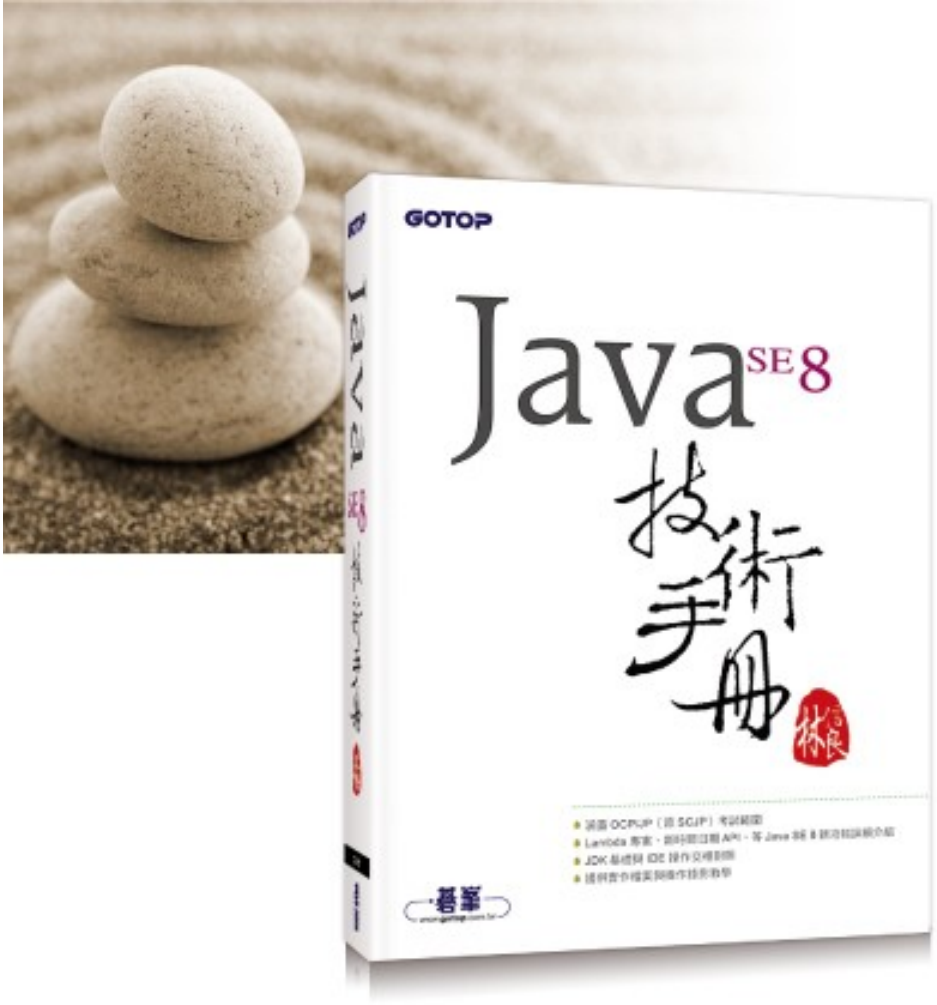


Java SE 8 技術手冊

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

通用 API

學習目標

- 使用日誌 API
- 瞭解國際化基礎
- 運用規則表示式
- 認識 JDK8 API 增強

簡介日誌 API

- **`java.util.logging`** 套件提供了日誌功能相關類別與介面，它們是從 JDK1.4 之後加入標準 API
- 要取得 `Logger` 實例，必須使用 `Logger` 的靜態方法 **`getLogger()`**：

```
Logger logger = Logger.getLogger("cc.openhome.Main");
```

簡介日誌 API

- 呼叫 `getLogger()` 時，必須指定 `Logger` 實例所屬名稱空間（Name space）
- 名稱空間以 "." 作為階層區分，名稱空間階層相同的 **`Logger`**，其父 **`Logger`** 組態相同

```
Logger logger = Logger.getLogger(Main.class.getName());
```

