**TECNOLÓGICO ESPÍRITU SANTO**

**“Lambda Expressions (Java 7, 8)”**

****

**INTEGRANTES:**

**Melanny Núñez Vera**

**Anggie Medina**

**Wellington Mora**

**MATERIA:**

**Fundamentos De Programación**

**INSTRUCTOR:**

**Eddye Lino**

**BIMESTRE:**

**I Bimestre**

**AÑO:**

**2017-2018**

INDICE

[**I.** **INTRODUCCIÓN** 3](#_Toc482709965)

[**II.** **ENTORNO** 6](#_Toc482709966)

[**III.** **TIPOS** 6](#_Toc482709967)

[**IV.** **REFERENCIAS A MÉTODOS** 9](#_Toc482709968)

[**V.** **SINTAXIS** 11](#_Toc482709969)

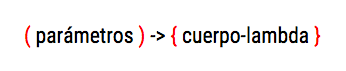
[**VI.** **CASO DE USO IDEAL PARA EXPRESIONES LAMBDA** 20](#_Toc482709970)

[**VII.** **CONCLUSIONES** 21](#_Toc482709972)

[**VIII.** **BIBLIOGRAFÍA** 22](#_Toc482709973)

1. **INTRODUCCIÓN**

Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase.

Su sintaxis básica se detalla a continuación:

* El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.
* Parámetros:
* Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
* Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
* Cuerpo de lambda:
* Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula return en el caso de que deban devolver valores.
* Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula return en el caso de que la función deba devolver un valor.
* Las expresiones lambda proporcionan un medio para escribir códigos claros y concisos con el uso de estas.
* Con la introducción de las expresiones lambda en java se puedan aprovechar los beneficios del lenguaje funcional dentro de un lenguaje orientado a objetos.
* Las expresiones lambda le permiten expresar instancias de clases de un solo método más compacta.

Algunos ejemplos de expresiones lambda pueden ser:

* z -> z + 2
* () -> System.out.println(” Mensaje 1 “)
* (int longitud, int altura) -> { return altura \* longitud; }
* (String x) -> { String retorno = x;

retorno = retorno.concat(” \*\*\*”);

return retorno; }

Como hemos visto las expresiones lambda son funciones anónimas y pueden ser utilizadas allá donde el tipo aceptado sea una interfaz funcional.

Una interfaz funcional es una interfaz con uno y solo un método abstracto. La declaración es exactamente igual que las interfaces normales con dos características adicionales:

* Tiene un único método abstracto, como ya hemos dicho.
* De manera opcional puede estar anotada como @FunctionalInterface.

El motivo de que la interfaz tenga un único método abstracto es que será la expresión lambda la que proveerá de la implementación para dicho método.

A continuación algunos ejemplos de interfaz funcional

|  |  |
| --- | --- |
|  | *@FunctionalInterface*  public interface Runnable {  public abstract void run();  } |

|  |
| --- |
| *public interface MiInterfaz {*  *default void saluda() {*  *System.out.println ("Un saludo!");*  *}*  *public abstract int calcula(int dato1, int dato2);*  *}* |

* **EXPRESIONES LAMBDA**

Por medio de expresiones lambda podemos referenciar métodos anónimos o métodos sin nombre, lo que nos permite escribir código más claro y conciso que cuando usamos clases anónimas. Una expresión lambda se compone de:

* Listado de parámetros separados por comas y encerrados en paréntesis, por ejemplo: (a, b).
* El símbolo de flecha hacia la derecha: ->
* Un cuerpo que puede ser un bloque de código encerrado entre llaves o una sola expresión.

@FunctionalInterface

public interface Comparator {

// Se eluden los métodos default y estáticos

int compare(T o1, T o2);

// El método equals (Object obj) es implícitamente implementado por la clase objeto.

boolean equals(Object obj);

}

1. **ENTORNO**

El tutorial está escrito usando el siguiente entorno:

* Hardware: MacBook Pro 17′ (2.66 GHz Intel Core i7, 8GB DDR3 SDRAM).
* Sistema Operativo: Mac OS X Lion 10.10.3.
* NVIDIA GeForce GT 330M 512Mb.
* Crucial MX100 SSD 512 Gb.

1. **TIPOS**

Las expresiones lambda pueden clasificarse de la siguiente manera:

* Consumidores.
* Proveedores.
* Funciones.
* Operadores Unarios.
* Operadores Binarios.
* Predicados.

