Lambdas, Streams, Date/Time API

# Lambdas

## Definición

Las expresiones lambda se introdujeron en Java 8 y se consideran la característica más importante de Java 8. Las expresiones lambda facilitan la programación funcional y simplifican mucho el desarrollo.

Esencialmente, son funciones anónimas, que no necesitan una clase y que trabajan basadas en una interfaz. Son básicamente un objeto que se compone de un conjunto de parámetros y un bloque de código a ejecutar.

Una expresión lambda se caracteriza por la siguiente sintaxis.

* Declaración de tipo es opcional - No es necesario declarar el tipo de un parámetro. El compilador puede inferir el mismo a partir del valor del parámetro.
* Paréntesis opcional alrededor del parámetro - No es necesario declarar un solo parámetro entre paréntesis. Para múltiples parámetros, los paréntesis son necesarios.
* Corchetes son opcionales - No es necesario usar corchetes en el cuerpo de la expresión si el cuerpo contiene una sola declaración.
* Palabra clave return es opcional - El compilador devuelve automáticamente el valor si el cuerpo tiene una única expresión para devolver el valor. Las llaves son necesarias para indicar que la expresión devuelve un valor.

Ejemplos:

1. 1 parámetro

(arg) -> { body }

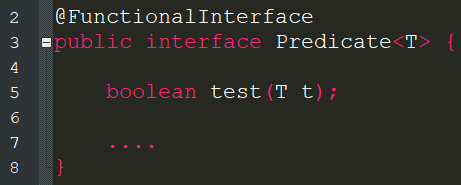
1. Varios parámetros que resultan ser el mismo tipo de objeto.

(arg1, arg2...) -> { body }

1. Varios parámetros que resultan ser diferentes tipos de objetos.

(type1 arg1, type2 arg2...) -> { body }

Los Lambdas deben corresponderse con una interfaz funcional, las cuales se encuentra en el paquete java.util.function. Esta interfaz solo debe tener un método abstracto e indicarse la anotación @FunctionalInterface.



## Interfaces funcionales

Las interfaces funcionales tienen la única funcionalidad de exhibir. Por ejemplo, una interfaz Comparable con un único método 'compareTo' se utiliza para fines de comparación. Java 8 ha definido una gran cantidad de interfaces funcionales que se utilizan ampliamente en las expresiones lambda. A continuación se muestra la lista de interfaces funcionales definidas en el paquete java.util.Function. Todas las interfaces funcionales del paquete pueden agruparse en 3 categorías:

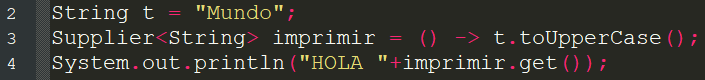
* Predicate

La interfaz Predicate<T> se utiliza para evaluar una condición booleana mediante el uso de tu método test(). Ejemplo:



* Supplier

La interfaz Supplier<T> se encarga de ejecutar alguna operativa sin recibir argumentos utilizando para ello el método get(). Ejemplo:

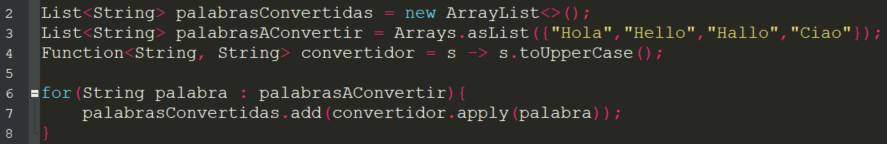


* Consumer

La interfaz Consumer<T> se encarga de realizar una acción en base a unos argumentos suministrados mediante el uso del método accept().

