

Principales métodos de Stream



Conteo y procesamiento

Métodos
finales

➤ `long count()`. Devuelve el número de elementos de un Stream.

```
Stream st=Stream.of(2,5,7,3,6,2,3);  
//indica el total de elementos  
System.out.println(st.count()); //7
```

➤ `void forEach(Consumer<? super T> action)`. Realiza una acción para cada elemento del stream.

```
Stream st=Stream.of(2,5,8,3,6,2,10);  
//muestra todos los elementos  
st.forEach(n->System.out.println(n));  
System.out.println(st.count()); //Error!!
```

Tras llamar a un método el stream se cierra y **no puede** volver a utilizarse

Extracción de datos

Métodos
intermedios

➤ **Stream<T> distinct().** Devuelve un Stream eliminando los elementos duplicados, según aplicación de *equals()*.

```
Stream<Integer> st=Stream.of(2,5,3,3,6,2,4);  
//Cuenta el total de números no repetidos  
System.out.println(st.distinct().count()); //5
```

➤ **Stream<T> limit(long n).** Devuelve un nuevo Stream con los *n* primeros elementos del mismo.

```
Stream<Integer> st=Stream.of(2,5,8,3,6,2,10);  
//Devuelve un Stream formado por 2,5 y 8  
Stream<Integer> st2=st.limit(3);
```

➤ **Stream<T> skip(long n).** Devuelve un nuevo Stream, saltándose los *n* primeros elementos.

Comprobaciones

Métodos
finales

➤ **boolean anyMatch(Predicate<? super T> predicate).** Devuelve *true* si algún elemento del Stream cumple con la condición del predicado:

```
Stream st=Stream.of(2,5,7,3,6,2,3);  
//indica si alguno es mayor de 5  
System.out.println("alguno mayor 5? "+st.anyMatch(n->n>5)); //true
```

➤ **boolean allMatch(Predicate<? super T> predicate).** Devuelve *true* si todos cumplen con la condición del predicado.

➤ **boolean noneMatch(Predicate<? super T> predicate).** Devuelve *true* si ninguno cumple con la condición del predicado.

Funcionan en modo
cortocircuito

Revisión conceptos



Dado el siguiente stream de nombres, escribe un bloque de código que nos indique si alguno de los nombres finaliza por la letra "d"

```
Stream<String> nombres=Stream.of("Mario", "Ana", "David", "Belén");
```

Respuesta

```
System.out.println(nombres  
                    .anyMatch(n->n.endsWith("d"));
```

Filtrado

Método
intermedio

➤ `Stream<T> filter(Predicate<? super T> predicate)`. Aplica un filtro sobre el Stream, devolviendo un nuevo Stream con los elementos que cumplen el predicado:

pipeline

```
Stream<Integer> st=Stream.of(7,5,7,3,6,2,3,8,5);  
//cuenta el total de números mayores de 3  
//no duplicados  
System.out.println(st  
    .distinct()  
    .filter(s->s>3)  
    .count());
```

Búsquedas

Métodos
finales

- **Optional<T> findFirst()**. Devuelve el primer elemento del Stream, o un Optional vacío si no hay nada

```
Stream<Integer> st=Stream.of(11,5,8,3,9);  
//devuelve el primer par  
Optional<Integer> op=st  
    .filter(s->s%2==0)  
    .findFirst();  
if(op.isPresent()){  
    System.out.println("El primer par es "+op.get());  
}
```

- **Optional<T> findAny()**. Devuelve cualquiera de los elementos del Stream. Normalmente, el primero

La clase Optional<T>

- Encapsula resultados de una operación final de un Stream
- Podemos utilizar los siguientes métodos para manipularlo:
 - `T get()`. Devuelve el valor encapsulado. Si no hay ningún valor, lanza una `NoSuchElementException`
 - `T orElse(T other)`. Devuelve el valor encapsulado. Si no hay ninguno, entonces devuelve el valor pasado como parámetro.
 - boolean `isPresent()`. Permite comprobar si contiene o no algún valor.
- Existen las variantes `OptionalInt` y `OptionalDouble` que encapsulan tipos primitivos

Obtención de extremos

Métodos
finales

- **Optional<T> max(Comparator<? super T> comparator).**
Devuelve el mayor de los elementos, según el criterio de comparación del objeto Comparator:

```
Stream<Integer> nums=Stream.of(20,5,8,3,9);  
//muestra el mayor de los números del Stream  
Optional<Integer> op=nums.max((a,b)->a-b);  
System.out.println("mayor: "+op.get());
```

- **Optional<T> min(Comparator<? super T> comparator).**
Operación contraria a max.

