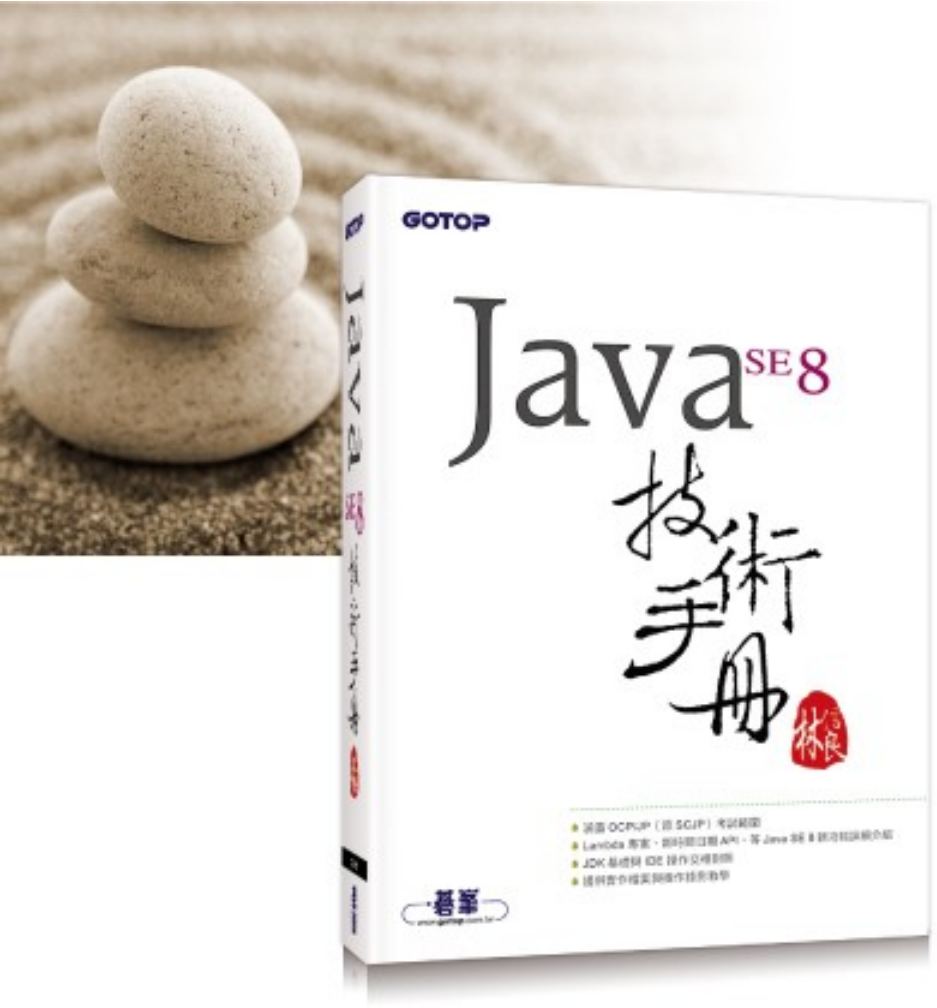

Java SE 8 技術手冊

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

Lambda

學習目標

- 認識 Lambda 語法
- 運用方法參考
- 瞭解介面預設方法
- 善用 Functional 與 Stream API
- Lambda 與平行化

Lambda 語法概覽

- 匿名類別的應用場合

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
});
```

Lambda 語法概覽

- 稍微改變 `Arrays.sort()` 該行的可讀性

```
Comparator<String> byLength = new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
};

String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, byLength);
```

Lambda 語法概覽

- 使用 JDK8 的 Lambda 表示式

```
Comparator<String> byLength =
     (String name1, String name2) -> name1.length() - name2.length();
```

- 編譯器可以從 `byLength` 變數的宣告型態, `name1` 與 `name2` 的型態

```
Comparator<String> byLength = (name1, name2) -> name1.length() - name2.length();
```

Lambda 語法概覽

- 直接放到 `Arrays` 的 `sort()` 方法中

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, (name1, name2) -> name1.length() - name2.length());
System.out.println(Arrays.toString(names));
```

Lambda 語法概覽

- 將一些字串排序時可能的方式都定義出來

```
public class StringOrder {
    public static int byLength(String s1, String s2) {
        return s1.length() - s2.length();
    }

    public static int byLexicography(String s1, String s2) {
        return s1.compareTo(s2);
    }

    public static int byLexicographyIgnoreCase(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
}
```