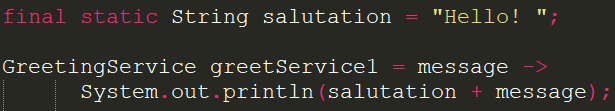


Este tipo de interfaz es el más utilizado y uno de sus usos más comunes es para la conversión de datos. Esto lo realiza mediante la generación de una función que recibe un dato, lo trata y su resultado es almacenado en una variable destino, como en el siguiente ejemplo:



## Scope

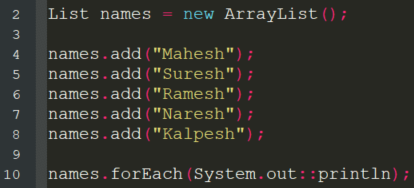
Usando la expresión lambda, puedes referirte a cualquier variable final o efectivamente final (que se asigna sólo una vez). La expresión lambda arroja un error de compilación si a una variable se le asigna un valor la segunda vez. Ejemplo:



## Referencia a métodos

Las referencias a métodos ayudan a señalar los métodos por sus nombres. Una referencia a un método se describe con el símbolo "::" y puede utilizarse para señalar los siguientes tipos de métodos:

* Métodos estáticos
* Métodos de instancia
* Constructores que utilizan el operador new (TreeSet::new)



# Streams

Stream representa una secuencia de objetos de una fuente que soporta operaciones de agregación. A continuación, algunas características de un flujo:

* **Secuencia de elementos** - Un flujo proporciona un conjunto de elementos de tipo específico de forma secuencial. Un flujo obtiene/calcula elementos bajo demanda. Nunca almacena los elementos.
* **Fuente** - El flujo toma Colecciones, Arrays, o recursos de E/S como fuente de entrada.
* **Operaciones de agregación** - El flujo soporta operaciones de agregación como filtrar, mapear, limitar, reducir, encontrar, coincidir, etc.
* **Pipelining** - La mayoría de las operaciones de flujo devuelven el flujo mismo para que su resultado pueda ser canalizado. Estas operaciones se llaman operaciones intermedias y su función es tomar la entrada, procesarla y devolver la salida al objetivo. El método collect() es una operación terminal que normalmente está presente al final de la operación de canalización para marcar el final del flujo.
* **Iteraciones automáticas** - Las operaciones de flujo hacen las iteraciones internamente sobre los elementos de origen proporcionados, en contraste con las Colecciones donde se requiere una iteración explícita.

Adicionalmente, se caracterizan por:

* Trabajan perfectamente con lambdas.
* No almacenan sus elementos.
* Son inmutables.
* No son reutilizables.
* No permiten acceso por índice a sus elementos.
* Son fácilmente paralelizables.
* Se ejecutan de forma Lazy.

### Generando Streams

Con la versión 8 de java, la interfaz Collection tiene dos métodos para generar un Stream.

* stream() - Devuelve un flujo secuencial considerando la colección como su fuente.



* parallelStream() - Devuelve un flujo paralelo considerando la colección como su fuente.



Los streams permiten realizar operaciones de filtro, mapeo y reducción, sobre colecciones de datos. Está basada en la interface Stream<T> . Esta en el package: **java.util.stream**.

### Streams Primitivos

A parte de que trabaja solo con objetos (Colecciones), también existen Streams para tratar tipos de datos primitivos como:

* IntStream
* LongStream
* DoubleStream

De estos, podemos generarlos de varias formas como:

1. A partir de un listado:



1. A partir de un rango:



A partir del ejemplo obtenemos un listado de valores que va de 1 a 5.

1. A partir de un iterador:



Genera un listado de valores pares que va de 0 a 18, es decir 0, 2, 4, 6, 8, 10, 12, 14, 16, 18.

1. A partir de un generador:



A diferencia del método “iterate”, no crea un listado de secuencia, sino que crea un listado de 10 valores invocando una lambda que calcula números aleatorios.

### Operaciones con Streams

Los streams permiten realizar el procesamiento de los datos de forma lineal, paso a paso, mediante lo que llamamos un “Pipeline” o tubería. Este comportamiento permite que podamos realizar pasos seguidos uno a tras de otro tomando en cuenta que la salida de un paso es la entrada del otro. Ejemplo:

Stream.of(-5,-2,-3,1,2,3,4,5).filter(dato -> dato > 0).foreach(System.out::println);

En este caso, lo que hacemos es crear un stream con ese conjunto de números y seguido ejecutamos la operación filter que filtrará los valores del listado para solo permitir que continúen en el pipeline los que son mayor a 0 y con su salida entonces se ejecuta el foreach para entonces imprimirlos en consola. En otras palabras:

1. Creamos Stream.
2. La salida de la creación del Stream es la entrada de la operación Filter, quien filtrará y arrojará como resultado un Stream con los números que sean mayor a 0.
3. La salida de la operación filter será la entrada del foreach, quien imprimirá uno a uno los valores en pantalla.