A continuación iremos detallando uno a uno y contando en qué consisten.

* **CONSUMIDORES**

Se trata de aquellas expresiones lambda que aceptan un solo valor y no devuelven valor alguno.

* String message -> System.out.println(message);

Las expresiones BiConsumidoras, un caso especial de las expresiones consumidoras, son aquellas que toman dos valores como parámetro y no devuelven resultado.

* (String key, String valué) -> System.out.println("Key: %s, valué: %s%n", key, valué);
* **PROVEEDORES**

En este caso se trata de expresiones que no tienen parámetros pero devuelven un resultado.

* () -> return createRandomInteger()
* **FUNCIONES**

Aquellas expresiones que aceptan un argumento y devuelven un valor como resultado y cuyos tipos no tienen por qué ser iguales.

* Order persistedOrder -> persistedOrder.getIdientifier();

Las BiFunciones son aquellas expresiones de tipo función que aceptan dos argumentos y devuelven un resultado.

* (Address address, String name) -> new Person(name, address);
* **OPERADORES UNARIOS**

Caso especial de funciones en las que tanto el parámetro como el valor devuelto son del mismo tipo.

* String message -> message. ToLowerCase()
* **OPERADORES BINARIOS**

Igual que en el caso de los Operadores Unarios, se trata de un caso especial de funciones en las que los dos argumentos y el resultado son del mismo tipo.

* (String message, String anotherMesssage) -> message.concat(anotherMessage);
* **PREDICADOS**

Se trata de expresiones que aceptan un parámetro y devuelven un valor lógico.

* String message -> message.length > 50

Como en los casos anteriores, se pueden tener Predicados, predicados que en lugar de tener un parámetro, tienen dos.

* (path, attr) -> String.valueOf(path).endsWith(".js") && attr.size() > 1024

1. **REFERENCIAS A MÉTODOS**

Las referencias a los métodos nos permiten reutilizar un método como expresión lambda. Para hacer uso de las referencias a métodos basta con utilizar la siguiente sintaxis: referencia Objetivo: nombreDelMetodo.

* File::canRead // en lugar de File f -> f.canRead();

Con las referencias a los métodos se ofrece una anotación más rápida para expresiones lambda simple y existen 3 tipos diferentes:

* Métodos estáticos.
* Métodos de instancia de un tipo.
* Métodos de instancia de un objeto existente.

**MÉTODOS ESTÁTICOS**

Cuando el método invocado es estático, la forma de escribir la expresión lambda usando métodos de referencia es la siguiente: NombreClase::métodoEstático, donde NombreClase es el nombre de la clase que contiene el método y métodoEstático es el nombre del método estático a invocar. En el siguiente ejemplo, definimos una operación de suma por medio del nuevo método estático +Integer.sum(int,int):int el cual suma los dos parámetros y retorna su resultado.

Primero veamos cómo se escribiría usando una expresión lambda:

* BinaryOperator<Integer> sum = (a,b) ­> Integer.sum(a,b);

Y ahora usando métodos de referencia:

* BinaryOperator<Integer> sum = Integer::sum;

Nótese el uso de la interface funcional java.util.function.BinaryOperator, la cual define una función que recibe dos parámetros del mismo tipo y retorna un resultado del mismo tipo de sus parámetros:

+apply(T,T):T

**MÉTODOS DE INSTANCIA DE UN OBJETO**

Cuando contamos con una referencia a un objeto y deseamos invocar alguno de sus métodos de instancia dentro de la expresión lambda, la forma en la que la escribiríamos usando métodos de referencia es la siguiente: RefObjeto::métodoInstancia, donde RefObjeto es la referencia al objeto y métodoInstancia es el nombre del método a invocar. Por ejemplo, la clase java.lang.System tiene una referencia a un objeto de tipo java.io.PrintStream denominada out, usaremos esa referencia para nuestro siguiente ejemplo.

