Introducción a Expresiones Lambda y API Stream en Java SE 8 – Parte 2
Por Alexis Lopez
Publicado en Octubre 2015

Las características más importantes de Java SE 8 son la adición de **Expresiones Lambda** y la **API Stream**. Con la adición de expresiones lambda podemos crear código más conciso y significativo, además de abrir la puerta hacia la programación funcional en Java, en donde las funciones juegan un papel fundamental. Por otro lado, la API Stream nos permite realizar operaciones de tipo filtro/mapeo/reducción sobre colecciones de datos de forma secuencial o paralela y que su implementación sea transparente para el desarrollador. Lambdas y Stream son una combinación muy poderosa que requiere un cambio de paradigma en la forma en la que hemos escrito código Java hasta el momento.

En la primera parte, describimos por qué la necesidad de expresiones lambda, su sintáxis y funcionamiento, así como las adiciones y cambios al lenguaje que soportan esta nueva característica.

En esta segunda parte, mostraremos el uso de la API Stream y la combinación ganadora Lambdas + Stream.

API Stream
Stream se define como una secuencia de elementos que provienen de una fuente que soporta operaciones para el procesamiento de sus datos:

De forma declarativa usando expresiones lambda.
Permitiendo el posible encadenamiento de varias operaciones, con lo que se logra tener un código fácil de leer y con un objetivo claro.
De forma secuencial o paralela (Fork/Join).
Las estructuras que soportan esta nueva API se encuentran en el paquete *java.util.stream* y en especial, la interface *java.util.stream.Stream* define un Stream.

La API nos permite realizar operaciones sobre colecciones de datos usando el modelo filtro/mapeo/reducción, en el cual se seleccionan los datos que se van a procesar (filtro), se convierten a otro tipo de dato (mapeo) y al final se obtiene el resultado deseado (reducción).

El siguiente ejemplo nos permite visualizar las tres (3) partes que componen un Stream:

Fuente de información → La lista de transacciones
Cero o más operaciones intermedias → Se puede apreciar la operación **filter** y la operación **mapToInt**, operaciones que serán explicadas más adelante en este mismo artículo.
Operación terminal que produce un resultado o un efecto sobre los datos → Se puede apreciar la operación **sum**, la cual se explicará más adelante en este mismo artículo.

```
List transacciones = ...  int  sum = transacciones.stream().
     filter(t ->  t.getProveedor().getCiudad().equals("Cali")).
     mapToInt(Transaccion::getPrecio).
     sum();
```

Ahora veamos algunas de las propiedades y características de un Stream:

Las operaciones intermedias retornan otro Stream. Esto permite que podamos hacer un encadenamiento de operaciones .
Las operaciones intermedias son encoladas hasta que una operación terminal sea invocada. Es muy importante tener esto claro, ninguna de las operaciones intermedias es ejecutada hasta que se invoca una operación terminal.
Un Stream puede ser recorrido una sola vez, intentar recorrerlo de nuevo generará una excepción del tipo *IllegalStateException*.
Al usar esta API estamos cambiando el paradigma **Iteración externa** vs. **Iteración interna**. En iteración interna, ésta sucede automáticamente permitiendo al desarrollador enfocarse en qué hacer con los datos y no en cómo hacerlo. Por otro lado, permite "esconder" complejidades y permite que la iteración secuencial o paralela sea transparente para el desarrollador.
Existen versiones "primitivas" de Stream que evitan el Autoboxing y Unboxing innecesario, mejorando el desempeño de las aplicaciones:
java.util.stream.DoubleStream → Stream cuyos elementos son tipos de dato double.
java.util.stream.IntStream → Stream cuyos elementos son tipos de dato int.
java.util.stream.LongStream → Stream cuyos elementos son tipos de dato long.
¿Cómo obtener instancias de Stream?
Existen varias formas de obtener instancias de un *Stream*, a continuación veremos algunas de ellas:

Stream.of(T...): Stream<T>
Retorna un Stream ordenado y secuencial de los elementos pasados por parámetro.

```
Stream orquestas = Stream.of("Grupo Niche",  "Guayacán", "Son de Cali");
```

Stream.empty():Stream
Retorna un Stream secuencial y vacío.

