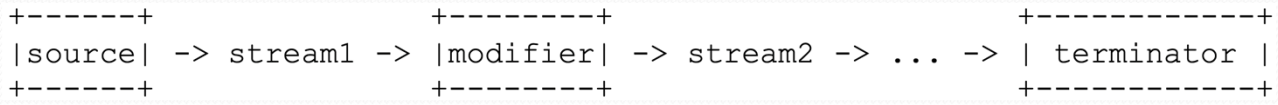
Streams

Streams

- **Stream**. Una secuencia de elementos que admite operaciones.
- Un stream consta de una *fuente*, cero o más *operaciones intermedias* y una *operación terminal*.



- La fuente puede ser un arreglo, una colección, una función generadora, un canal de I/O, etc.)
- Las operaciones intermedias transforman un stream en otro stream.
- La operación terminal produce una salida o un efecto secundario.

Primer ejemplo

- Encontrar la suma de los cuadrados de los enteros entre 1 y 5.
- Usando ciclos:

```
// Compute the sum of the squares of integers 1-5
int sum = 0;
for (int i = 1; i <= 5; i++) {
    sum = sum + i * i;
}
```

Primer ejemplo

- Con streams:

```
int sum = IntStream.range(1, 6).map(n -> n * n).sum();
```

Streams en Java

- En Java, Stream<E> representa una secuencia de elementos de tipo E.

Fuentes

- Formas de obtener un stream:
- Un stream de enteros con IntStream.range().
- Desde una Collection a través del método stream().
- Desde un arreglo a través de Arrays.stream(Object []).
- Usando métodos en la clase Stream como Stream.of().
- Las líneas de un archivo se pueden obtener con BufferedReader.lines() o con Files.lines(Paths.get("nombre")).
- Se pueden obtener secuencias de números aleatorios de Random.ints().

Fuentes

- Otros métodos, incluidos BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence) y JarFile.stream().

Operaciones intermedias

- Las operaciones intermedias devuelven un nuevo stream.
- Esto permite llamar varias operaciones en cadena.
- **Importante**: el stream de entrada no se modifica.

Operaciones intermedias

Operación	Descripción
filter()	Devuelve los elementos que coinciden con el predicado dado
map()	Aplica la función dada a cada elemento del stream
flatMap()	Aplana un stream
distinct()	Devuelve elementos únicos del stream
sorted()	Devuelve los elementos del stream en orden
peek()	Examina solo el primer elemento del stream
limit()	Devuelve un número limitado de elementos del stream
skip()	Omite un número de elementos del stream

Operaciones terminales

- Las operaciones terminales devuelven valores que no son stream.
- Pueden devolver algún tipo primitivo, objeto o nada (void).
- Un stream solo puede tener una operación terminal y debe ser la última operación.

Operaciones terminales

Operación	Tipo	Descripción
forEach	void	Realiza una acción en todos los elementos del stream
toArray	Object[]	Regresa un arreglo conteniendo los elementos del stream
reduce	tipo T	Realiza una operación de reducción en los elementos del stream usando un valor inicial y una operación binaria
collect	Contenedor de tipo T	Regresa un contenedor mutable como List o Set
min	Optional<T>	Regresa el elemento mínimo en el stream
max	Optional<T>	Regresa el elemento máximo en el stream
count	long	Regresa el número de elementos del stream

Operaciones terminales

Operación	Tipo	Descripción
anyMatch	boolean	Regresa true si alguno de los elementos del stream coincide con el predicado
allMatch	boolean	Regresa true si todos los elementos del stream coinciden con el predicado
noneMatch	boolean	Regresa true si ninguno de los elementos del stream coincide con el predicado
findFirst	Optional<T>	Regresa el primer elemento del stream
findAny	Optional<T>	Regresa un elemento aleatorio del stream

Ejemplo: imprimir una lista

```
List<Integer> intList = Arrays.asList(1, 4, 9, 16);
Stream<Integer> intStream = intList.stream();
intStream.forEach(System.out::println);    // 1 4 9 16
```