Para poder establecer correctamente un pipeline, debemos:

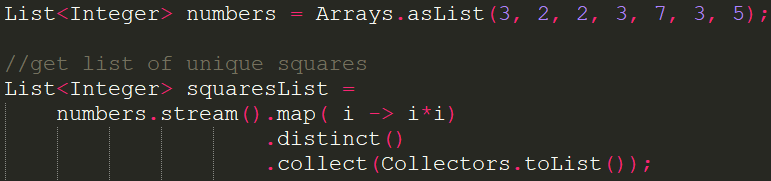
1. Crear un stream.
2. Aplicar cero o más operaciones intermedias (que devuelven otro stream).
3. Aplicar una operación final que genera el resultado.

#### Operaciones intermedias

Entre las diferentes operaciones intermedias de un pipeline podemos encontrar:

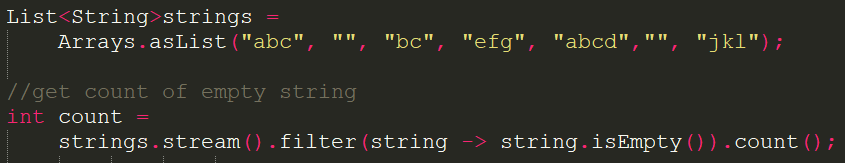
|  |  |
| --- | --- |
| **Operacion** | **Descripcion** |
| filter | Devuelve un stream con los elementos que cumplen la condición |
| map | Devuelve un stream con los elementos resultantes de aplicar una función. |
| flatMap | Función que recibe una entrada y devuelve varias salidas para esa entrada. Esa es la diferencia con Map que tiene una entrada y devuelve una única salida. En este caso hemos convertido el array en un stream |
| sorted | Devuelve un stream con los elementos ordenados respecto a un criterio de ordenación |
| distinct | Devuelve un stream sin elementos duplicados |
| limit | Devuelve un stream con el número de elementos especificados |

* **Map y distinct**



1. Creamos stream a partir del listado de números.
2. Ejecutamos la operación map para que cada elemento se convierta en un número resultado de multiplicar el valor por si mismo.
3. Eliminamos los duplicados mediante la operación distinct.
4. Recolectamos todos valores en una lista.

* **filter**



1. Creamos un Stream a partir de un listado.
2. Filtramos los valores que no estén vacíos.
3. Como resultado contamos el total de elementos que quedan después del filtro.

* **sorted y limit**



1. Creamos un Streams de enteros aleatorios.
2. Tomamos los 10 primeros.
3. Ordenamos los valores utilizando el comparador por defecto: de menor a mayor.
4. Recorremos cada valor del stream y los imprimimos en pantalla.

### Operadores finales

Para culminar el pipeline, esta necesita una operación para recuperar el resultado de las ejecuciones. Alguna de las operaciones que podemos usar son:

|  |  |
| --- | --- |
| **Operacion** | **Descripcion** |
| count | Devuelve un stream con los elementos que cumplen la condición |
| forEach | Devuelve un stream con los elementos resultantes de aplicar una función. |
| collect | Función que recibe una entrada y devuelve varias salidas para esa entrada. Esa es la diferencia con Map que tiene una entrada y devuelve una única salida. En este caso hemos convertido el array en un stream |
| max | Devuelve un stream con los elementos ordenados respecto a un criterio de ordenación |
| min | Devuelve un stream sin elementos duplicados |
| anyMatch | Devuelve un stream con el número de elementos especificados |
| orElse | Devuelve un stream con el número de elementos especificados |
| orElseThrow | Devuelve un stream con el número de elementos especificados |

* forEach



1. Creamos el Stream de enteros aleatorios.
2. Tomamos los primeros 10.
3. Recorremos uno a uno y los imprimimos en consola.