Primero veamos cómo se escribiría usando una expresión lambda:

* Consumer<Integer> print = (a) ­> System.out.println(a);

Y ahora usando métodos de referencia:

* Consumer<Integer> print = System.out::println;

Nótese que la referencia al objeto la tenemos en System.out e invocamos su método de instancia +println(int):void

* **MÉTODOS DE INSTANCIA DE ALGÚN TIPO**

Este caso es parecido al anterior, pero se diferencia en que no contamos con una referencia a un objeto, solo conocemos su tipo y podríamos escribir la expresión lambda de la siguiente forma: Tipo::métodoInstancia, donde Tipo es la clase y métodoInstancia es el nombre del método de instancia a invocar. El siguiente ejemplo define un java.lang.Comparator que nos permitirá comparar cadenas sin importar si son mayúsculas/minúsculas.

Primero veamos cómo se escribiría usando una expresión lambda:

* Comparator<String> upper = (a, b) ­> a.compareToIgnoreCase(b);

Y ahora usando métodos de referencia:

* Comparator<String> upper = String::compareToIgnoreCase;

Nótese que en este caso no contamos con la referencia a un objeto como tal, pero sabemos que queremos comprar objetos de tipo String y con eso es suficiente para que podamos escribir nuestra expresión lambda usando métodos de instancia de algún tipo.

Ejemplo de uso con método estático:

* (String info) -> System.out.println (info) // Expresión lambda sin referencias.
* System.out.println // Expresión lambda con referencia a método estático.

Ejemplo de uso con método de instancia de un tipo:

* (Student student, int registryIndex) -> student.getRegistry (registryIndex) // Expresión lambda sin referencias.
* Student: getRegistry // Expresión lambda con referencia a método de un tipo.

Ejemplo de uso con método de instancia de un objeto existente:

* Student student -> getMarks (student) // Expresión lambda sin referencias.
* This: getMarks // Expresión lambda con referencia a método de un objeto existente.

Las expresiones lambda se introducen en Java 8 y se promociona a ser la característica más grande de Java 8. Expresión Lambda facilita la programación funcional, y simplifica el desarrollo mucho.

1. **SINTAXIS**

Una expresión lambda se caracteriza por la siguiente sintaxis

Parameter -> expression body

Las siguientes son las características importantes de una expresión lambda -

* OPCIONAL DECLARACIÓN DE TIPO:

No hay necesidad de declarar el tipo de un parámetro. El compilador puede inferencia de la misma a partir del valor del parámetro.

* OPCIONAL PARÉNTESIS ALREDEDOR DE PARÁMETROS:

No hay necesidad de declarar un solo parámetro entre paréntesis. Para múltiples parámetros, se requieren paréntesis.

* LLAVES OPCIONALES:

No hay necesidad de utilizar llaves en el cuerpo de expresión si el cuerpo contiene una única instrucción.

* PALABRA CLAVE OPCIONAL RETURN:

El compilador devuelve automáticamente el valor si el cuerpo tiene una sola expresión para devolver el valor. Se requieren llaves para indicar que la expresión devuelve un valor.

* **EJEMPLOS USANDO LAMBDA**
* **EJEMPLO 1**

public class Java8Tester {

public static void main(String args[]){

Java8Tester tester = new Java8Tester() //with type declaration

MathOperation addition = (int a, int b) -> a + b; //with out type declaration

MathOperation subtraction = (a, b) -> a - b; //with return statement along with curly braces

MathOperation multiplication = (int a, int b) -> { return a \* b; //without return statement and without curly braces

MathOperation division = (int a, int b) -> a / b;

System.out.println("10 + 5 = " + tester.operate(10, 5, addition));

System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));

System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));

System.out.println("10 / 5 = " + tester.operate(10, 5, división)); //with parenthesis

GreetingService greetService1 = message ->

System.out.println("Hello " + message) //without parenthesis

GreetingService greetService2 = (message) ->

System.out.println("Hello " + message);

greetService1.sayMessage("Mahesh");

greetService2.sayMessage("Suresh");

}

interface MathOperation {

int operation(int a, int b);

}

interface GreetingService {

void sayMessage(String message);

}

prívate int opérate(int a, int b, MathOperation mathOperation){

return mathOperation.operation(a, b);

}

}

Comprobar el resultado

10 + 5 = 15

10 - 5 = 5

10 x 5 = 50

10 / 5 = 2

Hello Mahesh

Hello Suresh

* **EJEMPLO 2**



Los siguientes son los puntos importantes a tener en cuenta en el ejemplo anterior.

