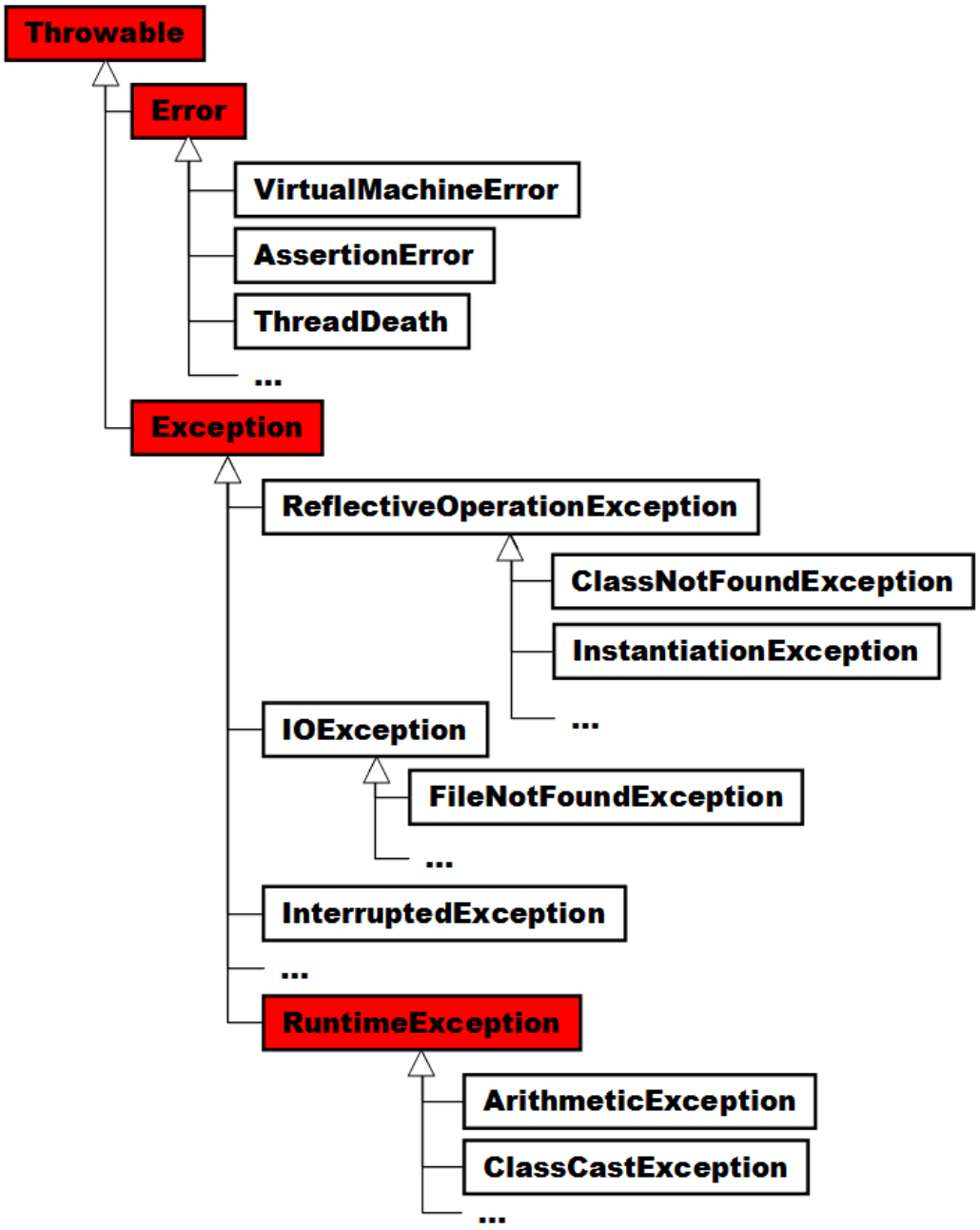
例外繼承架構

- 先前的Average範例中，雖然沒有撰寫try、catch語句，照樣可以編譯執行，如果如下撰寫，編譯卻會錯誤？



例外繼承架構

- 要解決這個錯誤訊息有兩種方式
  - 使用try、catch包裹System.in.read()
  - 在main()方法旁宣告throws java.io.IOException


