Working with Streams and Lambda expressions

1. Streams

1.1. OPTIONAL

Constructores:

```
Optional.empty();
Optional.of(value);
Optional.ofNullable(value);
```

Uso:

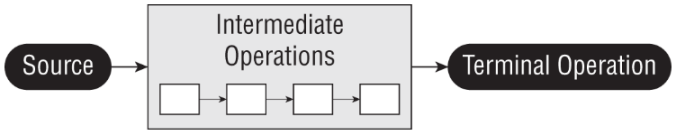
```
# Old version
Optional o = (value== null) ? Optional.empty(): Optional.of(value);

# Using Optional
Optional o = Optional.ofNullable(value);
```

Otros metodos:

Method	When Optional is empty	When Optional contains a value
`get()`	Throws an exception	Returns value
`ifPresent(Consumer c)`	Does nothing	Calls Consumer with value
`isPresent()`	Returns false	Returns true
`orElse(T other)`	Returns other parameter	Returns value
`orElseGet(Supplier s)`	Returns result of calling Supplier	Returns value
`orElseThrow()`	Throws `NoSuchElementException`	Returns value
`orElseThrow(Supplier s)`	Throws exception created by calling Supplier	Returns value

1.2. STREAMS



Creando una Fuente:

Method	Finite or infinite?	Notes
`Stream.empty()`	Finite	Creates Stream with zero elements
`Stream.of(varargs)`	Finite	Creates Stream with elements listed
`coll.stream()`	Finite	Creates Stream from a Collection

`coll.parallelStream()`	Finite	Creates Stream from a Collection where the stream can run in parallel
`Stream.generate(supplier)`	Infinite	Creates Stream by calling the Supplier for each element upon request
`Stream.iterate(seed, unaryOperator)`	Infinite	Creates Stream by using the seed for the first element and then calling the UnaryOperator for each subsequent element upon request
`Stream.iterate(seed, predicate, unaryOperator)`	Finite or infinite	Creates Stream by using the seed for the first element and then calling the UnaryOperator for each subsequent element upon request. Stops if the Predicate returns false

Terminal operators:

Method	What happens for infinite streams	Return value	Reduction
`count()`	Does not terminate	`long`	Yes
`min()  max()`	Does not terminate	`Optional<T>`	Yes
`findAny()  findFirst()`	Terminates	`Optional<T>`	No
`allMatch()  anyMatch()  noneMatch()`	Sometimes terminates	`boolean`	No
`forEach()`	Does not terminate	`void`	No
`reduce()`	Does not terminate	Varies	Yes
`collect()`	Does not terminate	Varies	Yes

Common Itermedia operators

Method Signature	Notes
`Stream<T> filter(Predicate<? super T> predicate)`	
`Stream<T> distinct()`	
`Stream<T> limit(long maxSize)`	
`Stream<T> skip(long n)`	
`<R> Stream<R> map(Function<? super T, ? extends R> mapper)`	
`<R> Stream<R> flatMap(` `  Function<? super T, ? extends Stream<? extends R>> mapper)`	
`Stream<T> sorted()`	
`Stream<T> sorted(Comparator<? super T> comparator)`	
`Stream<T> peek(Consumer<? super T> action)`	

Common primitive stream methods

Method	Primitive stream	Description

`OptionalDouble average()`	`IntStream`
	`LongStream`
	`DoubleStream`
		The arithmetic mean of the elements

`Stream<T> boxed()`	`IntStream`
	`LongStream`
	`DoubleStream`
		A Stream<T> where T is the wrapper class associated with the primitive value