Lambda 語法概覽

- 原本的依名稱長度排序就可以改寫為：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, (name1, name2) -> StringOrder.byLength(name1, name2));
```

- JDK8 提供了方法參考（Method reference）

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, StringOrder::byLength);
System.out.println(Arrays.toString(names));
```

Lambda 語法概覽

- 方法參考的特性，在重用現有 API 上扮演了重要角色。重用現有方法實作，可避免到處寫下 Lambda 運算式

Lambda 語法概覽

- byLexicography() 方法實作中，只不過是呼叫 String 的 compareTo() 方法

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, StringOrder::byLexicography);
```

- 可直接參考 String 類別的 compareTo 方法

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, String::compareTo);
System.out.println(Arrays.toString(names));
```

Lambda 語法概覽

- 想對名稱按照字典順序排序，但忽略大小寫差異

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, String::compareToIgnoreCase);
```

- 方法參考不僅避免了重複撰寫 Lambda 運算式，也可以讓程式碼更為清楚

Lambda 表示式與函式介面

- 等號右邊是 Lambda 表示式（Expression），等號左邊是作為 Lambda 表示式的目標型態（Target type）

```
Comparator<String> byLength =
     (String name1, String name2) -> name1.length() - name2.length();
```

Lambda 表示式與函式介面

- Lambda 表示式：

```
(String name1, String name2) -> name1.length() - name2.length()


(String name1, String name2) -> {
    String name1 = name1.trim();
    String name2 = name2.trim();
    ...
    return name1.length() - name2.length();
}

() -> "Justin"                  // 不接受參數，傳回字串
() -> System.out.println()      // 不接受參數，沒有傳回值
```

Lambda 表示式與函式介面

- 如果有目標型態的話，在編譯器可推斷出類型的情況下，就可以不寫出 Lambda 表示式的參數型態

```
Comparator<String> byLength = (name1, name2) -> name1.length() - name2.length();
```

- Lambda 表示式本身是中性的，不代表任何型態的實例，同樣的 Lambda 表示式，可用來表示不同目標型態的物件實作

```
Func<String, Integer> func = (name1, name2) -> name1.length() - name2.length();
```

Lambda 表示式與函式介面

- 函式介面就是介面，要求僅具單一抽象方法

```
public interface Runnable {
    void run();
}

public interface Callable<V> {
    V call() throws Exception;
}

public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Lambda 表示式與函式介面

- 匿名類別不是不好，只不過有其應用的場合
- 在許多時候，特別是介面只有一個方法要實作時，你會只想關心參數及實作本體

```
Arrays.sort(names, new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
});
```

```
Arrays.sort(names, (name1, name2) -> name1.length() - name2.length());
```

Lambda 表示式與函式介面

- 如果函式介面上定義的方法只接受一個參數

```
public interface Func {
    public void apply(String s);
}

Func func = (s) -> out.println(s);

Func func = s -> out.println(s);
```

Lambda 表示式與函式介面

- @FunctionalInterface 在 JDK8 中被引入

```
@FunctionalInterface
public interface Func<P, R> {
    R apply(P p);
}
```

- 並非函式介面的話會編譯錯誤，例如：

```
@FunctionalInterface
public interface Function<P, R> {
    R call(P p);
    R call(P p1, P p2);
}
```

Lambda 遇上 this 與 final

- Lambda 表示式並不是匿名類別的語法蜜糖

```
class Hello {
    Runnable r1 = new Runnable() {
        public void run() {
            out.println(this);
        }
    };

    Runnable r2 = new Runnable() {
        public void run() {
            out.println(toString());
        }
    };

    public String toString() {
        return "Hello, world!";
    }
}
```

Lambda 遇上 this 與 final

```
public class ThisDemo {
    public static void main(String[] args) {
        Hello hello = new Hello();
        hello.r1.run();
        hello.r2.run();
    }
}
```

```
cc.openhome.Hello$1@15db9742
cc.openhome.Hello$2@6d06d69c
```

Lambda 遇上 this 與 final

```
class Hello2 {
    Runnable r1 = () -> out.println(this);
    Runnable r2 = () -> out.println(toString());

    public String toString() {
        return "Hello, world!";
    }
}

public class ThisDemo2 {
    public static void main(String[] args) {
        Hello2 hello = new Hello2();
        hello.r1.run();
        hello.r2.run();
    }
}
```

