Programación Funcional

BOOTCAMP JAVA-MICROSERVICIOS | Perú 2022

Por: Angelo Rafael Angulo Méndez

aangulom@emeal.nttdata.com

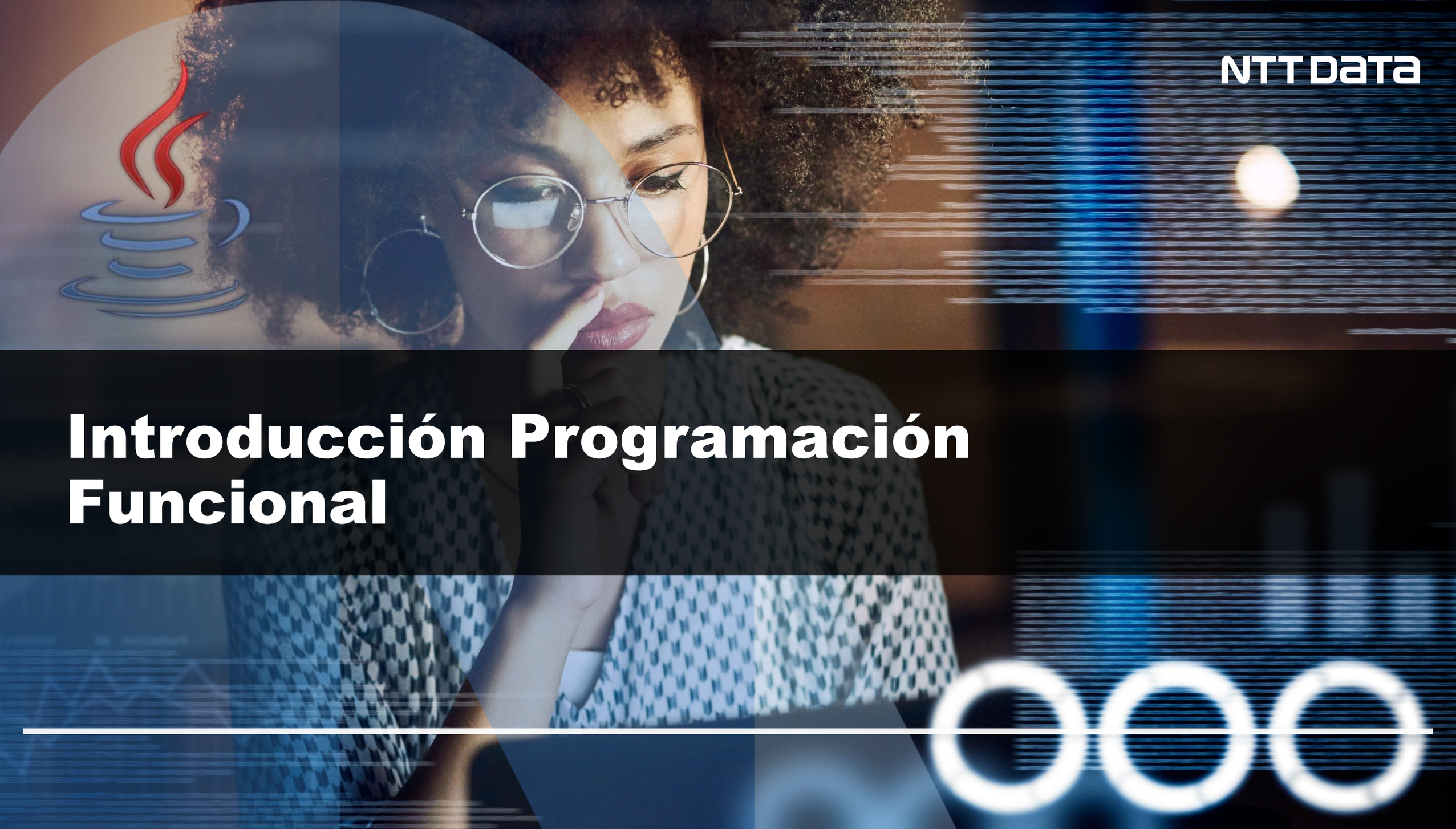
AGENDA

PROGRAMACIÓN FUNCIONAL

EXPRESIONES LAMBDA

API STREAM

MÉTODOS POR REFERENCIA



Introducción Programación Funcional

Introducción a la Programación Funcional

- ✓ *En informática, la programación funcional es un **paradigma** de **programación declarativa** basado en el uso de **funciones puras** y **valores inmutables***
- ✓ *Se prioriza el uso de recursividad y aplicación de funciones de **primera clase** para resolver problemas que en otros lenguajes se resolverían mediante estructuras de control (por ejemplo, ciclos). .*
- ✓ *Java no es un lenguaje diseñado netamente para Programación Funcional sin embargo desde Java 8 se agregaron varias características para facilitar un estilo de programación funcional*

Ventajas

- Escribir programas más rápidamente.
- Escribir programas más concisos.
- Escribir programas con menos errores.
- Escribir programas más fáciles de paralelizar.