Throwable
Error
VirtualMachineError
AssertionError
ThreadDeath
...
Exception
ReflectiveOperationException
ClassNotFoundException
InstantiationException
...
IOException
FileNotFoundException
...
InterruptedException
...
RuntimeException
ArithmeticException
ClassCastException
...

例外繼承架構

- 錯誤會包裝為物件，這些物件都是可拋出的，因此設計錯誤物件都繼承自**java.lang.Throwable**類別
- Throwable定義了取得錯誤訊息、堆疊追蹤（Stack Trace）等方法，它有兩個子類別：**java.lang.Error**與**java.lang.Exception**

例外繼承架構

- **Error**與其子類別實例代表嚴重系統錯誤
- 雖然也可以使用try、catch來處理Error物件，但並不建議，發生嚴重系統錯誤時，Java應用程式本身是無力回復的
- **Error**物件拋出時，基本上不用處理，任其傳播至JVM為止，或者是最多留下日誌訊息

例外繼承架構

- **程式設計本身的錯誤，建議使用`Exception`或其子類別實例來表現，所以通常稱錯誤處理為例外處理**
- 單就語法與繼承架構上來說...
  - 如果某個方法宣告會拋出`Throwable`、`Exception`或子類別實例，但又不屬於**`java.lang.RuntimeException`**或其子類別實例，就必須明確使用`try`、`catch`語法加以處理，或者在方法用`throws`宣告這個方法會拋出例外，否則會編譯失敗

例外繼承架構

- 呼叫System.in.read()時，in其實是System的靜態成員，其型態為java.io.InputStream

read

```
public abstract int read()
                    throws IOException
```

例外繼承架構

- **`Exception`**或其子物件，但非屬於**`RuntimeException`**或其子物件，稱為受檢例外（Checked Exception）
  - 受檢例外存在之目的，在於API設計者實作某方法時，某些條件成立時會引發錯誤，而且認為呼叫方法的客戶端有能力處理錯誤，要求編譯器提醒客戶端必須明確處理錯誤

例外繼承架構

- 屬於**RuntimeException**衍生出來的類別實例，稱為非受檢例外（Unchecked Exception）
  - 代表API設計者實作某方法時，某些條件成立時會引發錯誤，而且認為API客戶端應該在呼叫方法前做好檢查，以避免引發錯誤

例外繼承架構

- 使用陣列時，若存取超出索引就會拋出`ArrayIndexOutOfBoundsException`，但編譯器並沒有強迫你在語法上加以處理

java.lang

Class ArrayIndexOutOfBoundsException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IndexOutOfBoundsException
          java.lang.ArrayIndexOutOfBoundsException

例外繼承架構

- 例如Average範例中，InputMismatchException設計為一種RuntimeException：

java.util

Class InputMismatchException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.util.NoSuchElementException
          java.util.InputMismatchException

例外繼承架構

- 如果父類別例外物件在子類別例外物件前被捕捉，則catch子類別例外物件的區塊將永遠不會被執行，編譯器會檢查出這個錯誤

```
try {
    System.in.read()
} catch(Exception ex
    ex.printStackTra
} catch(IOException ex) {
    ex.printStackTrace();
}
```

exception IOException has already been caught

(Alt-Enter shows hints)

例外繼承架構

- 要完成這個程式的編譯，必須更改例外物件捕捉的順序：

```
try {
    System.in.read();
} catch(java.io.IOException e) {
    e.printStackTrace();
} catch(Exception e) {
    e.printStackTrace();
}
```