Arrays.stream(T[]):Stream<T>
Retorna un Stream secuencial del arreglo pasado por parámetro. Si T[] es un arreglo de datos "primitivos" entonces retorna: *DoubleStream*, *IntStream* o *LongStream* según el caso.

```
int[]  enteros = new int[]{1,2,3,4,5};
IntStream  streamEnteros = Arrays.stream(enteros);
```

Collection<E>.stream():Stream<E>
Retorna un Stream secuencial de los elementos de la colección, para obtener una versión en paralelo basta con usar:
*Collection<E>.**parallelStream()**:Stream<E>*

```
List<String>  canciones = ...;
Stream<String>  streamCanciones = canciones.stream();
```

Stream.iterate(T, UnaryOperator<T>):Stream<T>
Retorna un Stream infinito, ordenado y secuencial a partir del valor inicial T y de aplicar la función pasada por parámetro *UnaryOperator* al valor inicial para obtener los demás elementos. Para limitar su tamaño, se puede usar el método *+limit(long):Stream*

```
//Primeros 10 números impares positivos iniciando en  el número 1
Stream  impares = Stream.iterate(1, x -> x + 2).limit(10);
```

Stream.generate(Supplier<T>):Stream<T>
Retorna un Stream infinito, secuencial pero no ordenado, a partir de una función de tipo *Supplier* que provee los elementos.

Operaciones sobre colecciones de datos
Ahora que sabemos de qué trata la API Stream y cómo podemos crear/obtener objetos de tipo *java.util.stream.Stream*, es el momento de que veamos lo que podemos lograr con esta nueva API a la hora de realizar operaciones sobre colecciones de datos.

Iniciamos con la selección de los datos. Para filtrar los elementos de un Stream podemos usar los siguientes métodos:

+filter(Predicate<T>):Stream<T>
Retorna un Stream que contiene sólo los elementos que cumplen con el predicado pasado por parámetro.

```
List<String> ciudades = ...

//Stream de ciudades cuya primera letra es C de Cali
Stream  stream = ciudades.stream()
        .filter(s ->  s.charAt(0) == 'C');
```

+distinct():Stream<T>
Retorna un Stream sin elementos duplicados. Depende de la implementación de *+equals(Object):boolean*.

```
List<String> ciudades =  Arrays.asList("Cali", "Bogotá", "Medellín",  "Cali");

//Stream sin ciudades repetidas: Cali, Bogotá,  Medellín
Stream  stream = ciudades.stream().distinct();
```

+limit(long):Stream<T>
Retorna un Stream cuyo tamaño no es mayor al número pasado por parámetro. Los elementos son cortados hasta ese tamaño.

```
List<String> ciudades =  Arrays.asList("Cali", "Bogotá", "Medellín");

//Stream limitado a los dos primeros elementos: Cali,  Bogotá
Stream  stream = ciudades.stream().limit(2);
```

+skip(long):Stream<T>
Retorna un Stream que descarta los primeros N elementos, donde N es el número pasado por parámetro. Si el Stream contiene menos elementos que N, entonces retorna un Stream vacío.

```
List<String> ciudades =  Arrays.asList("Cali", "Bogotá", "Medellín");

//Stream que ha saltado los dos primeros elementos,  quedando solo: Medellín
Stream  stream = ciudades.stream().skip(2);
```

El siguiente paso es el mapeo de datos. Una vez hemos realizado la selección (filtro), podemos transformar los elementos de un Stream al extraer información de éstos. Para lograrlo usamos alguno de los siguientes métodos:

+map(Function<T, R>): Stream<R>
Retorna un Stream que contiene el resultado de aplicar la función pasada por parámetro a todos los elementos del Stream. Transforma los elementos del tipo T al tipo R.

El siguiente ejemplo muestra cómo transformar un Stream de cadenas a otro Stream de esas mismas cadenas pero en mayúsculas. La función usada en este caso es una función (expresión lambda) dónde T y R son de tipo *java.lang.String*:

```
List<String>  paises = Arrays.asList("Colombia", "Perú",  "Panamá");

//Stream cuyos elementos son los países en mayúsculas

Stream<String>  stream = paises.stream().map(String::toUpperCase);
```

El anterior método también existe en sus versiones “primitivas”:

+mapToDouble(ToDoubleFunction<T>): DoubleStream
Transforma un Stream<T> a un DoubleStream por medio de la función pasada por parámetro.

```
List<Transaccion> trxs = ...

//Stream de decimales cuyos elementos son el valor de  las transacciones
DoubleStream  stream = trxs.stream().mapToDouble(Transaccion::getValor);
```