Lambda 遇上 this 與 final

- 以下在 JDK8 中不會有錯：

```
String[] names = {"Justin", "Monica", "Irene"}; // JDK8 前必須加上 final
Runnable runnable = new Runnable() {
    public void run() {
        for(String name : names) {
            out.println(name);
        }
    }
};
```

```
String[] names = {"Justin", "Monica", "Irene"};
Runnable runnable = () -> {
    for(String name : names) {
        out.println(name);
    }
};
```

Lambda 遇上 this 與 final

- 如果 Lambda 表示式中捕獲的區域變數本身等效於 final 區域變數，可以不用在區域變數上加上 final。
- 可以在 Lambda 表示式中改變被捕獲的區域變數值嗎？答案是不行！

方法與建構式參考

- 可以使用靜態方法來定義函式介面實作

```
public class StringOrder {
    public static int byLength(String s1, String s2) {
        return s1.length() - s2.length();
    }
    ...
}
```

```
Comparator<String> byLength  = (s1, s2) -> s1.length() - s2.length();
```

```
Comparator<String> byLength = StringOrder::byLength;
```

方法與建構式參考

- 方法參考（Method references）可以避免你到處寫下 Lambda 表示式，儘量運用現有的 API 實作，也可以改善可讀性

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, (name1, name2) -> name1.length() - name2.length());
```

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, StringOrder::byLength);
```

方法與建構式參考

- 可以參考特定物件的實例方法

```
List<String> names = Arrays.asList("Justin", "Monica", "Irene");
names.forEach(name -> out.println(name));
new HashSet(names).forEach(name -> out.println(name));
new ArrayDeque(names).forEach(name -> out.println(name));

List<String> names = Arrays.asList("Justin", "Monica", "Irene");
names.forEach(out::println);
new HashSet(names).forEach(out::println);
new ArrayDeque(names).forEach(out::println);
```

方法與建構式參考

- 可以參考類別上定義的非靜態方法

```
Comparator<String> naturalOrder = String::compareTo;


String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, String::compareTo);
...
Arrays.sort(names, String::compareToIgnoreCase);
```

方法與建構式參考

- 建構式參考（Constructor references）用來重用現有 API 的物件建構流程

```
List<String> names = Arrays.asList(args);
List<Person> persons = map(names, name -> new Person(name));
```

```
List<String> names = Arrays.asList(args);
List<Person> persons = map(names, Person::new);
```

介面預設方法

- 在 JDK8 中，`interface` 定義時可以加入預設實作，或者稱為預設方法（Default methods）

```
@FunctionalInterface
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

介面預設方法

- `forEach()` 方法本身已有實作，所以不會破壞 `Iterable` 現有的其他實作

```
List<String> names = Arrays.asList("Justin", "caterpillar", "Monica");
names.forEach(out::println);
```

- 預設方法中不能使用資料成員
- 預設方法中不能有直接變更狀態的流程

介面預設方法

- JDK8 新增了預設方法，這也給了共用相同實作的方便性

```
public interface Comparable<T> {
    int compareTo(T that);

    default boolean lessThan(T that) {
        return compareTo(that) < 0;
    }
    default boolean lessOrEquals(T that) {
        return compareTo(that) <= 0;
    }
    default boolean greaterThan(T that) {
        return compareTo(that) > 0;
    }
    ...
}
```

介面預設方法

- 如果有個 Ball 類別打算實作這個自定義的 Comparable 介面的話

```
public class Ball implements Comparable<Ball> {
    private int radius;
    ...
    public int compareTo(Ball that) {
        return this.radius - that.radius;
    }
}
```

介面預設方法

- 實作介面是廣義的多重繼承
- JDK8 中允許有預設實作，引入了強大的威力，也引入了更多的複雜度
- 你得留意到底採用的是哪個方法版本

介面預設方法

- 父介面中的抽象方法，可以在子介面中以預設方法實作
- 父介面中的預設方法，可以在子介面中被新的預設方法重新定義
- 重新定義父類別中預設方法實作為抽象方法

```
public interface BiIterable<T> extends Iterable<T> {
    Iterator<T> iterator();
    void forEach(Consumer<? super T> action);
    ...
}
```

介面預設方法

- 如果重新定義為預設方法時，想明確呼叫某個父介面的 `draw()` 方法，必須使用介面名稱與 `super` 明確指定