Ejemplo: convertir un arreglo a lista

```
// En tres líneas
String[] stringArray = new String[] {“a”, “b”, “c”};
Stream<String> stringStream = Arrays.stream(stringArray);
List<String> list = stringStream.toList();
System.out.println(list);    // [a, b, c]

// En una línea
List<String> list =  Arrays.stream(new String[] {“a”, “b”, “c”}).toList();
System.out.println(list);    // [a, b, c]
```

Ejemplo: arreglo de aleatorios

```
// 20 números aleatorios entre 0 y 100 en un arreglo
Random random = new Random();
int[] numbers = random.ints(20, 0, 101).toArray();
System.out.println(Arrays.toString(numbers));

// 20 números aleatorios entre 0 y 100 ordenados en un arreglo
Random random = new Random();
int[] numbers = random.ints(20, 0, 101).sorted().toArray();
System.out.println(Arrays.toString(numbers));
```

Ejemplo: lista de aleatorios

```
// 20 números aleatorios entre 0 y 100 ordenados en una lista
Random random = new Random();
List<Integer> listNumbers = random.ints(20, 0, 101).sorted().boxed().toList();
System.out.println(listNumbers);
```

Operaciones con streams

- Se verán tres operaciones: map, filter y reduce.

Map

- La operación map aplica una función unaria a cada elemento del stream y devuelve un stream nuevo que contiene los resultados en el mismo orden.
- **map : Stream<E> × (E → F) → Stream<F>**
- Ejemplo:

```
// Mapea cada entero a su raíz cuadrada
List<Integer> list = Arrays.asList(1, 4, 9, 16);
Stream<Integer> s = list.stream();
Stream<Double> t = s.map(Math::sqrt);
System.out.println(t.toList());            // [1.0, 2.0, 3.0, 4.0]
```

Map

- La instrucción se puede escribir de forma compacta mediante encadenamiento de llamadas:

```
List<Double> v = Arrays.asList(1, 4, 9, 16).stream().map(Math::sqrt).toList();
System.out.println(v);
```

- O, también:

```
List<Double> w = Stream.of(1, 4, 9, 16).map(Math::sqrt).toList();
System.out.println(w);
```

Map

- Otro ejemplo:

```
// Mapea cada string a su versión en mínusculas
Object[] x = Stream.of(“A”, “b”, “C”).map(String::toLowerCase).toArray();
String[] y = Arrays.copyOf(x, x.length, String[].class);
System.out.println(Arrays.toString(y));      // [a, b, c]
```

Filter

- La operación filter prueba cada elemento de un stream de entrada con un predicado unario.
- Un predicado unario es una función de T a boolean.
- Sólo los elementos que satisfacen el predicado pasan al stream de salida.
- **filter : Stream<E> × (E → boolean) → Stream<E>**

Ejemplos

```
// Filtra los caracteres que no son letras
Character[] z = Stream.of('x', 'y', '2', '3', 'a').filter(Character::isLetter).
                                                    toArray(Character[]::new);
System.out.println(Arrays.toString(z));      // ['x', 'y', 'a']

// Filtra los números pares
Integer[] a = Stream.of(1, 2, 3, 4).filter(b -> b % 2 == 1).toArray(Integer[]::new);
System.out.println(Arrays.toString(a));      // [1, 3]
```

Ejemplos

```
// Filtra los strings vacíos
String[] c = Stream.of("abc", "", "d").filter(d -> !d.isEmpty()).
                                    toArray(String[]::new);
System.out.println(Arrays.toString(c));    // ["abc", "d"]

// Filtra los puntajes menores a 60
List<Integer> scores = random.ints(20, 0, 101).sorted().boxed().toList();
System.out.println(scores);
List<Integer> upperScores = scores.stream().filter(n -> n >= 60).toList();
System.out.println(upperScores);
```

Reduce

- La operación reduce combina los elementos de un stream utilizando una función binaria.
- Sus parámetros son un stream, una función binaria y un valor inicial.
- Si el stream está vacío, el valor de retorno es el valor inicial.
- **reduce : Stream<E> × E × (E × E → E) → E**

Reduce

- stream.reduce(init, f) combina los elementos del stream.
- Una forma en que se puede calcular el resultado:

$result_0 = init$

$result_1 = f(result_0, stream[0])$

$result_2 = f(result_1, stream[1])$

...