* Las expresiones lambda se utilizan principalmente para definir la aplicación en línea de una interfaz funcional, es decir, una interfaz con un solo método. En el ejemplo anterior, hemos utilizado varios tipos de expresiones lambda para definir el método de funcionamiento de la interfaz MathOperation. A continuación, hemos definido la aplicación de sayMessage de GreetingService.
* Expresión lambda elimina la necesidad de la clase anónima y da una capacidad de programación funcional muy simple pero potente para Java.
* ALCANCE

El uso de la expresión lambda, se puede hacer referencia a la última variable o variable con eficacia final (que se asigna una sola vez). Expresión lambda lanza un error de compilación, si una variable se le asigna un valor de la segunda vez.

* **Ejemplo alcance**

Cree el siguiente programa Java utilizando el editor y guardar en alguna carpeta como C: \> JAVA.

Public class Java8Tester {

Final static String salutation = "Hello! ";

Public static void main (String args []) {

GreetingService greetService1 = message ->

System.out.println (salutatión + message);

greetService1.sayMessage ("Mahesh");

}

Interface GreetingService {

Void sayMessage (String message);

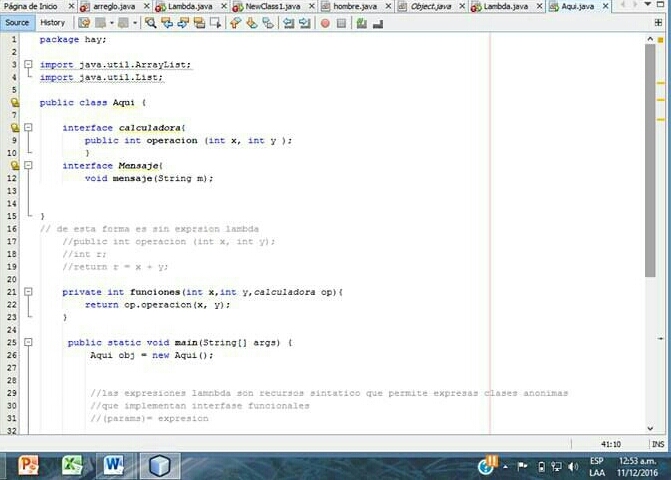
}

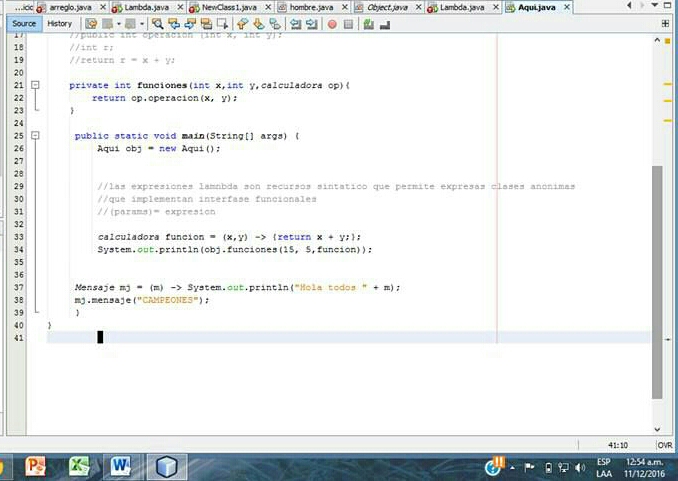
}

Comprobar el resultado

Hello! Mahesh

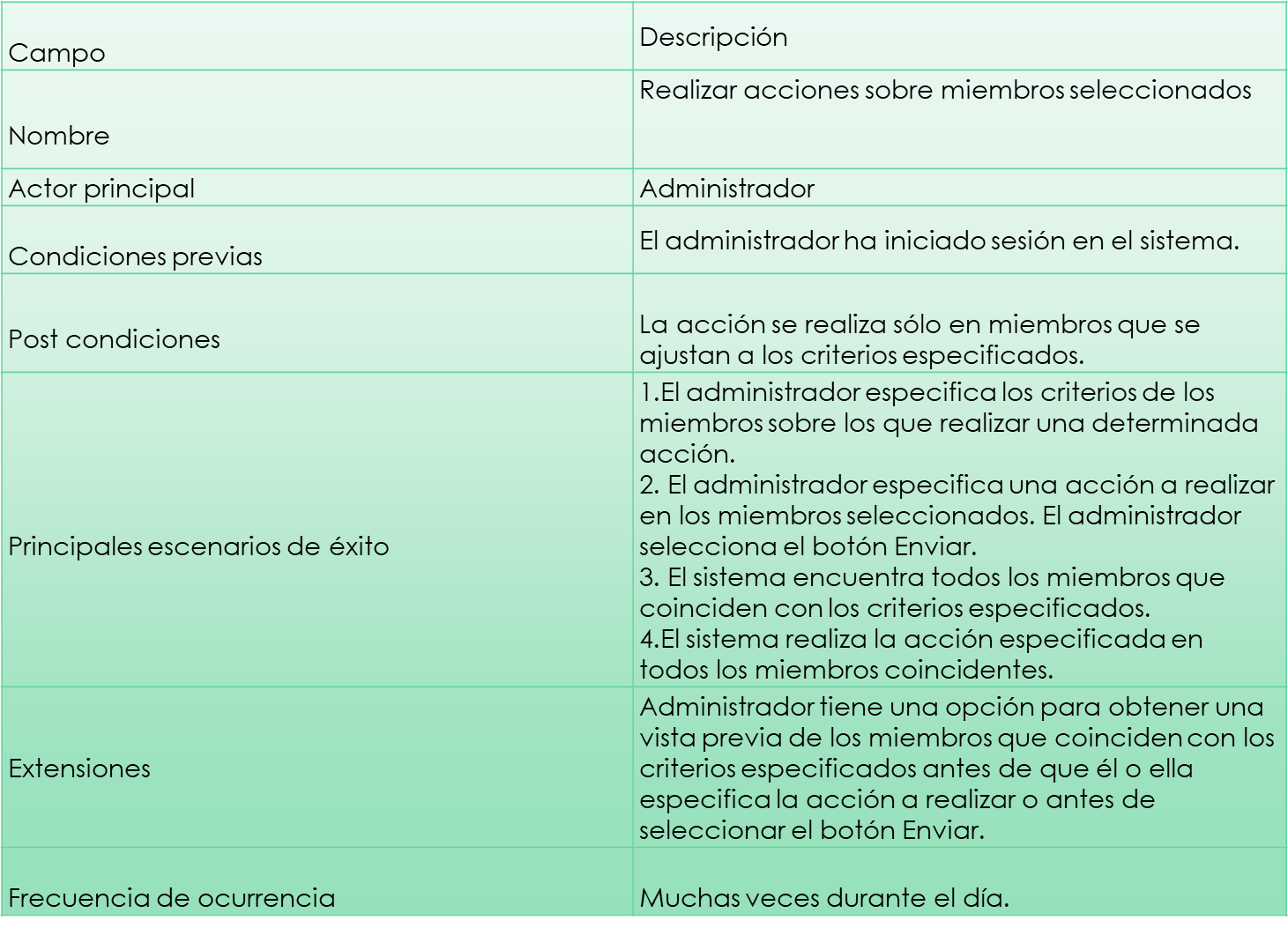
