Java Functionality

Programación Funcional

@javacf

Programación Funcional

- La programación funcional es un paradigma de programación donde el enfoque está en ***"qué resolver"*** en contraste con un paradigma **procedimental** donde el enfoque es *"cómo resolver".*

Que?

Functional

Declarative

Cómo?

POO

Imperative

- Las funciones de alto orden dependen de la existencia de funciones de primera clase.
- Son funciones que:

  Tomar una función como argumento
  Devuelven una función

El operador lambda, `->` separa el parámetro del cuerpo.

Ej. A esta función (lambda) se le pasa un valor, que se multiplica por dos y luego se devuelve 4, de la siguiente manera:

```
x -> 2 * x
```

- En la siguiente expresión lambda, se pasa un argumento y no se devuelve nada:

```
x->Sistema.fuera.println(x)
```

- Uso

```
Integer arr[] = {1,2,3,4,5};
List<Integer> list = Arrays.asList(arr);
list.forEach(x->System.out.println(x));
```

¿Qué son Java Lambdas?

- Una **expresión lambda** es una sola línea o bloque de código Java que contiene cero o más parámetros y puede devolver un valor. Desde un punto de vista simplificado, una lambda es como **un método anónimo que no pertenece a ningún objeto**:

() -> System.out.println("Hello, lambda!")

```
(<parámetros>) -> { <cuerpo> };
```

- Flecha: La ->(flecha) separa los parámetros del cuerpo lambda. es el equivalente a un cálculo lambda.

Equivalencia

```
// FUNCTIONAL INTERFACE (implicit)

interface HelloWorld {
  String sayHello(String name);
}


// AS ANONYMOUS CLASS

var helloWorld = new HelloWorld() {

  @Override
  public String sayHello(String name) {
    return "hello, " + name + "!";
  }
};

// AS LAMBDA

HelloWorld helloWorldLambda = name -> "hello, " + name + "!";
```

- En **programación funcional una función es también un tipo de dato** lo que ahora permite en programación funcional se pueda pasar como parámetro y retornar a otras funciones.

```java
public static Function<String,String> functionExample() {
    System.out.println("Functional Demo");
    //.........Type,Result
    Function<String, String> function = new Function<String, String>() {

        @Override
        public String apply(String msg) {
            return msg + " @@@ ";
        }
    };
    return function; // retorna una función
}
```

```java
Function funtionExample =Function1.functionExample();
System.out.println(funtionExample.apply( t: "Hello"));
```

Lambda es una función la cual no tiene un nombre(función anónima)

```java
public class Function2 {
    1 usage
    public static Function<String,Integer> functionLambdaLength() {
        return x -> x.length();
    }
}
```

Ejecutando:

```java
Function functionLambda=Function2.functionLambdaLength();
System.out.println(functionLambda.apply( t: "Hello Word"));
```

Predicate es un tipo de función que trabaja sobre un tipo y **devuelve un boolean**. **Su función principal es comprobar si una condición es valida o no.**

```java
public class Alumno {
    4 usages
    public String name;
    5 usages
    public Integer score;
    1 usage
    public Alumno(String name, Integer score) {
        super();
        this.name = name;
        this.score = score;
    }
```

```java
public class Main2 {
    no usages
    public static void main(String[] args) {
        //Example Predicate
        Alumno alumno = new Alumno( name: "Juan",  score: 18);
        Predicate<Alumno> predicate = x -> x.score >=13;
        System.out.println("Alumno approved: " + predicate.test(alumno));
    }
}
```

Un **Consumer** es una expresión que a**cepta un solo valor y no devuelve ninguno**. Su método mas importante es **accept**. U**n uso de Consumer es realizar operaciones sobre un tipo de datos** por ejemplo de un listado guardar su contenido en una base de datos.