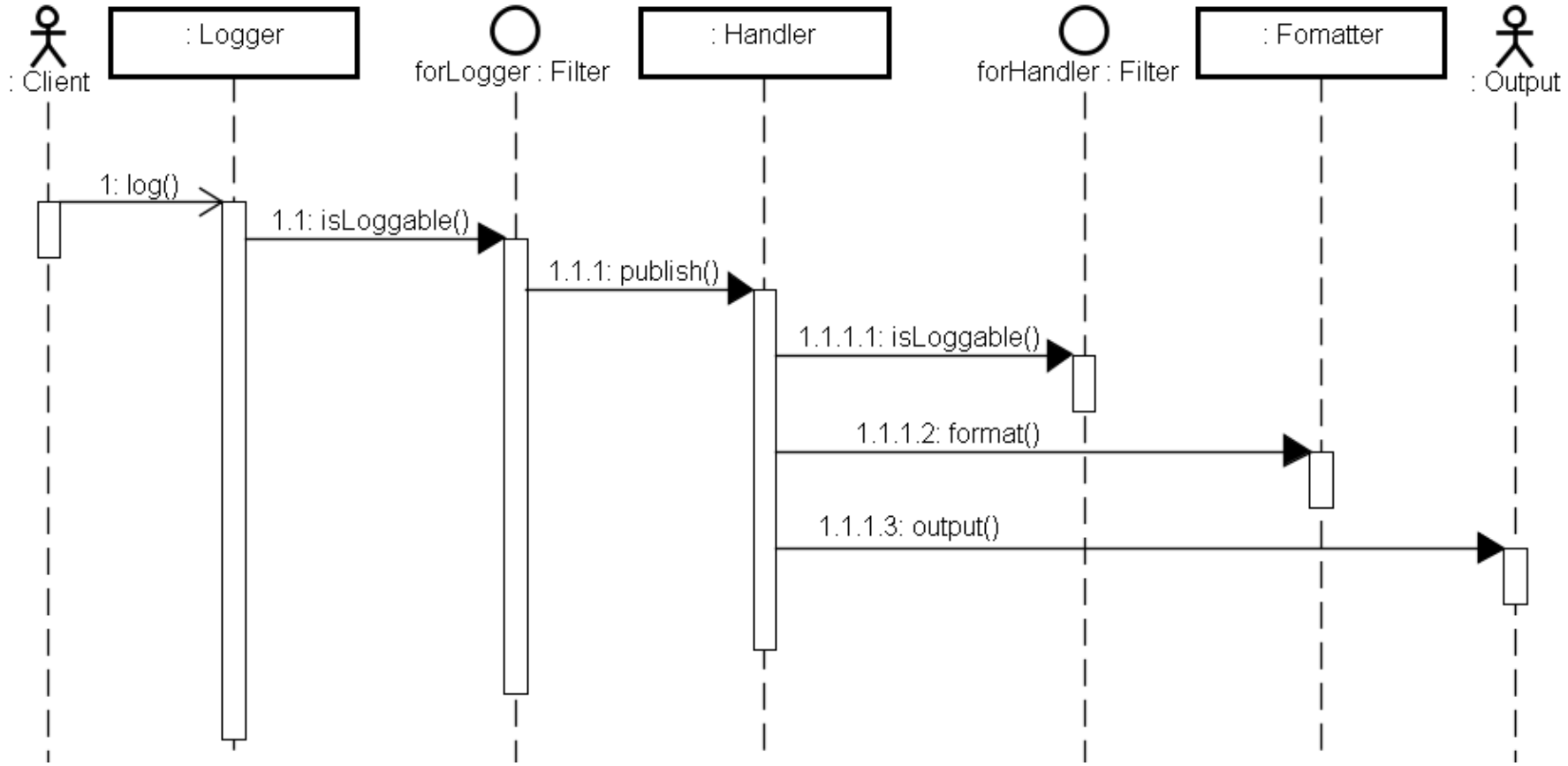
簡介日誌 API

- 取得 `Logger` 實例之後，可以使用 **`log()`** 方法輸出訊息，輸出訊息時可以使用 `Level` 的靜態成員指定訊息層級（Level）：

```
Logger logger = Logger.getLogger(LoggerDemo.class.getName());
logger.log(Level.WARNING, "WARNING 訊息");
logger.log(Level.INFO, "INFO 訊息");
logger.log(Level.CONFIG, "CONFIG 訊息");
logger.log(Level.FINE, "FINE 訊息");
```

```
五月 07, 2014 2:20:14 下午 cc.openhome.LoggerDemo main
警告: WARNING 訊息
五月 07, 2014 2:20:14 下午 cc.openhome.LoggerDemo main
資訊: INFO 訊息
```

簡介日誌 API


: Client
: Logger
forLogger : Filter
: Handler
forHandler : Filter
: Fomatter
: Output
1: log()
1.1: isLoggable()
1.1.1: publish()
1.1.1.1: isLoggable()
1.1.1.2: format()
1.1.1.3: output()

簡介日誌 API

- **Logger** 有階層關係，名稱空間階層相同的 **Logger**，父 **Logger** 組態會相同
- 每個 **Logger** 處理完自己的日誌動作後，會向父 **Logger** 傳播，讓父 **Logger** 也可以處理日誌

指定日誌層級

- Logger 與 Handler 預設都會先依 Level 過濾訊息
- 如果沒有作任何修改，取得的 Logger 實例之父 Logger 組態，就是 **Logger.GLOBAL_LOGGER_NAME** 名稱空間 Logger 實例的組態，這個實例的 Level 設定為 INFO

指定日誌層級

- 可透過 Logger 實例的 **getParent()** 取得父 Logger 實例，可透過 **getLevel()** 取得設定的 Level 實例

```
Logger logger = Logger.getLogger(Some.class.getName());
Logger global = Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
out.println(logger.getLevel());   // 顯示 null
out.println(logger.getParent().getLevel());   // 顯示 INFO
out.println(global.getParent().getLevel());   // 顯示 INFO
```

指定日誌層級

- 在沒有作任何組態設定的情況下，預設取得的 **Logger** 實例，層級必須大於或等於 **Logger.GLOBAL_LOGGER_NAME** 名稱空間 **Logger** 實例設定的 **Level.INFO**，才有可能輸出訊息