+mapToInt(ToIntFunction<T>): IntStream
Transforma un Stream<T> a un IntStream por medio de la función pasada por parámetro.

```
List<String> paises =  Arrays.asList("Colombia", "México", "Guatemala");

//Stream de enteros cuyos elementos son el num de  caracteres de los países
IntStream  stream = paises.stream().mapToInt(String::length);
```

+mapToLong(ToLongFunction<T>): LongStream
Transforma un Stream<T> a un LongStream por medio de la función pasada por parámetro.

```
List<Long> longs = …

//Stream cuyos elementos son la versión primitiva de  la lista definida arriba
LongStream  stream = longs.stream().mapToLong(Long::longValue);
```

La ventaja de usar las versiones primitivas radica en que se evita el uso de Autoboxing y Unboxing, lo que en algunas situaciones puede ser deseado por temas de rendimiento.

+flatMap(Function<T, Stream<R>):Stream<R>
Permite transformar cada elemento del Stream en otro Stream y al final concatenarlos todos en uno solo. Es un poco confuso, pero pensemos que cuando tengamos un Stream<Stream<R>>, este método nos permitirá transformarlo en Stream<R>.

En el siguiente ejemplo, vamos a crear un Stream de las distintas palabras de una lista de cadenas:

```
List<String> lista = Arrays.asList("Taller", "Taller Lambdas y API  Stream");
Stream stream = lista.stream()
               .map(s ->  s.split(" ")) // Stream<String[]>
               .map(Arrays::stream) //  Stream<Stream<String>>
               .flatMap(Function.identity()) // Stream<String>
               .distinct(); //  Stream<String> de 5 elementos
```

Nótese que primero obtuvimos las palabras de cada cadena, esa operación nos deja con un Stream de arreglos de cadenas. Luego invocamos el método estático *+Arrays.stream(T[]):Stream<T>* el cual nos deja con un Stream de Stream de cadenas. Es en ese momento donde *flatMap* es de

gran utilidad, ya que nos permite concatenar todos esos Streams en uno solo.

Para efectos de claridad, el ejemplo anterior primero obtiene un Stream<Stream<String>> y luego usa *flatmap* para quedar solo con Stream<String>. Una versión más directa del ejemplo anterior se puede escribir como sigue a continuación:

```
List<String>  lista = Arrays.asList("Taller", "Taller Lambdas y API  Stream");
Stream stream = lista.stream()
              .map(s -> s.split(" ")) //  Stream<String[]>
              .flatMap(Arrays::stream)  // Stream<String>
              .distinct(); //  Stream<String> de 5 elementos
```

El anterior método también existe en sus versiones "primitivas":

+flatMapToDouble(Function<T, DoubleStream): DoubleStream
+flatMapToInt(Function<T, IntStream): IntStream
+flatMapToLong(Function<T, LongStream): LongStream

Nuevamente, la ventaja de usar las versiones primitivas radica en que se evita el uso de Autoboxing y Unboxing, lo que en algunas situaciones puede ser deseado por temas de rendimiento.

El último paso son las operaciones terminales, las cuales provocan que todas las operaciones intermedias sean ejecutadas y pueden catalogarse así:

Operaciones terminales cuyo propósito es el consumo de los elementos del Stream, por ejemplo: *+foreach(Consumer<T>):void*
Operaciones terminales que permiten obtener datos del Stream como: conteo, mínimo, máximo, búsqueda, y en general reducir el Stream a un valor.
Operaciones terminales que permiten recolectar los elementos de un Stream en estructuras mutables como por ejemplo listas.
A continuación revisaremos algunas de las operaciones terminales de las dos últimas categorías y ofreceremos ejemplos que ayuden a entender de una mejor manera su funcionamiento.