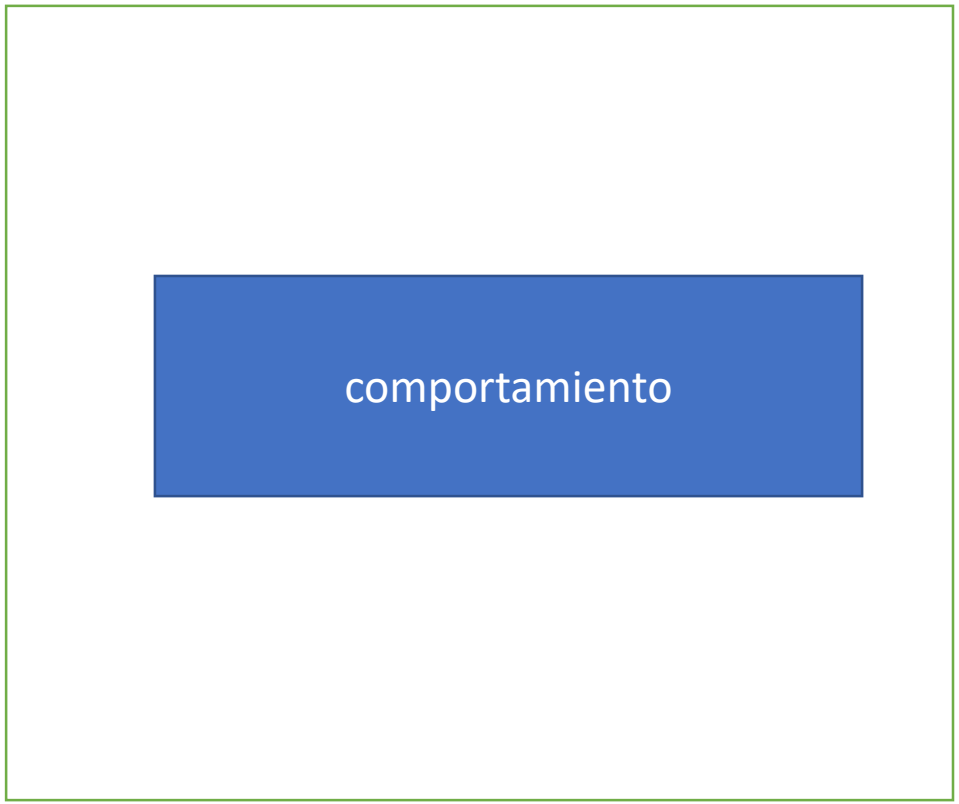
```java
public static void main(String[] args) {
    Alumno alumno = new Alumno ( name: "Juan Consumer",  score: 9);
    Consumer<Alumno> studentConsumer = oneStudent -> System.out.println(oneStudent.toString());
    studentConsumer.accept(alumno);
}
```

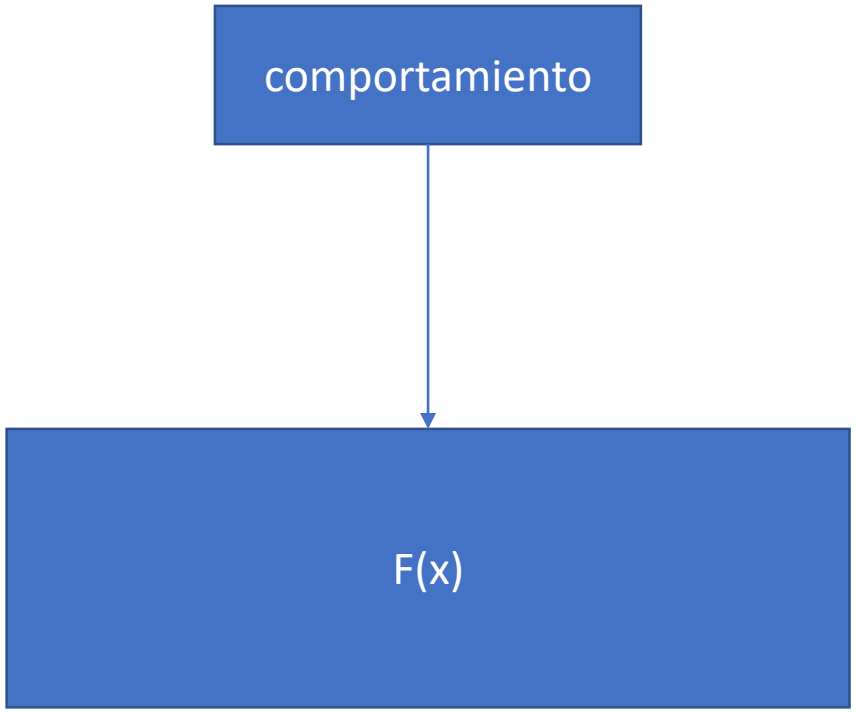
Java nos provee de interfaces con las cuales podemos escribir nuestras propias funciones como pueden ser Predicate, Consumer, Supplier, etc. Pero **también podemos definir nuestra propia estructura de funciones ocupando las interfaces funcionales.** Una interfaz funcional utiliza la anotación **@FunctionalInterface** y puede tener solo un método abstracto es decir que no tiene código implementado.

```java
@FunctionalInterface
public interface ADMInterface2<S, U, P> {
    1 usage
    public Double getScore(S s1, U s2); //abstracto
    1 usage
    public default boolean isApproved(S s1, U s2) {
        if(getScore(s1, s2)>=6) {
            return true;
        }
        return false;
    }
```

```java
    no usages
    public static void main(String[] args) {
        ADMInterface2<Double, Double, Double> finalScoreMath = (s1,s2) -> s1*.4 + s2*.6;
        System.out.println(finalScoreMath.isApproved(5.5, 7.0));
    }
```