指定日誌層級

- 可以透過 Logger 的 **setLevel()** 指定 Level 實例：
  - Level.OFF（Integer.MAX_VALUE）
  - Level.SEVERE（1000）
  - Level.WARNING（900）
  - Level.INFO（800）
  - Level.CONFIG（700）
  - Level.FINE（500）
  - Level.FINER（400）
  - Level.FINEST（300）
  - Level.ALL（Integer.MIN_VALUE）

指定日誌層級

- 在經過 Logger 過濾之後，還得再經過 Handler 的過濾
- 一個 Logger 可以擁有多個 Handler，可透過 Logger 的 **addHandler()** 新增 Handler 實例

指定日誌層級

- 實際上進行訊息輸出時，目前 **Logger** 的 **Handler** 處理完，還會傳播給父 **Logger** 的所有 **Handler** 處理（在通過父 **Logger** 層級的情況下）

指定日誌層級

- 可透過 **getHandlers()** 方法來取得目前已有的 Handler 實例陣列：

```
Logger logger = Logger.getLogger(Some.class.getName());
System.out.println(logger.getHandlers().length); // 顯示 0，表示沒有 Handler
// 以下會顯示兩行，一行包括 java.util.logging.ConsoleHandler 字樣
// 一行包括 INFO 字樣
for(Handler handler : logger.getParent().getHandlers()) {
    out.println(handler);
    out.println(handler.getLevel());
}
```

指定日誌層級

- 在沒有作任何組態設定的情況下，取得的 **Logger** 實例，只會使用 **Logger.GLOBAL_LOGGER_NAME** 名稱空間 **Logger** 實例擁有的 **Handler**
- 預設是使用 **ConsoleHandler**，為 **Handler** 的子類別，作用是在主控台下輸出日誌訊息，預設的層級是 **Level.INFO**

指定日誌層級

- 若要顯示 INFO 以下的訊息，不僅要將 Logger 的層級設定為 Level.INFO，也得將 Handler 的層級設定為 Level.INFO

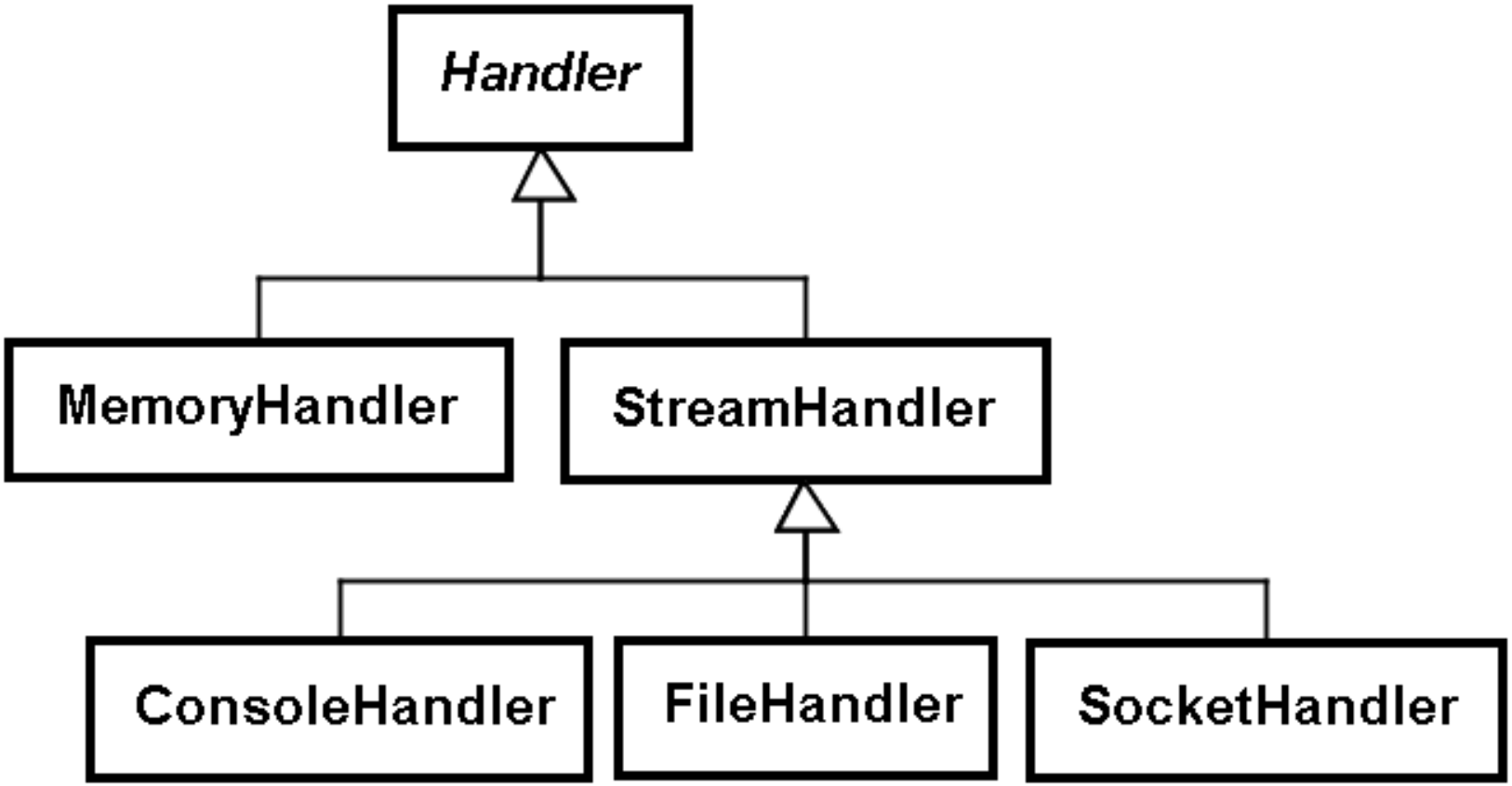
```
Logger logger = Logger.getLogger(LoggerDemo2.class.getName());
logger.setLevel(Level.FINE);
for(Handler handler : logger.getParent().getHandlers()) {
    handler.setLevel(Level.FINE);
}
logger.log(Level.WARNING, "WARNING 訊息");
logger.log(Level.INFO, "INFO 訊息");
logger.log(Level.CONFIG, "CONFIG 訊息");
logger.log(Level.FINE, "FINE 訊息");
```