$result_n = f(result_{n-1}, stream[n-1])$

- $result_n$ es el resultado final de un stream de n elementos.

Ejemplo

```
// Calcula (((0 + 1) + 2) + 3) para producir el entero 6
int sum = Arrays.asList(1, 2, 3).stream().reduce(0, (x, y) -> x + y);
```

- También se puede utilizar una referencia a método:

```
int sum = Stream.of(1, 2, 3).reduce(0, Integer::sum);
```

Valor inicial

- Hay tres opciones de diseño en la operación reduce.
- La primera es decidir si se requiere un valor inicial.
- En Java, el valor inicial se puede omitir, en cuyo caso reduce utiliza el primer elemento del stream como valor inicial.
- Si el stream está vacío, entonces reduce no tiene ningún valor que devolver, por lo que esta versión de reduce tiene un tipo de retorno Optional<E>.
- Esto facilita el uso de reductores como max, que no tienen un valor inicial bien definido:

Ejemplo: máximo de un stream

```
// Computes max(max(max(5, 8), 3), 1) and returns an Optional<Integer> value containing 8
Optional<Integer> m = Stream.of(5, 8, 3, 1).reduce(Math::max);
System.out.println(m.get().intValue());    // 8
```

Orden de las operaciones

- La segunda elección de diseño es el orden en que se acumulan los elementos.
- La operación reduce de Java requiere que el operador sea asociativo, como + y max.
- Para los operadores asociativos, el orden de combinación no hace ninguna diferencia.

Orden de las operaciones

- Otros lenguajes permiten el uso de operadores no asociativos en reducciones y, entonces, el orden de combinación importa.
- El reduce de Python, llamado fold-left en otros lenguajes de programación, combina la secuencia comenzando desde la izquierda:

Operación fold-left

$result_0 = init$

$result_1 = f(result_0, stream[0])$

$result_2 = f(result_1, stream[1])$

....

$result_n = f(result_n - 1, stream[n - 1])$

Operación fold-right

- La operación fold-right va en la otra dirección:

$result_0 = init$

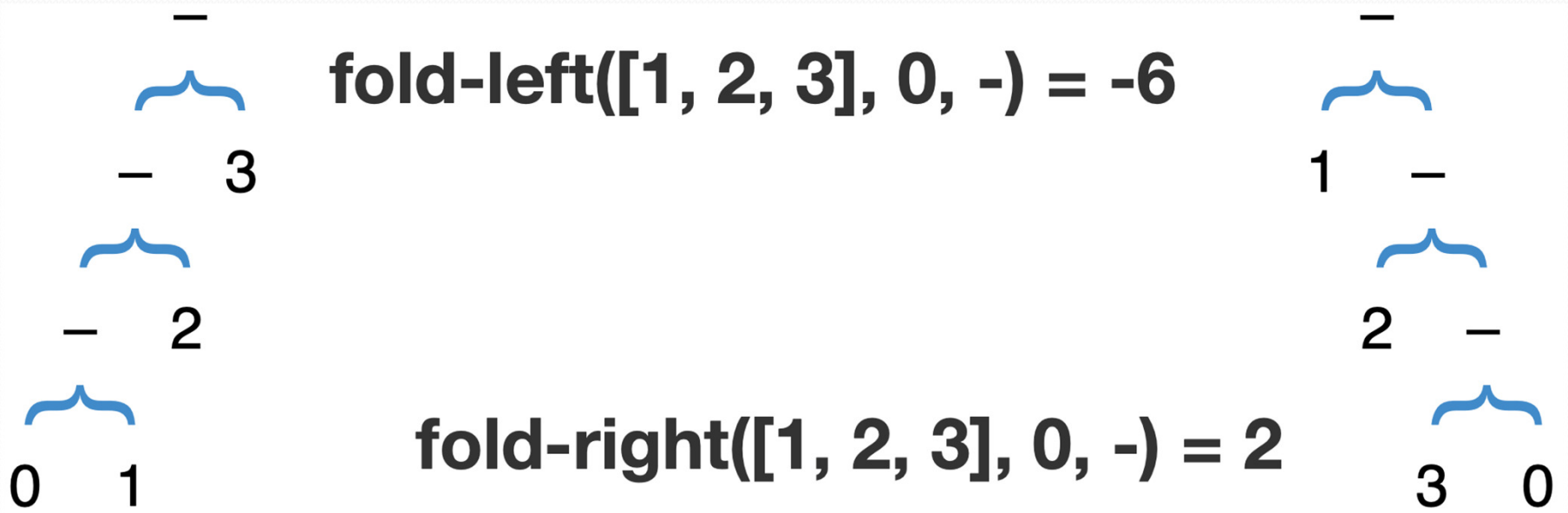
$result_1 = f(stream[n - 1], result_0)$

$result_2 = f(stream[n - 2], result_1)$

...

$result_n = f(stream[0], result_n - 1)$

Comparación



Fuente: https://web.mit.edu/6.031/www/fa17/classes/26-map-filter-reduce/

Reduce en Java

- En Java la función debe ser asociativa.
- La implementación de reduce es libre de elegir cualquier orden de combinación, incluidas combinaciones parciales dentro de la secuencia.
- La reducción

```
Stream.of(1, 2, 3).reduce(0, Integer::sum)
```

- se puede calcular en cualquier orden:

Reduce en Java

$$((0 + 1) + 2) + 3 = 6$$

$$(0 + 1) + (2 + 3) = 6$$

$$0 + ((1 + 2) + 3) = 6$$

$$0 + (1 + (2 + 3)) = 6$$

Reducción a otro tipo

- La tercera opción de diseño es el tipo de retorno de la operación reduce.
- No necesariamente tiene que coincidir con el tipo de los elementos del stream.
- Por ejemplo, se puede usar reduce para concatenar una secuencia de números enteros (tipo E) en un string (tipo F).
- Debido a que Java no especifica el orden de combinación, se requieren dos operadores binarios con firmas de tipo ligeramente diferentes:

Reducción a otro tipo

- Un acumulador $\odot : F \times E \rightarrow F$ que suma un elemento de la secuencia (de tipo E) al resultado creciente (de tipo F).
- Un combinador $\otimes : F \times F \rightarrow F$ que combina dos resultados parciales, cada uno acumulado de parte de la secuencia, en un resultado creciente.
- Entonces, la forma más general de reducción en Java se ve así:
- **reduce : Stream<E> × F × (F × E → F) × (F × F → F) → F**

Reducción a otro tipo

- Ejemplo que concatena una secuencia de números enteros en un string:

```
String z = Stream.of(1, 2, 3)
        .reduce(
            "",                                  // identity value
            (String s, Integer n) -> s + n,      // accumulator
            (String s, String t) -> s + t        // combiner
        );
// Regresa "123"
```

Explicación

- El acumulador ⊙ y el combinador ⊗ deben ser ambos asociativos y también deben ser consistentes entre sí en el sentido de que cualquier orden de aplicación de acumuladores y combinadores al valor inicial y a los elementos de la secuencia debe producir el mismo resultado:

  (("" ⊙ 1) ⊙ 2) ⊙ 3 = "123"

  ("" ⊙ 1) ⊗ (("" ⊙ 2) ⊙ 3) = "123"

  ("" ⊙ 1) ⊗ (("" ⊙ 2) ⊗ ("" ⊙ 3)) = "123"

Ejemplo: arreglo de aleatorios

- Generar un arreglo con 20 enteros aleatorios entre 0 y 1000:

```
Random random = new Random();
int[] numbers = random.ints(20, 0, 1001).toArray();
System.out.println(Arrays.toString(numbers));
```

- El arreglo se puede generar ordenado en formas ascendente usando el método sorted():

```
int[] numbers = random.ints(20, 0, 1001).sorted().toArray();
```

- Suponer que se desea que el arreglo salga ordenado en forma descendente.