Resumen





Predicate Implementando Interface

```java
class BiggerThanFivePredicate<E> implements Predicate<Integer> {

    @Override
    public boolean test(Integer v) {

        Integer five = 5;

        return v > five;
    }
}
```

```java
public static void main(String[] args) {
    List<Integer> nums = List.of(2, 3, 1, 5, 6, 7, 8, 9, 12);
    BiggerThanFivePredicate<Integer> btf = new BiggerThanFivePredicate<>();
    nums.stream().filter(btf).forEach(System.out::println);
}
```

Predicate simplificada usando Lambda

```java
public class Main2WithLambda {
    public static void main(String[] args) {
        List<Integer> nums = List.of(2, 3, 1, 5, 6, 7, 8, 9, 12);
        Predicate<Integer> btf = n -> n > 5; // forma simplificada de crear un predicado
        nums.stream().filter(btf).forEach(System.out::println);
    }
}
```

Procesamiento de Datos

- Casi cualquier programa tiene que lidiar con el procesamiento de datos, muy probablemente en forma de colecciones.
- Un enfoque ***imperativo*** usa bucles para iterar sobre los elementos, trabajando con cada elemento en secuencia. Sin embargo, los **lenguajes funcionales** prefieren un enfoque ***declarativo*** y, a veces, ni siquiera tienen una declaración de bucle clásica, para empezar.

Streams

La *API* ***Streams*** , introducida en Java 8, proporciona un enfoque totalmente declarativo y evaluado de forma *perezosa* para el procesamiento de datos que se beneficia de las adiciones funcionales de Java al utilizar funciones de orden superior para la mayoría de sus operaciones.

La clase **Stream** es una especie de lista que puede tener elementos y se puede iterar, la diferencia entre colecciones y Stream es que Stream es autoiterable. Imaginemos a Stream como un flujo de datos moviéndose sin esperar que alguien los mueva.

```java
public static Stream<String>createStream(){
    Stream<String> stringNumbers =  Stream.of( ...values: "1","2","3","4","5","6","7","8","9","10");
    return stringNumbers;
}
```

no usages

```java
public static void printStream(Stream<String> stringNumbers) { stringNumbers.forEach(System.out::println); }
```

1 usage

```java
public static void streamMap(Stream<String> stringNumbers) {
    Stream<Integer> integerStream = stringNumbers.map(number->Integer.parseInt(number)*10);
    integerStream.forEach(System.out::println);//integerStream.forEach(s -> System.out.println(s));
}
```

1 usage

```java
public static void streamFilter(Stream<String> stringNumbers) {
    stringNumbers.map(number->Integer.parseInt(number)*10).filter(x-> x>= 60).forEach(System.out::println);
}
```

- **Recuerda que la programación funcional se trata de ocuparnos mas en que se va a realizar mas que en como realizarlo.**

