Java程式設計進階
Stream API

鄭安翔
ansel.cheng@hotmail.com

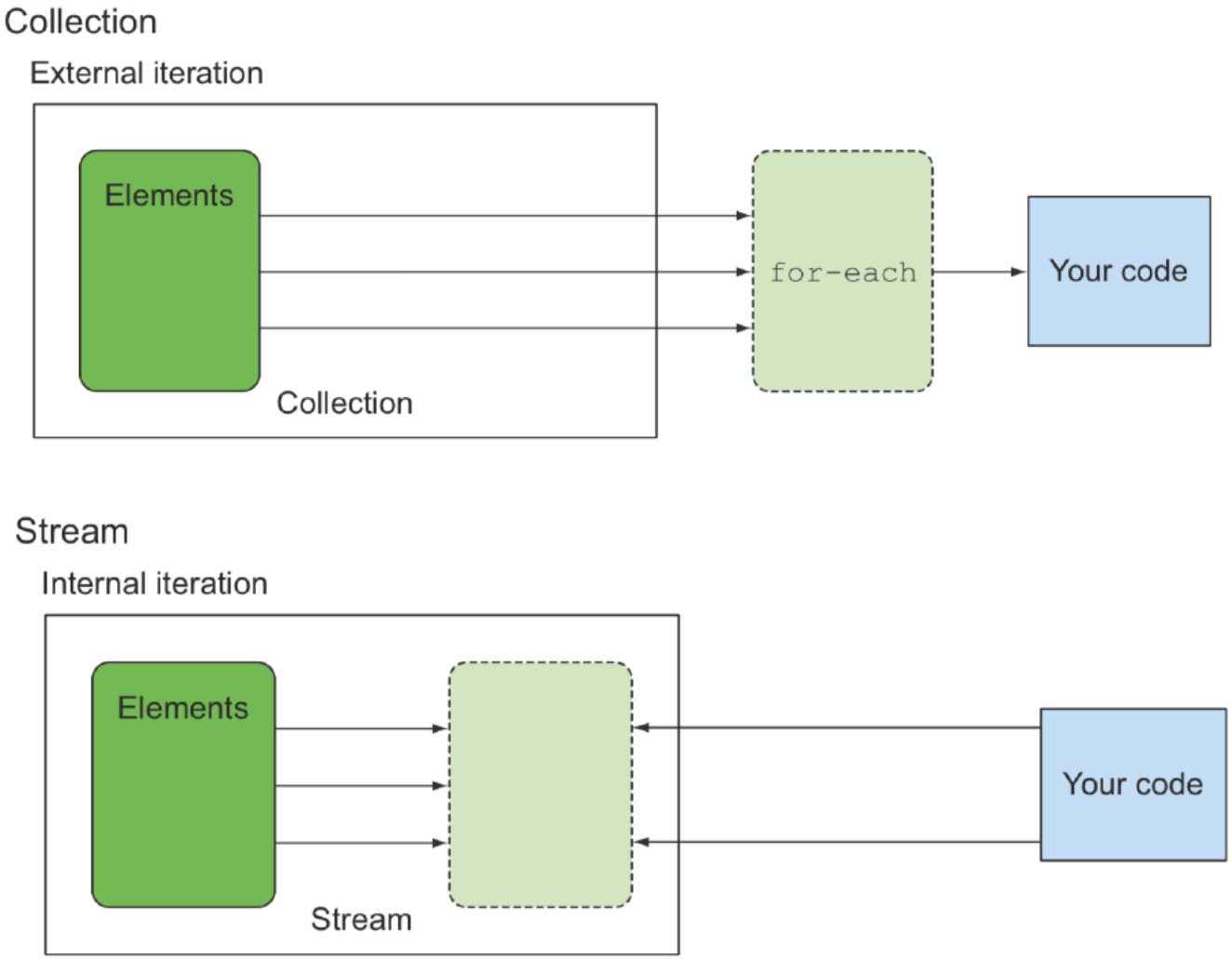
課程大綱

1) **集合串流操作**
   - **迭代 vs 串流 ForEach 操作**
   - **迭代 vs 串流 Filter 操作**
   - **連鎖呼叫 Method Chaining**
2) Stream API
3) Stream 進階

外部 vs. 內部迭代

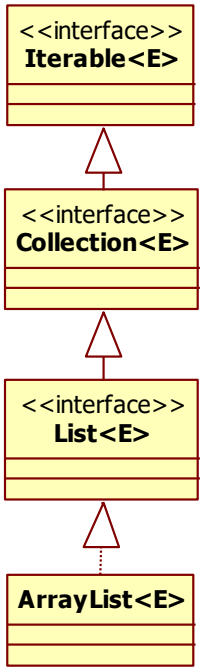
- Java 8 之前只能使用外部迭代 External Iteration
  - 撰寫 for 迴圈逐一讀取或操作集合元素
  - 程式碼較複雜
  - 無法進行並行運算
- Java 8 提供內部迭代 Internal Iteration功能
  - JDK 串流 (Stream) 物件可逐一讀取或操作集合元素
    - 使用 Lambda 表示式設定串流操作規則
  - 程式碼較簡單易懂
  - 可搭配可使用方法鏈接呼叫 (Method Chaining)
  - 可進行並行運算

外部 vs. 内部迭代


Collection
External iteration
Elements
Collection
for-each
Your code
Stream
Internal iteration
Elements
Stream
Your code

集合串流操作

- Java 8 在 Iterable interface 上新增 default 方法 forEach()

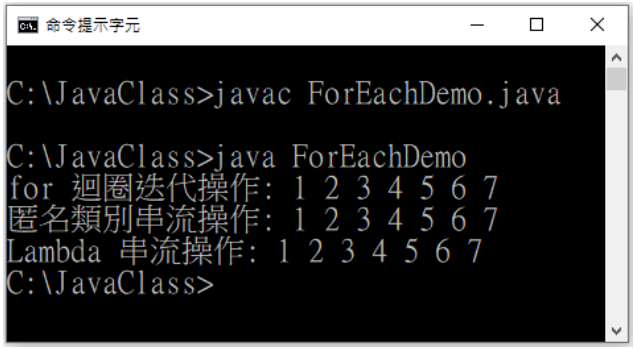


java.util.Iterable 介面

方法名稱	傳回值	說明
forEach(Consumer<? super T> action)	boolean	將集合中每一個元素, 以傳入的action物件所定義之accept()方法處理, 傳回成功或失敗boolean結果

集合迭代 vs 串流操作

```
import java.util.*;
import java.util.function.Consumer;
public class ForEachDemo {
        public static void main(String[] args) {
                List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
                //for 迴圈迭代操作
                System.out.print("for 迴圈迭代操作:");
                for(Integer n: list) {
                        System.out.print(" "+n);
                }

                //匿名類別進行forEach()串流操作
                System.out.print("\n匿名類別串流操作:");
                list.forEach(new Consumer<Integer>() {
                        public void accept(Integer n) {
                                System.out.print(" "+n);
                        }
                });

                // Lambda Express進行forEach()串流操作
                System.out.print("\nLambda 串流操作:");
                list.forEach(n -> System.out.print(" "+n));

        }
}
```



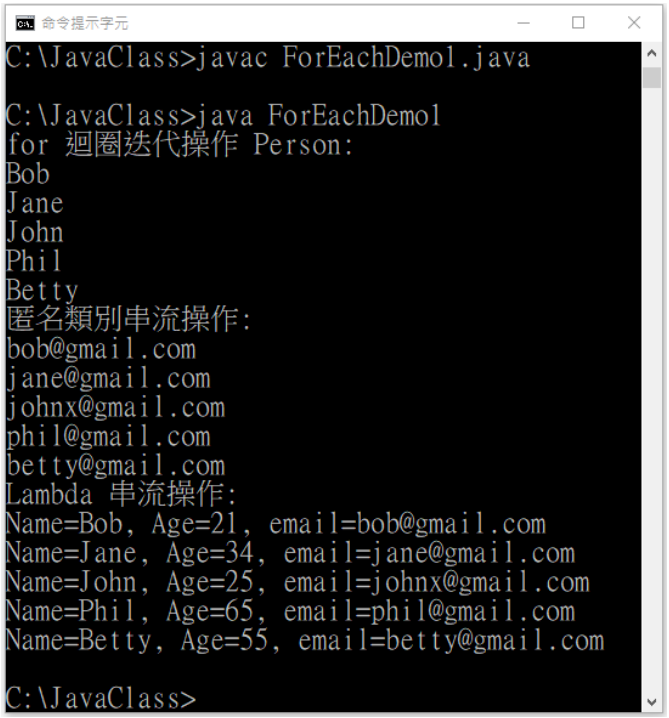
Person類別

```
import java.util.*;
public class Person implements Comparable<Person> {
	private String name, email;
	private int age;
	private List<String> hobbies;
	public Person(String name, String email,
	                int age, List<String> hobbies) {
		this.name = name;	this.age = age;
		this.email = email;	 this.hobbies = hobbies;
	}
	public String getName() {	return name;	}
	public int getAge() {	return age;	}
	public String getEmail() {	return email;	}
	public List<String> getHobbies() {	return hobbies;	}
	public int compareTo(Person p) { … }
	public int compareAgeTo(Person p) { … }
	public static int compareNameLength(Person a, Person b) { … }
```

```
	@Override
	public String toString() {
		return "Name=" + name +
			", Age=" + age +
			", email=" + email;
	}
	public static Person[] createArray() {
		List<Person> pl = createList();
		return pl.toArray(new Person[pl.size()]);
	}
	public static List<Person> createList() {
		List<Person> people = new ArrayList<>();
		people.add(new Person("Bob", "bob@gmail.com", 21, Arrays.asList("Piano", "Baseball", "Movie")));
		people.add(new Person("Jane", "jane@gmail.com", 34, Arrays.asList("Music", "Movie", "Swimming")));
		people.add(new Person("John", "johnx@gmail.com", 25, Arrays.asList("Music", "Baseball")));
		people.add(new Person("Phil", "phil@gmail.com", 65, Arrays.asList("Basketball", "Movie")));
		people.add(new Person("Betty", "betty@gmail.com", 55, Arrays.asList( "Swimming", "Piano", "Movie")));
		return people;
	}
}
```