* Max y Min



1. Crea una lista de números.
2. A partir de la lista, créame un stream.
3. Del stream, búscame el valor máximo utilizando la siguiente función para comparación.
4. Retorna el objeto resultante.

Este ejemplo funcionaría igual con la operación “Min”

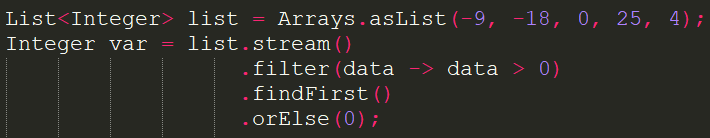


* anyMatch



1. Crea una lista de números.
2. A partir de la lista, créame un stream.
3. Del stream, valida si al menos 1 elemento cumple el predicado indicado.

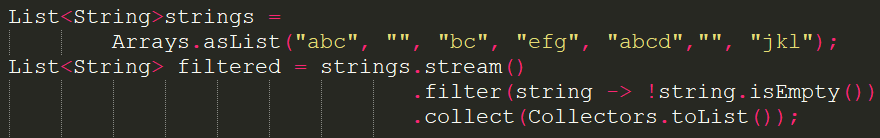
* findAny, orElse y orElseThrow



1. Crea una lista de números.
2. A partir de la lista, créame un stream.
3. De la lista, filtra los valores mayores a 0.
4. Del stream resultante, obtén el primero valor.
5. En caso de que no encuentra ninguno, retorna un 0.

Siguiendo este mismo ejemplo, en caso de implementar orElseThrow, el resultado será el lanzamiento de una excepción en vez de un valor fijo.

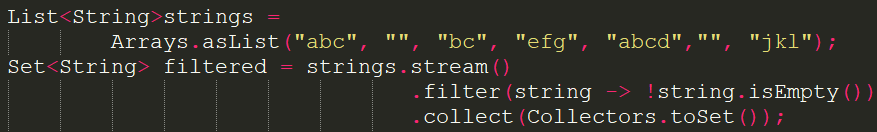
* Collect



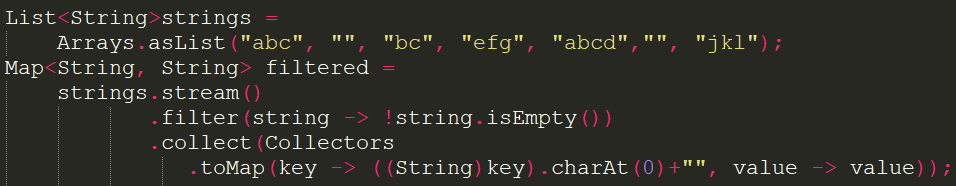
1. Crea una lista de String.
2. A partir de la lista, créame un stream.
3. Del stream, filtra los que no estén vacíos.
4. Recolecta todos los valores del stream y créame una lista con todos ellos.

Este ejemplo es el mas sencillo de las recolecciones de datos. Al igual que toList(), existen otros métodos que agrupan los valores en otras colecciones como:

* toSet



* toMap



1. Crea una lista de String.
2. A partir de la lista, créame un stream.
3. Del stream, filtra los que no estén vacíos.
4. Recolecta todos los valores del stream y créame un mapa. Para indicar como calcular la llave y el valor, recibe 2 lambas: uno que obtiene la primera letra del String como llave y otro que indica que el valor es el String completo.

Existen muchos otros tipos de agrupaciones mas complejas con diferentes propósitos como agrupar bajo un criterio (groupingBy), reducir el Stream a 1 valor bajo una serie de operaciones matemáticas (reduce), entre otros.

## Date/Time API

Dentro de la versión 8 de java se introduce una nueva API para manejo de fechas basadas en dos conceptos:

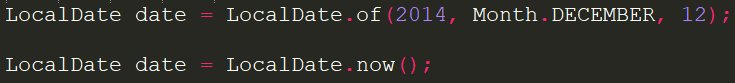
1. Local: simplifica el concepto date y time sin incluir el manejo de la zona horaria.
2. Zoned: especializa a date y time incluyendo la zona horaria.