Iteración Externa

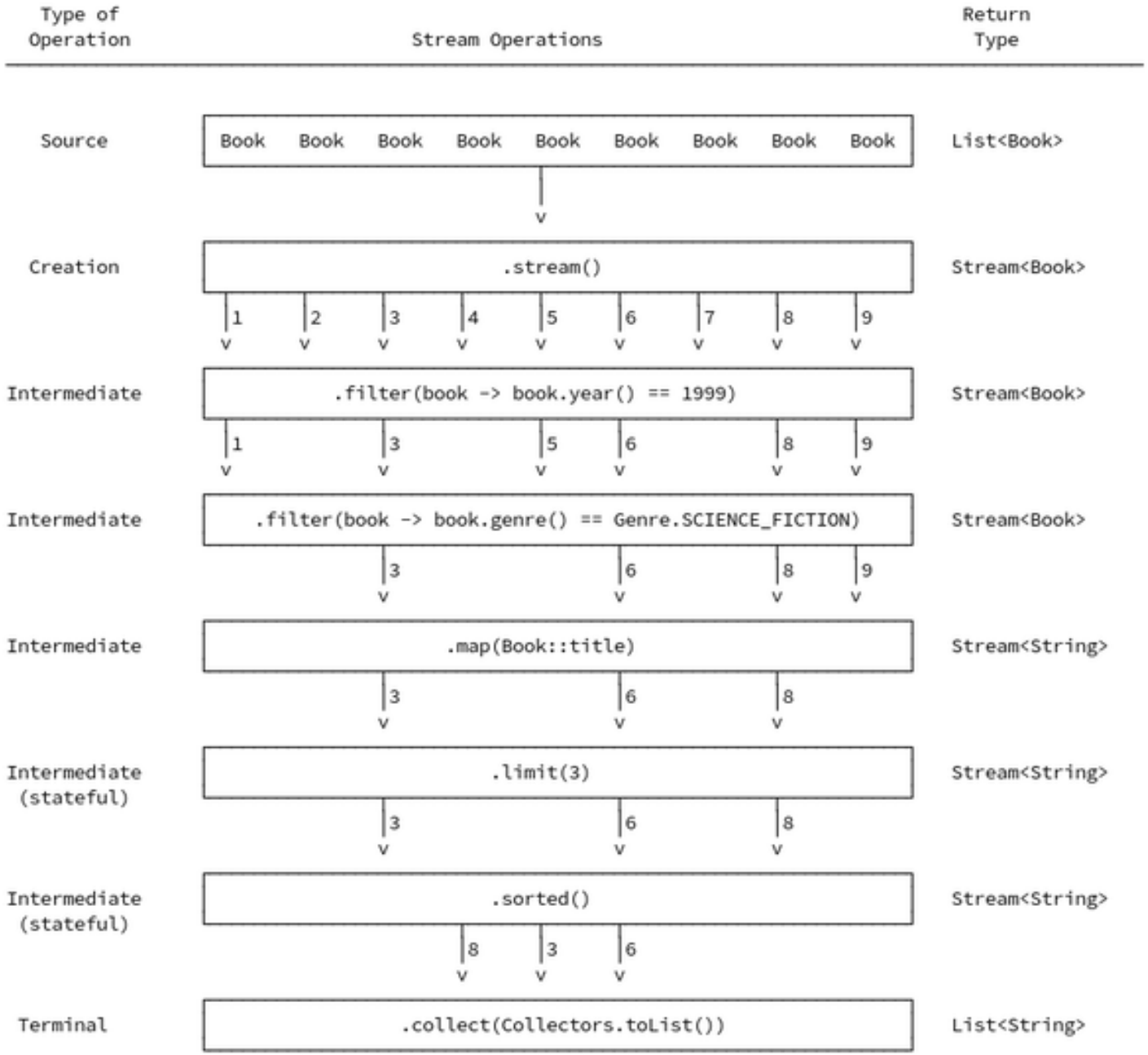
- El origen de estos problemas es mezclar “lo que estás haciendo” (trabajar con datos) y “cómo se hace” (iterar elementos). Este tipo de iteración se llama *iteración externa* .

```
no usages
public class Main {
    public static void main(String[] args) {
        // books must be mutable for sorting
        List<Book> books = new ArrayList<>(
                List.of(new Book( title: "Dracula",  year: 1897, Genre.HORROR),
                        new Book( title: "Brave New World",  year: 1932, Genre.DYSTOPIAN),
                        new Book( title: "1984",  year: 1949, Genre.DYSTOPIAN),
                        new Book( title: "Dune",  year: 1965, Genre.SCIENCE_FICTION),
                        new Book( title: "Do Androids Dream of Electric Sheep",  year: 1968, Genre.SCIENCE_FICTION),
                        new Book( title: "The Shining",  year: 1977, Genre.HORROR),
                        new Book( title: "Neuromancer",  year: 1984, Genre.SCIENCE_FICTION),
                        new Book( title: "The Handmaid's Tale",  year: 1985, Genre.DYSTOPIAN)));

        Collections.sort(books, Comparator.comparing(Book::title));

        List<String> result = new ArrayList<>();
```