Revisión conceptos



¿Qué se mostrará al ejecutar el siguiente código?:

```
Stream<Integer> nums=Stream.of(15,-3,21,7,9,-4,7);  
System.out.println(nums  
    .filter(n->n<0)  
    .max((a,b)->a-b)  
    .get());
```

Respuesta

Se mostrará -3

Transformación

Métodos
intermedios

➤ **Stream<R> map(Function<? super T, ? extends R> mapper).**
Transforma cada elemento del Stream en otro según el criterio definido por el objeto Function que se le pasa como parámetro:

```
Stream<String> st=Stream.of("Juan," "Maria", "Ana");  
//genera un Stream con los nombres en mayúsculas  
Stream<String> st2=st.map(s->s.toUpperCase());
```

➤ **IntStream mapToInt(ToIntFunction<? super T> mapper).**
Aplica una función a cada elemento del Stream que genera un int de cada elemento. El resultado se devuelve como IntStream:

```
Stream<String> st=Stream.of("Juan," "Maria", "Ana");  
//muestra la suma de todos los caracteres de los nombres  
System.out.println(st  
    .mapToInt(s->s.length())  
    sum());
```

Stream de tipos primitivos

➤ Las interfaces `IntStream`, `DoubleStream` y `LongStream`, cuyos objetos son obtenidos mediante los métodos `mapToInt`, `mapToDouble` y `mapToLong`, respectivamente, proporcionan los siguientes métodos de cálculo:

- `int sum()`. Método final que devuelve la suma de todos los elementos del stream. En `DoubleStream` y `LongStream` el tipo de devolución es `double` y `long`, respectivamente
- `OptionalDouble average()`. Método final que devuelve la media encapsulada en un `OptionalDouble` en los tres casos
- `OptionalInt max()` y `min()`. Devuelven el mayor y menor de los números, respectivamente. En `DoubleStream` y `LongStream` el tipo de devolución es `OptionalDouble` y `OptionalLong`, respectivamente.

Transformación y aplanamiento

Método
intermedio

➤ `Stream<R> flatMap(Function<T, Stream<R>> mapper)`.
Devuelve un nuevo Stream, resultante de unir los Streams generados por la aplicación de una función sobre cada elemento.

▪ Ejemplo: Partiendo de una lista de listas de nombres, calcular cuantos nombres en total tienen más de 4 caracteres:

```
List<List<String>> datos=Arrays.asList(Arrays.asList("Gema","María","Carlos"),  
                                       Arrays.asList ("Laura","Ana","Luis"));  
System.out.println(datos.stream()  
                    .flatMap(l->l.stream().map(s->s.length()))  
                    .filter(n->n>4)  
                    .count());
```

Revisión conceptos

Dado el siguiente array bidimensional de Integer, escribir un bloque de código que nos muestre el mayor de todos los números de dicho array:

```
Integer[][] datos = {{8,3,5},{2,11,7},{4,1,9,6}};
```

Respuesta

```
System.out.println(Arrays.stream(datos) //convertimos a Stream
.flatMap(n->Arrays.stream(n)) //cada array se convierte a Stream
.mapToInt(n->n) //transformamos a IntStream
.max() //aplicamos método de cálculo
.getAsInt()); //extraemos resultado
```