* **EJEMPLOS PROPIOS**





1. **CASO DE USO IDEAL PARA EXPRESIONES LAMBDA**

Suponga que está creando una aplicación de red social. Desea crear una función que permita a un administrador realizar cualquier tipo de acción, como enviar un mensaje, a los miembros de la aplicación de redes sociales que cumplan ciertos criterios. La siguiente tabla describe este caso de uso en detalle:

****

1. **CONCLUSIONES**

Las expresiones lambda de Java 8 nos ofrecen varias mejoras con respecto a las versiones anteriores:

* Nos acerca a la programación funcional.
* Hace nuestro código más preciso y legible, mejorando, en consecuencia, su mantenibilidad.
* Su utilización junto con la API Stream hace más fácil la ejecución concurrente de tareas.

En este tutorial hemos mostrado en qué consisten y qué tipos hay, cuándo pueden utilizarse y las diferentes maneras de utilizarlas. Pendiente de posteriores tutoriales queda tratar aspectos más avanzados como streams finitos e infinitos, utilización de colectores o debugging de streams y expresiones lambda.

1. **BIBLIOGRAFÍA**

<https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm>

<http://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-api-stream-java-2633852-esa.html>

<https://www.youtube.com/watch?v=8man0ZNx3FU>

<http://www.arquitecturajava.com/java-8-lambda-expressions/>