Ejemplo: arreglo de aleatorios

- Si se usa sorted(Collections.reverseOrder()), el código no compila.
- Razón: el método random.ints() produce un IntStream.
- IntStream es un stream de variables primitivas int.
- Java no ofrece una forma de ordenar int en orden descendente y el método IntStream.sorted() no acepta parámetros.
- Solución: convertir el stream de IntStream a Stream<Integer>, ordenarlo y luego regresarlo a IntStream.

```
int[] numbers = random.ints(20, 0, 1001).boxed()
.sorted(Collections.reverseOrder()).mapToInt(i -> i).toArray();
```

Ejemplo: operaciones con empleados

- Suponer la siguiente clase:

```
public record Employee(int id, String name, int years, double salary) { }
```

- Y la siguiente lista con datos:

```
List<Employee> employees = new ArrayList<>();
employees.add(new Employee(1000, “Carlos”, 22, 25000));
employees.add(new Employee(7000, “Ana”, 7, 35000));
employees.add(new Employee(2000, “Daniel”, 22, 17000));
employees.add(new Employee(6000, “Blanca”, 9, 21000));
employees.add(new Employee(9000, “Eugenia”, 11, 18000));
```

Ejemplo: operaciones con empleados

```
// Lista de los empleados con antigüedad mayor a 10 años
List<Employee> oldEmployees = employees.stream()
                                .filter(e -> e.years() > 10).toList();
oldEmployees.forEach(System.out::println);    // Carlos, Daniel, Eugenia

// Sueldo mayor
Optional<Double> maxSalary = employees.stream()
                        .map(Employee::salary).max(Double::compare);
System.out.println("Sueldo: " + maxSalary.get());        // 35000
```

Ejemplo: operaciones con empleados

```
// Empleado con el sueldo mayor (max regresa uno)
Optional<Employee> maxEmployee = employees.stream()
                .max(Comparator.comparingDouble(Employee::salary));
System.out.println("Nombre: " + maxEmployee.get().name());      // Ana
```

Ejemplo: operaciones con empleados

```
// Empleados con mayor antigüedad (si son más de uno)
Optional<Integer> maxYears = employees.stream().map(Employee::years)
                                              .max(Integer::compare);
List<Employee> oldestEmployees = employees.stream()
                                  .filter(e -> e.years == maxYears.get()).toList();
System.out.println(oldestEmployees);          // Carlos, Daniel
```

Ejemplo: operaciones con empleados

```
// Suma de los sueldos
double sumSalary = employees.stream().map(Employee::salary)
                .mapToDouble(f -> f).sum();
System.out.println("Suma de sueldos: " + sumSalary);        // 116000.0
```

Ejemplo: operaciones con archivos

- Considerar el archivo points.txt, disponible en la página del curso, que contiene un listado de cien puntos:

```
278 250 1
354 347 1
325 56 1
…
```

- Y considerar la siguiente clase para representar un punto:

```
public record Data(int x, int y, int color) {}
```

Ejemplo: operaciones con archivos

- El método Files.lines(file) regresa un Stream<String> con las líneas del archivo.

```
// Genera una lista con las líneas del archivo
List<String> lines = Files.lines(Paths.get("points.txt")).toList();
lines.forEach(System.out::println);
// Genera una lista con las líneas del archivo quitando las líneas vacías
List<String> lines = Files.lines(Paths.get("points.txt"))
                          .filter(s -> !s.isEmpty()).toList();
lines.forEach(System.out::println);
```

Ejemplo: operaciones con archivos

```
// Cuenta el número de líneas no vacías
long lineCount = Files.lines(Paths.get("points.txt"))
                      .filter(s -> !s.isEmpty()).count();
System.out.println("Number of lines: " + lineCount);      // 100
```