Entre las operaciones terminales que permiten obtener datos del Stream tenemos:

+count():long
Retorna la cantidad de elementos en el Stream . En el siguiente ejemplo, vemos como podemos obtener la cantidad de transacciones que existen en el Stream cuyo valor es mayor a dos mil:

```
List<Transaccion> trxs = ...
 long  count = trxs.stream()
             .filter(t -> t.getValor()  > 2000)
             .count();
```

+max(Comparator<T>):Optional<T>
Retorna el elemento máximo del Stream basado en el comparador pasado por parámetro. Nótese que el retorno es de tipo *java.util.Optional*, clase nueva en Java SE 8 que representa un contenedor que puede o no tener un valor no-nulo. El siguiente ejemplo hace uso de un nuevo método estático de la interface *java.util.Comparator*, el cual extrae el valor de un atributo de tipo *java.lang.Double* de los elementos del Stream y retorna un comparador que hace uso de dicho atributo para hacer la comparación:

```
List<Transaccion> trxs = ...
Optional<Transaccion>  max =
         trxs.stream()
             .max(Comparator.comparingDouble(Transaccion::getValor));
```

+min(Comparator<T>):Optional<T>
Retorna el elemento mínimo del Stream basado en el comparador pasado por parámetro. Nótese que el retorno es de tipo Optional (por si acaso el Stream se encuentra vacío).

También contamos con operaciones terminales que permiten realizar búsquedas entre los elemento de un Stream:

+allMatch(Predicate<T>):boolean
Verifica si todos los elementos del Stream satisfacen el predicado pasado por parámetro. Si durante la verificación alguno no lo cumple entonces se detiene la verificación y retorna falso, es decir, no requiere procesar todo el Stream para producir el resultado. El siguiente ejemplo revisa que el tamaño de todas las cadenas de texto sea de al menos 4 caracteres. Nótese que el predicado lo hemos pasado como una expresión lambda:

```
List<String>  palabras = Arrays.asList("Java", "Lambdas",  "Stream", "API");

//Verifica si todas las palabras tienen un tamaño de 4  caracteres
boolean  longitud = palabras.stream().allMatch(s -> s.length() >= 4);
```

+anyMatch(Predicate<T>):boolean
Verifica si alguno de los elementos del Stream satisface el predicado pasado por parámetro. Si durante la verificación alguno lo cumple entonces se detiene la verificación y retorna verdadero, es decir, no requiere procesar todo el Stream para producir el resultado. En el siguiente ejemplo buscaremos un elemento en el Stream usando una expresión lambda como predicado:

```
List<String>  palabras = Arrays.asList("Java", "Lambdas",  "Stream", "API");

//Verifica si existe la cadena "lambda" dentro del  Stream
boolean  anymatch = palabras.stream()
               .anyMatch(s ->  s.equalsIgnoreCase("lambda"));
```

+noneMatch(Predicate<T>):boolean
Contrario a *+allMatch(Predicate<T>):boolean*, verifica si todos los elementos del Stream NO satisfacen el predicado pasado por parámetro. Si alguno SÍ lo cumple entonces se detiene la verificación y retorna falso, es decir, no requiere procesar todo el Stream para producir el resultado.

+findAny():Optional<T>
Retorna algún elemento del Stream. Se recomienda usar este método cuando no se requiera un orden o cuando se esté usando Streams en paralelo. Nótese que el retorno es de tipo Optional (por si acaso el Stream se encuentra vacío).

```
//Obtiene alguno de los elementos del Stream
Optional<String>  alguno = palabras.stream().findAny();
```

+findFirst():Optional<T>
Retorna el primer elemento del Stream. No se recomienda usar este método cuando se tengan Streams en paralelo, debido a que obligaría a la API a sincronizar los hilos para no perder el orden de los elementos. Nótese que el retorno es de tipo Optional (por si acaso el Stream se encuentra vacío).

En general, si queremos reducir un Stream a un valor, podemos hacer uso del método mostrado a continuación, pero se aclara que existe una forma aún más genérica de reducir un Stream que se sale del alcance de este artículo:

+reduce(BinaryOperator<T>):Optional<T>
Realiza la reducción del Stream usando una función asociativa. Nótese que el retorno es de tipo Optional (por si acaso el Stream se encuentra vacío). En el siguiente ejemplo, obtenemos el mayor entero par que se encuentra en el Stream. La función que pasamos como parámetro usa el nuevo método estático *+Integer.max(int, int):int*

```
List<Integer> numeros = …

//Obtiene el posible número mayor par del Stream
Optional<Integer>  opt = numeros.stream()
                .filter(x -> x % 2 == 0)
                .reduce(Integer::max);
```

+reduce(T, BinaryOperator<T>):T
Realiza la reducción del Stream usando un valor inicial y una función asociativa. A continuación sumaremos todos los elementos del Stream usando el valor inicial 0, si el Stream se encuentra vacío, ese sería nuestro resultado:

```
List<Integer> numeros = …

//Obtiene la suma de los elementos del Stream
Integer suma = numeros
        .stream()
        .reduce(0, (x,y) -> x + y);
```

La otra categoría de operaciones terminales nos permite recolectar los elementos de un Stream en estructuras mutables como por ejemplo listas o maps. Para ello usaremos algo llamado *collect* que es una operación terminal que permite recolectar los elementos de un Stream en dichas estructuras.