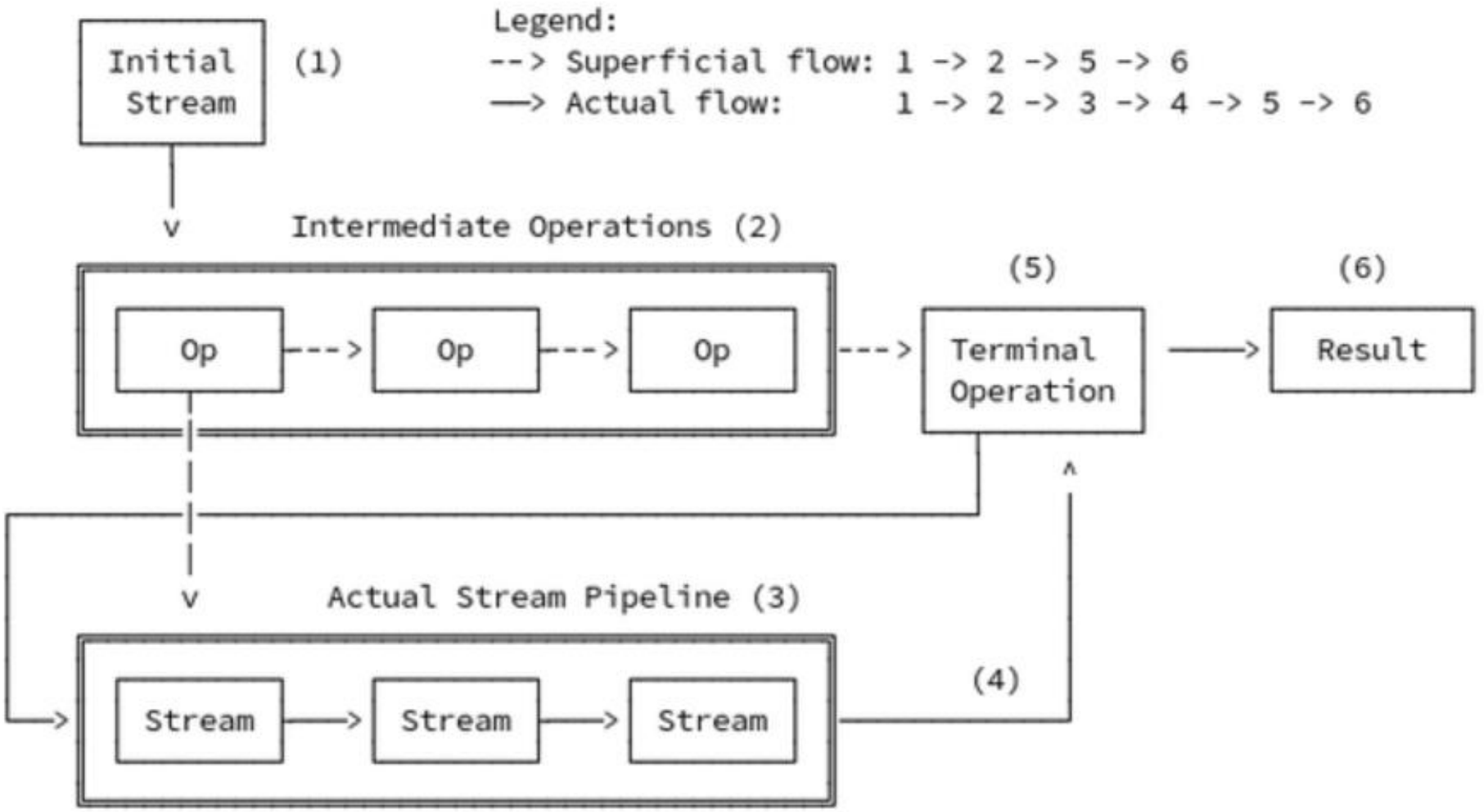
```java
        List<String> result = new ArrayList<>();

        for (var book : books) {

            if (book.year() >= 1970) {
                continue;
            }

            if (book.genre() != Genre.SCIENCE_FICTION) {
                continue;
            }

            var title = book.title();
            result.add(title);

            if (result.size() == 3) {
                break;
            }
        }
        System.out.println(result);
    }
}
```

```java
public class MainStreams {
    public static void main(String[] args) {
        // books must be mutable for sorting
        List<Book> books = new ArrayList<>(
                List.of(new Book( title: "Dracula",  year: 1897, Genre.HORROR),
                        new Book( title: "Brave New World",  year: 1932, Genre.DYSTOPIAN),
                        new Book( title: "1984",  year: 1949, Genre.DYSTOPIAN),
                        new Book( title: "Dune",  year: 1965, Genre.SCIENCE_FICTION),
                        new Book( title: "Do Androids Dream of Electric Sheep",  year: 1968, Genre.SCIENCE_FICTION),
                        new Book( title: "The Shining",  year: 1977, Genre.HORROR),
                        new Book( title: "Neuromancer",  year: 1984, Genre.SCIENCE_FICTION),
                        new Book( title: "The Handmaid's Tale",  year: 1985, Genre.DYSTOPIAN)));

        List<String> result =
                books.stream() Stream<Book>
                        .filter(book -> book.year() < 1970)
                        .filter(book -> book.genre() == Genre.SCIENCE_FICTION)
                        .map(Book::title) Stream<String>
                        .sorted()
                        .limit( maxSize: 3)
                        .collect(Collectors.toList());
        System.out.println(result);
    }
}
```