集合迭代 vs 串流操作

```java
import java.util.*;
import java.util.function.Consumer;
public class ForEachDemo1 {
	public static void main(String[] args) {
		List<Person> persons = Person.createList();
		//for 迴圈迭代操作 Person
		System.out.println("for 迴圈迭代操作 Person:");
		for(Person p: persons) {
			System.out.println(p.getName());
		}

		//匿名類別進行forEach()串流操作Person
		System.out.println("匿名類別串流操作:");
		persons.forEach(new Consumer<Person>() {
			public void accept(Person p) {
				System.out.println(p.getEmail());
			}
		});

		// Lambda Express進行forEach()串流操作
		System.out.println("Lambda 串流操作:");
		persons.forEach(p -> System.out.println(p));

	}
}
```

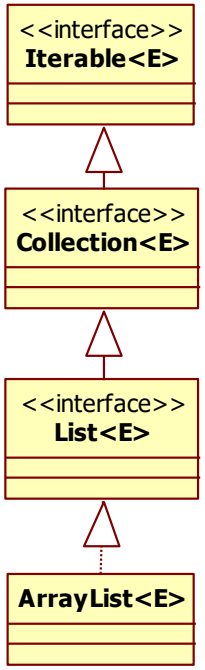


集合串流操作

- Java 8 在 Collection interface 上新增 default 方法 stream()

java.util.collection 介面

方法名稱	傳回值	說明
stream()	default Stream<E>	將集合中元素依序轉換為 Stream 物件



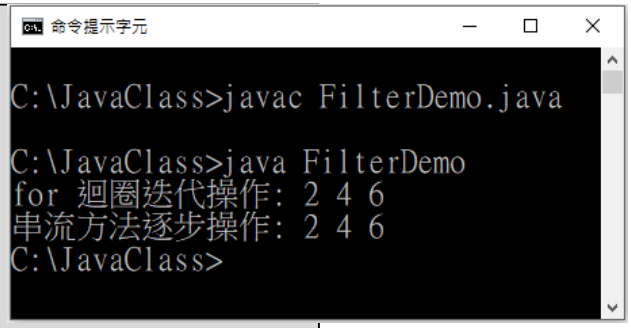
- Java 8 新增 Stream interface

java.util.stream.Stream介面

方法名稱	傳回值	說明
filter(Predicate<? super T> predicate)	Stream<T>	以指定的 Predicate 物件或Lambda Expression 為過濾條件傳回的 Stream 物件僅包含判定為ture的元素

集合迭代 vs 串流操作

```java
import java.util.*;
import java.util.stream.Stream
public class FilterDemo {
        public static void main(String[] args) {
                List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
                // for 迴圈迭代操作
                System.out.print("for 迴圈迭代操作:");
                for(Integer n: list) {
                     if(n%2==0)
                                System.out.print(" "+n);
                }

                // 串流方法逐步操作
                System.out.print("\n串流方法逐步操作:");
                Stream<Integer> stream = list.stream(); //轉換為串流物件
                Stream<Integer> evenStream = stream.filter(n -> n%2==0); //中間操作:過濾
                evenStream.forEach(n -> System.out.print(" "+n)); //終端操作:forEach

        }
}
```



集合迭代 vs 串流操作

```java
import java.util.*;
import java.util.stream.Stream
public class FilterDemo1 {
        public static void main(String[] args) {
                List<Person> persons = Person.createList();
                // for 迴圈迭代操作
                System.out.println("for 迴圈迭代操作:");
                for(Person p: persons) {
                    if(p.getAge()<30)
                            System.out.println(p);
                }

                //串流方法逐步操作
                System.out.println("串流方法逐步操作:");
                Stream<Person> stream = persons.stream(); //轉換為串流物件
                Stream<Person> youngStream = stream.filter(p -> p.getAge()<30); //中間操作:過濾
                youngStream.forEach(p -> System.out.println(p)); //終端操作:forEach
        }
}
```

命令提示字元

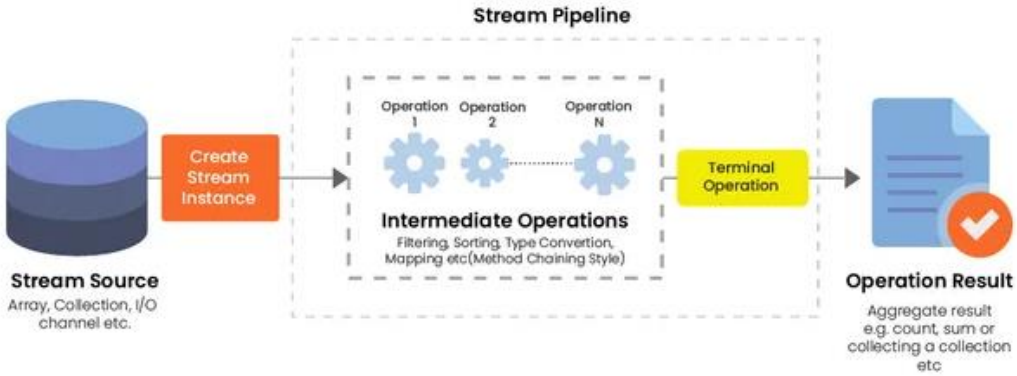
```
C:\JavaClass>javac FilterDemo1.java

C:\JavaClass>java FilterDemo1
for 迴圈迭代操作:
Name=Bob, Age=21, email=bob@gmail.com
Name=John, Age=25, email=johnx@gmail.com
串流方法逐步操作:
Name=Bob, Age=21, email=bob@gmail.com
Name=John, Age=25, email=johnx@gmail.com

C:\JavaClass>
```

方法連鎖呼叫

- 方法連鎖呼叫 Method Chaining
  - 物件連續呼叫多個方法
    - 傳回本身物件型態

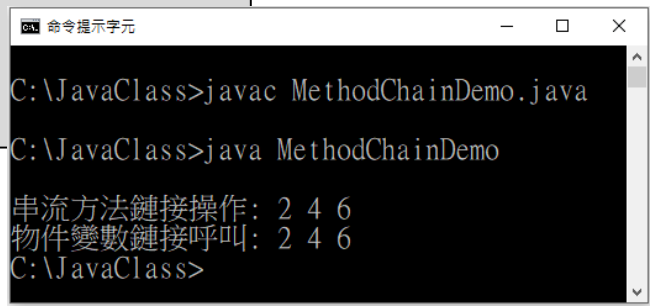
```
list .stream().filter(n -> n%2==0).forEach(n -> System.out.println(" "+n));
```

  - 用少量程式碼表達複雜操作
  - 提高程式碼可讀性
  - 實現fluent interface



方法連鎖呼叫

```
import java.util.*;
import java.util.function.*;
public class MethodChainDemo {
        public static void main(String[] args) {
                List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

                //串流方法鏈接操作
                System.out.print("\n串流方法鏈接操作:");
                list.stream().filter(n -> n%2==0).forEach(n -> System.out.println(" "+n));

                // 將Lambda 表示式宣告為物件變數進行鏈接呼叫
                System.out.print("\n物件變數鏈接呼叫:");
                Predicate<Integer> criteria = n -> n%2==0;
                Consumer<Integer> action = n -> System.out.println(" "+n);
                list.stream().filter(criteria).forEach(action);

        }
}
```



方法連鎖呼叫

```
import java.util.*;
import java.util.function.*;
public class MethodChainDemo1 {
        public static void main(String[] args) {
                List<Person> persons = Person.createList();

                //串流方法鏈接操作
                System.out.println("串流方法鏈接操作:");
                persons.stream().filter(p -> p.getAge()>30).forEach(p -> System.out.println(p));

                // 將Lambda 表示式宣告為物件變數進行鏈接呼叫
                System.out.print("\n物件變數鏈接呼叫:");
                Predicate<Integer> criteria = p -> p.getAge()>30;
                Consumer<Integer> action = p -> System.out.println(p);
                list.stream().filter(criteria).forEach(action);

        }
}
```

命令提示字元

```
C:\JavaClass>javac MethodChainDemo1.java

C:\JavaClass>java MethodChainDemo1
串流方法鏈接操作:
Name=Jane, Age=34, email=jane@gmail.com
Name=Phil, Age=65, email=phil@gmail.com
Name=Betty, Age=55, email=betty@gmail.com
物件變數鏈接呼叫:
Name=Jane, Age=34, email=jane@gmail.com
Name=Phil, Age=65, email=phil@gmail.com
Name=Betty, Age=55, email=betty@gmail.com

C:\JavaClass>
```

課程大綱

1) 集合串流操作
2) **Stream API**
   - **管線**
   - **中間操作**
   - **終端操作**
   - **捷徑終端操作**
3) Stream 進階

