

Procesamiento de datos con streams de Java SE 8 - Parte 1

Por Raoul-Gabriel Urma

Uso de operaciones de streams para expresar consultas de procesamiento de datos complejas

¿Qué haríamos sin las colecciones? Casi todas las aplicaciones de Java *crean* y *procesan* colecciones. Son esenciales para muchas tareas de programación: permiten agrupar y procesar datos. Por ejemplo, el desarrollador podría querer crear una colección de transacciones bancarias para representar el extracto de un cliente. Luego, tal vez quiera procesar toda la colección para averiguar cuánto dinero gastó el cliente. A pesar de su importancia, el procesamiento de colecciones en Java dista de ser perfecto.

En primer lugar, los patrones de procesamiento de colecciones típicos son similares a las operaciones del estilo de las que se usan en SQL para "buscar" (por ejemplo, buscar la transacción de mayor valor) o "agrupar" (por ejemplo, agrupar todas las transacciones relacionadas con compras de almacén). La mayoría de las bases de datos permiten establecer operaciones como esas de manera declarativa. Por ejemplo, la siguiente consulta de SQL permite buscar la identificación de la transacción de mayor valor: "SELECT id, MAX(value) from transactions".

Como puede verse, no es necesario programar *cómo* calcular el valor máximo (por ejemplo, mediante bucles y una variable para hacer el seguimiento del mayor valor). Solo se expresa *qué* resultado se espera. Así, debemos preocuparnos menos acerca de cómo codificar explícitamente las consultas; el lenguaje lo hace por nosotros. ¿Por qué no se puede hacer algo parecido con las colecciones? ¿Quién no se ha encontrado codificando esas operaciones con bucles una y otra vez?

En segundo lugar, ¿cómo podemos hacer para procesar colecciones realmente grandes con eficiencia? Idealmente, para acelerar el procesamiento conviene trabajar con arquitecturas de núcleos múltiples. No obstante, programar código paralelo es una tarea ardua y en la que es fácil cometer errores.

Presentamos una posibilidad extraordinaria: Esas dos operaciones pueden generar elementos "infinitamente".

Y todo gracias a Java SE 8. Los diseñadores de la interfaz API de Java han incorporado en su actualización una nueva abstracción denominada *Stream*, que permite procesar datos de modo declarativo. Más aún, los streams permiten aprovechar las arquitecturas de núcleos múltiples sin necesidad de programar líneas de código multiproceso. Suena bien, ¿no? Eso es lo que exploraremos en esta serie de artículos.

Antes de ahondar en lo que se puede hacer con streams, veamos un ejemplo para tener una idea del nuevo estilo de programación que posibilitan los streams de Java SE 8. Imaginemos que necesitamos encontrar todas las transacciones del tipo grocery y obtener un listado de identificaciones de transacciones ordenadas de mayor a menor por valor de transacción. En Java SE 7, usaríamos el código que se muestra en la **Secuencia 1**. En Java SE 8, usaremos el código que se muestra en la **Secuencia 2**.

```
List<Transaction> groceryTransactions = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.GROCERY){
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
    transactionIds.add(t.getId());
}
```

Secuencia 1

```
List<Integer> transactionIds =
    transactions.stream()
        .filter(t -> t.getType() == Transaction.GROCERY)
        .sorted(comparing(Transaction::getValue).reversed())
        .map(Transaction::getId)
        .collect(toList());
```

Secuencia 2

En la **Figura 1** se ilustra el código de Java SE. En primer lugar, obtenemos un stream del listado de transacciones (los datos) con el método `stream()` disponible para `List`. Luego, se encadenan varias operaciones (`filter`, `sorted`, `map`, `collect`) en un proceso, que puede verse como una consulta respecto de los datos.