指定日誌層級

- JDK8 帶入了 Lambda 之後，`severe()`、`warning()`、`info()`、`config()`、`fine()`、`finer()`、`finest()` 方法也多了重載版本

```
logger.debug(() -> expansiveLogging());
```

使用 Handler 與 Formatter



使用 Handler 與 Formatter

- Logger 可以使用 **addHandler()** 新增 Handler 實例，使用 **removeHandler()** 移除 Handler：

```
Logger logger = Logger.getLogger(HandlerDemo.class.getName());
logger.setLevel(Level.CONFIG);
FileHandler handler = new FileHandler("%h/config.log");
handler.setLevel(Level.CONFIG);
logger.addHandler(handler);
logger.config("Logger 組態完成");
```

使用 Handler 與 Formatter

- FileHandler 預設會以 XML 格式儲存：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2014-05-07T14:21:51</date>
  <millis>1399443711854</millis>
  <sequence>0</sequence>
  <logger>cc.openhome.HandlerDemo</logger>
  <level>CONFIG</level>
  <class>cc.openhome.HandlerDemo</class>
  <method>main</method>
  <thread>1</thread>
  <message>Logger 組態完成</message>
</record>
</log>
```

使用 Handler 與 Formatter

- FileHandler 預設的 Formatter 是 **XMLFormatter**，先前看過的 ConsoleHandler 預設則使用 **SimpleFormatter**
- 這兩個類別是 Formatter 的子類別，可以透過 Handler 的 setFormatter() 方法設定 Formatter

使用 Handler 與 Formatter

- 如果你不想讓父 Logger 的 Handler 處理日誌，可以呼叫 Logger 實例的 **setUseParentHandlers()** 設定為 false
- 可以使用 Logger 實例的 **setParent()** 方法指定父 Logger

自訂 Handler、 Formatter 與 Filter

- 可以繼承 Handler 類別，實作抽象方法 **publish()**、 **flush()** 與 **close()** 方法來自訂 Handler

自訂 Handler、 Formatter 與 Filter

```
...
public class CustomHandler extends Handler {
        ...
    public void publish(LogRecord logRecord) {
        if (!isLoggable(record)) {
            return;
        }
        String logMsg = getFormatter().format(logRecord);
        out.write(logMsg); // out 是輸出目的地物件
    }
    public void flush() {
         ....出清訊息
    }

    public void close() {
         ...關閉輸出物件
    }
}
```

自訂 Handler、 Formatter 與 Filter

- Handler 有預設的 isLoggable() 實作

```
...
    public boolean isLoggable(LogRecord record) {
        int levelValue = getLevel().intValue();
        if (record.getLevel().intValue() < levelValue ||
             levelValue == offValue) {
            return false;
        }
        Filter filter = getFilter();
        if (filter == null) {
            return true;
        }
        return filter.isLoggable(record);
    }
...
```

自訂 Handler、 Formatter 與 Filter

- 如果要自訂 Formatter，可以繼承 Formatter 後實作抽象方法 **format()**，這個方法會傳入 **LogRecord**，儲存有所有日誌訊息

自訂 Handler、 Formatter 與 Filter

```
Logger logger = Logger.getLogger(FormatterDemo.class.getName());
logger.setLevel(Level.CONFIG);
ConsoleHandler handler = new ConsoleHandler();
handler.setLevel(Level.CONFIG);
handler.setFormatter(new Formatter() {
    @Override
    public String format(LogRecord record) {
        return "日誌來自 " + record.getSourceClassName()
                                + record.getSourceMethodName() + "\n"
                + "\t 層級\t: " + record.getLevel() + "\n"
                + "\t 訊息\t: " + record.getMessage() + "\n"
                + "\t 時間\t: "  + new Date(record.getMillis())
                + "\n";
    }
});
logger.addHandler(handler);
logger.config("自訂 Formatter 訊息");
```

自訂 Handler、 Formatter 與 Filter

- Logger 與 Handler 都有 **setFilter()** 方法，可以指定 Filter 實作物件，
- 如果想讓 Logger 與 Handler 除了依層級過濾之外，還可以加入額外過濾條件，就可以實作 Filter 介面：