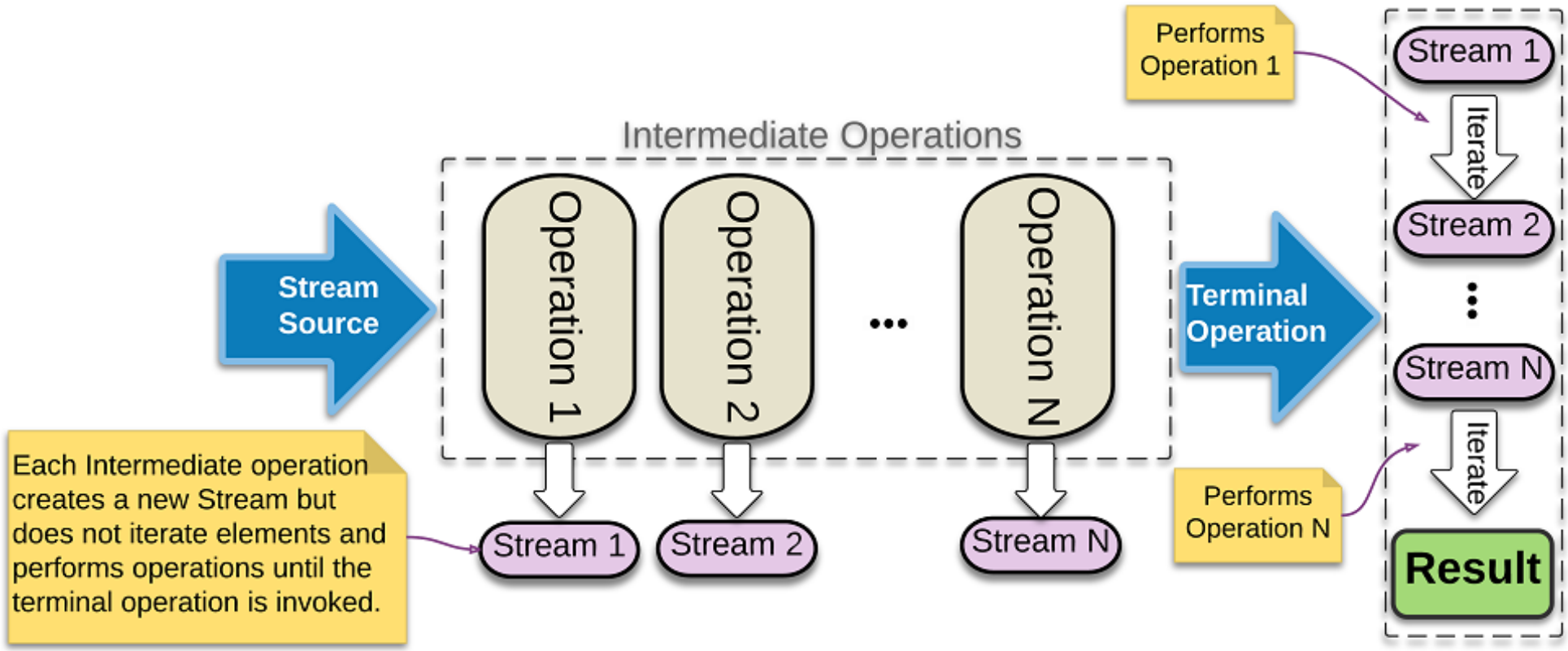
Iteración Interna


Type of Operation
Stream Operations
Return Type
Source
Book Book Book Book Book Book Book Book Book
List<Book>
Creation
.stream()
Stream<Book>
1 2 3 4 5 6 7 8 9
Intermediate
.filter(book -> book.year() == 1999)
Stream<Book>
1 3 5 6 8 9
Intermediate
.filter(book -> book.genre() == Genre.SCIENCE_FICTION)
Stream<Book>
3 6 8 9
Intermediate
.map(Book::title)
Stream<String>
3 6 8
Intermediate (stateful)
.limit(3)
Stream<String>
3 6 8
Intermediate (stateful)
.sorted()
Stream<String>
8 3 6
Terminal
.collect(Collectors.toList())
List<String>