```
public interface Lego extends Part, Canvas {
    default void draw() {
        Part.super.draw();
    }
}
```

介面預設方法

- 如果實作時有兩個介面都定義了相同方法簽署的預設方法，那麼會引發衝突
- 解決的方式是明確重新定義，無論是重新定義為抽象或預設方法
- 如果重新定義為具體方法時，想明呼叫某個介面的方法，也是得使用介面名稱與 `super` 明確指定

介面預設方法

- 如果類別實作的兩個介面擁有相同的父介面，其中一個介面重新定義了父介面的預設方法，而另一個介面沒有，那麼實作類別會採用重新定義了的版本
- 如果子類別繼承了父類別同時實作某介面，而父類別中的方法與介面中的預設方法具有相同方法簽署，則採用父類別的方法定義

介面預設方法

- 簡單來說，類別中的定義優先於介面中的定義，如果有重新定義，就以重新定義的為主，必要時使用介面與 `super` 指定採用哪個預設方法
- JDK8 除了讓介面可以定義預設方法之外，也開始允許在介面中定義靜態方法

介面預設方法

- Iterable 介面新增 forEach() 預設方法

```
:
public interface Iterable<T> {
    ...
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
    ...
}
```

介面預設方法

- Iterator 也有個 forEachRemaining() 的預設實作

```
public interface Iterator<E> {
    …
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

介面預設方法

- Comparator 也定義了一些預設方法

```
Comparator<Customer> byLastName = comparing(Customer::getLastName);

customers.sort(
        byLastName
        .thenComparing(Customer::getFirstName)
        .thenComparing(Customer::getZipCode)
);
```

使用 Optional 取代 null

- 你經常會與 NullPointerException 奮戰
- null 的最根本問題在於語意含糊不清

```
public static void main(String[] args) {
    String nickName = getNickName("Duke");
    if (nickName == null) {
        nickName = "Openhome Reader";
    }
    out.println(nickName);
}

static String getNickName(String name) {
    Map<String, String> nickNames = new HashMap<>(); // 假裝的鍵值資料庫
    nickNames.put("Justin", "caterpillar");
    nickNames.put("Monica", "momor");
    nickNames.put("Irene", "hamimi");
    return nickNames.get(name); // 鍵不存在時會傳回 null
}
```

使用 Optional 取代 null

- 呼叫方法時如果傳回型態是 Optional，應該立即想到它可能包裹也可能不包裹值

```
static Optional<String> getNickName(String name) {
    Map<String, String> nickNames = new HashMap<>();
    nickNames.put("Justin", "caterpillar");
    nickNames.put("Monica", "momor");
    nickNames.put("Irene", "hamimi");
    String nickName = nickNames.get(name);
    return nickName == null ? Optional.empty() : Optional.of(nickName);
}
```

使用 Optional 取代 null

- 在 Optional 沒有包含值的情況下， get 會拋出 NoSuchElementException

```
String nickName = getNickName("Duke").get();
out.println(nickName);

Optional<String> nickOptional = getNickName("Duke");
String nickName = "Openhome Reader";
if(nickOptional.isPresent()) {
    nickName = nickOptional.get();
}
out.println(nickName);

Optional<String> nickOptional = getNickName("Duke");
out.println(nickOptional.orElse("Openhome Reader"));
```

使用 Optional 取代 null

- Optional 的 ofNullable() 來銜接程式庫中會傳回 null 的方法

```
static Optional<String> getNickName(String name) {
    Map<String, String> nickNames = new HashMap<>();
    nickNames.put("Justin", "caterpillar");
    nickNames.put("Monica", "momor");
    nickNames.put("Irene", "hamimi");
    return Optional.ofNullable(nickNames.get(name));
}
```

標準 API 的函式介面

- JDK8 已經定義了幾個通用的函式介面
- 基本上可以分為 `Consumer`、 `Function`、`Predicate` 與 `Supplier` 四個類型

標準 API 的函式介面

- 如果需要的行為是接受一個引數，然後處理後不傳回值，就可以使用 Consumer 介面

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    ...
}

default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

標準 API 的函式介面

- 接受了引數但沒有傳回值，這行為就像純粹消耗了引數，就是命名為 `Consumer` 的原因
- 真的有結果產生，就是以副作用（Side effect）形式呈現

```
Arrays.asList("Justin", "Monica", "Irene").forEach(out::println);
```