```
package java.util.logging;
public interface Filter {
    public boolean isLoggable(LogRecord record);
}
```

使用 logging.properties

```
############################################################
#  全域 Logger 組態
############################################################

# "handlers" 可以逗號分隔指定多個 Handler 類別
# 在 JVM 啟動後會完成 Handler 設定，指定的類別必須在 CLASSPATH 中
# 預設是 ConsoleHandler
handlers= java.util.logging.ConsoleHandler

# 底下是同時設定 FileHandler 與 ConsoleHandler 的範例
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# 全域 Logger 預設層級（不是 Handler 預設層級）
# 預設是 INFO
.level= INFO
```

```
############################################################
# Handler 預設組態
############################################################

# FileHandler 預設組態
# Formatter 預設是 XMLFormatter
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# ConsoleHandler 預設組態
# 層級預設是 INFO
# Formatter 預設是 SimpleFormatter
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# 要自訂 SimpleFormatter 輸出格式，可用以下範例：
#     <level>: <log message> [<date/time>]
# 例如：
# java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n

############################################################
# 特定名稱空間 Logger 組態
############################################################

# 例如設定 com.xyz.foo 名稱空間 Logger 的層級為 SEVERE
com.xyz.foo.level = SEVERE
```

國際化基礎、日期

- 應用程式根據不同地區使用者，呈現不同語言、日期格式等稱為本地化（Localization）
- 如果應用程式設計時，可在不修改應用程式情況下，根據不同使用者直接採用不同語言、日期格式等，這樣的設計考量稱為國際化（internationalization），簡稱 i18n

關於 i18n

- 在程式中有很多字串訊息會被寫死在程式中

```
public class Hello {
     public static void main(String[] args) {
         System.out.println("Hello!World!");
     }
}
```

關於 i18n

I18N messages.properties

```
cc.openhome.welcome=Hello
cc.openhome.name=World
```

```
public class Hello {
    public static void main(String[] args) {
        ResourceBundle res = ResourceBundle.getBundle("messages");
        out.print(res.getString("cc.openhome.welcome") + "!");
        out.println(res.getString("cc.openhome.name") + "!");
    }
}
```

```
Hello!World!
```

關於 i18n

- 地區資訊代表了特定的地理、政治或文化區，地區資訊可由一個語言編碼（Language code）與可選的地區編碼（Country code）來指定
- 地區（Locale）資訊的對應類別是 `Locale`

```
Locale locale = new Locale("zh", "TW");
```

關於 i18n

- 資源包中包括了特定地區的相關資訊，`ResourceBundle` 物件，就是 JVM 中資源包的代表物件
- 代表同一組訊息但不同地區的各個資源包會共用相同的基礎名稱
- 使用 `ResourceBundle` 的 `getBundle()` 時指定的名稱，就是在指定基礎名稱

關於 i18n

- `ResourceBundle` 的 `getBundle()` 時若僅指定“`messages`”，會嘗試用預設 `Locale`（由 `Locale.getDefault()` 取得的物件）取得 .properties 檔案
  - 若預設 `Locale` 代表 `zh_TW`，則 `ResourceBundle` 的 `getBundle()` 時若指定 "messages"，則會嘗試取得 messages_zh_TW.properties 檔案中的訊息，若找不到，再嘗試找 messages.properties 檔案中的訊息

關於 i18n

- 可以在 messages_zh_TW.txt 中撰寫以下內容

```
cc.openhome.welcome=哈囉
cc.openhome.name=世界
```

```
> native2ascii -encoding Big5 messages_zh_TW.txt messages_zh_TW.properties
```

I18N messages_zh_TW.properties

```
cc.openhome.welcome=\u54c8\u56c9
cc.openhome.name=\u4e16\u754c
```

關於 i18n

- 如果想將 Unicode 編碼表示的 .properties 轉回中文，則可以使用 -reverse 引數

```
> native2ascii -reverse -encoding UTF-8 messages_zh_TW.properties
messages_zh_TW.txt
```

關於 i18n

- 如果執行先前的 `Hello` 類別，而你的系統預設 `Locale` 為 `zh_TW`，則會顯示“哈囉！世界！” 的結果
- 如果你提供 messages_en_US.properties ：

I18N messages_en_US.properties

```
cc.openhome.welcome=Hi
cc.openhome.name=Earth
```

關於 i18n

- 如果如下撰寫程式，就是顯示 "Hi!Earth!" ：

```
Locale locale = new Locale("en", "US");
ResourceBundle res = ResourceBundle.getBundle("messages", locale);
out.print(res.getString("cc.openhome.welcome") + "!");
out.println(res.getString("cc.openhome.name") + "!");
```

關於 i18n

- 使用 `ResourceBundle` 時，如何根據基礎名稱取得對應的訊息檔案：
  - 使用指定的 `Locale` 物件取得訊息檔案
  - 使用 `Locale.getDefault()` 取得的物件取得訊息檔案
  - 使用基礎名稱取得訊息檔案

簡介規則表示式

```
// 根據逗號切割
for(String token : "Justin,Monica,Irene".split(",")) {
    out.println(token);
}
// 根據 Orz 切割
for(String token : "JustinOrzMonicaOrzIrene".split("Orz")) {
    out.println(token);
}
// 根據 Tab 字元切割
for(String token : "Justin\tMonica\tIrene".split("\\t")) {
    out.println(token);
}
```