Figura 1

¿Y cómo se hace para tener código paralelo? En Java SE 8 es fácil: solo es necesario reemplazar la instrucción `stream()` por `parallelStream()`, como se muestra en la **Secuencia 3**, y la API de streams descompondrá internamente la consulta para aprovechar los núcleos múltiples de la computadora.

```
List<Integer> transactionIds =
    transactions.parallelStream()
        .filter(t -> t.getType() == Transaction.GROCERY)
        .sorted(comparing(Transaction::getValue).reversed())
        .map(Transaction::getId)
        .collect(toList());
```

Secuencia 3

La aparente complejidad del código no debe preocuparnos. En las próximas secciones, veremos en detalle cómo funciona. No obstante, vale destacar el uso de expresiones lambda (tales como `t-> t.getCategory() == Transaction.GROCERY`) y referencias a métodos (por ejemplo, `Transaction::getId`), con los que el desarrollador seguramente ya esté familiarizado. (Para recordar cómo se usan las expresiones lambda, recomendamos consultar artículos anteriores de *Java Magazine* y otros recursos enumerados al final de este artículo.)

Por ahora, podemos entender un stream como una abstracción para expresar operaciones eficientes al estilo SQL con relación a una colección de datos. Además, esas operaciones pueden parametrizarse sucintamente mediante expresiones lambda.

Una vez que haya leído la serie completa de artículos sobre streams de Java SE 8, el desarrollador sabrá usar la API de streams para programar código similar al de la **Secuencia 3** a fin de expresar consultas potentes.

Programación con streams: primeros pasos

Empecemos por ver algo de teoría. ¿Cuál es la definición de stream? En pocas palabras, podría decirse que es una "secuencia de elementos de un origen que admite operaciones concatenadas". Ahora desglosemos la definición:

Secuencia de elementos: Un stream brinda una interfaz para un conjunto de valores secuenciales de un tipo de elemento particular. No obstante, los streams no almacenan elementos; estos se calculan cuando se recibe la solicitud correspondiente.

Origen: Los streams toman su insumo de un origen de datos, como colecciones, matrices o recursos de E/S.

Operaciones concatenadas: Los streams admiten operaciones estilo SQL y operaciones comunes a la mayoría de los lenguajes de programación funcionales, como filter, map, reduce, find, match y sorted, entre otras.

Más aún, las operaciones de los streams tienen dos características fundamentales que las distinguen de las operaciones con colecciones:

Estructura de proceso: Muchas operaciones de stream devuelven otro stream. Así, es posible encadenar operaciones para formar un proceso más abarcador. Esto, a su vez, permite lograr ciertas optimizaciones, por ejemplo mediante las nociones de "pereza" (*laziness*) y "corte de circuitos" (*short-circuiting*), que analizaremos más adelante.

Iteración interna: A diferencia del trabajo con colecciones, en que la iteración es explícita (*iteración externa*), las operaciones del stream llevan a cabo la iteración tras bambalinas.

Repasemos el ejemplo de código anterior para explicar estas ideas. En la **Figura 2** se ilustra la **Secuencia 2** en mayor detalle.