標準 API 的函式介面

- 接受一個引數，然後以該引數進行計算後傳回結果，就可以使用 Function 介面

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

- 行為就像是數學函數 y=f(x)，給予 x 值計算出 y 值的概念，因此命名為 Function

標準 API 的函式介面

- 接受一個引數，然後只傳回 boolean 值，也就是根據傳入的引數直接論斷真假的行為，就可以使用 Predicate 函式介面

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    ...
}

long count = Stream.of(fileNames)
                   .filter(name -> name.endsWith("txt"))
                   .count();
```

標準 API 的函式介面

- 需要的行為是不接受任何引數，然後傳回值，那可以使用 `Supplier` 函式介面

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

collect

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

使用 Stream 進行管線操作

- 在正式瞭解 Stream 介面的作用之前

```
String fileName = args[0];
String prefix = args[1];
String firstMatchdLine = "no matched line";
for (String line : Files.readAllLines(Paths.get(fileName))) {
    if(line.startsWith(prefix)) {
        firstMatchdLine = line;
        break;
    }
}
out.println(firstMatchdLine);
```

使用 Stream 進行管線操作

- 在 JDK8 中，這類的需求，建議改用以下的程式來完成：

```
String fileName = args[0];
String prefix = args[1];
Optional<String> firstMatchdLine =
        Files.lines(Paths.get(fileName))
             .filter(line -> line.startsWith(prefix))
             .findFirst();
System.out.println(firstMatchdLine.orElse("no matched line"));
```

使用 Stream 進行管線操作

- 能夠達到這類惰性求值（Lazy evaluation）的效果，功臣就是 Stream 實例
- 第一個程式片段搭配 for 迴圈進行外部迭代（External iteration）第二個程式片段內部迭代（Internal iteration）
- 因為內部迭代的行為是被隱藏的，因此多了很多可以實現效率的可能性。

使用 Stream 進行管線操作

- 絕大多數的 Stream 並不需要呼叫 close() 方法
- JDK8 中要 close() 的是 Files.lines()、 Files.list() 與 Files.walk() 方法

使用 Stream 進行管線操作

- JDK8 引入了 Stream API，也引入了管線操作風格
  - 來源（Source）
  - 零或多個中介操作（Intermediate operation）
  - 一個最終操作（Terminal operation）

使用 Stream 進行管線操作

- 原本有個程式片段：

```
List<Player> players = ...;
List<String> names = new ArrayList<>();
for(Player player : players) {
    if(player.getAge() > 15) {
        names.add(player.getName().toUpperCase());
    }
}
for(String name : names) {
    System.out.println(name);
}
```

使用 Stream 進行管線操作

- 在 JDK8 中可以改為以下的風格：

```
List<Player> players = Arrays.asList(
        new Player("Justin", 39),
        new Player("Monica", 36),
        new Player("Irene", 6)
);
players.stream()
        .filter(player -> player.getAge() > 15)
        .map(Player::getName)
        .map(String::toUpperCase)
        .collect(toList())
        .forEach(out::println);
```

使用 Stream 進行管線操作

- 如果你的程式在 for 迴圈中使用了 if：

```
for(Player : players) {
    if(player.getAge() > 15) {
        // 這是下一個小任務
    }
}

players.stream()
    .filter(player -> player.getAge() > 15)
    ... // 接下來的中介或最終操作
```

使用 Stream 進行管線操作

- 如果你的程式在 for 迴圈中從一個型態對應至另一個型態：

```
for(Player player: players) {
    String upperCase = player.getName().toUpperCase();
    ...下一個小任務
}

players.stream()
       .map(Player::getName)
       .map(String::toUpperCase)
       ...下一個小任務
```

使用 Stream 進行管線操作

- 許多時候，`for` 迴圈中就是滲雜了許多小任務，從而使 `for` 迴圈中的程式碼艱澀難懂
- `Stream` 只能迭代一次，重複對 Stream 進行迭代，會引發 `IllegalStateException`

Stream 的 reduce 與 collect

- 程式設計中不少地方存在類似需求

```
List<Employee> employees = ...;
int sum = 0;
for(Employee employee : employees) {
    if(employee.getGender() == Gender.MALE) {
        sum += employee.getAge();
    }
}
int average = sum / employees.size();
```

```
int max = 0;
for(Employee employee : employees) {
    if(employee.getGender() == Gender.MALE) {
        if(employee.getAge() > max) {
            max = employee.getAge();
        }
    }
}
```

Stream 的 reduce 與 collect

- 程式中這類需求都存在著類似的流程結構，而你也不斷重複撰寫著類似結構，而且從閱讀程式碼角度來看，無法一眼察覺程式意圖

Stream 的 reduce 與 collect

- 在 JDK8 中，可以改寫為：