Java Streams

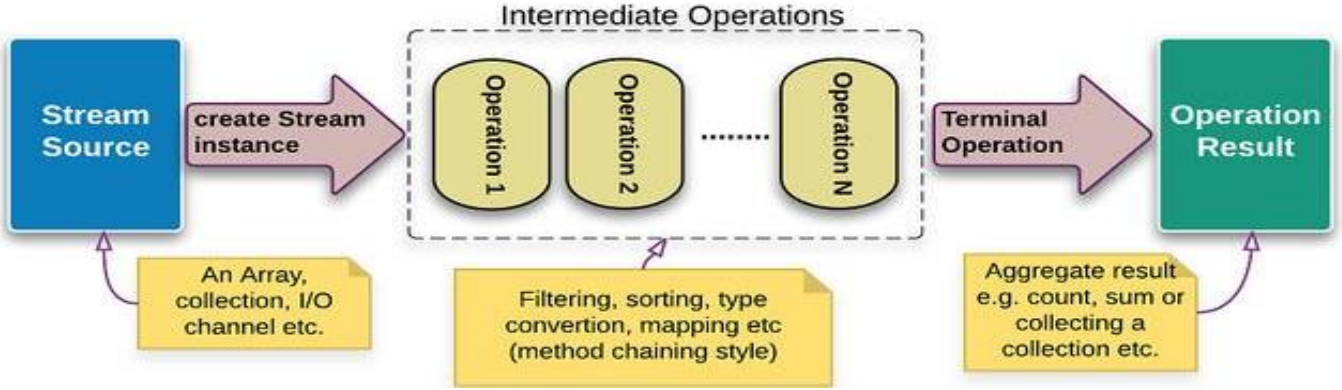
- 定義於 java.util.stream套件
- 將序列的物件元素轉換為串流物件
- 可套用多種可鏈接方法chaining methods
- Collection vs. Stream：
  - Collection 類別提供元素的管理規則 (List, Set, Queue) 和存取方法。
  - Stream 類別沒有提供存取元素的方法，以宣告的方式，定義即將對 stream 來源(通常是集合) 進行的各式操作。
    - 不可使用索引存取

Java Streams 特性

- Streams 特性：
  - Stream 元素為 immutable (不可變改)
  - Stream 元素只能使用一次(流水不能回頭)
  - Stream 串流可進行方法鏈接(chaining methods)
    - 稱為 Pipeline Operations (管線操作)
  - 惰性運算 Lazy Evaluation
  - 2種處理方式：
    - Sequential 循序處理 (預設)
    - Parallel 平行處理

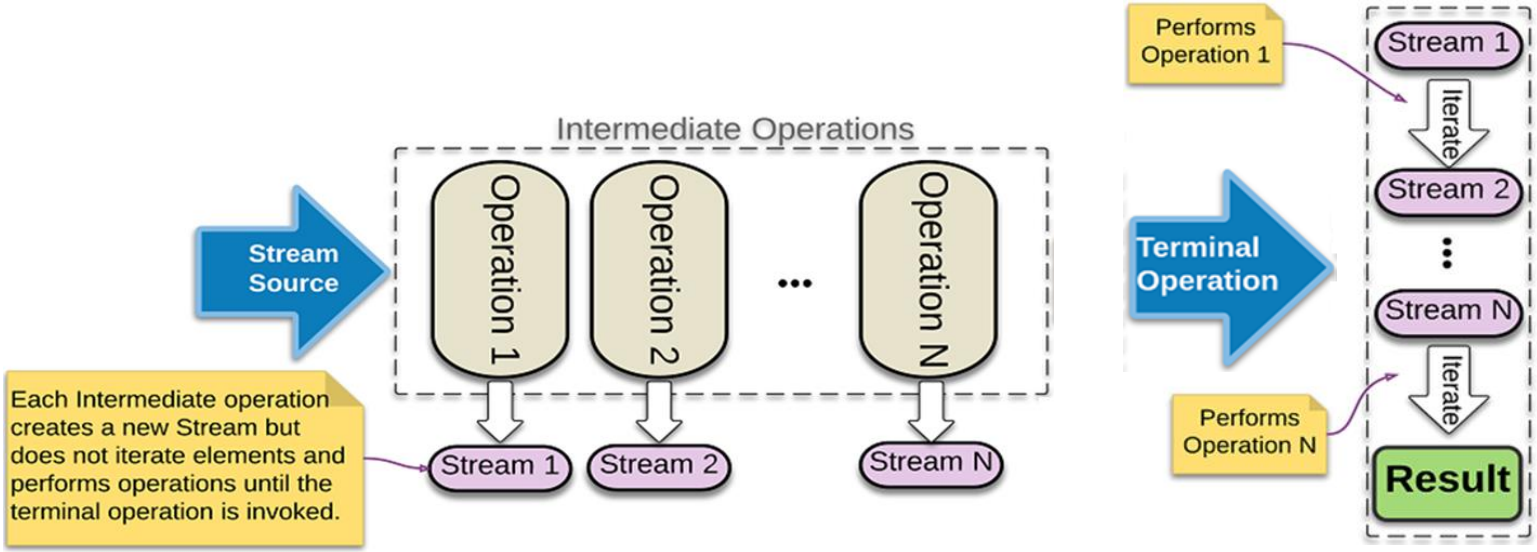
管線運作 Pipeline Operation

- 串流支援管線運作 Pipeline Operation
  - Source (來源)：將 Collection 物件、檔案物件轉換成 Stream 物件
  - Intermediate Operation (中間操作)：可呼叫多個，執行結果為Stream物件
  - Terminal Operation (終端操作) ：最終輸出結果，只有一個



Java Streams Laziness

- Streams Lazy Evaluation：
  - 串流順著每段管線向下，會先確認終端操作業方式後才開始輸送資料
  - 只在要開始執行時才要求輸送資料



Pipeline Operations 的種類

- Source 串流來源建立
  - Collection.stream()、Stream.of()、Arrays.stream()
- Intermediate Operation (中間操作)
  - filter()、peek()、skip()、limit()、distinct()、sorted()、map()、flatMap()
- Terminal Operation (終端操作)
  - forEach()、count()、 sum()、average()、min()、max()、collect()
- Short-Circuit Terminal Operation (捷徑終端操作)
  - findFirst()、findAny()、anyMatch()、allMatch()、noneMatch()

建立串流物件

- java.util.collection 介面

方法名稱	傳回值	說明
stream()	default Stream<E>	將集合中元素依序轉換為 Stream 物件

- java.util.Arrays 類別

方法名稱	傳回值	說明
stream(T[] array)	static <T> Stream<T>	將指定泛型陣列元素依序轉換為 Stream 物件

- java.util.stream.Stream 介面

方法名稱	傳回值	說明
of(T... values)	static <T> Stream<T>	將指定泛型陣列元素依序轉換為 Stream 物件
iterate(T seed, UnaryOperator<T> f)	static <T> Stream<T>	由起始元素seed重複帶入函式 f，生成無限的有序Stream物件 f(seed)、f(f(seed))、f(f(f(seed)))、f(f(f(f(seed))))
generate(Supplier<T> s)	static <T> Stream<T>	由Supplier函式生成的無限無序Stream物件(常數或隨機元素串流)

建立串流物件

```
import java.util.*;
import java.util.stream.*;
public class CreateStreamDemo {
    public static void main(String[] args) {
        Person[] persons = Person.createArray();

        Stream<Integer>  i1 = Arrays.asList(1, 2, 3, 4).stream();
        Stream<String>   s1 = Arrays.asList("Bob", "Jane", "John", "Phil", "Betty").stream();
        Stream<Person> p1 = Person.createList().stream();

        Stream<Integer>  i2 = Stream.of(1, 2, 3, 4);
        Stream<String>   s2 = Stream.of("Bob", "Jane", "John", "Phil", "Betty");
        Stream<Person> p2 = Stream.of(persons[0], persons[1], persons[2], persons[3], persons[4]);

        Stream<Integer>   i3 = Arrays.stream(new Integer[] {1, 2, 3, 4});
        Stream<String>   s3 = Arrays.stream(new String[]{"Bob", "Jane", "John", "Phil", "Betty"});
        Stream<Person> p3 = Arrays.stream(persons);
        ....
    }
}
```

命令提示字元

```
C:\JavaClass>java CreateStreamDemo
使用stream()建構 int串流: 1 2 3 4
使用stream()建構 String串流: Bob Jane John Phil Betty
使用stream()建構 Person串流:
 Name=Bob, Age=21, email=bob@gmail.com
 Name=Jane, Age=34, email=jane@gmail.com
 Name=John, Age=25, email=johnx@gmail.com
 Name=Phil, Age=65, email=phil@gmail.com
 Name=Betty, Age=55, email=betty@gmail.com

使用Stream.of()建構 int串流: 1 2 3 4
使用Stream.of()建構 String串流: Bob Jane John Phil Betty
使用Stream.of()建構 Person串流:
 Name=Bob, Age=21, email=bob@gmail.com
 Name=Jane, Age=34, email=jane@gmail.com
 Name=John, Age=25, email=johnx@gmail.com
 Name=Phil, Age=65, email=phil@gmail.com
 Name=Betty, Age=55, email=betty@gmail.com

使用Arrays.stream()建構 int串流: 1 2 3 4
使用Arrays.stream()建構 String串流: Bob Jane John Phil Betty
使用Arrays.stream()建構 Person串流:
 Name=Bob, Age=21, email=bob@gmail.com
 Name=Jane, Age=34, email=jane@gmail.com
 Name=John, Age=25, email=johnx@gmail.com
 Name=Phil, Age=65, email=phil@gmail.com
 Name=Betty, Age=55, email=betty@gmail.com

C:\JavaClass>
```

建立串流物件

```
import java.util.stream.*;
import java.util.Random;
public class CreateStreamDemo1 {
	public static void main(String[] args) {
		Stream<Integer>  i1 = Stream.iterate(1, i -> i+1);
		Stream<Integer>  i2 = Stream.generate(() -> (int)(Math.random()*100));
		Stream<Double> d1 = Stream.generate(() -> new Random().nextDouble());

		System.out.print("Stream.iterate()建構: ");
		i1.limit(8).forEach(i -> System.out.print(i));
		System.out.print("\nStream.generate()建構: ");
		i2.limit(8).forEach(i -> System.out.print(" "+i));
		System.out.print("\nStream.generate()建構: ");
		d1.limit(4).forEach(d -> System.out.print(" "+d));
	}
}
```

命令提示字元

```
C:\JavaClass>javac CreateStreamDemo1.java

C:\JavaClass>java CreateStreamDemo1
Stream.iterate()建構: 12345678
Stream.generate()建構:  94 76 0 11 91 92 99 5
Stream.generate()建構:  0.6287918834972305 0.6523967499334471 0.07923926505074497 0.15487023407196998
C:\JavaClass>
```

Intermediate Operation (中間操作)