### LocalDate, LocalTime y LocalDateTime

Estas clases simplifican en desarrollo donde las zonas horarias (timezones) no son requeridas, como el tratamiento de fechas puntuales, horas, timestamps, entre otros.

#### LocalDate

Objeto que representa una fecha en concreto. Puede ser inicializad indicando la fecha en especifico u obteniendo la fecha del instante.

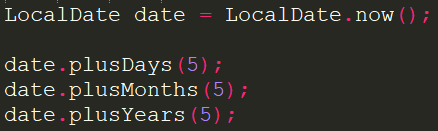


Una vez inicializada la fecha, pueden realizarse diferentes operaciones de forma muy amigable como sumarle días, meses, años, ajustarla a, entre otros; así como obtener información sobre esa fecha de forma amigable, como por ejemplo el día actual, de la semana o del año, el mes tanto en valor numérico como en nombre, el año, entre otros. Ejemplos:

* getDayOfMonth: obtiene el valor numérico del día en el mes indicado en la fecha.



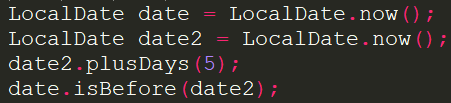
* plus, plusDays, plusMonths, plusYears: suma la cantidad indicada al elemento deseado, ya sea la fecha completa o una cantidad de días, meses o años. Para restar, se usa el opuesto, es decir minuts, minusDays, minusMonths y minusYears.



* isLeapYear: indica si es un año bisiesto o no.



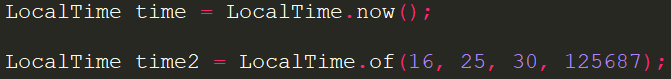
* isBefore, isAfter, isEquals: sirven para comparar dos fechas e indicar si una ocurre antes, después o al mismo tiempo.



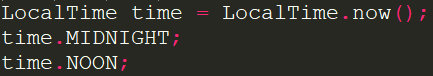
En este ejemplo, el resultado sería verdadero (true) ya que como a date2 se le suman 5 días si que termina siendo una fecha futura al compararla con date.

#### LocalTime

Esta clase representa el tiempo, no la fecha. Tiene los mismos comportamientos de LocalDate pero orientado a horas, minutos, segundos y nanosegundos.

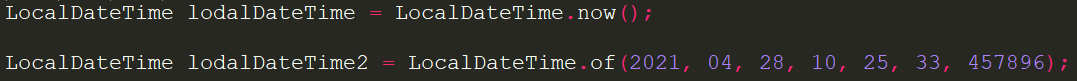


Adicional, mediante esta clase también e pueden obtener las horas que representan la media noche (midnight) o medio día (noon).



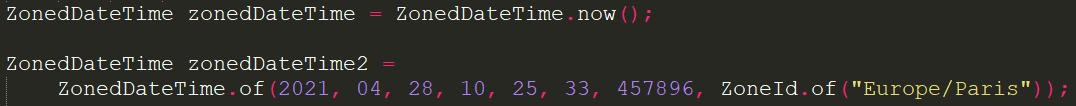
#### LocalDateTime

Esta clase es una combinación de LocalDate y LocalTime, es decir maneja fecha y hora y con ello todos los comportamientos o métodos que pueden realizarse con ellos.



### ZonedDateTime

La clase ZonedDateTime incluye la zona horaria para agregar un factor de complejidad que ayuda determinar y controlar ciertos aspectos que se ven afectados por ella.



Posee los mismos métodos que LocalDateTime, agregando solo los parámetros o algún comportamiento relacionado a la zona horaria.

### Chronounits

ChronoUnit es una enumeración localizada en el paquete java.time.temporal.ChronoUnit que ayuda a centralizar los diferentes valores cronológicos que existen o pueden utilizarse para representar el tiempo en Java.