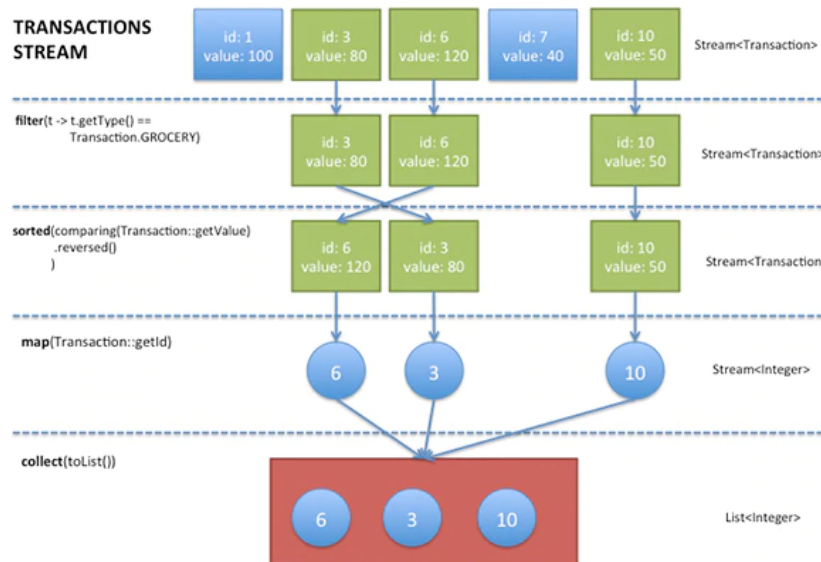


Figura 2

En primer lugar, obtendremos un stream del listado de transacciones llamando al método `stream()`. El origen de datos es el listado de transacciones, que le proporcionará una secuencia de elementos al stream. Luego, aplicaremos una serie de operaciones concatenadas al stream: `filter` (para filtrar elementos según un predicado particular), `sorted` (para ordenar los elementos según un comparador) y `map` (para extraer información). Todas las operaciones, a excepción de `collect`, devuelven un Stream, por lo que es posible encadenarlas y formar un proceso, que puede verse como consulta respecto de los datos del origen.

En realidad no se lleva a cabo ninguna tarea hasta que se invoca la operación `collect`. Esta última comenzará a abordar el proceso para devolver un resultado (que no será un Stream; en este caso, se trata de un listado `List`). Olvidémonos de `collect` por el momento; exploraremos esa operación en detalle en otro artículo. Mientras, podemos entenderla como una operación que como argumento emplea diversas recetas para acumular los elementos de un stream en un resultado resumido. Aquí, `toList()` describe una receta para convertir un Stream en un List.

Antes de explorar los diferentes métodos disponibles para un stream, conviene hacer una pausa y reflexionar sobre la diferencia conceptual entre un stream y una colección.

Streams vs. colecciones

Tanto la noción de colecciones que ya existía en Java como la nueva noción de streams se refieren a interfaces con secuencias de elementos. Entonces, ¿cuál es la diferencia? En resumen, las colecciones hacen referencia a datos mientras que los streams hacen referencia a cómputos.

Pensemos por ejemplo en una película almacenada en un DVD. Se trata de una colección (de bytes o de fotografías; precisarlo no es importante para el ejemplo) porque contiene toda la estructura de datos. Ahora imaginemos el mismo video, pero esta vez lo reproducimos desde Internet. En este caso hablamos de un stream (de bytes o fotografías). El reproductor de video por secuencias (*streaming*) necesita descargar solo unos pocos fotografías más allá de los que está viendo el usuario; así, es posible comenzar a mostrar los valores del comienzo del stream antes de que la mayor parte del stream se haya computado (la transmisión de secuencias o streaming puede pensarse como un partido de fútbol en vivo).

En términos simples, la diferencia entre las colecciones y los streams se relaciona con *cuándo* se hacen los cómputos. Las colecciones son estructuras de datos que se almacenan en la memoria, donde se encuentran todos los valores que tiene la estructura de datos en un momento dado; cada elemento de la colección debe calcularse antes de que se lo pueda agregar a la colección. En cambio, los streams son estructuras de datos fijas conceptualmente cuyos elementos se computan cuando se recibe la solicitud correspondiente.

Cuando se emplea la interfaz `Collection`, es el usuario quien debe ocuparse de la iteración (por ejemplo, mediante `foreach`, bucle `for` mejorado); ese enfoque se denomina iteración externa.