```
int sum = employees.stream()
        .filter(employee -> employee.getGender() == Gender.MALE)
        .mapToInt(Employee::getAge)
        .sum();

int average = (int) employees.stream()
        .filter(employee -> employee.getGender() == Gender.MALE)
        .mapToInt(Employee::getAge)
        .average()
        .getAsDouble();

int max = employees.stream()
        .filter(employee -> employee.getGender() == Gender.MALE)
        .mapToInt(Employee::getAge)
        .max()
        .getAsInt();
```

Stream 的 reduce 與 collect

- 先前的迴圈結構，實際上有個步驟都是將一組數據逐步取出削減，然而透過指定運算以取得結果的結構
- JDK8 將這個流程結構通用化，定義了 `reduce()` 方法來達到自訂運算需求

Stream 的 reduce 與 collect

```
int sum = employees.stream()
        .filter(employee -> employee.getGender() == Gender2.MALE)
        .mapToInt(Employee2::getAge)
        .reduce((total, age) ->  total + age)
        .getAsInt();

long males = employees.stream()
        .filter(employee -> employee.getGender() == Gender2.MALE)
        .count();

long average = employees.stream()
        .filter(employee -> employee.getGender() == Gender2.MALE)
        .mapToInt(Employee2::getAge)
        .reduce((total, age) ->  total + age)
        .getAsInt() / males;

int max = employees.stream()
        .filter(employee -> employee.getGender() == Gender2.MALE)
        .mapToInt(Employee2::getAge)
        .reduce(0, (currMax, age) -> age > currMax ? age : currMax);
```

Stream 的 reduce 與 collect

- 如果你想將一組員工的男性收集至另一個 List<Employee> 呢？

```
List<Employee> males = employees.stream()
          .filter(employee -> employee.getGender() == Gender.MALE)
          .collect(toList()); // toList()是 java.util.stream.Collectors 的靜態方法
```

Stream 的 reduce 與 collect

- Collector 主要的四個方法是：
  - suppiler() 傳回 Suppiler，定義收集結果的新容器如何建立
  - accumulator() 傳回 BiConsumer，定義如何使用結果容器收集物件
  - combiner() 傳回 BinaryOperator，定義若有兩個結果容器，如何合併為一個結果容器
  - finisher() 傳回 Function，選擇性地定義如何將結果轉換為最後的結果容器。

Stream 的 reduce 與 collect

- 來看看 Stream 的 collect() 方法另一版本

```
List<Employee> males = persons.stream()
                    .filter(employee -> employee.getGender() == Gender.MALE)
                    .collect(
                         () -> new ArrayList<>(),
                         (maleLt, employee) -> maleLt.add(employee),
                         (maleLt1, maleLt2) -> maleLt1.addAll(maleLt2)


List<Employee> males = employees.stream()
             .filter(employee -> employee.getGender() == Gender.MALE)
             .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

Stream 的 reduce 與 collect

- 可以先看看 Collectors 上提供了哪些 Collector 實作

```
Map<Gender, List<Employee>> males = employees.stream()
                    .collect(
                        groupingBy(Employee::getGender));
```

```
Map<Gender, List<String>> males = employees.stream()
                    .collect(
                        groupingBy(Employee::getGender,
                            mapping(Employee::getName, toList()))
                    );
```

Stream 的 reduce 與 collect

```
Map<Gender, Integer> males = employees.stream()
                    .collect(
                            groupingBy(Employee::getGender,
                                 reducing(0, Employee::getAge, Integer::sum))
                    );


Map<Gender, Double> males = employees.stream()
                     .collect(
                             groupingBy(Employee::getGender,
                                  averagingInt(Employee::getAge))
                     );
```

關於 flatMap() 方法

- 在程式設計中有時會出現巢狀或瀑布式的流程，就結構來看每一層運算極為類似，只是傳回的型態不同，很難抽取流程重用

```
Customer customer = order.getCustomer();
if(customer != null) {
    String address = customer.getAddress();
    if(address != null) {
        return address;
    }
}
return "n.a.";
```

關於 flatMap() 方法

- 巢狀的層次可能還會更深，像是 ...

```
Customer customer = order.getCustomer();
if(customer != null) {
    Address address = customer.getAddress();
    if(address != null) {
        City city = address.getCity();
        if(city != null) {
            ....
        }
    }
}
return "n.a.";
```

