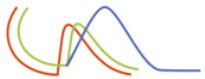
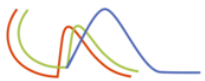
El paquete java.util.function en Java

Programación funcional moderna en Java



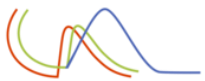
Contexto y fundamentos

- Evolución hacia la programación funcional en Java
- Qué es una interfaz funcional
- Sintaxis de lambdas
- Ejemplo: comparación Java 7 vs Java 8



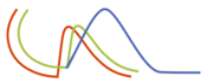
Principales interfaces funcionales

Nombre	**Función**	**Método a llamar**
Predicate	Evalúa una condición	test
Consumer	Usa un valor, no devuelve nada	accept
Supplier	Genera un valor sin recibir nada	get
Function<T,R>	Transforma un valor en otro	apply
UnaryOperator/BinaryOperator	Versiones de Function del mismo tipo	apply



Composición y aplicación práctica

- Encadenamiento de funciones: andThen, compose, negate
- Uso conceptual dentro de Stream API
- Beneficios: reusabilidad, expresividad, claridad
- Cierre conceptual: impacto de java.util.function

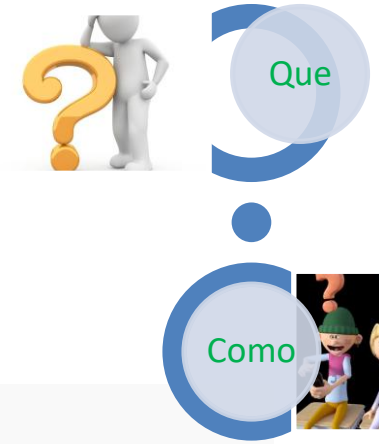


De las lambdas a los Streams

- La Stream API usa las interfaces del paquete java.util.function.
- Cada operación (map, filter, forEach, reduce) recibe una función como parámetro.
- Las lambdas permiten escribir esas funciones de forma concisa.
- Programación declarativa: decimos qué queremos hacer, no cómo.
- Juntas, las lambdas y los Streams permiten un estilo funcional en Java.

```
lista.stream()
     .filter(x -> x > 10)
     .map(x -> x * 2)
     .forEach(System.out::println);
```

Contexto y fundamentos



- Evolución hacia la programación funcional en Java

```
// Java 7
Collections.sort(lista, new Comparator<String>() {
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
});

// Java 8+
lista.sort((a, b) -> a.length() - b.length());
```

Sintaxis

Parámetros opcionales. Entre paréntesis y separados por comas

Operador flecha para separar los parámetros del cuerpo

Cuerpo. Una expresión (expresión lambda) o un bloque (bloque lambda)

```
// Java 8+
lista.sort((a, b) -> a.length() - b.length());
```

Reglas básicas de su sintaxis

1. Inferencia de tipos
2. Paréntesis opcionales
3. Valor de retorno
4. Bloques con {}
5. Uso de variables externas – final
6. Atributos de clase

```
n -> n * n;
(n) -> n * n; // también válido
```

```
x -> x + 1;           // return implícito
x -> { return x + 1; } // return explícito
```

```
(a, b) -> a + b;          // correcto, tipos inferidos
(int a, int b) -> a + b; // correcto, tipos explícitos
(int a, b) -> a + b;     // error: tipos mezclados
```

Encadenamiento

- Ejemplo 1 - andThen

```
Function<Integer, Integer> duplicar = x -> x * 2;
Function<Integer, Integer> sumarTres = x -> x + 3;
Function<Integer, Integer> combinada = duplicar.andThen(sumarTres);

System.out.println(combinada.apply(5)); // (5*2) + 3 = 13
```

- Ejemplo 2 – Predicate – negate, and or

```
Predicate<Integer> esPar = n -> n % 2 == 0;
Predicate<Integer> mayorQueDiez = n -> n > 10;
Predicate<Integer> esParYMayorQueDiez = esPar.and(mayorQueDiez);
```

Encadenamiento

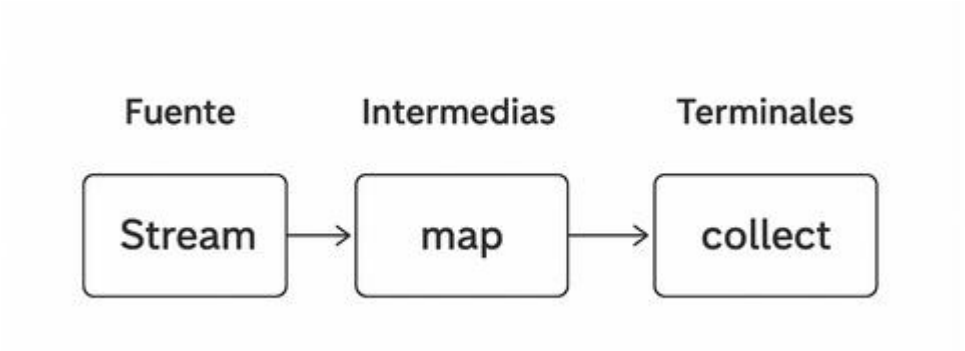
- Ejemplo 1 – filter, map foreach

```
lista.stream()
     .filter(esPar)
     .map(duplicar)
     .forEach(System.out::println);
```

¿Qué es un Stream en Java?

- El principal tipo en este paquete es la interfaz Stream<T>.También existen IntStream, LongStream y DoubleStream para tipos primitivos.
- Un Stream es una secuencia de elementos.Los Streams permiten segmentar y encadenar operaciones (pipelining), lo que facilita el procesamiento paralelo.
- El origen más común de un Stream son los objetos de tipo Collection (List, Set, Map),pero también pueden crearse de manera independiente.

Secuencia de operaciones en un Stream



- Las fuentes no transforman datos, únicamente los proveen al pipeline
- Intermedias: transforman o filtran los elementos sin consumirlos todavía
  - Filter(),Map(), Distinct(), Sorted(), Limit(),
  - Son lazy, se ejecutan hasa que llegue una operación terminal
- Terminales: Son las que consume el Stream y producen un valor final
  - forEach(), reduce(), collect(), count(), sum(), FindFirst(), anyMatch()
  - Una vez ejecutada, el stream se cierra y no puede reutilizarse

Conclusión

- java.util.function proporciona interfaces funcionales listas para usar
- Facilita pasar comportamiento como parámetro
- Base del enfoque declarativo en Java moderno
- Clave para trabajar con Streams, APIs reactivas y programación funcional