En contraste, la biblioteca Streams recurre a la iteración interna; se ocupa de la iteración y de almacenar en algún lugar el valor del stream resultante; el usuario solo provee una función que dice qué debe hacerse. En el código de la **Secuencia 4** (iteración externa con una colección) y de la **Secuencia 5** (iteración interna con un stream) se ilustra esa diferencia.

```

List<String> transactionIds = new ArrayList<>();
for(Transaction t: transactions){
    transactionIds.add(t.getId());
}
  
```

Secuencia 4

```

List<Integer> transactionIds =
    transactions.stream()
        .map(Transaction::getId)
        .collect(toList());
  
```

Secuencia 5

En la **Secuencia 4**, generamos la iteración explícita del listado de transacciones secuencialmente para extraer la identificación de cada transacción y agregarla al acumulador. En cambio, cuando se usa un stream, no existe iteración explícita. Con el código de la **Secuencia 5** se crea una consulta en la que se ha parametrizado la operación `map` para extraer las identificaciones de transacciones, y la operación `collect` convierte el Stream resultante en un listado `List`.

En este punto, seguramente el desarrollador ya tenga una buena idea de qué son los streams y de para qué puede usarlos. Observemos ahora las diferentes operaciones que admiten los streams para expresar las propias consultas de procesamiento de datos.

Operaciones de streams: Cómo aprovechar los streams para procesar datos

La interfaz `Stream` de `java.util.stream.Stream` define varias operaciones que pueden agruparse en dos categorías. En el ejemplo de la **Figura 1** es posible identificar las siguientes operaciones:

filter, *sorted* y *map*, que pueden conectarse para formar un proceso;
collect, que cierra el proceso y devuelve un resultado.

Las operaciones de streams que pueden conectarse entre sí se llaman *operaciones intermedias*. Se pueden conectar porque la salida que devuelven es de tipo `Stream`. Las operaciones que cierran un proceso de stream se llaman *operaciones terminales*. A partir de un proceso producen un resultado de tipo `List`, `Integer` o incluso `void` (de tipos distintos de `Stream`).

¿Por qué es importante la distinción? Bien, las operaciones intermedias no llevan a cabo tareas de procesamiento hasta que se invoca una operación terminal en el proceso del stream; son "perezosas". Eso se debe a que a menudo la operación terminal puede "fusionar" y procesar diversas operaciones intermedias en una sola acción.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
numbers.stream()
    .filter(n -> {
        System.out.println("filtering " + n);
        return n % 2 == 0;
    })
    .map(n -> {
        System.out.println("mapping " + n);
        return n * n;
    })
    .limit(2)
    .collect(toList());
Secuencia 6
```

Por ejemplo, tomemos el código de la **Secuencia 6**, que calcula dos potencias pares a partir de un listado de números. Tal vez resulte sorprendente que muestre lo siguiente:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Eso se debe a que en `limit(2)` se *corta el circuito*; solo es necesario procesar parte del stream, no todo, para devolver un resultado. Una situación similar se da al evaluar una expresión booleana extensa encadenada con el operador `and`: tan pronto la expresión devuelve el valor `false`, es posible deducir que toda la expresión es `false` sin evaluarla en su totalidad. En este caso, `limit` devuelve un stream de tamaño 2.

La API de streams descompondrá internamente la consulta para aprovechar los núcleos múltiples de la computadora.

Además, las operaciones `filter` y `map` han sido fusionadas en una única acción.

Para resumir lo que hemos aprendido hasta ahora, cuando se usan streams en general se trabaja con tres elementos:

- origen de datos (p. ej., una colección) respecto del cual se hará una consulta;
- cadena de operaciones intermedias, que forman un proceso;
- operación terminal, que ejecuta el proceso del stream y produce un resultado.

Veamos ahora algunas operaciones que pueden usarse con streams. Pueden consultarse el listado completo en la interfaz `java.util.stream.Stream` u otros ejemplos en los recursos enumerados al final de este artículo.

Filtrado. Diversas operaciones pueden usarse para filtrar elementos de un stream:

filter(Predicate): Toma un predicado (`java.util.function.Predicate`) como argumento y devuelve un stream que incluye todos los elementos que coinciden con el predicado indicado.

distinct: Devuelve un stream con elementos únicos (según sea la implementación de `equals` para un elemento del stream).

limit(n): Devuelve un stream cuya máxima longitud es `n`.

skip(n): Devuelve un stream en el que se han descartado los primeros `n` números.

Búsquedas e identificación de coincidencias. Un patrón común en el procesamiento de datos consiste en determinar si algunos elementos se ajustan a una propiedad dada. Es posible usar las operaciones `anyMatch`, `allMatch` y `noneMatch` para lograr ese fin. Todas toman como argumento un predicado y devuelven un valor booleano (es decir que son operaciones terminales). Por ejemplo, se puede usar `allMatch` para verificar que todos los elementos de un stream de transacciones tengan valores superiores a 100, como se muestra en la **Secuencia 7**.