關於 flatMap() 方法

- 如果能修改 getCustomer() 傳回 Optional<Customer>、也修改 getAddress() 傳回 Optional<String>

```
String addr = "n.a.";
Optional<Customer> customer = order.getCustomer();
if(customer.isPresent()) {
    Optional<String> address = customer.get().getAddress();
    if(address.isPresent()) {
        addr = address.get();
    }
}
return addr;
```

關於 flatMap() 方法

- 每一層都是 Optional 型態了，而每一層都是 isPresent() 的判斷，然後將 Optional<T> 轉換為 Optional<U>

```
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Objects.requireNonNull(mapper.apply(value));
    }
}
```

關於 flatMap() 方法

- 直接使用 Optional 的 flatMap() 方法：

```
return order.getCustomer()
            .flatMap(Customer::getAddress)
            .orElse("n.a.");
```

- 第二個程式片段，改寫為以下就清楚多了…

```
return order.getCustomer()
            .flatMap(Customer::getAddress)
            .flatMap(Address::getCity)
            .orElse("n.a.");
```

關於 flatMap() 方法

- flatMap() 就像是從盒子取出另一盒子（flat 就是平坦化的意思）
- Lambda 表示式指定了前一個盒子中的值與下一個盒子之間的轉換關係
- 運算情境被隱藏了，使用者可明確指定感興趣的特定運算，從而使程式碼意圖顯露出來
- 可接暢地接續運算，以避免巢狀或瀑布式的複雜檢查流程。

關於 flatMap() 方法

- 如果你沒辦法修改傳回型態怎麼辦？

```
return Optional.ofNullable(order)
               .map(Order::getCustomer)
               .map(Customer::getAddress)
               .map(Address::getCity)
               .orElse("n.a.");
```

```
public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Optional.ofNullable(mapper.apply(value));
    }
}
```

關於 flatMap() 方法

- 連續取得 List

```
List<String> itemNames = new ArrayList<>();
for(Order order : orders) {
    for(LineItem lineItem : order.getLineItems()) {
        itemNames.add(lineItem.getName());
    }
}
```

關於 flatMap() 方法

- 用 List 的 stream() 方法取得 Stream 之後，使用 flatMap() 方法

```
List<String> itemNames = orders.stream()
        .map(Order::getLineItems)
        .flatMap(lineItems -> lineItems.stream())
        .map(LineItem::getName)
        .collect(toList());
```

關於 flatMap() 方法

- 從盒子中取出盒子的操作（一個 Stream 接著一個 Stream）可以接續下去

```
List<String> itemNames = orders.stream()
        .map(Order::getLineItems)
        .flatMap(lineItems -> lineItems.stream())
        .map(LineItem::getPremiums)
        .flatMap(premiums -> premiums.stream())
        .map(Premium::getName)
        .collect(toList());
```

關於 flatMap() 方法

- 如果能瞭解 Optional、Stream（或其他型態）的 flatMap() 方法，其實就是取得盒子中的值，讓你指定這個值與下個盒子間的關係，那在撰寫與閱讀程式碼時，忽略掉 flatMap 這個名稱，就能比較清楚程式碼的主要意圖

Stream 與平行化

- 要獲得平行處理能力在 JDK8 中可以說很簡單

```
List<Person> males = persons.stream()
                .filter(person -> person.getGender() == Person.Gender.MALE)
                .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);


List<Person> males = persons.parallelStream()
                .filter(person -> person.getGender() == Person.Gender.MALE)
                .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

Stream 與平行化

- JDK8 希望你想要進行平行處理時，必須有明確的語義
- 想要知道 `Stream` 是否為平行處理，可以呼叫 `isParallel()` 來得知

Stream 與平行化

- 天下沒有白吃的午餐
  - 留意平行處理時的順序需求
  - 不要干擾 Stream 來源
  - 一次做一件事

Stream 與平行化

- 使用了 `parallelStream()`，不代表一定會平行處理而使得執行必然變快
- 得思考處理過程是否能夠分而治之而後合併結果，如果可能，方能從中獲益

Stream 與平行化

- Collectors 有 groupingBy() 與 groupingByConcurrent()

```
Map<Person.Gender, List<Person>> males = persons.parallelStream()
                    .collect(
                        groupingByConcurrent(Person::getGender));
```

Stream 與平行化

- Stream 實例若具有平行處理能力，處理過程會分而治之

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
numbers.parallelStream()
       .forEach(out::println);
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
numbers.parallelStream()
       .forEachOrdered(out::println);
```