- 傳回新的串流物件
- Laziness Seeking
- Stateless 無狀態操作
  - filter()、map()、flatMap()
- Stateful 有狀態操作：需保存元素於緩衝區
  - skip()、limit() ：bounded 有界
  - distinct()、sorted()：unbounded 無界

常用中間操作

- java.util.stream.Stream 介面

方法名稱	傳回值	說明
filter(Predicate<? super T> predicate)	Stream<T>	以指定的 Predicate 物件或Lambda Expression 為條件過濾原串流，傳回的 Stream 物件僅包含判定為true的元素
peek(Consumer<? super T> action)	Stream<T>	以指定的 Consumer 物件或Lambda Expression 為條件操作原串流傳回個元素執行後的 Stream 物件
skip(long n)	Stream<T>	排除原串流中前N個元素，傳回排除後的Stream 物件
limit(long maxSize)	Stream<T>	將原串流由首元素開始長度截斷不超過maxSize，傳回個數上限為maxSize的Stream 物件
distinct()	Stream<T>	排除原串流中重複的元素，(根據Object.equals(…)方法)傳回不重複的Stream 物件
sorted()	Stream<T>	根據元素自然排序規則將原串流排序，傳回排序後的Stream 物件
sorted(Comparator<? super T> comparator)	Stream<T>	將原串流元素根據comparator排序規則排序，傳回排序後的Stream 物件

常用終端操作

- java.util.stream.Stream 介面

方法名稱	傳回值	說明
forEach(Consumer<? super T> action)	void	對串流所有元素執行指定函式操作action，無傳回值
collect(Collector<? super T,A,R> collector)	<R,A> R	收集串流所有元素，儲存於指定型態集合中傳回
count()	long	傳回串流集合元素個數
max(Comparator<? super T> comparator)	Optional<T>	依據指定comparator比較規則，傳回最大的串流元素
min(Comparator<? super T> comparator)	Optional<T>	依據指定comparator比較規則，傳回最小的串流元素
reduce(T identity, BinaryOperator<T> accumulator)	T	指定初值T，依指定函式accumulator累加，將串流元素減少為單一數值後傳回

Collectors 收集器

- java.util.stream.Collectors

方法名稱	傳回值	說明
toList()	static <T> Collector<T,?,List<T>>	傳回可將串流物件元素收集到List中的收集器
toSet()	static <T> Collector<T,?,Set<T>>	傳回可將串流物件元素收集到Set中的收集器
toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)	static <T,K,U> Collector<T,?,Map<K,U>>	傳回可將串流元素分別轉換為Key及Value，收集到Map中的收集器

常用中間操作

```
import java.util.stream.*;
public class IntermediateOperation {
    public static void main(String[] args) {
        Stream<Integer> iStream = Stream.generate(() -> (int)(Math.random()*100));
        System.out.print("Integer串流filter():");
        List<Integer> iList = iStream.limit(10).peek(i -> System.out.print(" "+i))
                                        .filter(i->i%2==0).collect(Collectors.toList());
        System.out.println("\n列表:"+iList);

        Stream<String>  sStream = Stream.of("Bob","Jane","John","Phil","John","Betty","Bob");
        System.out.print("\nString串流distinct():");
        long count = sStream.peek(s->System.out.print("\n"+s)).distinct()
                                .peek(i->System.out.print(i)).count();
        System.out.println("\n不重複字串個數:"+count);

        Stream<Person>  pStream = Person.createList().stream();
        System.out.print("\nPerson串流skip():");
        pStream.peek(p -> System.out.print("\n"+p.getName())).skip(2)
                .forEach(p-> System.out.print(" "+p.getAge()));
    }
}
```

```
命令提示字元
C:\JavaClass>javac IntermediateOperation.java

C:\JavaClass>java IntermediateOperation
Integer串流filter(): 45 78 19 32 34 46 81 94 73 46
列表:[78, 32, 34, 46, 94, 46]

String串流distinct():
BobBob
JaneJane
JohnJohn
PhilPhil
John
BettyBetty
Bob
不重複字串個數:5

Person串流skip():
Bob
Jane
John 25
Phil 65
Betty 55
C:\JavaClass>
```

Java 8 Comparator 介面

- java.util.stream.Comparator 介面

方法名稱	傳回值	說明
comparing(Function<? super T, ? extends U> keyExtractor)	static <T,U extends Comparable <? super U>> Comparator<T>	建立自訂比較器，傳入一個函數，該函數從物件T中取得一個整數作為排序鍵
thenComparing(Comparator<? super T> other)	default Comparator<T>	建立第二順位比較器
reversed()	default Comparator<T>	返回將此比較器反向順序的比較器。

串流排序

```
import java.util.*;
import java.util.stream.*;
public class SortedDemo {
    public static void main(String[] args) {
        Stream<Integer> iStream1 = Stream.of(86,66,26,95,87,17,1
        List<Integer> iList1 = iStream1.sorted().collect(Collectors
        System.out.println("Integer串流自然排序:"+iList1);

        Stream<String> sStream1 = Stream.of("Bob", "jane", "Ph
        List<String> sList1 = sStream1.sorted().collect(Collectors
        System.out.println("String串流自然排序:"+sList1);

        Stream<Person> pStream1 = Person.createList().stream()
        System.out.println("Person串流自然排序: ");
        pStream1.sorted().forEach(p->System.out.println(p));

        Stream<Integer> iStream2 = Stream.of(86,66,26,95,87,17,1,21,58,51);
        List<Integer> iList2 = iStream2.sorted((i1,i2)->i2.compareTo(i1)).collect(Collectors.toList());
        System.out.println("\nInteger串流自訂排序:"+iList2);

        Stream<String> sStream2 = Stream.of("Bob", "jane", "Phil", "John", "betty");
        List<String> sList2 = sStream2.sorted((s1,s2)->s1.compareToIgnoreCase(s2)).collect(Collectors.toList());
        System.out.println("String串流自訂排序:"+sList2);

        Stream<Person> pStream2 = Person.createList().stream();
        System.out.println("Person串流自訂排序: ");
        pStream2.sorted((p1,p2)->p1.compareAgeTo(p2)).forEach(p->System.out.println(p));
    }
}
```

命令提示字元

```
C:\JavaClass>javac SortedDemo.java

C:\JavaClass>java SortedDemo
Integer串流自然排序:[1, 17, 21, 26, 51, 58, 66, 86, 87, 95]
String串流自然排序:[Bob, John, Phil, betty, jane]
Person串流自然排序:
Name=Betty, Age=55, email=betty@gmail.com
Name=Bob, Age=21, email=bob@gmail.com
Name=Jane, Age=34, email=jane@gmail.com
Name=John, Age=25, email=john@gmail.com
Name=Phil, Age=65, email=phil@gmail.com

Integer串流自訂排序:[95, 87, 86, 66, 58, 51, 26, 21, 17, 1]
String串流自訂排序:[betty, Bob, jane, John, Phil]
Person串流自訂排序:
Name=Bob, Age=21, email=bob@gmail.com
Name=John, Age=25, email=john@gmail.com
Name=Jane, Age=34, email=jane@gmail.com
Name=Betty, Age=55, email=betty@gmail.com
Name=Phil, Age=65, email=phil@gmail.com

C:\JavaClass>
```

串流排序

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
public class SortedDemo1 {
    public static void main(String[] args) {
        Stream<String> sStream = Stream.of("Bob", "jane", "Phil", "John", "betty");
        List<String> sList = sStream.sorted((s1,s2)->s1.compareToIgnoreCase(s2))
                                    .collect(Collectors.toList());
        System.out.println("String串流自訂Comparator排序:"+sList);

        Stream<Person> pStream = Person.createList().stream();
        Function<Person, Integer> getNameLength = p->p.getName().length();
        Function<Person, Integer> getAge = p->p.getAge();
        Comparator<Person> comparator = Comparator.comparing(getNameLength)
                                                  .thenComparing(getAge)
                                                  .reversed();
        System.out.println("Person串流自訂Comparator排序: ");
        pStream.sorted(comparator).forEach(p->System.out.println(p));

    }
}
```

命令提示字元

```
C:\JavaClass>javac SortedDemo1.java

C:\JavaClass>java SortedDemo1
String串流自訂Comparator排序:[betty, Bob, jane, John, Phil]
Person串流自訂Comparator排序:
Name=Betty, Age=55, email=betty@gmail.com
Name=Phil, Age=65, email=phil@gmail.com
Name=Jane, Age=34, email=jane@gmail.com
Name=John, Age=25, email=john@gmail.com
Name=Bob, Age=21, email=bob@gmail.com

C:\JavaClass>
```

常用中間操作 - 對應轉換

- Map
  - 對成員套用函式，將傳入元素一對一轉換為另一元素
- FlatMap
  - 對成員套用函式，將傳入元素一對多轉換為多個元素
  - 將元素扁平化傳回：二(多)維變為一維
- java.util.stream.Stream 介面

方法名稱	傳回值	說明
map(Function<? super T, ? extends R> mapper)	<R> Stream<R>	已指定mapper函式轉換串流元素，傳回轉換後的串流
flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	<R> Stream<R>	已指定mapper函式轉換串流元素，傳回轉換後並扁平化的串流

對應轉換

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
public class MapDemo {
	public static void main(String[] args) {
		Stream<Integer> iStream = Stream.of(1, 2, 3, 4, 5);
		Function<Integer, Integer> iMapper = i -> i*i;
		List<Integer> iList = iStream.map(iMapper).collect(Collectors.toList());
		System.out.println("Integer串流map()轉換: "+iList);

		Stream<String> sStream = Stream.of("Bob", "jane", "Phil", "John", "betty");
		List<String> sList = sStream.map(s->"Hi "+s.toUpperCase()).collect(Collectors.toList());
		System.out.println("String串流map()轉換: "+sList);

		Stream<Person> pStream = Person.createList().stream();
		List<String> emailList = pStream.map(p->p.getEmail()).collect(Collectors.toList());
		System.out.println("Person串流map()轉換: "+emailList);
	}
}
```