Java SE 8 introduce una nueva clase utilitaria llamada *java.util.stream.Collectors* que provee métodos estáticos que retornan los recolectores más usados. Dichos recolectores pueden ser agrupados en 3 tipos:

Reducción y resúmen: Reducen el Stream y permite obtener valores agregados como cantidad de elementos, sumas, promedios, etc.
Agrupamiento: Agrupa elementos en un Map usando una "función de clasificación", que permite establecer a qué grupo pertenece cada elemento.
Particionamiento: Agrupamiento donde la función de clasificación es un predicado y por lo tanto agrupa los elementos en un Map de dos llaves: false y true
Nota: Para facilitar la lectura de código, es mejor hacer la importación estática de la clase utilitaria:

```
Import  static java.util.stream.Collectors.*;
```

A continuación algunos recolectores que permiten obtener información agregada de un Stream:

Counting
Recolector que realiza la cuenta de elementos de un Stream. Es fácil pensar que ya existe una operación terminal que realiza la cuenta de elementos de un Stream (*+count():long)* y que no es necesario tener otra. Sin embargo, el tener esta operación como un recolector, nos permitirá usarla en conjunto con otros recolectores como veremos más adelante:

```
Import  static java.util.stream.Collectors.*;
 ...
 List<Integer> numeros = …

//Cantidad de elementos en el Stream
long  cuenta = numeros.stream().collect(counting());
```

MaxBy, MinBy
Estos recolectores permiten obtener el elemento máximo y mínimo respectivamente. Requieren que se les pase como parámetro un comparador que permita realizar las comparaciones. Su retorno es de tipo Optional por si acaso el Stream se encuentra vacío. En el siguiente ejemplo vamos a obtener el máximo entero del Stream de acuerdo a un comparador que compara enteros basado en su orden natural:

```
Import  static java.util.stream.Collectors.*;
...
List<Integer> numeros = …

//Máximo elemento en el Stream de acuerdo al  comparador
Optional<Integer>  max = numeros.stream()
                .collect(maxBy(Comparator.naturalOrder()));
```

SummingInt, SummingDouble, SummingLong
Recolectores que permiten realizar la suma de los elementos del Stream. Requieren que se les pase una función de tipo Supplier (es decir de las que retornan el valor requerido) para obtener el valor a sumar. En el siguiente ejemplo podemos ver que la función que pasamos como parámetro obtiene el valor entero del elemento que queremos sumar:

```
Import  static java.util.stream.Collectors.*;
 ...
 List<Integer> numeros = …

//Obtiene la suma los elementos del Stream como un  entero
int  suma = numeros
        .stream()
        .collect(summingInt(x -> x.intValue()));
```

AveragingInt, AveragingDoble, AveragingLong
Recolectores que permiten obtener el promedio de los elementos del Stream. Tambień requieren que se les pase una función de tipo Supplier (es decir de las que retornan el valor requerido) para obtener el valor a promediar.

SummarizingInt, SummarizingDouble, SummarizingLong
Recolectores que nos retornan instancias de clases que agrupan la información agregada del Stream. Tambień requieren que se les pase una función de tipo Supplier (es decir de las que retornan el valor requerido) para obtener el valor a procesar. Las instancias que retornan son de tipo IntSummaryStatistics, DoubleSummaryStatistics o LongSummaryStatistics.

```
Import  static java.util.stream.Collectors.*;
...
List<Integer>  numeros = …

IntSummaryStatistics res = numeros
         .stream()
         .collect(summarizingInt(Integer::intValue));
```

Joining
Recolector que nos permite concatenar en una sola cadena, todas las cadenas retornadas por *+toString():String* de cada elemento del Stream. También es posible pasar como parámetro el delimitador que queremos usar entre elementos:

```
Import  static java.util.stream.Collectors.*;
...
List<Integer>  numeros = …

String  csv = numeros.stream().map(Object::toString)
        .collect(joining(", "));
```

Existe una versión genérica que permite crear todos los recolectores que hemos visto hasta el momento, pero que se sale del alcance de este artículo. Sin embargo, si el lector desea indagar más al respecto, existe un recolector que se puede obtener invocando *Collectors.reducing*.