```
boolean expensive =
transactions.stream()
    .allMatch(t -> t.getValue() > 100);
Secuencia 7
```

Además, la interfaz de *Stream* incluye operaciones como `findFirst` y `findAny` para recuperar elementos arbitrarios de un stream. Pueden usarse en conjunto con otras operaciones de *stream*, tales como *filter*. Tanto `findFirst` como `findAny` devuelven un objeto *Optional*, como se muestra en la **Secuencia 8**.

```
Optional<Transaction> =
transactions.stream()
    .filter(t -> t.getType() == Transaction.GROCERY)
    .findAny();
Secuencia 8
```

Optional<T> (`java.util.Optional`) es una clase contenedora que representa la existencia o ausencia de un valor. En la **Secuencia 8**, puede suceder que `findAny` no encuentre ninguna transacción del tipo *grocery*. La clase *Optional* contiene diversos métodos para poner a prueba la existencia de un elemento.

Por ejemplo, si una transacción está presente, podemos optar por aplicar una operación sobre el objeto opcional usando el método *ifPresent*, como se muestra en la **Secuencia 9** (donde acabamos de mostrar la transacción).

```
transactions.stream()
    .filter(t -> t.getType() == Transaction.GROCERY)
    .findAny()
    .ifPresent(System.out::println);
Secuencia 9
```

Mapeo. Los streams admiten el método `map`, que emplea una función (`java.util.function.Function`) como argumento para proyectar los elementos del stream en otro formato. La función se aplica a cada elemento, que se "mapea" o asocia con un nuevo elemento.

Por ejemplo, podría ser necesario usarla para extraer información de cada elemento de un stream. En el ejemplo de la **Secuencia 10**, se devuelve una lista con las longitudes de todas las palabras de un listado.

Reducción. Las operaciones terminales que vimos hasta ahora devuelven objetos booleano (`allMatch` y similares), `void` (`forEach`) u `Optional` (`findAny` y similares). También hemos usado `collect` para combinar todo el conjunto de elementos de un `Stream` en un objeto `List`.

```
List<String> words = Arrays.asList("Oracle", "Java", "Magazine");
List<Integer> wordLengths =
words.stream()
.map(String::length)
.collect(toList());
Secuencia 10
```

Otra posibilidad es combinar todos los elementos de un stream para formular consultas de procesos más complicadas, como "¿cuál es la transacción con la identificación más alta?" o "calcular la suma de los valores de todas las transacciones". Para ello, se puede usar la operación `reduce` con streams; esta operación aplica reiteradamente una operación (por ejemplo, la suma de dos números) a cada elemento hasta que se genera un resultado. En el ámbito de la programación funcional se la suele llamar *operación fold* (de pliegue) porque se asimila a la acción de plegar repetidamente un largo trozo de papel (el stream) hasta que queda un pequeño cuadrado, el resultado de la operación de pliegue.

Es útil pensar primero cómo podríamos calcular la suma de los elementos de una lista con un bucle *for*:

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

Cada elemento de la lista de números se combina iterativamente empleando el operador de suma para generar un resultado. Esencialmente, hemos "reducido" la lista de números a uno solo. El código incluye dos parámetros: el valor inicial de la variable `sum`, en este caso 0, y la operación que combina todos los elementos de la lista, en este caso `+`.

Empleando el método *reduce* con un *stream*, podemos sumar todos los elementos de un stream como se muestra en la **Secuencia 11**. El método *reduce* lleva dos argumentos:

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
Secuencia 11
```

un valor inicial, acá 0;

un *BinaryOperator<T>* para combinar dos elementos y generar un nuevo valor.