命令提示字元

```
C:\JavaClass>javac MapDemo.java

C:\JavaClass>java MapDemo
Integer串流map()轉換: [1, 4, 9, 16, 25]
String串流map()轉換: [Hi BOB, Hi JANE, Hi PHIL, Hi JOHN, Hi BETTY]
Person串流map()轉換: [bob@gmail.com, jane@gmail.com, john@gmail.com, phil@gmail.com, betty@gmail.com]

C:\JavaClass>
```

扁平化對應轉換

```
import java.util.*;
import java.util.stream.*;
public class FlatMapDemo {
	public static void main(String[] args) {
		List<Integer> oddList = Arrays.asList(1, 3, 5, 7);
		List<Integer> evenList = Arrays.asList(2, 4, 6, 8);
		List<List<Integer>> listOfList = Arrays.asList(oddList, evenList);
		System.out.println("扁平化前: "+listOfList);

		List<Integer> flattenList = listOfList.stream().flatMap(l -> l.stream())
								.collect(Collectors.toList());
		System.out.println("扁平化後: "+flattenList);	}
}
```

命令提示字元

```
C:\JavaClass>javac FlatMapDemo.java

C:\JavaClass>java FlatMapDemo
扁平化前: [[1, 3, 5, 7], [2, 4, 6, 8]]
扁平化後: [1, 3, 5, 7, 2, 4, 6, 8]

C:\JavaClass>
```

扁平化對應轉換

```
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
public class FlatMapDemo1 {
	public static void main(String[] args) {
		Stream<Integer> iStream = Stream.of(1, 2, 3, 4, 5);
		Function<Integer, Stream<Integer>> iMapper = i -> Stream.of(i, 2*i, i*i);
		List<Integer> iList = iStream.flatMap(iMapper).collect(Collectors.toList());
		System.out.println("Integer串流flatMap()轉換: "+iList);

		Stream<String> sStream = Stream.of("Bob", "jane", "Phil", "John", "betty");
		List<String> sList = sStream.flatMap(s->Stream.of("Hi "+s, s.toUpperCase(),
								s.toLowerCase())).collect(Collectors.toList());
		System.out.println("String串流flatMap()轉換: "+sList);

		Stream<Person> pStream = Person.createList().stream();
		List<String> hobbyList = pStream.map(p->p.getHobbies())
								.flatMap(l->l.stream())
								.distinct().sorted()
								.collect(Collectors.toList());
		System.out.println("Person嗜好串流 flatMap()轉換: "+hobbyList);
	}
}
```

命令提示字元

```
C:\JavaClass>javac FlatMapDemo1.java

C:\JavaClass>java FlatMapDemo1
Integer串流flatMap()轉換: [1, 2, 1, 2, 4, 4, 3, 6, 9, 4, 8, 16, 5, 10, 25]
String串流flatMap()轉換: [Hi Bob, BOB, bob, Hi jane, JANE, jane, Hi Phil, PHIL, phil, Hi John, JOHN, john, Hi betty, BETTY, betty]
Person嗜好串流 flatMap()轉換: [Baseball, Basketball, Movie, Music, Piano, Swimming]

C:\JavaClass>
```

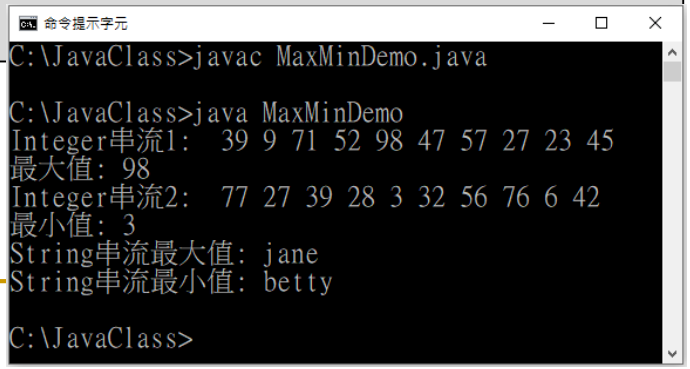
Optional<T> 類別

- Optional 類別
  - 避免null值檢查，或NullPointerException出現
  - 容器物件 (container object)，可以包含/不包含指定泛型的 非null 物件
- java.util.Optional<T> 類別

方法名稱	傳回值	說明
empty()	static <T> Optional<T>	建立一個空的Option物件傳回
of(T value)	static <T> Optional<T>	建立一個包含指定型態物件value的Option物件傳回
isPresent()	boolean	檢驗容器內容物是否存在
get()	T	取得容器內容值，如果內容不存在丟出NoSuchElementException
ifPresent(Consumer<? super T> consumer)	void	檢驗容器內容物是否存在，如果存在，對容器內容物件執行指定消費函式consumer
orElse(T other)	T	傳回容器內容值，如果內容不存在傳回other
orElseGet(Supplier<? extends T> other)	T	傳回容器內容值，如果內容不存在，對傳回執行提供函式other後產生之物件

串流最大最小值

```
import java.util.*;
import java.util.stream.*;
public class MaxMinDemo{
	public static void main(String[] args) {
		Stream<Integer> iStream1 = Stream.generate(() -> (int)(Math.random()*100)).limit(10);
		System.out.print("Integer串流1: ");
		Optional<Integer> maxInt = iStream1.peek(i->System.out.print(" "+i)).max((i1,i2)->i1.compareTo(i2));
		System.out.println("\n最大值: "+maxInt.get());
		Stream<Integer> iStream2 = Stream.generate(() -> (int)(Math.random()*100)).limit(10);
		System.out.print("Integer串流2: ");
		Optional<Integer> minInt = iStream2.peek(i->System.out.print(" "+i)).min((i1,i2)->i1.compareTo(i2));
		System.out.println("\n最小值: "+minInt.get());

		Stream<String> sStream1 = Stream.of("Bob", "jane", "Phil", "John", "betty");
		Optional<String> maxStr = sStream1.max((s1,s2)->s1.compareTo(s2));
		System.out.println("String串流最大值: "+maxStr.get());
		Stream<String> sStream2 = Stream.of("Bob", "jane", "Phil", "John", "betty");
		Optional<String> minStr = sStream2.min((s1,s2)->s1.compareToIgnoreCase(s2));
		System.out.println("String串流最小值: "+minStr.get());	}
}
```



串流最大最小值

```
import java.util.*;
import java.util.stream.*;
public class MaxMinDemo1{
	public static void main(String[] args) {
		Stream<Person> pStream1 = Person.createList().stream();
		//Optional<Person> minPerson = pStream1.min((p1,p2)->p1.getAge()-p2.getAge());
		Optional<Person> minPerson = pStream1.min((p1,p2)->p1.compareAgeTo(p2));
		if(minPerson.isPresent())
			System.out.println("年紀最小:"+minPerson.get());

		Stream<Person> pStream2 = Person.createList().stream();
		//Optional<Person> maxPerson = pStream2.max((p1,p2)->p1.getName().length()-p2.getName().length());
		Optional<Person> maxPerson = pStream2.max ((p1,p2)-> Person.compareNameLength(p1,p2));
		if(maxPerson.isPresent())
			System.out.println("名字最長:"+maxPerson.get());
	}
}
```

命令提示字元

```
C:\JavaClass>javac MaxMinDemo1.java

C:\JavaClass>java MaxMinDemo1
年紀最小:Name=Bob, Age=21, email=bob@gmail.com
名字最長:Name=Betty, Age=55, email=betty@gmail.com

C:\JavaClass>javac MaxMinDemo1.java
```

常用捷徑終端操作

- java.util.stream.Stream 介面

方法名稱	傳回值	說明
findFirst()	Optional<T>	傳回包含串流中第一個元素的Optional物件，若無元素，傳回空的Optional物件
findAny()	Optional<T>	傳回包含串流中某個元素的Optional物件(在並行運算中獲得佳效能)，若無元素，傳回空的Optional物件
anyMatch(Predicate<? super T> predicate)	boolean	是否串流中有任一元素符合指定判定條件predicate
allMatch(Predicate<? super T> predicate)	boolean	是否串流中有所有元素均符合指定判定條件predicate
noneMatch(Predicate<? super T> predicate)	boolean	是否串流中沒有任何元素符合指定判定條件predicate

串流Find

```
import java.util.*;
import java.util.stream.*;
public class FindDemo {
	public static void main(String[] args) {
		Stream<Integer> iStream1 = Stream.of(1, 2, 3, 4, 5);
		Optional<Integer> firstInt = iStream1.findFirst();
		System.out.println("第一個Integer: "+firstInt.get());

		Stream<String> sStream1 = Stream.of("Bob", "jane", "Phil", "John", "betty");
		Optional<String> firstStr = sStream1.findFirst();
		System.out.println("第一個String: "+firstStr.get());

		Stream<Integer> iStream2 = Stream.of(1, 2, 3, 4, 5);
		Optional<Integer> anyInt = iStream2.findAny();
		System.out.println("\n任一個Integer: "+anyInt.get());

		Stream<String> sStream2 = Stream.of("Bob", "jane", "Phil", "John", "betty");
		Optional<String> anyStr = sStream2.findAny();
		System.out.println("任一個String: "+anyStr.get());
	}
}
```

命令提示字元

```
C:\JavaClass>javac FindDemo.java

C:\JavaClass>java FindDemo
第一個Integer: 1
第一個String: Bob
任一個Integer: 1
任一個String: Bob

C:\JavaClass>
```

串流Find Laziness

```
import java.util.*;
import java.util.stream.*;
public class FindDemo1 {
	public static void main(String[] args) {
		List<Person> pList = Person.createList();

		Optional<Person> firstResult = pList.stream().peek(p->System.out.println("Name:"+p.getName()))
				.filter(p->p.getName().startsWith("B")).peek(p->System.out.println("Starts with B"))
				.filter(p->p.getAge()>30).peek(p->System.out.println("Age>30:"+p.getAge()))
				.findFirst();
		if(firstResult.isPresent()){
			System.out.println(firstResult.get());
		} else {
			System.out.println("無符合條件成員");
		}

		firstResult = pList.stream().peek(p->System.out.println("Name:"+p.getNa
				.filter(p->p.getName().startsWith("J")).peek(p->System.out.p
				.filter(p->p.getAge()>40).peek(p->System.out.println("Age>4
				.findFirst();
		if(firstResult.isPresent()){
			System.out.println(firstResult.get());
		} else {
			System.out.println("無符合條件成員");
		}
	}
}
```