例外繼承架構

- 發現到數個型態catch區塊在作相同的事情

```
try {
    作一些事...
} catch(IOException e) {
    e.printStackTrace();
} catch(InterruptedException e) {
    e.printStackTrace();
} catch(ClassCastException e) {
    e.printStackTrace();
}
```

例外繼承架構

- 在JDK7開始，可以如下使用多重捕捉（multi-cath）語法：

```
try {
    作一些事...
} catch(IOException | InterruptedException | ClassCastException e) {
    e.printStackTrace();
}
```

例外繼承架構

- catch括號中列出的例外不得有繼承關係，否則會發生編譯錯誤：

```
try {
    Syst
} catch(IOException | Exception ex) {
    ex.printStackTrace();
}
```

Alternatives in a multi-catch statement cannot be related by subclassing
Alternative IOException is a subclass of alternative Exception

The catch(java.lang.Exception) is too broad, the actually caught exception is java.io.IOException
---
(Alt-Enter shows hints)

要抓還是要拋？

- 開發一個程式庫，其中有個功能是讀取純文字檔案，並以字串傳回檔案中所有文字

```
public static String readFile(String name) {
    StringBuilder text = new StringBuilder();
    try {
        Scanner console = new Scanner(new FileInputStream(name));
        while(console.hasNext()) {
            text.append(console.nextLine())
                .append('\n');
        }
    } catch (FileNotFoundException ex) {
        ex.printStackTrace();
    }
    return text.toString();
}
```

要抓還是要拋？

- 老闆有說這個程式庫會用在文字模式中嗎？

```
public static String readFile(String name)
                                    throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    Scanner console = new Scanner(new FileInputStream(name));
    while(console.hasNext()) {
        text.append(console.nextLine())
            .append('\n');
    }
    return text.toString();
}
```

要抓還是要拋？

- 在例外發生時，可使用try、catch處理當時環境可進行的例外處理，當時環境下無法決定如何處理的部份，可以拋出由呼叫方法的客戶端處理

要抓還是要拋？

```
public class FileUtil {
    public static String readFile(String name) throws FileNotFoundException {
        StringBuilder text = new StringBuilder();
        try {
            Scanner console = new Scanner(new FileInputStream(name));
            while(console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
            throw ex;
        }
        return text.toString();
    }
}
```

❶ 宣告方法中會拋出例外

❷ 執行時拋出例外

要抓還是要拋？

- 如果原先有個方法實作是這樣的：

```
public static void doSome(String arg)
            throws FileNotFoundException, EOFException {
    try {
        if("one".equals(arg)) {
            throw new FileNotFoundException();
        } else {
            throw new EOFException();
        }
    } catch(FileNotFoundException ex) {
        ex.printStackTrace();
        throw ex;
    } catch(EOFException ex) {
        ex.printStackTrace();
        throw ex;
    }
}
```

要抓還是要拋？

- 以下的寫法在JDK6之前都會出錯：

```
static void doSome(String arg)
        throws FileNotFoundException, EOFException {
    try {
        if ("one".equals(arg)) {
            throw new FileNotFoundException();
        } else {
            throw new EOFException();
        }
    } catch (IOEx
        ex.print
        throw ex;
    }
}
```

unreported exception IOException; must be caught or declared to be thrown
---
(Alt-Enter shows hints)

要抓還是要拋？

- **在JDK7中，編譯器對於重新拋出的例外型態可以更精準判斷（more-precise-rethrow）**，因此上面的程式片段，在JDK7中不會再出現編譯錯誤

要抓還是要拋？

- 父類別某個方法宣告throws某些例外，子類別重新定義該方法時可以：
  - 不宣告throws任何例外
  - 可throws父類別該方法中宣告的某些例外
  - 可throws父類別該方法中宣告例外之子類別
  - 但是不可以：
  - throws父類別方法中未宣告的其它例外
  - throws父類別方法中宣告例外之父類別