Ejemplo: operaciones con archivos

```
// Genera una lista de objetos Data
List<Data> points = new ArrayList<>();
Files.lines(Paths.get("points.txt"))          // Stream<String>
           .map(l -> l.split("\\s+"))         // Stream<String[]>
           .forEach(a -> points.add(new Data(Integer.parseInt(a[0]),
                        Integer.parseInt(a[1]), Integer.parseInt(a[2]))));
points.forEach(System.out::println);
```

Ejemplo: operaciones con archivos

```
// Cuenta el número de palabras
long wordsCount = Files.lines(Paths.get("points.txt"))      // Stream<String>
        .flatMap(str -> Stream.of(str.split("[ ,.!?\r\n]")))    // Stream<String>
        .filter(s -> !s.isEmpty()).count();
System.out.println("Number of words: " + wordsCount);        // 300
```

- El método flatMap(function) reemplaza un stream con un stream producido al aplicar la función.

Ejemplo: flatMap

```
// Arreglo con el tercer caracter de una lista de palabras
Character[] second = Stream.of("Hello", "World", "Hermosillo",
                "Sonora", "Mexico")                    // Stream<String>
        .flatMap(s -> Stream.of(s.charAt(2)))          // Stream<Character>
        .toArray(Character[]::new);
System.out.println(Arrays.toString(second));           // [l, r, r, n, x]
```

Ejemplo: flatMap

```
// Suma de los elementos de una matriz
int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
int sum = Arrays.stream(matrix)                          // Stream<int[]>
          .flatMap(e -> Arrays.stream(e).boxed())        // Stream<Integer>
          .mapToInt(e -> e)                              // IntStream
          .sum();
System.out.println(sum);          // 21
```

Ejemplo: flatMap

```
// Combinar tres listas de números
List<Integer> primeNumbers = List.of(5, 7, 11, 13);
List<Integer> oddNumbers = List.of(1, 3, 5, 7, 9);
List<Integer> evenNumbers = List.of(2, 4, 6, 8, 10);
List<List<Integer>> allLists = List.of(primeNumbers, oddNumbers, evenNumbers);
System.out.println(allLists);      // [[5, 7, 11, 13], [1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
List<Integer> numbers = allLists.stream()     // Stream<List<Integer>
        .flatMap(list -> list.stream())       // Stream<Integer>
        .toList();
System.out.println(numbers);     // [5, 7, 11, 13, 1, 3, 5, 7, 9, 2, 4, 6, 8, 10]
```

Ejemplo: flatMap

```
// Elimina la palabra “a” de una matriz con palabras
String[][] array = {{“a”, “b”}, {“c”, “d”}, {“e”, “f”}};
List<String> letters = Stream.of(array)          // Stream<String[]>
        .flatMap(Stream::of)                     // Stream<String>
        .filter(x -> !“a”.equals(x))
        .toList();
System.out.println(letters);                     // [b, c, d, e, f]
```

Ejemplo: flatMap

```
// Considerar dos conjuntos con nombres de libros
Set<String> books1 = new HashSet<>();
books1.add("Java 8 in Action");
books1.add("Spring Boot in Action");
books1.add("Effective Java");

Set<String> books2 = new HashSet<>();
books2.add("Learning Python");
books2.add("Effective Java");
```

Ejemplo: flatMap

```
// Lista con los dos conjuntos
List<Set<String>> allBooks = List.of(books1, books2);
// Conjunto con los libros de Java
Set<String> javaBooks = allBooks.stream()   // Stream<Set<String>>
        .flatMap(x -> x.stream())                  // Stream<String>
        .filter(x -> !x.toLowerCase().contains("python"))
        .collect(Collectors.toSet());
System.out.println(javaBooks); // [Spring Boot in Action, Effective Java, Java 8 in Action]
```

Resumen

- Un stream es una secuencia de elementos que soporta operaciones.
- Arreglos, colecciones, strings, rangos de enteros, archivos y otras fuentes pueden ser convertidos en streams.
- Las operaciones sobre streams incluyen `map` (aplicar una operación a cada elemento), `filter` (mantener o remover elementos basado en algún criterio) y `reduce` (combinar múltiples elementos en un solo elemento).
- Los streams permiten hacer operaciones sobre secuencias de objetos sin utilizar ciclos.