命令提示字元

```
C:\JavaClass>java FindDemo1
Name:Bob
Starts with B
Name:Jane
Name:John
Name:Phil
Name:Betty
Starts with B
Age>30:55
Name=Betty, Age=55, email=betty@gmail.com

Name:Bob
Name:Jane
Starts with J
Name:John
Starts with J
Name:Phil
Name:Betty
無符合條件成員

C:\JavaClass>
```

串流Match

```
import java.util.*;
import java.util.stream.*;
public class MatchDemo {
    public static void main(String[] args) {
        Stream<Integer> iStream1 = Stream.generate(() -> (int)(Math.random()*1
        System.out.print("Integer串流1: ");
        boolean iResult1 = iStream1.peek(i->System.out.print(" "+i)).allMatch(i->
        System.out.println("\n所有數值都大於10: "+iResult1);
        Stream<String> sStream1 = Stream.of("Bob", "jane", "Phil", "John", "be
        boolean sResult1 = sStream1.allMatch(s->s.length()<6);
        System.out.println("所有字串長度均小於6: "+sResult1);

        Stream<Integer> iStream2 = Stream.generate(() -> (int)(Math.random()*100)).limit(6);
        System.out.print("\nInteger串流2: ");
        boolean iResult2 = iStream2.peek(i->System.out.print(" "+i)).anyMatch(i->i%5==0);
        System.out.println("\n數值中有5的倍數: "+iResult2);
        Stream<String> sStream2 = Stream.of("Bob", "jane", "Phil", "John", "betty");
        boolean sResult2 = sStream2.anyMatch(s->s.equals("Phil"));
        System.out.println("字串中包含Phil: "+sResult2);

        Stream<Integer> iStream3 = Stream.generate(() -> (int)(Math.random()*100)).limit(6);
        System.out.print("\nInteger串流3: ");
        boolean iResult3 = iStream3.peek(i->System.out.print(" "+i)).noneMatch(i->i<10);
        System.out.println("\n沒有小於10的數值: "+iResult3);
        Stream<String> sStream3 = Stream.of("Bob", "jane", "Phil", "John", "betty");
        boolean sResult3 = sStream3.noneMatch(s->s.equals("Phil"));
        System.out.println("字串中不包含Phil: "+sResult3);
    }
}
```

命令提示字元

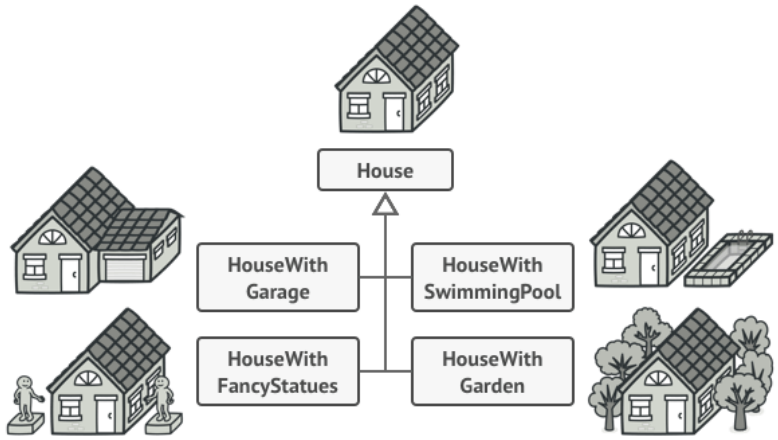
```
C:\JavaClass>javac MatchDemo.java

C:\JavaClass>java MatchDemo
Integer串流1:  88 40 39 17 52 24
所有數值都大於10: true
所有字串長度均小於6: true

Integer串流2:  37 82 16 38 41 40
數值中有5的倍數: true
字串中包含Phil: true

Integer串流3:  16 31 94 77 96 9
沒有小於10的數值: false
字串中不包含Phil:: false

C:\JavaClass>
```

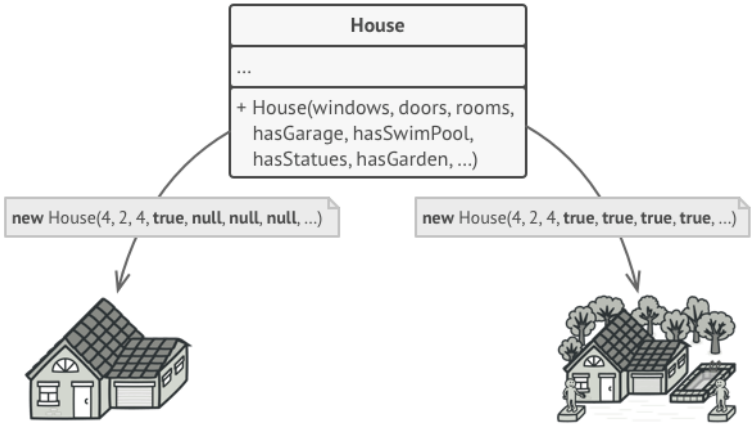
課程大綱

1) 集合串流操作
2) Stream API
3) **Stream 進階**
   - **Builder Design Pattern**
   - **基本型別串流**
   - **Stream API 並行操作**

Builder(生成者) Design Pattern

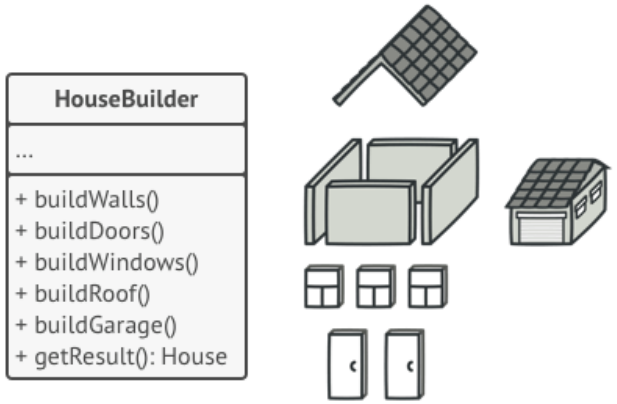
- 問題 / 需求
  - 類別有許多選擇性的屬性時，需要很多建構子多載才能滿足需求
  - 使用建構子建構物件實體時，需要同時提供所有的屬性



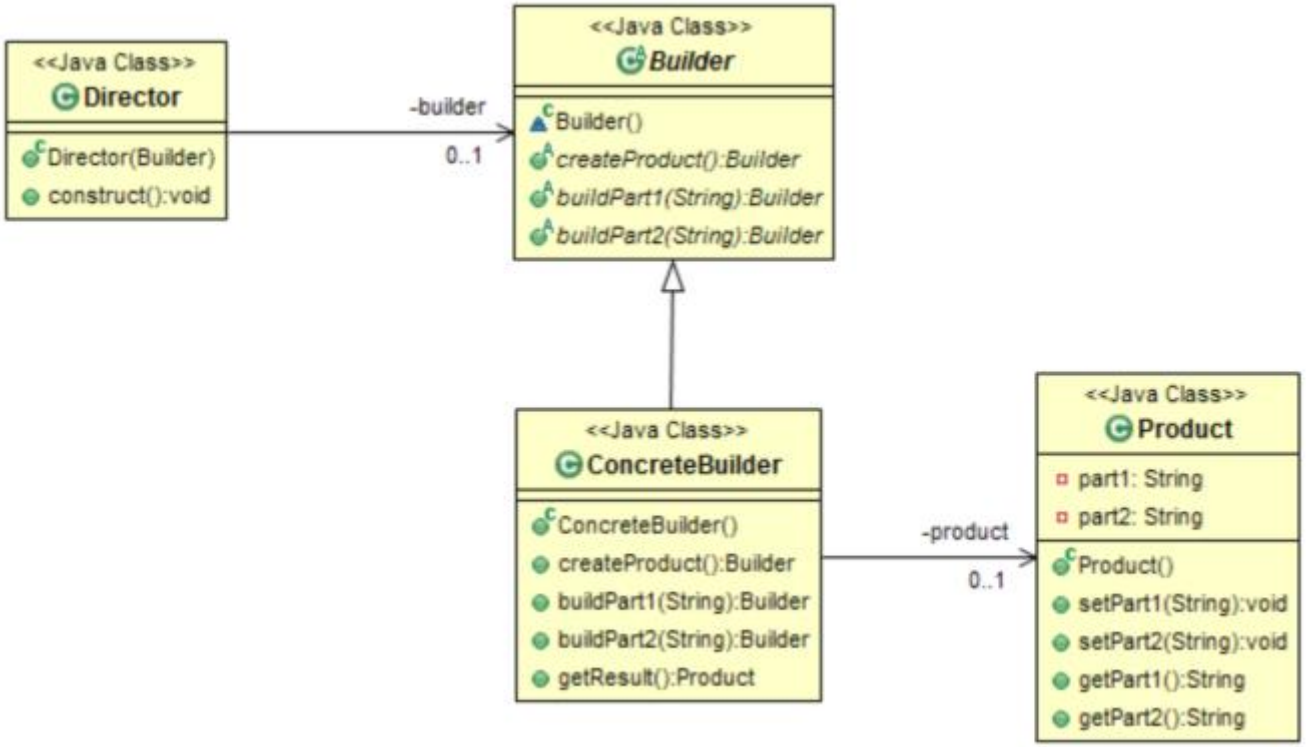


Builder(生成者) Design Pattern

- 宣告靜態巢狀的Builder 類別來建構物件，取代以類別的建構子
  - 類別中每個屬性，Builder 類別中宣告對應屬性及其設定方法
  - 設定方法傳回值為 Builder 物件本身
  - 宣告建立物件方法： build()
- 類別建構子宣告為私有權限
  - 以Builder物件為傳入參數
- 使用步驟
  - 建構Builder物件
  - 呼叫需設定屬性的設定方法
  - 呼叫 build() 方法建立物件