貼心還是造成麻煩？

- 就撰寫本書的時間點來說，Java是唯一採用受檢例外（Checked exception）的語言
  - 文件化
  - 提供編譯器資訊

貼心還是造成麻煩？

- 有些錯誤發生引發例外時，你根本無力處理
- 例如使用JDBC撰寫資料庫連線程式時，經常要處理的`java.sql.SQLException`
- 假設方法是在整個應用程式非常底層被呼叫

```
public Customer getCustomer(String id) throws SQLException {
    ...
}
```

貼心還是造成麻煩？

- 為了讓例外往上浮現，你也許會選擇在每個方法呼叫上都宣告`throws SQLException`
  - 這樣的作法也許會造成許多程式碼的修改（更別說要重新編譯了）
  - 如果你根本無權修改應用程式的其他部份，這樣的作法顯然行不通。

貼心還是造成麻煩？

- 受檢例外本意良好，有助於程式設計人員注意到例外的可能性並加以處理
- 應用程式規模增大時，會對逐漸對維護造成困難
- 不一定是自訂API時發生，也可能是在底層引入了一個會拋出受檢例外的API而發生類似情況。

貼心還是造成麻煩？

- 重新拋出例外時，也可以考慮為應用程式自訂專屬例外類別，讓例外更能表現應用程式特有的錯誤資訊
- 通常建議繼承自`Exception`或其子類別
- 若不是繼承`Error`或`RuntimeException`，那麼就會是受檢例外

貼心還是造成麻煩？

- 如果認為呼叫API的客戶端應當有能力處理未處理的錯誤，那就自訂受檢例外

```
public class CustomizedException extends Exception { // 自訂受檢例外
    ...
}
```

- 如果認為呼叫API的客戶端沒有準備好就呼叫了方法，才會造成還有未處理的錯誤，那就自訂非受檢例外

```
public class CustomizedException extends RuntimeException { // 自訂非受檢例外
    ...
}
```

貼心還是造成麻煩？

- 一個基本的例子是這樣的：

```
try {
    ....
} catch(SomeException ex) {
    // 作些可行的處理
    // 也許是 Logging 之類的
    throw new CustomizedException("error message..."); // Checked 或 Unchecked?
}
```

貼心還是造成麻煩？

- 如果流程中要拋出例外，也要思考一下，這是客戶端可以處理的例外嗎？還是客戶端沒有準備好前置條件就呼叫方法，才引發的例外？

```
if(someCondition) {
    throw new CustomizedException("error message"); // Checked 或 Unchecked?
}
```

貼心還是造成麻煩？

- 有些開發者在設計程式庫時，乾脆就選擇完全使用非受檢例外
  - 選擇給予開發人員較大的彈性來面對例外（也許也需要開發人員更多的經驗）
- 隨著應用程式的演化，例外也可以考慮演化，也許一開始是設計為受檢例外，在經過考量後，可演化為非受檢例外

認識堆疊追蹤

```
public class StackTraceDemo {
    public static void main(String[] args) {
        try {
            c();
        } catch(NullPointerException ex) {
            ex.printStackTrace();
        }
    }

    static void c() {
        b();
    }

    static void b() {
        a();
    }

    static String a() {
        String text = null;
        return text.toUpperCase();
    }
}
```

認識堆疊追蹤

- 當例外發生而被捕捉後，可以呼叫 `printStackTrace()` 在顯示堆疊追蹤：

```
java.lang.NullPointerException
        at cc.openhome.StackTraceDemo.a(StackTraceDemo.java:22)
        at cc.openhome.StackTraceDemo.b(StackTraceDemo.java:17)
        at cc.openhome.StackTraceDemo.c(StackTraceDemo.java:13)
        at cc.openhome.StackTraceDemo.main(StackTraceDemo.java:6)
```

認識堆疊追蹤

- 要善用堆疊追蹤，前題是程式碼中不可有私吞例外的行為

```
try {
    ...
} catch(SomeException ex) {
    // 什麼也沒有，絕對不要這麼作！
}
```