Stream 與平行化

- 使用 `forEachOrdered()` 這類的有序處理時，可能會（或完全失去）失去平行化的一些優勢
- 實際上中介操作亦有可能如此，例如 `sorted()` 方法
- API 文件上基本上會記載終結操作時是否依來源順序

Stream 與平行化

- reduce() 基本上是按照來源順序，而 collect() 得視給予的 Collector 而定

```
List<Person> males = persons.parallelStream()
                    .filter(person -> person.getGender() == Gender.MALE)
                    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);

List<Person> males = persons.parallelStream()
                    .filter(person -> person.getGender() == Gender.MALE)
                    .collect(toList());
```

Stream 與平行化

- 在 collect() 操作時若想要有平行效果
  - Stream 必須有平行處理能力。
  - Collector 必須有 Collector.Characteristics.CONCURRENT 特性。
  - Stream 是無序的（Unordered）或 Collector 具 Collector.Characteristics.UNORDERED 特性。

Stream 與平行化

- 當 API 在處理小任務時，你不應該進行干擾
- 這樣的程式會引發 ConcurrentModifiedException

```
numbers.parallelStream()
       .filter(number -> {
            numbers.add(7);
            return number > 5;
       })
       .forEachOrdered(out::println);
```

Stream 與平行化

- 思考處理的過程中，實際上是由哪些小任務組成

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> alsoLt = new ArrayList<>();

for(Integer number : numbers) {
    if(number > 5) {
        alsoLt.add(number + 10);
        out.println(number);
    }
}
```

Stream 與平行化

- 記得一次只做一件事

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> biggerThan5 = numbers.stream()
                .filter(number -> number > 5)
                .collect(toList());

biggerThan5.forEach(out::println);

List<Integer> alsoLt = biggerThan5.stream()
                .map(number -> number + 10)
                .collect(toList());
```

Stream 與平行化

- 避免寫出以下的程式：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> alsoLt = new ArrayList<>();

numbers.stream()
       .filter(number -> {
           boolean isBiggerThan5 = number > 5;
           if(isBiggerThan5) {
               alsoLt.add(number + 10);
           }
           return isBiggerThan5;
        })
        .forEach(out::println);
```

Stream 與平行化

- 如果你試圖進行平行化處理時，就會發現，`alsoLt` 的順序並不照著 `numbers` 的順序
- 然而一次處理一個任務的版本，可以簡單地改為平行化版本，而又沒有順序問題

使用 CompletableFuture

- 非同步（Asynchronous）讀取文字檔案

```
public static Future readFileAsync(String file, Consumer<String> success,
                        Consumer<IOException> fail, ExecutorService service) {
    return service.submit(() -> {
        try {
            success.accept(new String(Files.readAllBytes(Paths.get(file))));
        } catch (IOException ex) {
            fail.accept(ex);
        }
    });
}

readFileAsync(args[0],
    content -> out.println(content),   // 成功處理
    ex -> ex.printStackTrace(),        // 失敗處理
    Executors.newFixedThreadPool(10)
);
```

使用 CompletableFuture

- 回呼地獄（Callback hell）

```
readFileAsync(args[0],
    content -> processContentAsync(content,
                processedContent -> out.println(processedContent) ,
                ex -> ex.printStackTrace(), service),
    ex -> ex.printStackTrace(), service);
```

使用 CompletableFuture

- JDK8 新增了 CompletableFuture

```
public static CompletableFuture<String> readFileAsync(
                    String file, ExecutorService service) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            return new String(Files.readAllBytes(Paths.get(file)));
        } catch(IOException ex) {
            throw new UncheckedException(ex);
        }
    }, service);
}

readFileAsync(args[0], poolService).whenComplete((ok, ex) -> {
    if(ex == null) {
        out.println(ok);
    } else {
        ex.printStackTrace();
    }
}).join();  // join()是為了避免 main 執行緒在任務完成前就關閉 ExecutorService
```

使用 CompletableFuture

- 繼續以非同步方式來處理結果

```
readFileAsync(args[0], poolService)
      .thenApplyAsync(String::toUpperCase)
      .whenComplete((ok, ex) -> {
          if(ex == null) {
             out.println(ok);
          } else {
             ex.printStackTrace();
          }
      });
```