Builder Design Pattern



Person類別

```
import java.util.*;
public class Person2 implements Comparable<Person2>{
    private String name, email;
    private int age;
    private List<String> hobbies;
    private Person2(Builder builder){
        this.name = builder.name;     this.age = builder.age;
        this.email = builder.email;     this.hobbies = builder.hobbies;
    }
    public String getName() {     return name;     }
    public int getAge() {     return age;     }
    public String getEmail() {     return email;     }
    public List<String> getHobbies() {     return hobbies;     }
    public void addHobby(String hobby){   this.hobbies.add(hobby);   }
```

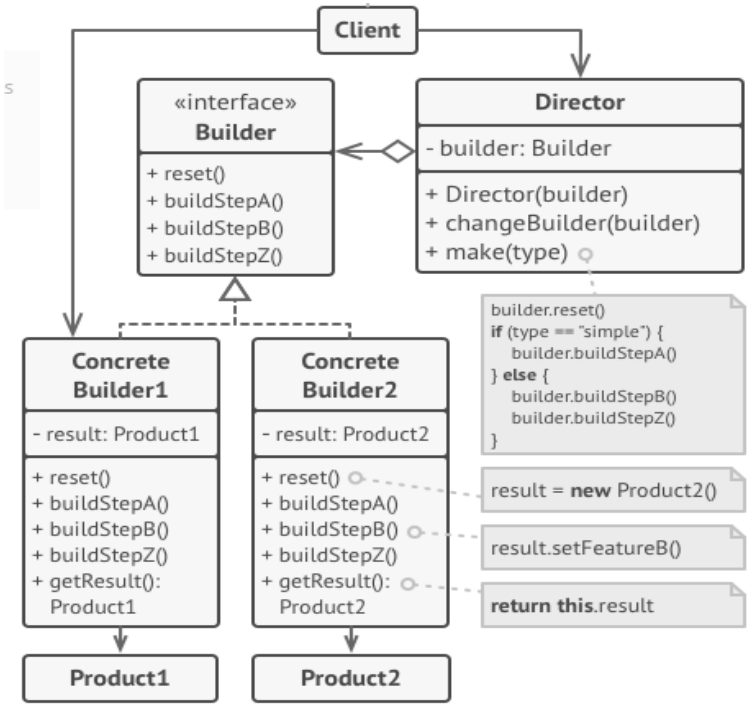
```
    public static class Builder {
        private String name = "", email = "";
        private int age = 0;
        private List<String> hobbies = new ArrayList<>();
        public Builder name(String name) {
            this.name = name;          return this;
        }
        public Builder age(int val) {
            this.age = val;          return this;
        }
        public Builder email(String val) {
            this.email = val;     return this;
        }
        public Builder hobby(String hobby) {
            this.hobbies.add(hobby);     return this;
        }
        public Person2 build() {    return new Person2(this);    }
    }
```

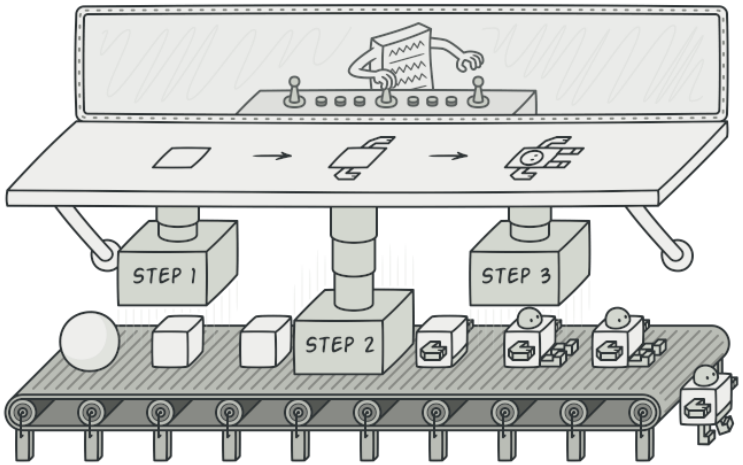
```
    public static List<Person2> createList() {
        List<Person2> people = new ArrayList<>();
        people.add(new Builder().name("Bob").email("bob@gmail.com").age(21).hobby("Piano").hobby("Baseball").hobby("Movie").build());
        people.add(new Builder().name("Jane").email("jane@gmail.com").age(34).hobby("Music").hobby("Movie").hobby("Swim").build());
        people.add(new Builder().name("John").email("john@gmail.com").age(25).hobby("Music").hobby("Baseball").build());
        people.add(new Builder().name("Phil").email("phil@gmail.com").age(65).hobby("Basketball").hobby("Movie").build());
        people.add(new Builder().name("Betty").email("betty@gmail.com").age(55).hobby("Swim").hobby("Piano").hobby("Movie").build());
        return people;
    }
}
```

Builder Design Pattern

- 優點
  - 程式碼更容易理解
    - 設定方法明確表示要設定的屬性
  - 更有彈性的物件建立方式
    - 無資料的選擇性屬性，不呼叫設定方法即可
  - 讓建立物件可以方法鏈式呼叫(method chaining) 的方式進行
    - 程式碼更加 fluent (流暢)

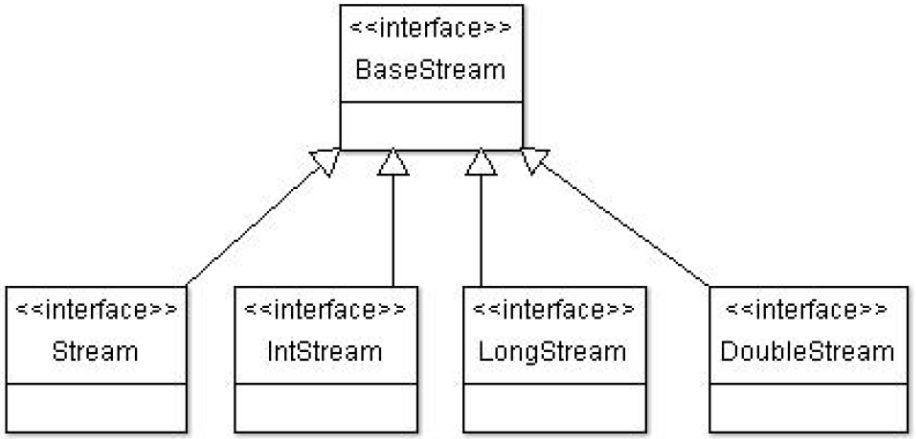
Builder Design Pattern





基本型別串流介面

- 參考型別串流
  - Stream<T>
- 基本型別串流
  - IntStream
  - LongStream
  - DoubleStream
  - 提高效率，避免耗資源的自動拆裝箱操作



建立IntStream串流物件

- java.util.stream.IntStream

方法名稱	傳回值	說明
of(int... values)	IntStream	將指定int陣列元素依序轉換為 Stream 物件
range(int startInclusive, int endExclusive)	IntStream	傳回由startInclusive到endExclusive-1間距為1的IntStream 物件
rangeClosed(int startInclusive, int endInclusive)	IntStream	傳回由startInclusive到endInclusive 間距為1的IntStream 物件

- java.util.Arrays 類別

方法名稱	傳回值	說明
stream(int[] array)	static IntStream	將指定 int 陣列元素依序轉換為 IntStream 物件

IntStream常用方法

- java.util.stream.IntStream

方法名稱	傳回值	說明
sum()	int	傳回串流整數元素的和
average()	OptionalDouble	傳回串流整數元素的平均值的optional容器
summaryStatistics()	IntSummaryStatistics	傳回包含串流元素各項統計數據的IntSummaryStatistics物件可由此物件取得串流的sum/average/count/max/min等資訊
asLongStream()	LongStream	傳回LongStream，將每個int元素轉換為long元素
asDoubleStream()	DoubleStream	傳回DoubleStream，將每個int元素轉換double元素
boxed()	Stream<Integer>	傳回泛型為Integer的Stream，將int元素裝箱為Integer元素
mapToLong(IntToLongFunction mapper)	LongStream	傳回將串流元素執行指定轉換程式mapper後的LongStream
mapToDouble(IntToDoubleFunction mapper)	DoubleStream	傳回將串流元素執行指定轉換程式mapper後的DoubleStream
mapToObj(IntFunction<? extends U> mapper)	<U> Stream<U>	傳回將串流整數元素執行指定轉換程式mapper後的物件串流

建立IntStream串流

```
import java.util.*;
import java.util.stream.*;
public class CreateStreamDemo2 {
	public static void main(String[] args) {
		IntStream i1 = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8);
		IntStream i2 = IntStream.range(1, 9);
		IntStream i3 = IntStream.rangeClosed(1, 8);

		System.out.print("IntStream.of()建構: ");
		i1.forEach(i->System.out.print(i));

		System.out.print("\nIntStream.range()建構: ");
		i2.forEach(i->System.out.print(i));

		System.out.print("\nIntStream.rangeClosed(): ");
		i3.forEach(i->System.out.print(i));

	}
}
```

命令提示字元

```
C:\JavaClass>javac CreateStreamDemo2.java

C:\JavaClass>java CreateStreamDemo2
IntStream.of()建構: 12345678
IntStream.range()建構: 12345678
IntStream.rangeClosed(): 12345678
C:\JavaClass>
```