Ahora veremos cómo podemos agrupar los elementos de un Stream basándonos en una función que nos devuelve el grupo al que pertenece cada elemento. El siguiente ejemplo permite agrupar empleados de acuerdo a su departamento. Para este caso, la función de clasificación retorna el nombre del departamento al cual pertenece el empleado:

```
Import  static java.util.stream.Collectors.*;
…
List<Empleado>  empleados = …

//  Agrupamiento
Map<String, List<Empleado>> porDept = empleados
        .stream()
        .collect(groupingBy(Empleado::getDepartmento));
```

Por defecto, el recolector *Collectos.groupingBy* agrupa sus elementos en un Map dónde la llave es el valor retornado por la función de clasificación y los valores son listas que contienen los elementos de ese grupo. Las llaves son adicionadas al Map únicamente si existen elementos de ese grupo.

También es posible pasar otro recolector como parámetro, lo que nos permitiría obtener un valor diferente a una lista o establecer un segundo nivel de agrupamiento. En el ejemplo a continuación, queremos saber cuántos empleados hay en cada departamento:

```
Import  static java.util.stream.Collectors.*;
…
List<Empleado>  empleados = …

//  Agrupamiento
Map<String, Long> deptCant = empleados
        .stream()
        .collect(groupingBy(Empleado::getDepartmento),counting());
```

Y si lo deseamos, podemos pasar otro *Collector.groupingBy* como parámetro y obtendremos un segundo nivel de agrupamiento. En el ejemplo que sigue, vamos a agrupar los empleados primero por ciudad y luego por departamento al que pertenecen:

```
Import  static java.util.stream.Collectors.*;
…
List<Empleado>  empleados = …

//  Agrupamiento
Map<String, Map<String, List<Empleado>>> dptoCiu = null;
dptoCiu  = empleados
        .stream()
        .collect(groupingBy(Empleado::getDepartamento,
        groupingBy(Empleado::getCiudad)));
```

El caso de particionamiento es un caso especial de agrupamiento, en el cuál la función de clasificación es un predicado, lo que nos reduce la cantidad de grupos a los cuáles pertenecen los elementos del Stream a dos: true o false.

Es posible crear recolectores que realicen operaciones de reducción de Streams si creamos instancias de la interface *java.util.stream.Collector*. Dejamos al lector la tarea de profundizar en este tema.

Depuración
Una forma de hacer depuración de aplicaciones que usen lambas y Streams es usando el método *+peek(Consumer<T>):Stream<T>* , el cuál es una operación intermedia y por lo tanto no interrumpe el procesamiento del Stream. Cada elemento es pasado al *Consumer* y es allí cuando podemos hacer la depuración bien sea colocando un punto de depuración en la llamada a peek o implementando métodos de referencia y colocando puntos de depuración dentro de dichos métodos. Es importante recordar que no se deben modificar los elementos del Stream cuando se use este método.

```
List<String>  palabras = ...
 List<String> unicas = palabras.stream()
              .flatMap(w -> Stream.of(w.split(“ “)))
              //Colocar punto de  depuración
              .peek(s -> s.toString())
              .map(String::toLowerCase)
              //Punto de depuración dentro del método

              .peek(s -> metodoReferencia(s))
              .distinct()
              .collect(Collectors.toList());
```

Streams Paralelos
Todo lo que hemos visto hasta el momento aplica tanto para Streams secuenciales como paralelos. A continuación vamos a revisar algunas propiedades de Streams en paralelo que nos permitirán entender un poco más su funcionamiento:

Usan Fork/Join internamente para realizar la separación y asignación de tareas. Esto lo obtiene el desarrollador sin implementar una sola línea de código.
Usar *+parallel():Stream* para convertir a paralelo. Es así de sencillo, por medio de este método podemos convertir un Stream secuencial a uno paralelo. Usar *+sequential():Stream* para convertir a secuencial.
La última llamada a *+parallel():Stream* o *+sequential():Stream* es la que se tiene en cuenta. Siendo operaciones intermedias, éstas pueden ser llamadas muchas veces, pero es solo la última la que se tiene en cuenta.