```
Justin
Monica
Irene
Justin
Monica
Irene
Justin
Monica
Irene
```

簡介規則表示式

- 規則表示式是規則表示式，在 Java 中要將規則表示式撰寫在 "" 中是另一回事
- 首先得瞭解規則表示式如何定義 …

簡介規則表示式

- 字面字元（Literals）
  - 按照字面意義比對的字元
- 詮譯字元（Metacharacters）
  - 不按照字面比對，在不同情境有不同意義的字元
- 找出並理解詮譯字元想要詮譯的概念，對於規則表示式的閱讀非常重要

簡介規則表示式

字元	說明
字母或數字	比對字母或數字
\\	比對\
\0n	8 進位 0n 字元 (0 <= n <= 7)
\0nn	8 進位 0nn 字元 (0 <= n <= 7)
\0mnn	8 進位 0mnn 字元 (0 <= m <= 3, 0 <= n <= 7)
\xhh	16 進位 0xhh 字元
\uhhhh	16 進位 0xhhh 字元
\x{h...h}	16 進位 0xh...h 字元
\t	Tab (\u0009)
\n	換行 (\u000A)
\r	返回 (\u000D)
\f	換頁 (\u000C)
\a	響鈴 (\u0007)
\e	Esc (\u001B)
\cx	控制字元 x

簡介規則表示式

- 詮譯字元在規則表示式中有特殊意義，例如 ! $ ^ * ( ) + = { } [ ] | \ : . ? 等
- 若要比對這些字元，則必須加上忽略符號，例如要比對 !，則必須使用 \!，要比對 $ 字元，則必須使用 \$
- 如果不確定哪些標點符號字元要加上忽略符號，可以在每個標點符號前加上 \，例如比對逗號也可以寫 \,

定義規則表示式

- 若有個 Java 字串是“ Justin+Monica+Irene”，想使用 `split()` 方法依 + 切割
  - 使用的規則表示式是 \+
  - 要將 \+ 放至 "" 之間時，按照 Java 字串的規定，必忽略 \+ 的 \，所以必須撰寫為 "\\+"

簡介規則表示式

- 如果規則表示式為 `XY`，那麼表示比對「X 之後要跟隨著 Y」
- 如果想表示「X 或 Y」，可以使用 `X|Y`
- 如果有多個字元要以「或」的方式表示，例如「X 或 Y 或 Z」，則可以使用稍後會介紹的字元類表示為 `[XYZ]`

簡介規則表示式

- 想使用 split() 方法依 || 切割，要使用的規則表示式是 \|\|，要將 \|\| 放至 "" 之間時，按照 Java 字串規定必須忽略 \| 的 \，就必須撰寫為 "\\|\\|"

```
// 規則表示式\|\|撰寫為 Java 字串是"\\|\\|"
for(String token : "Justin||Monica||Irene".split("\\|\\|")) {
    out.println(token);
}
```

簡介規則表示式

- 如果有個字串是“ Justin\\Monica\\Irene”，也就是原始文字是 Justin\Monica\Irene 以 Java 字串表示
- 若想使用 `split()` 方法依 \ 切割，要使用的規則表示式是 \\，那就得如下撰寫：

```
// 規則表示式\\撰寫為 Java 字串是"\\\\"
for(String token : "Justin\\Monica\\Irene".split("\\\\")) {
    out.println(token);
}
```

簡介規則表示式

- 規則表示式中，多個字元可以歸類在一起，成為一個字元類（Character class）
- 字元類會比對文字中是否有「任一個」字元符合字元類中某個字元
- 規則表示式中被放在［］中的字元就成為一個字元類

簡介規則表示式

- 想要依 1 或 2 或 3 切割字串：