Esencialmente, el método *reduce* extrae el patrón que se aplica repetidamente. Otras consultas, como "calcular el producto" o "calcular el máximo" (ver **Secuencia 12**) son casos especiales de uso del método *reduce*.

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
int product = numbers.stream().reduce(1, Integer::max);
Secuencia 12
```

Streams numéricos

Acabamos de ver cómo usar el método *reduce* para calcular la suma de un stream de números enteros. No obstante, ese enfoque tiene una desventaja: se llevan a cabo muchas operaciones de *boxing* para sumar repetidamente objetos `Integer`. ¿No sería mejor poder llamar a un método `sum`, como se muestra en la **Secuencia 13**, para ser más explícitos acerca de la intención con que concebimos nuestro código?

```
int statement =
transactions.stream()
.map(Transaction::getValue)
.sum(); // error since Stream has no sum method
Secuencia 13
```

Java SE 8 incorpora tres interfaces que transforman streams primitivos en especializados para abordar ese problema: *IntStream*, *DoubleStream* y *LongStream*; cada una de ellas convierte los elementos de un stream de manera especializada para que sean de tipo *int*, *double* o *long*, respectivamente.

Los métodos más habituales para convertir un stream en una versión especializada son `mapToInt`, `mapToDouble` y `mapToLong`. Estos métodos funcionan exactamente igual que el método `map` que vimos anteriormente, pero devuelven un stream especializado en lugar de un `Stream<T>`. Por ejemplo, podríamos mejorar el código de la **Secuencia 13** como se muestra en la **Secuencia 14**. También es posible convertir un stream primitivo en un stream de objetos mediante la operación `boxed`.

```
int statementSum =
transactions.stream()
.mapToInt(Transaction::getValue)
.sum(); // works!
Secuencia 14
```

Por último, otro formato útil de streams numéricos es el de intervalos numéricos. Por ejemplo, podríamos querer obtener todos los números entre 1 y 100. Java SE 8 incorpora dos métodos estáticos para *IntStream*, *DoubleStream* y *LongStream* que ayudan a generar esos intervalos: `range` y `rangeClosed`.

Ambos métodos toman el valor inicial del intervalo como primer parámetro y el valor final como segundo parámetro. Sin embargo, `range` es exclusivo mientras que `rangeClosed` es inclusivo. La **Secuencia 15** es un ejemplo del uso de `rangeClosed` para devolver un stream de todos los números impares entre 10 y 30.

```
IntStream oddNumbers =
IntStream.rangeClosed(10, 30)
.filter(n -> n % 2 == 1);
Secuencia 15
```

Creación de streams

Hay varias maneras de crear streams. Hemos visto cómo obtener un stream a partir de una colección. Más aún, hicimos pruebas con streams de números. También se pueden crear streams a partir de valores, matrices o archivos. Incluso se puede crear un stream a partir de una función para generar streams infinitos.

A diferencia del trabajo con colecciones, en que la iteración es explícita (*iteración externa*), las operaciones del stream llevan a cabo la iteración tras bambalinas.

Crear un stream a partir de valores o de una matriz es muy sencillo: solo deben usarse los métodos estáticos `Stream.of` para valores y `Arrays.stream` para una matriz, como se muestra en la **Secuencia 16**.

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);
int[] numbers = {1, 2, 3, 4};
IntStream numbersFromArray = Arrays.stream(numbers);
Secuencia 16
```