¿Cuántos hilos se ejecutan en paralelo? Los que se tengan disponibles para la máquina virtual de acuerdo a *Runtime.getRuntime.availableProcessors()* . La propiedad de máquina virtual: "java.util.concurrent.ForkJoinPool.common.parallelism" modifica este valor. Sin embargo, actualmente no hay forma de modificar el valor y que esto aplique solo para algunos Stream.
El uso de Streams en paralelo requiere más trabajo internamente, por lo que no siempre terminará más rápido que Streams secuenciales.
Entonces, ¿cuándo usar Streams en paralelos? Debemos tener en cuenta las siguientes consideraciones si se desea usar Streams en paralelo:

Medir primero antes de hacer el cambio. Una forma de medir es usando la siguiente ecuación:

N = Número de items
Q = Costo de procesar un item
N*Q = Costo total de las operaciones.
Mientras más grande sea el valor de N*Q, mejor usar Streams paralelos .

Evitar el Autoboxing y Unboxing . Esto puede provocar deterioro en el rendimiento de los Streams, por lo que antes de pasar a Streams en paralelo se debe revisar que no se estén haciendo dichas operaciones o usar Streams primitivos.

Algunas operaciones no se comportan bien en paralelo debido a la distribución de tareas ente los diferentes hilos y por ende es mejor no hacer uso de ellas:

+limit(long):Stream<T>
+findFirst():Optional<T>
+sorted():Stream<T>
+distinct():Stream<T>
Por último, es mejor usar estructuras que sean fáciles de descomponer :

ArrayList -> Exelente
HashSet, TreeSet -> Bien
LinkedList -> Mal

Conclusión

En esta segunda parte analizamos el uso de la API Stream y pudimos observar la combinación Lambdas + Streams para realizar operaciones sobre colecciones de datos. Sin embargo, existen operaciones adicionales que invitamos al lector a revisar en la documentación oficial de la API.

Definimos Streams, sus propiedades y cómo obtener instancias de éstos.
Revisamos diferentes operaciones para filtrar elementos de un Stream (filtrado).
Revisamos operaciones para hacer transformación de elementos de un Stream (mapeo).
Revisamos los tipos de operaciones terminales que nos permiten obtener diferentes resultados de un Stream (reducción): Consumo de elementos, Obtener datos del Stream, Recolectar los elementos en otras estructuras.
Presentamos una forma de hacer depuración a aplicaciones que usan Lambdas + Streams por medio del método *+peek(Consumer<T>):Stream<T>* y el uso de métodos de referencia.
Analizamos propiedades y consideraciones a tener en cuenta a la hora de usar Streams en paralelo. No siempre un Stream en paralelo será más eficiente que un Stream secuencial. Regla de oro: Medir primero.
Para que puedas practicar lo aprendido en las dos partes de este artículo, he creado un laboratorio en GitHub que está basado en pruebas unitarias. La idea es implementar el cuerpo de diferentes métodos para que las pruebas sean exitosas:

https://github.com/aalopez/lambdas-lab

Información Adicional

Los siguientes enlaces ofrecen mayor información respecto a este tema:

Primera parte de este artículo
http://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-api-stream-java-2633852-esa.html

Tutorial de expresiones lambda creado por Oracle (en inglés):
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Documentación API Java SE 8
https://docs.oracle.com/javase/8/docs/api/

Libro recomendado: Java 8 in Action (en inglés)
http://www.manning.com/urma/

Alexis Lopez (@aa_lopez) es consultor independiente Java/ADF/BPM. Ha sido profesor universitario de cursos relacionados con Java y conferencista en congresos reconocidos como: Oracle Open World, JavaOne, Campus Party y OTN Tour. Cuenta con un título de ingeniero de sistemas y las siguientes certificaciones: SCJP, OCPJMAD, OCPWCD, especialista de implementación de Oracle ADF y Oracle BPM. Es líder del grupo de usuarios Java de Cali-Colombia (www.clojug.org), miembro del comité de dirección del grupo de usuarios virtual de Java (virtualjug.com) y blogger activo en www.java-n-me.com