Procesamiento intermedio

Método
intermedio

➤ `Stream<T> peek(Consumer<? super T> proceso)`. Aplica el consumer a cada elemento del Stream, devolviendo un nuevo stream idéntico para continuar con la manipulación de los elementos:

```
Stream<Integer> nums=Stream.of(20,5,8,3,9);  
//muestra los pares y el total de éstos  
System.out.println("total: "+nums  
    .filter(n->n%2==0)  
    .peek(n->System.out.println("par: "+n))  
    .count());
```



```
par: 20  
par: 8  
total: 2
```

Revisión conceptos

¿Qué se mostrará al ejecutar los siguientes códigos?:

```
Stream<Integer> nums=Stream.of(20,5,8,3,9);  
nums.peek(n->System.out.println(n));
```

Respuesta

No se mostrará nada, como las operaciones intermedias funcionan en modo lazy, se necesita realizar una operación final para que se lleven a término

```
Stream<Integer> nums=Stream.of(20,5,8,3,9);  
nums.peek(n->System.out.println(n)).allMatch(n->n>5);
```

Respuesta

Se mostrará 20 y 5. Como peek funciona en modo lazy, las operaciones no se procesan hasta la ejecución del método final y, como este funciona en modo cortocircuito, solo procesa los dos primeros elementos

Ordenación

Métodos
intermedios

- `Stream<T> sorted()`. Devuelve un Stream con los elementos ordenados, según el orden natural de los mismos
- `Stream<T> sorted(Comparator<? super T> comparator)`. Devuelve un Stream con los elementos ordenados, según el criterio de comparación especificado:

```
Stream<String> st=Stream.of("casa","pelota","lampara","disco");  
// muestra los nombres ordenados por número de caracteres  
st.sorted((a,b)->a.length()-b.length())  
  .forEach(s->System.out.println(s));
```

```
Stream<Persona> st=Stream.of(new Persona("marco",34), new Persona("ana",28));  
// muestra los nombres de las personas, ordenadas por edad  
st.sorted(Comparator.comparing(p->g.getEdad()))  
  .forEach(p->System.out.println(p.getNombre()));
```

Reducción

Método
final

➤ `Optional<T> reduce(BinaryOperator<T> accumulator)`.
Realiza la reducción de los elementos del stream a un único valor, utilizando la función proporcionada como parámetro:

```
Stream<Integer> nums=Stream.of(20,5,8,3,9);  
//Calcula la suma de todos los elementos del Stream  
System.out.println(nums  
    .reduce((a,b)->a+b)  
    .get());
```

Revisión conceptos



¿Qué código habría que escribir para sumar los 3 números pares más pequeños del siguiente stream?:

```
Stream<Integer> nums=Stream.of(11,20,5,8,3,9,4,10,2);
```

Respuesta

```
System.out.println(nums  
    .filter(n->n%2==0)  
    .sorted()  
    .limit(3)  
    .reduce((a,b)->a+b)  
    .get());
```

Reducción a colección

Método
final

➤ `R collect(Collector<? super T, A, R> collector)`. Devuelve un `List`, `Map` o `Set` con los datos del `Stream`, en función de la implementación de `Collector` proporcionada:

```
Stream<Integer> nums=Stream.of(20,5,8,5,3,3,9);  
//Genera una lista con los elementos del Stream sin duplicados  
List<Integer> lista=nums.distinct().collect(Collectors.toList());
```

```
Stream<Persona> personas=Stream.of(new Persona("Jaime",5431),  
                                   new Persona("Marta",5213),  
                                   new Persona("Pilar",6792));  
//genera una tabla con los datos de las personas, utilizando el dni como clave  
//y el nombre como valor  
Map<Integer,String> lista=personas  
    .collect(Collectors.toMap(p->p.getDni(),p->p.getNombre()));
```

Agrupación

➤ Utilizando el método `collect()` de `Stream`, se puede generar una agrupación de objetos utilizando el siguiente método de `Collectors`:

- `Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`. Devuelve un `Collector` que implementa una agrupación de tipo `groupBy`. El método recibe como parámetro un objeto `Function` con el criterio de agrupación. Con este tipo de `Collector`, la llamada a `collect()` devolverá un `Map` de listas. Cada elemento del mapa tiene una clave, que es el dato por el que se hace la agrupación, y un valor con la lista de objetos de cada grupo

```
Stream<Persona> st=Stream.of(new Persona("Juan",30,"jj@gmail.com"),
    new Persona("Ana",40,"anaj@gmail.com"),
    new Persona("Bea",35,"bae@gmail.com"),
    new Persona("Pedro",40,"bae@gmail.com"));
//agrupa las personas por edad
Map<Integer, List<Persona>> personas=st.collect(Collectors.groupingBy(p->p.getEdad()));
personas.forEach((k,v)->System.out.println(v));
```

Revisión conceptos



Indica que se mostrará al ejecutar el siguiente código:

```
Stream<Empleado> empleados=Stream.of(new Empleado("Marcos","ventas",1600),  
                                     new Empleado("Alicia","ventas",1200),  
                                     new Empleado("Juan","RRHH",1100),  
                                     new Empleado("Alfonso","ventas",1310),  
                                     new Empleado("Elena","RRHH",1550));  
  
empleados  
    .collect(Collectors.groupingBy(e->e.getDepartamento()))  
    .forEach((k,v)->System.out.println(k+"."+v.stream().mapToDouble(e->e.getSalario()).average()));
```

- A. Error de compilación
- B. Excepción
- C. RRHH:OptionalDouble[1325.0]
ventas:OptionalDouble[1370.0]
- D. RRHH:1325.0
ventas:1370.0

Respuesta

La correcta es la C, dado que `average()` de `DoubleStream` devuelve el resultado envuelto en un `OptionalDouble`

Partición

➤ Mediante el siguiente método de `Collectors` podemos proporcionar una implementación de `collect()` que genere una partición:

- `Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)`. Devuelve un `Collector` para generar una agrupación `Map` de clave boolean y valor lista de objetos. El método recibe como parámetro un `predicate` para aplicar la condición a cada elemento, de modo que los que la cumplan serán agrupados en una lista con clave *true*, y los que no en otra lista con clave *false*.

```
Stream<Persona> st=Stream.of(new Persona("Juan",15,"jj@gmail.com"),
                             new Persona("Ana",23,"anaj@gmail.com"),
                             new Persona("Bea",16,"bae@gmail.com"),
                             new Persona("Pedro",34,"bae@gmail.com"));
//agrupa las personas menores de edad por un lado, y mayores por otro
Map<Boolean,List<Persona>> personas=st.collect(Collectors.partitionBy(p->p.getEdad()>18));
```

Otras implementaciones de Collector

➤Collectors ofrece estos otros métodos de interés:

- `Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper)`. Permite calcular la media a partir de los valores devueltos por la función. Existe también `averagingInt` y `averagingLong`
- `Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)`. Permite calcular la suma a partir de los valores devueltos por la función. Existe también `summingLong` y `summingDouble`
- `Collector<CharSequence,?,String> joining(CharSequence delimiter)`. Devuelve un Collector que concatena en un único String todos los String resultantes de la llamada a `toString()` sobre cada objeto del Stream:

```
Stream<Persona> st=Stream.of(new Persona("Juan",30,"jj@gmail.com"),
    new Persona("Ana",40,"anaj@gmail.com"),
    new Persona("Bea",35,"bae@gmail.com"));
//imprime los nombres de todas las personas, separados por una coma
System.out.println(st
    .map(p->p.getNombre())
    .collect(Collectors.joining(",")));
```


Revisión conceptos

Dado el siguiente stream, indica de que dos formas distintas se podría calcular la temperatura media de todas las ciudades

```
Stream<Ciudad> ciudades=Stream.of(new Ciudad("Viena",10.7),  
    new Ciudad("Paris",14.8),  
    new Ciudad("Madrid",20.5),  
    new Ciudad("Sevilla",24.0));
```

Respuesta

Opción 1:

```
System.out.println(ciudades  
    .collect(Collectors.averagingDouble(c->c.getTemperatura())));
```

Opción 2:

```
System.out.println(ciudades  
    .mapToDouble(c->c.getTemperatura())  
    .average()  
    .getAsDouble());
```