`OptionalInt max()`	`IntStream`	The maximum element of the stream
`OptionalLong max()`	`LongStream`	
`OptionalDouble max()`	`DoubleStream`	
`OptionalInt min()`	`IntStream`	The minimum element of the stream
`OptionalLong min()`	`LongStream`	
`OptionalDouble min()`	`DoubleStream`	
`IntStream range(int a, int b)`	`IntStream`	Returns a primitive stream from a (inclusive) to b (exclusive)
`LongStream range(long a, long b)`	`LongStream`	
`IntStream rangeClosed(int a, int b)`	`IntStream`	Returns a primitive stream from a (inclusive) to b (inclusive)
`LongStream rangeClosed(long a, long b)`	`LongStream`	
`int sum()`	`IntStream`	Returns the sum of the elements in the stream
`long sum()`	`LongStream`	
`double sum()`	`DoubleStream`	
`IntSummaryStatistics summaryStatistics()`	`IntStream`	Returns an object containing numerous stream statistics such as the average, min, max, etc.
`LongSummaryStatistics summaryStatistics()`	`LongStream`	
`DoubleSummaryStatistics summaryStatistics()`	`DoubleStream`	

Mapping methods between types of streams

Source stream class	To create Stream	To create DoubleStream	To create IntStream	To create LongStream
`Stream<T>`	`map()`	`mapToDouble()`	`mapToInt()`	`mapToLong()`
`DoubleStream`	`mapToObj()`	`map()`	`mapToInt()`	`mapToLong()`
`IntStream`	`mapToObj()`	`mapToDouble()`	`map()`	`mapToLong()`
`LongStream`	`mapToObj()`	`mapToDouble()`	`mapToInt()`	`map()`

Function parameters when mapping between types of streams:

Source stream class	To create Stream	To create DoubleStream	To create IntStream	To create LongStream
`Stream<T>`	`Function<T,R>`	`ToDoubleFunction<T>`	`ToIntFunction<T>`	`ToLongFunction<T>`
`DoubleStream`	`Double Function<R>`	`DoubleUnary Operator`	`DoubleToInt Function`	`DoubleToLong Function`
`IntStream`	`IntFunction<R>`	`IntToDouble Function`	`IntUnary Operator`	`IntToLong Function`
`LongStream`	`Long Function<R>`	`LongToDouble Function`	`LongToInt Function`	`LongUnary Operator`

Optional types for primitives:

	OptionalDouble	OptionalInt	OptionalLong
Getting as a primitive	`getAsDouble()`	`getAsInt()`	`getAsLong()`

`orElseGet()` parameter type	`DoubleSupplier`	`IntSupplier`	`LongSupplier`
Return type of `max()` and min()	`OptionalDouble`	`OptionalInt`	`OptionalLong`
Return type of `sum()`	`double`	`int`	`long`
Return type of `average()`	`OptionalDouble`	`OptionalDouble`	`OptionalDouble`

Common functional interfaces for primitives:

Functional interfaces	# parameters	Return type	Single abstract method

`DoubleSupplier`
`IntSupplier`
`LongSupplier`
	0	`double`
		`int`
		`long`
			`getAsDouble`
			`getAsInt`
			`getAsLong`

`DoubleConsumer`
`IntConsumer`
`LongConsumer`
	`1 (double)`
	`1 (int)`
	`1 (long)`
		`void`	`accept`

`DoublePredicate`
`IntPredicate`
`LongPredicate`
	`1 (double)`
	`1 (int)`
	`1 (long)`
		`boolean`	`test`

`DoubleFunction<R>`
`IntFunction<R>`
`LongFunction<R>`
	`1 (double)`
	`1 (int)`
	`1 (long)`
		`R`	`apply`

`DoubleUnaryOperator`
`IntUnaryOperator`
`LongUnaryOperator`
	`1 (double)`
	`1 (int)`
	`1 (long)`
		`double`
		`int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

`DoubleBinaryOperator`
`IntBinaryOperator`
`LongBinaryOperator`
	`2 (double, double)`
	`2 (int, int)`
	`2 (long, long)`
		`double`
		`int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

Primitive-specific functional interfaces:

Functional interfaces	# parameters	Return type	Single abstract method

`ToDoubleFunction<T>`
`ToIntFunction<T>`
`ToLongFunction<T>`
	`1 (T)`	`double`
		`Int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

`ToDoubleBiFunction<T, U>`
`ToIntBiFunction<T, U>`
`ToLongBiFunction<T, U>`
	`2 (T, U)`	`double`
		`Int`
		`long`
			`applyAsDouble`
			`applyAsInt`
			`applyAsLong`

`DoubleToIntFunction`
`DoubleToLongFunction`
`IntToDoubleFunction`
`IntToLongFunction`
`LongToDoubleFunction`
`LongToIntFunction`
	`1 (double)`
	`1 (double)`
	`1 (int)`
	`1 (int)`
	`1 (long) 1 (long)`
		`Int`
		`long`
		`double`
		`long`
		`double`
		`int`
			`applyAsInt`
			`applyAsLong`
			`applyAsDouble`
			`applyAsLong`
			`applyAsDouble`
			`applyAsInt`

`ObjDoubleConsumer<T>`
`ObjIntConsumer<T>`
`ObjLongConsumer<T>`
	`2 (T, double)`
	`2 (T, int)`
	`2 (T, long)`
		`void`	`accept`

2. Functional Interfaces

Functional interface	Return type	Method name	# of parameters	Using
`Supplier<T>`	`T`	`get()`	0	Generar valores sin ninguna entrada
`Consumer<T>`	`void`	`accept(T)`	1 (T)	Hacer algo con el parámetro
`BiConsumer<T, U>`	`void`	`accept(T,U)`	2 (T, U)	Hacer algo con el parámetro
`Predicate<T>`	`boolean`	`test(T)`	1 (T)	Usalmente para hacer algun filtro o matching
`BiPredicate<T, U>`	`boolean`	`test(T,U)`	2 (T, U)	Usalmente para hacer algun filtro o matching
`Function<T, R>`	`R`	`apply(T)`	1 (T)	Convertir un parámetro a un tipo diferente y retornarlo
`BiFunction<T, U, R>`	`R`	`apply(T,U)`	2 (T, U)	Convertir un parámetro a un tipo diferente y retornarlo
`UnaryOperator<T>`	`T`	`apply(T)`	1 (T)	Transforma un valor a otro del mismo tipo
`BinaryOperator<T>`	`T`	`apply(T,T)`	2 (T, T)	Transforma/Mezcla dos valores a otro del mismo tipo

Métodos convenientes

Interface instance	Method return type	Method name	Method parameters	Using
`Consumer`	`Consumer`	`andThen()`	`Consumer`	Ejecuta dos funciones en secuencia
`Function`	`Function`	`andThen()`	`Function`	Aplica las funciones en secuencia
`Function`	`Function`	`compose()`	`Function`	Ejecuta en secuencia, pero primero la función parámetro
`Predicate`	`Predicate`	`and()`	`Predicate`	Encadena AND al resultado
`Predicate`	`Predicate`	`negate()`	–	Niega el resultado
`Predicate`	`Predicate`	`or()`	`Predicate`	Encadena OR al resultado

Ejemplo:

```
//Predicate
Predicate<String> brownEggs =
   s -> s.contains("egg") && s.contains("brown");
Predicate<String> otherEggs =
   s -> s.contains("egg") && ! s.contains("brown");

Predicate<String> brownEggs = egg.and(brown);
Predicate<String> otherEggs = egg.and(brown.negate());

//Consumer
Consumer<String> c1 = x -> System.out.print("1: " + x);
Consumer<String> c2 = x -> System.out.print(",2: " + x);

Consumer<String> combined = c1.andThen(c2);
combined.accept("Annie");              // 1: Annie,2: Annie

//Function
Function<Integer, Integer> before = x -> x + 1;
```

```
Function<Integer, Integer> after = x -> x * 2;

Function<Integer, Integer> combined = after.compose(before);
System.out.println(combined.apply(3));   // 8
```