```
for(String token : "Justin1Monica2Irene3".split("[123]")) {
    out.println(token);
}
```

字元類	說明
[abc]	a 或 b 或 c 任一字元
[^abc]	a、b、c 以外的任一字元
[a-zA-Z]	a 到 z 或 A-Z 任一字元
[a-d[m-p]]	a 到 d 或 m-p 任一字元，等於[a-dm-p]
[a-z&&[def]]	a 到 z 而且是 d、e、f 的任一字元，等於[def]
[a-z&&[^bc]]	a 到 z 而且不是 b 或 c 的任一字元，等於[ad-z]
[a-z&&[^m-p]]	a 到 z 而且不是 m 到 p 的任一字元，等於[a-lq-z]

簡介規則表示式

預定義字元類	說明
.	任一字元
\d	比對任一數字字元，即[0-9]
\D	比對任一非數字字元，即[^0-9]
\s	比對任一空白字元，即[\t\n\x0B\f\r]
\S	比對任一非空白字元，即[^\s]
\w	比對任一 ASCII 字元，即[a-zA-Z0-9_]
\W	比對任一非 ASCII 字元，即[^\w]

簡介規則表示式

- 如果想使用者輸入的手機號碼格式是否為 XXXX-XXXXXXX，其中 X 為數字，規則表示式可以使用 \d\d\d\d-\d\d\d\d\d\d\d
- 不

貪婪量詞	說明
X?	X 項目出現一次或沒有
X*	X 項目出現零次或多次
X+	X 項目出現一次或多次
X{n}	X 項目出現 n 次
X{n,}	X 項目至少出現 n 次
X{n,m}	X 項目出現 n 次但不超過 m 次

簡介規則表示式

- 看到貪婪量詞時，比對器（Matcher）會把剩餘文字整個吃掉，再逐步吐出（back-off）文字，看看是否符合貪婪量詞後的規則表示式，如果吐出部份符合，而吃下部份也符合貪婪量詞就比對成功
- 貪婪量詞會儘可能找出長度最長的符合文字

簡介規則表示式

- 文字 xfooxxxxxxfoo，使用規則表示式 `.*foo` 比對
  - 比對器會先吃掉整個 xfooxxxxxxfoo，再吐出 foo 符合 `foo` 部份，剩下的 xfooxxxxxx 也符合 `.*` 部份
- 得到的符合字串就是整個 xfooxxxxxxfoo

簡介規則表示式

- 如果在貪婪量詞表示法後加上？，將會成為逐步量詞（Reluctant quantifier）
- 比對器看到逐步量詞時，會一邊吃掉剩餘文字，一邊看看吃下的文字是否符合規則表示式
- 逐步量詞會儘可能找出長度最短的符合文字

簡介規則表示式

- 文字 xfooxxxxxxfoo 若用規則表示式 `.*?foo` 比對
  - 比對器在吃掉 xfoo 後發現符合 `*?foo`，接著繼續吃掉 xxxxxxfoo 發現符合
- 得到 xfoo 與 xxxxxxfoo 兩個符合文字

簡介規則表示式

- 如果在貪婪量詞表示法後加上 +，將會成為獨吐量詞（Possessive quantifier）
- 比對器看到獨吐量詞時，會先將剩餘文字全部吃掉，然後看看獨吐量詞部份是否可符合吃下的文字，如果符合就不會再吐出來了

簡介規則表示式

- 文字 xfooxxxxxxfoo，若使用規則表示式 `.*+foo` 比對
  - 比對器會先吃掉整個 xfooxxxxxxfoo，結果 `.*+` 就可以符合 xfooxxxxxxfoo 了，所以比對器就不會再吐出文字
- 因為沒有剩餘文字符合 foo 部份，所以結果就是沒有任何文字符合

簡介規則表示式

```
public class SplitDemo2 {
    public static void main(String[] args) {
        for(String str : "Justin dog Monica doggie Irene".split("dog")) {
            System.out.println(str.trim());
        }
    }
}
```

```
Justin
Monica
gie Irene
```

簡介規則表示式

- 可以使用 \b 標出單字邊界，例如 \bdog\b，這就只會比對出 dog 單字

```
public class SplitDemo3 {
    public static void main(String[] args) {
      for(String str : "Justin dog Monica doggie Irene".split("\\bdog\\b")) {
            System.out.println(str.trim());
        }
    }
}
```

```
Justin
Monica doggie Irene
```

簡介規則表示式

邊界比對	說明
^	一行開頭
$	一行結尾
\b	單字邊界
\B	非單字邊界
\A	輸入開頭
\G	前一個符合項目結尾
\Z	非最後終端機（final terminator）的輸入結尾
\z	輸入結尾

簡介規則表示式

- 可以使用（）來將規則表示式分組，除了作為子規則表示式之外，還可以搭配量詞使用
- 驗證電子郵件格式，@後網域名稱可以有數層，必須是大小寫英文字元或數字，規則表示式可以寫為(\[a-zA-Z0-9\]+\.)+

```
^[a-zA-Z]+\d*@([a-zA-Z0-9]+\.)+com
```

簡介規則表示式

- 分組計數是遇到的左括號來計數
- 如果有個規則表示式((A)(B(C)))，其中有四個分組：
  - ((A)(B(C)))
  - (A)
  - (B(C))
  - (C)

簡介規則表示式

- 分組回頭參考時，是在 \ 後加上分組計數，表示參考第幾個分組的比對結果
- \d\d 要求比對兩個數字，(\d\d)\1 的話，表示要輸入四個數字，輸入的前兩個數字與後兩個數字必須相同

簡介規則表示式

- `[“‘`][^“”’]* `[“‘`] 比對單引號或雙引號中 0 或多個字元，但沒有比對兩個都要是單引號或雙引號
- `(["'])[^"']*\1` 則比對出前後引號必須一致

Pattern 與 Matcher

- **java.util.regex.Pattern** 實例是規則表示式在 JVM 中的代表物件
- 必須透過 Pattern 的靜態方法 **compile()** 來取得，compile() 方法在剖析、驗證過規則表示式無誤後，將會傳回 Pattern 實例，之後你就可以重複使用這個實例

```
Pattern pattern = Pattern.compile(".*foo");
```

Pattern 與 Matcher

- `Pattern.compile()` 方法的另一版本，可以指定旗標（Flag）
- 例如想不分大小寫比對 dog 文字：

```
Pattern pattern = Pattern.compile("dog", Pattern.CASE_INSENSITIVE);
```

- 可以在規則表示式中使用嵌入旗標表示法（Embedded Flag Expression）：