- 這種程式碼會對應用程式維護造成嚴重傷害

認識堆疊追蹤

- 另一種對應用程式維護會有傷害的方式，就是對例外作了不適當的處理，或顯示了不正確的資訊

```
try {
    ...
} catch(FileNotFoundException ex) {
    作一些處理
} catch(EOFException ex) {
    作一些處理
}
```

```
try {
    ...
} catch(IOException ex) {
    System.out.println("找不到檔案");
}
```

認識堆疊追蹤

- 在使用throw重拋例外時，例外的追蹤堆疊起點，仍是例外的發生根源，而不是重拋例外的地方

```
public static void main(String[] args) {
    try {
        c();
    } catch(NullPointerException ex) {
        ex.printStackTrace();
    }
}

static void c() {
    try {
        b();
    } catch(NullPointerException ex) {
        ex.printStackTrace();
        throw ex;
    }
}

static void b() {
    a();
}

static String a() {
    String text = null;
    return text.toUpperCase();
}
```

```
java.lang.NullPointerException
        at cc.openhome.StackTraceDemo2.a(StackTraceDemo2.java:28)
        at cc.openhome.StackTraceDemo2.b(StackTraceDemo2.java:23)
        at cc.openhome.StackTraceDemo2.c(StackTraceDemo2.java:14)
        at cc.openhome.StackTraceDemo2.main(StackTraceDemo2.java:6)
java.lang.NullPointerException
        at cc.openhome.StackTraceDemo2.a(StackTraceDemo2.java:28)
        at cc.openhome.StackTraceDemo2.b(StackTraceDemo2.java:23)
        at cc.openhome.StackTraceDemo2.c(StackTraceDemo2.java:14)
        at cc.openhome.StackTraceDemo2.main(StackTraceDemo2.java:6)
```

```
public static void main(String[] args) {
    try {
        c();
    } catch(NullPointerException ex) {
        ex.printStackTrace();
    }
}

static void c() {
    try {
        b();
    } catch(NullPointerException ex) {
        ex.printStackTrace();
        Throwable t = ex.fillInStackTrace();
        throw (NullPointerException) t;
    }
}

static void b() {
    a();
}

static String a() {
    String text = null;
    return text.toUpperCase();
}
```

```
java.lang.NullPointerException
        at cc.openhome.StackTraceDemo3.a(StackTraceDemo3.java:28)
        at cc.openhome.StackTraceDemo3.b(StackTraceDemo3.java:23)
        at cc.openhome.StackTraceDemo3.c(StackTraceDemo3.java:14)
        at cc.openhome.StackTraceDemo3.main(StackTraceDemo3.java:6)
java.lang.NullPointerException
        at cc.openhome.StackTraceDemo3.c(StackTraceDemo3.java:17)
        at cc.openhome.StackTraceDemo3.main(StackTraceDemo3.java:6)
```

關於assert

- Java在JDK 1.4之後提供**assert**語法，有兩種使用的語法：

```
assert boolean_expression;
assert boolean_expression : detail_expression;
```

- `boolean_expression`若為`true`，則什麼事都不會發生，如果為`false`，則會發生**java.lang.AssertionError**

關於assert

- 為了避免JDK 1.3或更早版本程式使用assert作為變數導致名稱衝突問題，預設執行時不啟動斷言檢查
- 如果要在執行時啟動斷言檢查，可以在執行java指令時，指定**-enableassertions**或是**-ea**引數

關於assert

- 斷言客戶端呼叫方法前，已經準備好某些前置條件（通常在private方法之中）
- 斷言客戶端呼叫方法後，具有方法承諾的結果
- 斷言物件某個時間點下的狀態
- 使用斷言取代註解
- 斷言程式流程中絕對不會執行到的程式碼部份