建立IntStream串流

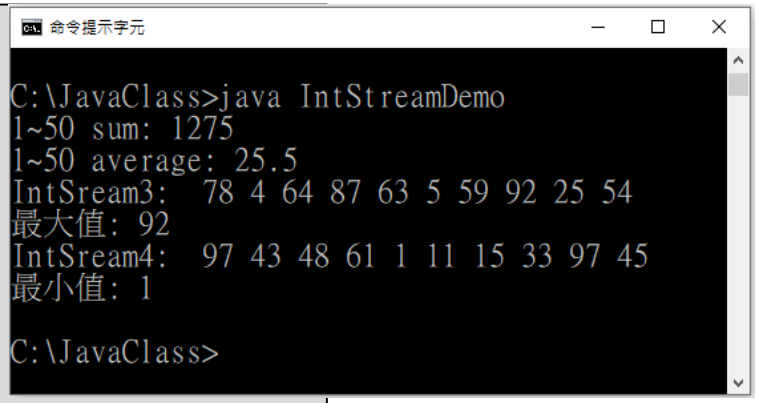
```
import java.util.*;
import java.util.stream.*;
public class CreateStreamDemo3 {
	public static void main(String[] args) {
		IntStream iStream = IntStream.generate(()->new Random().nextInt());
		DoubleStream dStream = DoubleStream.generate(()->new Random().nextDouble());

		System.out.print("\nIntStream.generate()建構: ");
		iStream.limit(6).forEach(i -> System.out.print(" "+i));

		System.out.print("\nDoubleStream.generate()建構: ");
		dStream.limit(3).forEach(d -> System.out.print(" "+d));
	}
}
```

命令提示字元

```
C:\JavaClass>javac CreateStreamDemo3.java

C:\JavaClass>java CreateStreamDemo3

IntStream.generate()建構:  -292655042 -894512267 506230397 -1241827408 -1731894406 1435458766
DoubleStream.generate()建構:  0.7559014221063045 0.19926108943377963 0.04289806830799081
C:\JavaClass>
```

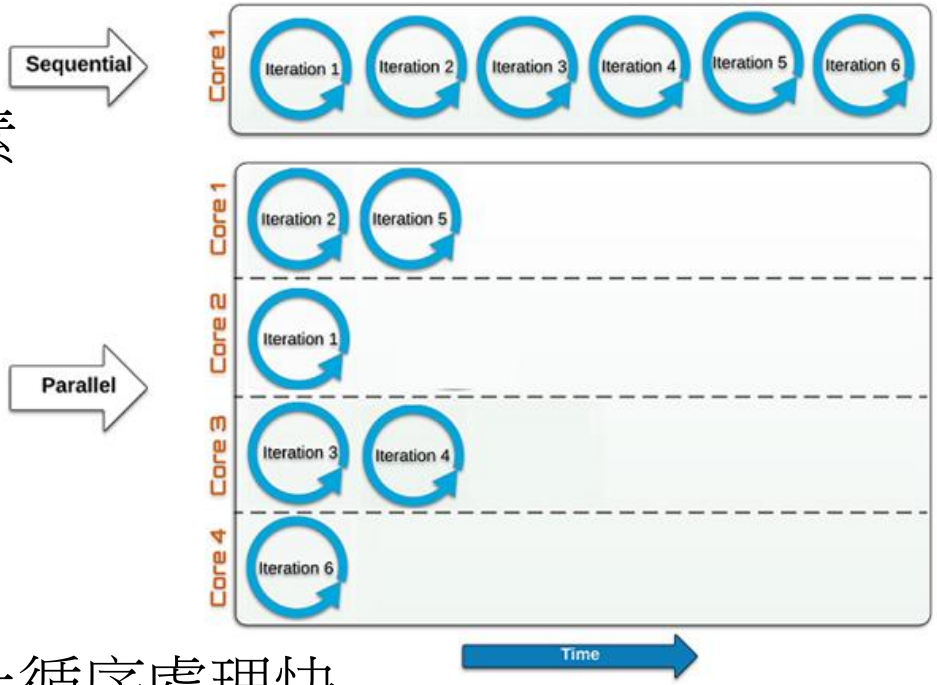
IntStream串流方法

```
import java.util.*;
import java.util.stream.*;
public class IntStreamDemo {
	public static void main(String[] args) {
		IntStream iStream1 = IntStream.rangeClosed(1,50);
		System.out.println("1~50 sum: "+iStream1.sum());

		IntStream iStream2 = IntStream.rangeClosed(1,50);
		OptionalDouble avg = iStream2.average();
		System.out.println("1~50 average: "+avg.orElse(-1));

		IntStream iStream3 = IntStream.generate(()->(int)(Math.random()*100)).limit(10);
		System.out.print("IntSream3: ");
		OptionalInt maxInt = iStream3.peek(i->System.out.print(" "+i)).max();
		System.out.println("\n最大值: "+maxInt.getAsInt());

		IntStream iStream4 = IntStream.generate(()->(int)(Math.random()*100)).limit(10);
		System.out.print("IntSream4: ");
		OptionalInt minInt = iStream4.peek(i->System.out.print(" "+i)).min();
		System.out.println("\n最小值: "+minInt.getAsInt());
	}
}
```



Stream API 並行操作

- Stream API 循序 vs. 並行操作
  - 多核心環境下並行操作可提升效能
    - 使用 Fork/Join 架構
  - 影響並行運算速度的因素
    - CPU 核心數
    - 資料大小
    - 資料結構
    - 資料是否裝箱
    - 處理聚合結果方式及時間
  - 並行處理不一定每次都比循序處理快



建立並行串流物件

- java.util.collection 介面

方法名稱	傳回值	說明
parallelStream()	default Stream<E>	將集合中元素依序轉換為 Parallel Stream 物件

- java.util.stream.BaseStream 介面

方法名稱	傳回值	說明
parallel()	Stream	將串流轉換為 Parallel Stream 物件

循序 vs. 並行串流

```
import java.util.*;
import java.util.stream.*;
public class SequentialParallelDemo {
	public static void main(String[] args) {
		List<String> sList = Arrays.asList("1","2","3","4","5","6","7","8","9",
		long start = System.currentTimeMillis();
		sList.stream().forEach(i -> {
			System.out.println(i + " - Thread:"+Thread.currentThread().getN
			try{
				Thread.sleep(100);
			} catch(Exception ex){ } }
		);
		long end = System.currentTimeMillis();
		System.out.println("Sequential Duration:"+(end-start));

		System.out.println("Processors:"+Runtime.getRuntime().availableProcessors());
		start = System.currentTimeMillis();
		sList.parallelStream().forEach(i -> {
			System.out.println(i + " - Thread:"+Thread.currentThread().getName());
			try{
				Thread.sleep(100);
			} catch(Exception ex){ } }
		);
		end = System.currentTimeMillis();
		System.out.println("Parallel Duration:"+(end-start));
	}
}
```

命令提示字元

```
C:\JavaClass>javac SequentialParallelDemo.java

C:\JavaClass>java SequentialParallelDemo
1 - Thread:main
2 - Thread:main
3 - Thread:main
4 - Thread:main
5 - Thread:main
6 - Thread:main
7 - Thread:main
8 - Thread:main
9 - Thread:main
10 - Thread:main
Sequential Duration:1056
Processors:8
7 - Thread:main
3 - Thread:ForkJoinPool.commonPool-worker-1
9 - Thread:ForkJoinPool.commonPool-worker-2
8 - Thread:ForkJoinPool.commonPool-worker-4
10 - Thread:ForkJoinPool.commonPool-worker-7
1 - Thread:ForkJoinPool.commonPool-worker-6
5 - Thread:ForkJoinPool.commonPool-worker-5
2 - Thread:ForkJoinPool.commonPool-worker-3
6 - Thread:main
4 - Thread:ForkJoinPool.commonPool-worker-1
Parallel Duration:218

C:\JavaClass>
```

循序 vs. 並行串流

```java
import java.util.*;
import java.util.stream.*;
public class SequentialParallelDemo1 {
	public static void main(String[] args) {
		IntStream iStream = IntStream.rangeClosed(1,10000);
		long start = System.currentTimeMillis();
		int sum1 = iStream.filter(i->i%2==0).sum();
		long end = System.currentTimeMillis();
		System.out.println("Sequential Sum:"+(end-start));
		System.out.println("2+...+10000="+sum1);

		IntStream iStream2 = IntStream.rangeClosed(1,10000);
		start = System.currentTimeMillis();
		int sum2 = iStream2.parallel().filter(i->i%2==0).sum();
		end = System.currentTimeMillis();
		System.out.println("Parallel Sum:"+(end-start));
		System.out.println("2+...+10000="+sum2);
	}
}
```

命令提示字元

```
C:\JavaClass>javac SequentialParallelDemo1.java

C:\JavaClass>java SequentialParallelDemo1
Sequential Sum:42
2+...+10000=25005000
Parallel Sum:4
2+...+10000=25005000

C:\JavaClass>
```

循序 vs. 並行串流

```
import java.util.*;
import java.util.stream.*;
public class SequentialParallelDemo2 {
	public static void main(String[] args) {
		long start = System.currentTimeMillis();
		List<String> hobbies = Person.createList().stream()
						.map(p -> p.getHobbies())
						.flatMap(l->l.stream())
						.distinct()
						.collect(Collectors.toList());
		long end = System.currentTimeMillis();
		System.out.println("Sequential getHobbies:"+(end-start));
		System.out.println("Hobbies: "+hobbies);

		start = System.currentTimeMillis();
		hobbies = Person.createList().parallelStream()
						.map(p -> p.getHobbies())
						.flatMap(l->l.stream())
						.distinct()
						.collect(Collectors.toList());
		end = System.currentTimeMillis();
		System.out.println("Parallel getHobbies:"+(end-start));
		System.out.println("Hobbies: "+hobbies);
	}
}
```

命令提示字元

```
C:\JavaClass>javac SequentialParallelDemo2.java

C:\JavaClass>java SequentialParallelDemo2
Sequential getHobbies:51
Hobbies: [Piano, Baseball, Movie, Music, Swimming, Basketball]
Parallel getHobbies:9
Hobbies: [Piano, Baseball, Movie, Music, Swimming, Basketball]

C:\JavaClass>
```

不適用並行串流狀況

- 並行處理不適用於處理
  - 有執行緒同步問題的物件
    - 每次處理結果可能不同
  - Stateful (有狀態) 的參數利用並行串流效能不會比循序串流好