```
Pattern pattern = Pattern.compile("(?i)dog");
```

Pattern 與 Matcher

- 因規則表示式有誤而導致 `compile()` 失敗，會拋出 **`java.util.regex.PatternSyntaxException`**
  - 使用 **`getDescription()`** 取得錯誤說明
  - 使用 **`getIndex()`** 取得錯誤索引
  - 使用 **`getPattern()`** 取得錯誤的規則表示式
  - 使用 **`getMessage()`** 會以多行顯示錯誤的索引、描述等綜合訊息

Pattern 與 Matcher

- 在取得 Pattern 實例後…
  - 使用 **split()** 方法將指定字串依規則表示式切割，效果等同於使用 String 的 split() 方法
  - 使用 **matcher()** 方法指定要比對的字串，這會傳回 **java.util.regex.Matcher** 實例，表示對指定字串的比對器
    - 可以使用 **find()** 方法看看是不是有下一個符合字串，或是使用 **lookingAt()** 看看字串開頭是否符合規則表示式，使用 **group()** 方法則可以傳回符合的字串

Pattern 與 Matcher

```
String[] regexs = {".*foo", ".*?foo", ".*+foo"};
for(String regex : regexs) {
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher("xfooxxxxxxfoo");
    out.printf("%s find ", pattern.pattern());
    while(matcher.find()) {
        out.printf(" \"%s\"", matcher.group());
    }
    out.println(" in \"xfooxxxxxxfoo\".");
}
```

```
.*foo find  "xfooxxxxxxfoo" in "xfooxxxxxxfoo".
.*?foo find  "xfoo" "xxxxxxfoo" in "xfooxxxxxxfoo".
.*+foo find  in "xfooxxxxxxfoo".
```

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    try {
        out.print("輸入規則表示式：");
        String regex = console.nextLine();
        out.print("輸入要比對的文字：");
        String text = console.nextLine();
        print(match(regex, text));
    } catch(PatternSyntaxException ex) {
            out.println("規則表示式有誤");
            out.println(ex.getMessage());
    }
}

private static List<String> match(String regex, String text) {
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(text);
    List<String> matched = new ArrayList<>();
    while(matcher.find()) {
        matched.add(String.format(
                "從索引 %d 開始到索引 %d 之間找到符合文字 \"%s\"%n",
                matcher.start(), matcher.end(), matcher.group()));
    }
    return matched;
}
```

Pattern 與 Matcher

- 在 Pattern 也因應 Stream API 而新增了 splitAsStream() 靜態方法，它傳回的是 Stream<String>

StringJoiner、Arrays 新增 API

- String 新增了 join() 靜態方法可以使用

```
String message = String.join("-", "Java", "is", "cool"); // 產生 "Java-is-cool"
```

- 如果你有一組實作了 CharSequence 的物件

```
List<String> strs = new ArrayList<>();
strs.add("Java");
strs.add("is");
strs.add("cool");
String message = String.join("-", strs); // 產生 "Java-is-cool"
```

StringJoiner、Arrays 新增 API

- 直接使用 JDK8 新增的 StringJoiner 類別

```
StringJoiner joiner = new StringJoiner("-");
String message = joiner.add("Java")
                       .add("is")
                       .add("cool")
                       .toString();   // 產生 "Java-is-cool"
```

StringJoiner、Arrays 新增 API

- 在管線化操作之後，想要進行字串的連結

```
List<Customer> customers = ...;
String joinedFirstNames = customers.stream()
                                   .map(Customer::getFirstName)
                                   .collect(joining(", "));
```

StringJoiner、Arrays 新增 API

- 在 Arrays 上新增了 parallelPrefix()、parallelSetAll() 與 parallelSort() 方法

```
int[] arrs = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(arrs, (left, right) -> left + right);
out.println(Arrays.toString(arrs)); // [1, 3, 6, 10, 15]

int[] arrs = new int[10000000];
Arrays.parallelSetAll(arrs, index -> -1);
```

Stream 相關 API

- 在許多 API 上，都可以取得 Stream 實例
  - Files 上的靜態方法
    lines()、list()、walk()
  - Pattern 上的 splitAsStream() 靜態方法
- 主要可適用於需要管線化、惰性操作的場合
- 想對陣列進行管線化操作
  - 使用 Arrays 的 asList() 方法傳回 List，而後呼叫 stream() 方法
  - 使用 Arrays 的 stream() 方法

Stream 相關 API

- Stream、IntStream、DoubleStream 等都有 of() 靜態方法，也各有 generate() 與 iterate() 靜態方法
- 想要產生一個整數範圍，IntStream 上有 range() 與 rangeClosed() 方法

```
range(0, 10000).forEach(out::println);
```

Stream 相關 API

- CharSequence 上新增了 chars() 與 codePoints() 兩個方法，都是傳回 IntStream

```
IntStream charStream = "Justin".chars();
IntStream codeStream = "Justin".codePoints();
```

Stream 相關 API

- JDK8 新增了 java.util.Random 類別

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(); // 0 到 1 間的隨機浮點數
IntStream intStream = random.ints(0, 100); // 0 到 100 間的隨機整數
```