關於assert

- 以第5章的CashCard物件為例

```
public void charge(int money) {
    if(money > 0) {
        if(money <= this.balance) {
            this.balance -= money;
        }
        else {
            out.println("錢不夠啦！");
        }
    }
    else {
        out.println("扣負數？這不是叫我儲值嗎？");
    }
}
```

```
public void charge(int money) throws InsufficientException {
    checkGreaterThanZero(money);
    checkBalance(money);
    this.balance -= money;

    // this.balance 不能是負數
}

private void checkGreaterThanZero(int money) {
    if(money < 0) {
        throw new IllegalArgumentException("扣負數？這不是叫我儲值嗎？");
    }
}

private void checkBalance(int money) throws InsufficientException {
    if(money > this.balance) {
        throw new InsufficientException("錢不夠啦！", this.balance);
    }
}
```

```
public void charge(int money) throws InsufficientException {
    assert money >= 0 : "扣負數？這不是叫我儲值嗎？";

    checkBalance(money);
    this.balance -= money;

    assert this.balance >= 0 : " this.balance 不能是負數";
}

private void checkBalance(int money) throws InsufficientException {
    if(money > this.balance) {
        throw new InsufficientException("錢不夠啦！", this.balance);
    }
}
```

關於assert

```
public static void play(int action) {
    switch(action) {
        case Action.STOP:
            out.println("播放停止動畫");
            break;
        case Action.RIGHT:
            out.println("播放向右動畫");
            break;
        case Action.LEFT:
            out.println("播放向左動畫");
            break;
        case Action.UP:
            out.println("播放向上動畫");
            break;
        case Action.DOWN:
            out.println("播放向下動畫");
            break;
        default:
            assert false : "非定義的常數";
    }
}
```

關於assert

- 控制流程不變量（Control flow invariant）判斷

```
public static void play(int action) {
    switch(action) {
        case Action.STOP:
            System.out.println("播放停止動畫");
            break;
        case Action.RIGHT:
            System.out.println("播放向右動畫");
            break;
        case Action.LEFT:
            System.out.println("播放向左動畫");
            break;
        case Action.UP:
            System.out.println("播放向上動畫");
            break;
        case Action.DOWN:
            System.out.println("播放向下動畫");
            break;
        default:
            assert false : "非定義的常數";
    }
}
```

使用finally

- 何時關閉資源呢？

```
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    Scanner console = new Scanner(new FileInputStream(name));
    while (console.hasNext()) {
        text.append(console.nextLine())
            .append('\n');
    }
    console.close();
    return text.toString();
}
```

使用finally

- **finally**區塊一定會被執行

```
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    Scanner console = null;
    try {
        console = new Scanner(new FileInputStream(name));
        while (console.hasNext()) {
            text.append(console.nextLine())
                .append('\n');
        }
    } finally {
        if(console != null) {
            console.close();
        }
    }
    return text.toString();
}
```

使用finally

- 如果程式撰寫的流程中先**return**了，而且也有寫**finally**區塊，那**finally**區塊會先執行完後，再將值傳回

```
public static void main(String[] args) {
    System.out.println(test(true));
}

static int test(boolean flag) {
    try {
        if(flag) {
            return 1;
        }
    } finally {
        System.out.println("finally...");
    }
    return 0;
}
```

自動嘗試關閉資源

- 在JDK7之後，新增了嘗試關閉資源（try-with-resources）語法

```
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    try(Scanner console = new Scanner(new FileInputStream(name))) {
        while (console.hasNext()) {
            text.append(console.nextLine())
                .append('\n');
        }
    }
    return text.toString();
}
```

