Working with Streams and Lambda expressions

1. Functional Interfaces

1.1. 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)	Usualmente para hacer algún filtro o matching
`BiPredicate<T, U>`	`boolean`	`test(T,U)`	2 (T, U)	Usualmente para hacer algún filtro o matching
`Function<T, R>`	`R`	`apply(T)`	1 (T)	Similar a una función de matemática con un parámetro. Convertir/Recibir un parámetro de un tipo y retorna otro tipo
`BiFunction<T, U, R>`	`R`	`apply(T,U)`	2 (T, U)	Similar a una función de matemática con un parámetro. Convertir/Recibir dos parámetros de tipos distintos y retorna otro tipo
`UnaryOperator<T>`	`T`	`apply(T)`	1 (T)	Caso especial de `Function`, los parámetros y retorno son del mismo tipo. Transforma un valor a otro del mismo tipo
`BinaryOperator<T>`	`T`	`apply(T,T)`	2 (T, T)	Caso especial de `Function`, los parámetros y retorno. Transforma/Mezcla dos valores a otro del mismo tipo

1.2. MÉTODOS IMPORTANTES

Métodos convenientes sobre las interfaces funcionales. Estos son métodos `default`.

Interface instance	Firma del Método	Using
`Consumer`	`Consumer<T> andThen(Consumer<? super T> after)`	Ejecuta dos funciones en secuencia

Function	<V> Function<T,V> andThen(Function<? super R,? extends V> after)	Aplica las funciones en secuencia
Function	<V> Function<V,R> compose(Function<? super V,? extends T> before)	Ejecuta en secuencia, pero primero la función parámetro
Predicate	Predicate<T> and(Predicate<? super T> other)	Encadena AND al resultado
Predicate	Predicate<T> negate()	Niega el resultado
Predicate	Predicate<T> or(Predicate<? super T> other)	Encadena OR al resultado

Ejemplo:

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

2. Optional

2.1. FACTORIAS

Método Factoría de métodos estáticos:

Métodos estáticos de Factoría	Notas
<T> Optional<T> empty()	Crea un `Optional` vacío
<T> Optional<T> of(T value)	Crea a partir de un Objeto
<T> Optional<T> ofNullable(T value)	Si `value` es `null`, crea un `Optional` vacío, sino crea a partir de `value`.

Uso:

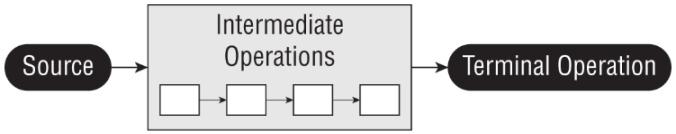
```
# Old version
Optional o = (value== null) ? Optional.empty(): Optional.of(value);

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

2.2. MÉTODOS IMPORTANTES

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

3. Streams

Tener en cuenta que el `Stream` usa *lazy operations*. Las operaciones se realizan cuando un terminador aparece. Esto es importante para reconocer donde salta una excepción.



3.1. CREANDO 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`

3.2. TERMINAL OPERATORS

Method	What happens for infinite streams	Reduction	Notes
`long count()`	No Termina	Yes	
`Optional<T> min(Comparator<? super T> comparator)` `Optional<T> max(Comparator<? super T> comparator)`	No Termina	Yes	
`Optional<T> findAny()` `Optional<T> findFirst()`	Termina	No	Método `findFirst` retorna el primero, pero `findAny` cualquiera, y cuando se usa en `parallel`, no es predecible.

`boolean anyMatch(Predicate <? super T> predicate)`
`boolean allMatch(Predicate <? super T> predicate)`
`boolean noneMatch(Predicate <? super T> predicate)`
	Depende	No	Para:
			`anymatch`, termina si para alguno true.
			`allMatch`, si una false, termina, sino continua
			`noneMatch`, si uno true, termina, sino continua.

`void forEach(Consumer<? super T> action)`	No Termina	No	Sólo consume.

`T reduce(T identity, BinaryOperator<T> accumulator)`
`Optional<T> reduce(BinaryOperator<T> accumulator)`
`<U> U reduce(U identity,`
`BiFunction<U,? super T,U> accumulator,`
`BinaryOperator<U> combiner)`
	No Termina	Yes	Parámetro `identity` es el valor inicial a usar por `accumulator`.
			Si no usamos el valor inicial, se devuelve un `Optional`.
			Si lo ejecutamos en un `parallel`, Se usa el `combiner` para que los resultados paralelos se combinen, sino es como si no se usara.

`<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`	No Termina	Yes	Método especial de `reduction`: `mutable reduction`. Utiliza un objeto mutable donde acumular. El `supplier` devuelve el objeto mutable, el `accumulator` usa el `supplier` y acumula. El `combiner` es para el uso con `parallel`. Finalmente devuelve el objeto mutable.
`<R,A> R collect(Collector<? super T, A,R> collector)`	No Termina	Yes	Este usa un tipo `Collector` que se puede crear con `Collectors`.

3.3. COMMON INTERMEDIA OPERATORS

Todos los métodos devuelven un Stream:

Method Signature	Notes
`Stream<T> filter(Predicate<? super T> predicate)`	Filtra los elementos para los cuales el `predicate` resulta `true`.
`Stream<T> distinct()`	Retorna un `stream` sin elementos repetidos. Usa por debajo el `equals`.
`Stream<T> limit(long maxSize)`	Solo siguen los primeros `maxSize` elementos.
`Stream<T> skip(long n)`	Salta los primeros `n` elementos.
`<R> Stream<R> map(Function<? super T, ? extends R> mapper)`	Se suele utilizar para transformar datos. De ahí el uso de `Function`.
`<R> Stream<R> flatMap(Function <? super T, ? extends Stream<? extends R>> mapper)`	La `Function mapper` devuelve un `Stream` que al final se reúne con todos los `Stream`'s, eliminando los elementos vacíos. El `Stream` del cual parte puede ser algo más complejo, convirtiéndolo en un único `Stream`. Ver ejemplo abajo.
`Stream<T> sorted()`	Ordena según orden natural. Los elementos deben implementar `Comparable`, sino excepción al usar un terminador.
`Stream<T> sorted(Comparator<? super T> comparator)`	Usa un `Comparator`.
`Stream<T> peek(Consumer<? super T> action)`	Se espera que no cambia el `Stream`, usual para debug.

Ejemplos:

```
List<String> zero = List.of();
var one = List.of("Bonobo");
var two = List.of("Mama Gorilla", "Baby Gorilla");
Stream<List<String>> animals = Stream.of(zero, one, two);
animals.flatMap(m -> m.stream())
   .forEach(System.out::println);
//salida
Bonobo
Mama Gorilla
Baby Gorilla
```

3.4. COMMON PRIMITIVE STREAM METHODS

Los primitive Stream son 3: `IntStream`, `LongStream`, `DoubleStream`. `Stream` tiene métodos para convertirlos:

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> arg0)
IntStream mapToInt(ToIntFunction<? super T> arg0)
LongStream mapToLong(ToLongFunction<? super T> arg0)
```

Los *Primitive Stream* tiene similares funciones terminadores e Intermedias. Las interfaces funcionales tienen prefijos según el tipo primitivo. De forma similar para los `Optional` con su prefijo del tipo de primitivo.

Method	Primitive stream	Description

`OptionalDouble average()`	`IntStream`
	`LongStream`
	`DoubleStream`
		Promedio de los elementos.

`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`	

3.5. 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()`

3.6. 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`

3.7. 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`

3.8. 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`

3.9. 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