LAZZY EVALUATION

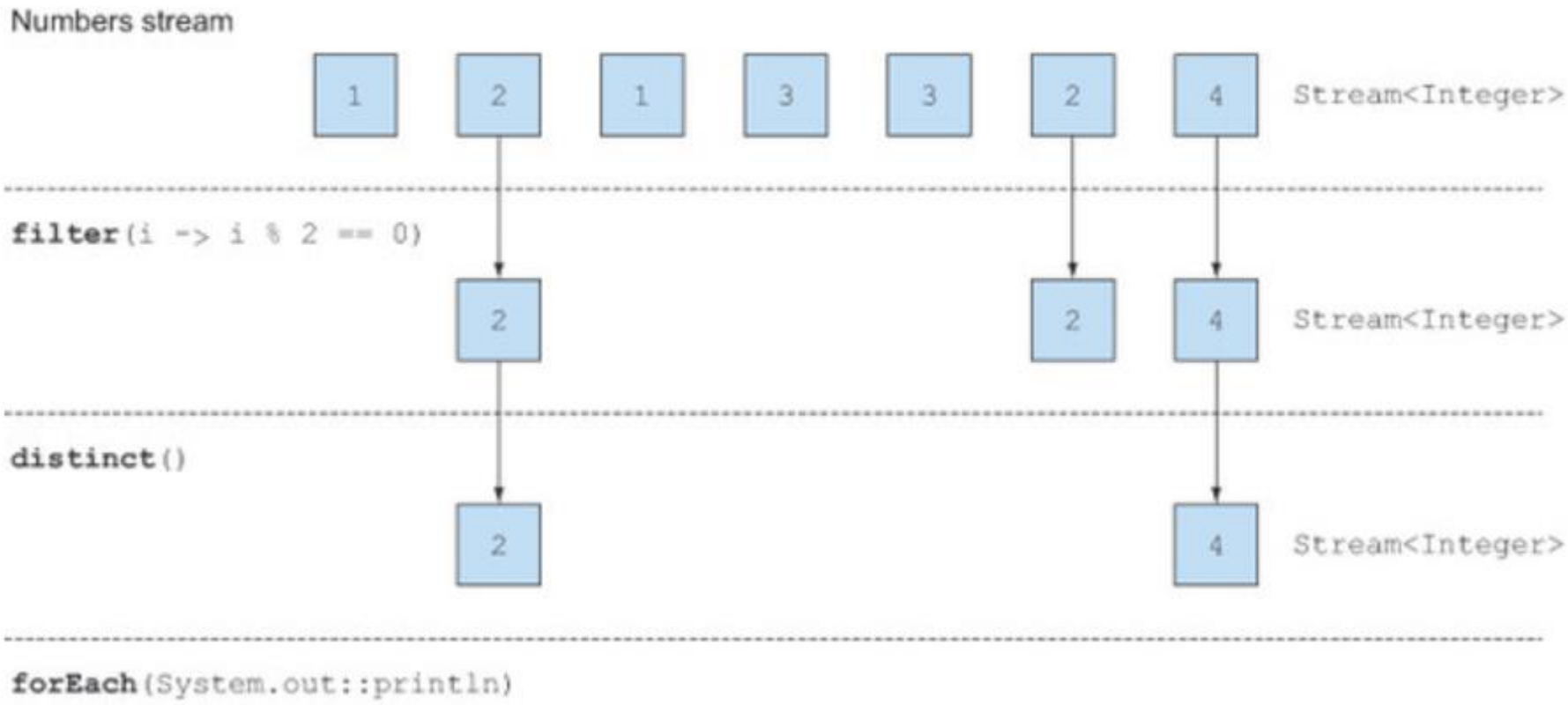


Stream Lazy Evaluation



Filtering

Figura 5.2.Filtrado de elementos únicos en una secuencia



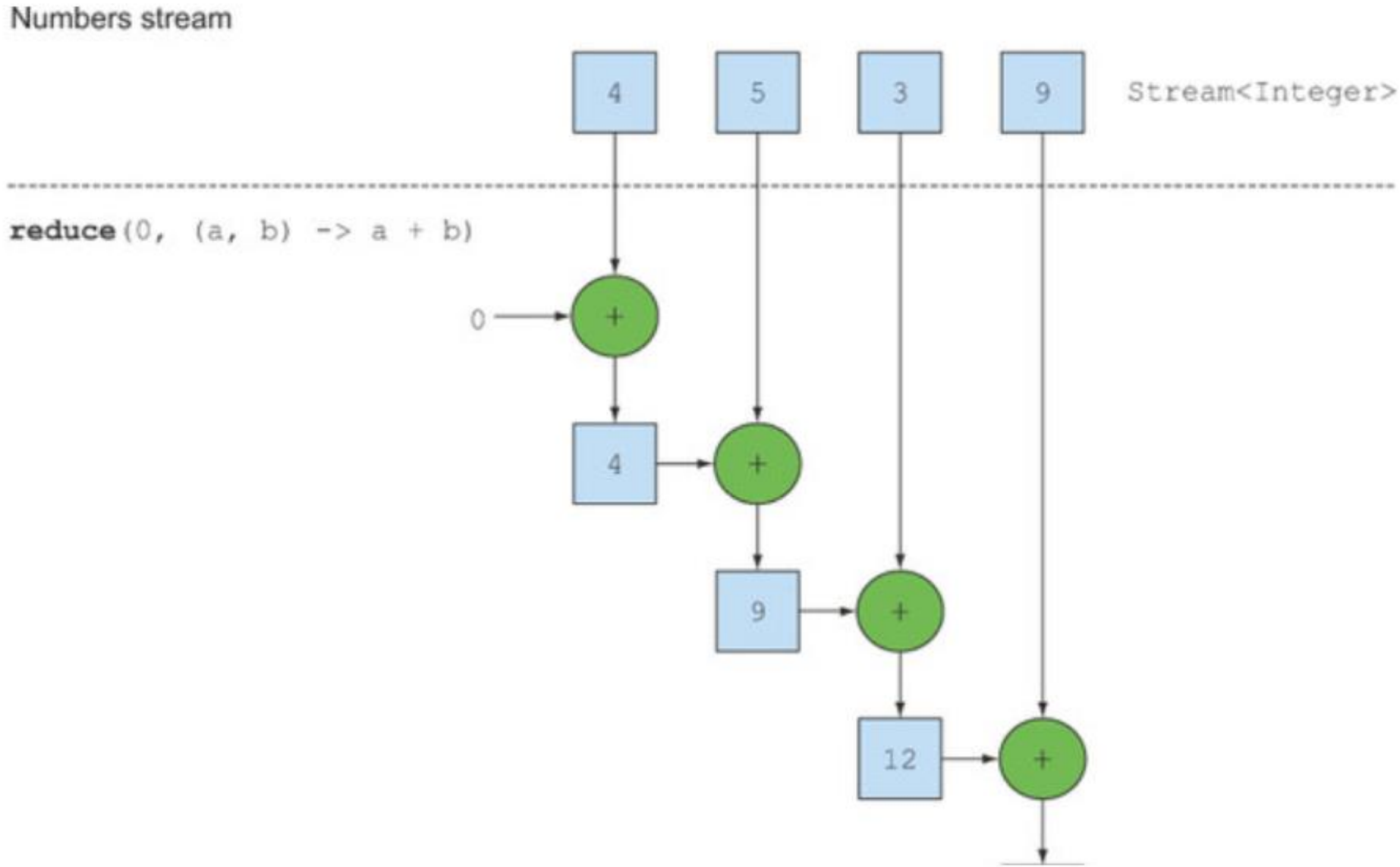
Mapping

- Los Streams admiten el método map, que toma una función como argumento. La función se aplica a cada elemento, mapeándolo en un nuevo elemento (se usa la palabra mapeo porque tiene un significado similar a transformar pero con el matiz de “crear una nueva versión de” en lugar de “modificar”)

Reduce

- Se puede combinar elementos de un flujo para expresar consultas más complicadas como "Calcular la suma de todas las calorías en el menú" o "¿Cuál es el plato con más calorías en el menú?" utilizando la operación **reduce**. Estas consultas combinan todos los elementos de la secuencia repetidamente para generar un único valor, como un valor Integer.
- Estas consultas se pueden clasificar como operaciones de reducción (un flujo se reduce a un valor).

```
Optional<Integer> suma = stream
                                //.reduce((a,b) -> a + b);
                                .reduce(Integer::sum);
 System.out.println("3b) Suma:" + suma.get());
```


Numbers stream
4
5
3
9
Stream<Integer>
reduce(0, (a, b) -> a + b)
0
+
4
+
9
+
12
+