自動嘗試關閉資源

- JDK7的嘗試關閉資源（try-with-resources）語法也是個編譯器蜜糖

```
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    Scanner console = new Scanner(new FileInputStream(name));
    Throwable localThrowable2 = null;
    try {
        while (console.hasNext()) {
            text.append(console.nextLine())
                .append('\n');
        }
    } catch (Throwable localThrowable1) {      // 嘗試捕捉所有錯誤
        localThrowable2 = localThrowable1;
        throw localThrowable1;
    }
    finally {
        if (console != null) {  // 如果 console 參考至 Scanner 實例
            if (localThrowable2 != null) {     // 若先前有 catch 到其他例外
                try {
                    console.close();           // 嘗試關閉 Scanner 實例
                } catch (Throwable x2) {       // 萬一關閉時發生錯誤
                    localThrowable2.addSuppressed(x2);    // 在原例外物件中記錄
                }
            } else {
                console.close();   // 若先前沒有發生任何例外，就直接關閉 Scanner
            }
        }
    }
    return text.toString();
}
```

自動嘗試關閉資源

- **addSuppressed()**方法是JDK7在java.lang.Throwable中新增的方法可將第二個例外記錄在第一個例外之中
- JDK7中與之相對應的是**getSuppressed()**方法，可傳回Throwable[]，代表先前被addSuppressed()記錄的各個例外物件

自動嘗試關閉資源

- 使用自動嘗試關閉資源語法時，也可以搭配 catch

```
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    try(Scanner console = new Scanner(new FileInputStream(name))) {
        while (console.hasNext()) {
            text.append(console.nextLine());
            text.append('\n');
        }
    } catch(FileNotFoundException ex) {
        ex.printStackTrace();
        throw ex;
    }
    return text.toString();
}
```

```
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    try(
        // 這個區塊中是自動嘗試關閉資源語法展開後的程式碼
        Scanner console = new Scanner(new FileInputStream(name));
        Throwable localThrowable2 = null;
        try {
            while (console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
        } catch (Throwable localThrowable1) {
            localThrowable2 = localThrowable1;
            throw localThrowable1;
        }
        finally {
            if (console != null) {
                if (localThrowable2 != null) {
                    try {
                        console.close();
                    } catch (Throwable x2) {
                        localThrowable2.addSuppressed(x2);
                    }
                } else {
                    console.close();
                }
            }
        }
    } catch(FileNotFoundException ex) {
        ex.printStackTrace();
        throw ex;
    }
    return text.toString();
}
```

java.lang.AutoCloseable介面

- JDK7的嘗試關閉資源語法可套用的物件，必須實作**java.lang.AutoCloseable**介面

java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:

Closeable, AutoCloseable, Iterator<String>

java.lang.AutoCloseable介面

- AutoCloseable是JDK7新增的介面，僅定義了close()方法：

```
package java.lang;
public interface AutoCloseable {
    void close() throws Exception;
}
```

GOTOP

Java SE 8 技術手冊

java.lang.AutoCloseable介面

java.lang

Interface AutoCloseable

All Known Subinterfaces:

AsynchronousByteChannel, AsynchronousChannel, BaseStream<T,S>, ByteChannel, CachedRowSet, CallableStatement, Channel, Clip, Closeable, Connection, DataLine, DirectoryStream<T>, DoubleStream, FilteredRowSet, GatheringByteChannel, ImageInputStream, ImageOutputStream, InterruptibleChannel, IntStream, JavaFileManager, JdbcRowSet, JMXConnector, JoinRowSet, Line, LongStream, MidiDevice, MidiDeviceReceiver, MidiDeviceTransmitter, Mixer, MulticastChannel, NetworkChannel, ObjectInput, ObjectOutput, Port, PreparedStatement, ReadableByteChannel, Receiver, ResultSet, RMIConnection, RowSet, ScatteringByteChannel, SecureDirectoryStream<T>, SeekableByteChannel, Sequencer, SourceDataLine, StandardJavaFileManager, Statement, Stream<T>, SyncResolver, Synthesizer, TargetDataLine, Transmitter, WatchService, WebRowSet, WritableByteChannel

All Known Implementing Classes:

AbstractInterruptibleChannel, AbstractSelectableChannel, AbstractSelector, AsynchronousFileChannel, AsynchronousServerSocketChannel, AsynchronousSocketChannel, AudioInputStream, BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, CheckedInputStream, CheckedOutputStream, CipherInputStream, CipherOutputStream, DatagramChannel, DatagramSocket, DataInputStream, DataOutputStream, DeflaterInputStream, DeflaterOutputStream, DigestInputStream, DigestOutputStream, FileCacheImageInputStream, FileCacheImageOutputStream, FileChannel, FileImageInputStream, FileImageOutputStream, FileInputStream, FileLock, FileOutputStream, FileReader, FileSystem, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, Formatter, ForwardingJavaFileManager, GZIPInputStream, GZIPOutputStream, ImageInputStreamImpl, ImageOutputStreamImpl, InflaterInputStream, InflaterOutputStream, InputStream, InputStream, InputStream, InputStreamReader, JarFile, JarInputStream, JarOutputStream, LineNumberInputStream, LineNumberReader, LogStream, MemoryCacheImageInputStream, MemoryCacheImageOutputStream, MLet, MulticastSocket, ObjectInputStream, ObjectOutputStream, OutputStream, OutputStream, OutputStream, OutputStreamWriter, Pipe.SinkChannel, Pipe.SourceChannel, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, PrivateMLet, ProgressMonitorInputStream, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, RMIConnectionImpl, RMIConnectionImpl_Stub, RMIConnector, RMIIIOPServerImpl, RMIJRMPServerImpl, RMIServerImpl, Scanner, SelectableChannel, Selector, SequenceInputStream, ServerSocket, ServerSocketChannel, Socket, SocketChannel, SSLServerSocket, SSLSocket, StringBufferInputStream, StringReader, StringWriter, URLClassLoader, Writer, XMLDecoder, XMLEncoder, ZipFile, ZipInputStream, ZipOutputStream

java.lang.AutoCloseable介面

```
public class AutoClosableDemo {
    public static void main(String[] args) {
        try(Resource res = new Resource()) {
            res.doSome();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

class Resource implements AutoCloseable {
    void doSome() {
        System.out.println("作一些事");
    }
    @Override
    public void close() throws Exception {
        System.out.println("資源被關閉");
    }
}
```

```
public class AutoClosableDemo2 {
    public static void main(String[] args) {
        try(ResourceSome some = new ResourceSome();
            ResourceOther other = new ResourceOther()) {
            some.doSome();
            other.doOther();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

class ResourceSome implements AutoCloseable {
    void doSome() {
        out.println("作一些事");
    }
    @Override
    public void close() throws Exception {
        out.println("資源 Some 被關閉");
    }
}

class ResourceOther implements AutoCloseable {
    void doOther() {
        out.println("作其他事");
    }
    @Override
    public void close() throws Exception {
        out.println("資源 Other 被關閉");
    }
}
```

```
try {
    ResourceSome some = new ResourceSome();
    Throwable localThrowable3 = null;
    try {
        ResourceOther other = new ResourceOther();
        Throwable localThrowable4 = null;
        try {
            some.doSome();
            other.doOther();
        } catch (Throwable localThrowable1) {
            localThrowable4 = localThrowable1;
            throw localThrowable1;
        } finally {  // 處理 ResourceOther 的關閉
            if (localThrowable4 != null) {
                try {
                    other.close();
                } catch (Throwable x2) {
                    localThrowable4.addSuppressed(x2);
                }
            } else {
                other.close();
            }
        }
    } catch (Throwable localThrowable2) {
        localThrowable3 = localThrowable2;
        throw localThrowable2;
    } finally {  // 處理 ResourceSome 的關閉
        if (localThrowable3 != null) {
            try {
                some.close();
            } catch (Throwable x2) {
                localThrowable3.addSuppressed(x2);
            }
        } else {
            some.close();
        }
    }
} catch (Exception ex) {
    ex.printStackTrace();
}
```

ResourceOther 的 try、catch、finally 部份

ResourceSome 的 try、catch、finally 部份