También se puede convertir un archivo en un stream de líneas con el método estático `Files.lines`. Por ejemplo, en la **Secuencia 17** se cuenta la cantidad de líneas de un archivo.

```
long numberOfLines =
Files.lines(Paths.get("yourFile.txt"), Charset.defaultCharset())
.count();
Secuencia 17
```

Streams infinitos. Por último, antes de cerrar este primer artículo sobre streams, presentamos una posibilidad extraordinaria. A esta altura, seguramente se haya comprendido que los elementos de un stream se generan cuando se recibe la solicitud correspondiente. Existen dos métodos estáticos —`Stream.iterate` y `Stream.generate`— que permiten crear un stream a partir de una función. Sin embargo, como esos elementos se

calculan cuando se recibe una solicitud, las dos operaciones pueden generar elementos "infinitamente". Eso es lo que llamamos *stream infinito*: un stream que no tiene un tamaño delimitado, a diferencia de lo que ocurre cuando el stream se crea a partir de una colección fija.

La Secuencia 18 es un ejemplo del uso de *iterate* para crear un stream de todos los múltiplos de 10. El método *iterate* lleva un valor inicial (acá, 0) y una expresión *lambda* (de tipo `UnaryOperator<T>`) para la aplicación sucesiva a cada nuevo valor que se genere.

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);
Secuencia 18
```

Es posible convertir un stream infinito en un stream de tamaño fijo con la operación *limit*. Por ejemplo, podemos limitar el tamaño del stream a 5, como se muestra en la **Secuencia 19**.

```
numbers.limit(5).forEach(System.out::println); // 0, 10, 20, 30, 40
Secuencia 19
```


Conclusión

Java SE 8 ha incorporado la API de *streams*, que permite expresar sofisticadas consultas de procesamiento de datos. En este artículo, hemos visto que los streams admiten diversas operaciones, como *filter*, *map*, *reduce* e *iterate* que pueden combinarse para generar consultas de procesamiento de datos concisas y expresivas. Esta nueva forma de programar es muy distinta del modo en que se procesaban las colecciones antes de Java SE 8; pero ofrece muchas ventajas. En primer lugar, la API de streams aprovecha diversas técnicas como la "pereza" y el "corte de circuitos" para optimizar las consultas de procesamiento de datos. En segundo lugar, los streams pueden emplearse en paralelo automáticamente para aprovechar las arquitecturas de núcleos múltiples. En el próximo artículo de esta serie, exploraremos operaciones más avanzadas, como *flatMap* y *collect*.

¡Hasta pronto!

Raoul-Gabriel Urma está terminando su doctorado en Ciencias de la Computación en la Universidad de Cambridge, donde desarrolla su investigación en lenguajes de programación. Asimismo, es autor de *Java 8 in Action: Lambdas, Streams, and Functional-style Programming* (Manning, 2014).

Este artículo ha sido revisado por el equipo de productos Oracle y se encuentra en cumplimiento de las normas y prácticas para el uso de los productos Oracle.

 E-mail this page  Printer View

Contáctenos

Ventas en EE. UU.:
+1.800.633.0738
Reciba una llamada de Oracle
Contactos globales
Directorio de soporte

Acerca de Oracle

Información de la empresa
Comunidades
Carreras

Cloud

Descripción general de Cloud
Solutions
Software (SaaS)
Plataforma (PaaS)
Infraestructura (IaaS)
Datos (DaaS)
Prueba gratuita de la nube

Eventos

Oracle Code
JavaOne
Todos los eventos de Oracle

Acciones principales

Descargue Java
Descargue Java para
desarrolladores
Pruebe Oracle Cloud
Suscríbese a los correos
electrónicos

Noticias

Sala de prensa
Revistas
Historias de éxito de los clientes
Blogs

Temas clave

ERP, EPM (Finanzas)
HCM (RR. HH. y talento)
Mercadeo
CX (Ventas, servicio, comercio)
Soluciones sectoriales
Base de datos
MySQL
Middleware
Java
Sistemas de ingeniería