Características

- Programación sin efectos secundarios.
- Funciones de primera clase.
- Cerraduras (closures) de funciones.
- Operaciones de orden superior sobre colecciones.

Programación Declarativa vs Imperativa

```java
 * Programación declarativa vs imperativa
public class Ejercicios01 {

     * Obtener la cantidad de numeros mayores a 10
    public static void main(String[] args) {
        List<Integer> numeros = List.of(18, 6, 4, 15, 55, 78, 12, 9, 8);
        declarative(numeros);
        imperative(numeros);
    }

    private static void imperative(List<Integer> numeros) {
        int contador = 0;
        for(int numero : numeros) {
            if(numero > 10) {
                contador ++;
            }
        }
        System.out.println(String.format("Hay %1$s numero mayores a 10 (Declarativo)", contador));
    }

    private static void declarative(List<Integer> numeros) {
        Long result = numeros.stream().filter(num -> num > 10).count();
        System.out.println(String.format("Hay %1$s numero mayores a 10 (Imperativo)",result));
    }
}
```

Nos enfocaremos en "**qué**" estamos haciendo y no en "**cómo**"

Funciones de Primera Clase

```
 */
@SuppressWarnings("rawtypes")
public class Ejercicios02 {


    static Function x;
    static Function y;

    public static void main(String[] args) {
        functionA();
        functionB(x);
        y = functionA();
    }

    private static Function functionA() {
        return x;
    }

    private static Function functionB(Function param) {
        return param;
    }
}
```

Es un elemento de un lenguaje de programación que está estrechamente **integrado** con el lenguaje y es compatible con la gama completa de **operaciones** generalmente disponibles para otras entidades en el lenguaje.

Funciones Puras

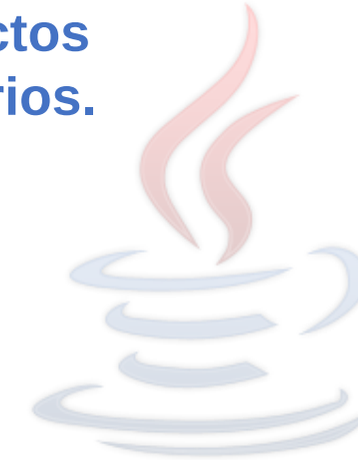
```java
public class Ejercicios03 {

    public static void main(String[] args) {
        System.out.println(functionA(1,2) == functionA(1,2)); //true
    }

    private static int functionA(int a, int b) {
        System.out.println(a+b);
        return a+b;
    }

}
```

Las funciones puras, no son más que funciones, las cuales, dado el mismo input, **siempre retornan el mismo output**, además de no tener **efectos secundarios.**



Valores Inmutables

```java
public class Employee {
    //Propiedad mutable
    private String name = "";

    //Propiedad inmutable
    private final LocalDateTime birthday = LocalDateTime.of(1991, 10, 28, 23, 0);

    public Employee(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getBirthday() {
        return birthday.getMonthValue() + "/" + birthday.getDayOfMonth();
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", birthday=" + birthday + "]";
    }

}
```

Un objeto inmutable es aquel que **no puede ser modificado** una vez haya sido creado.

Efectos Secundarios

```
5      public static int VALOR = 10;
6
7      public static void main(String[] args) {
8          functionA(1,2);
9      }
10
11     private static int functionA(int a, int b) {
12         VALOR = 5; //Valor alterado
13         System.out.println(VALOR);
14         return a+b;
15     }
16
17 }
```

Problems | Javadoc | Declaration | Console

```
<terminated> Ejercicios06 [Java Application] C:\Program Files\Java\jdk-11.0.3
5
```

Uno de los objetivos de la programación funcional es minimizar los efectos secundarios.

Un efecto secundario es cualquier **cambio de estado** en la aplicación que sea **observable fuera de la función llamada.**

NTT DATA

Expresiones Lambda

Expresiones Lambda

- ✓ *Una expresión lambda representa una **función anónima**, en java se utiliza desde la versión 8.*
- ✓ *No tienen nombre, pero tienen una lista de parámetros, cuerpo, tipo de retorno y posiblemente una lista de excepciones que puede lanzar*

Sintaxis de Expresiones Lambda

```
public static void main(String[] args) {

    IAddition addition =  (a, b) -> {
                              int c = a * 2;
                              return c + b;
                          };

    int resultado = addition.add(4, 5);
    System.out.println(resultado);

}
```

A más de 2 argumentos se requiere ()
A más de 1 sentencia requiere {}

Interfaces Funcionales

- ✓ *Las interfaces funcionales son interfaces que tienen un único método abstracto a implementar, pudiendo implementar uno o varios métodos **default** o **static**. Esto significa que cada interfaz creada que respeta esta premisa se convierte automáticamente en una interfaz funcional*

Interfaces Funcionales

```java
@FunctionalInterface
public interface IGreeting {
    void SayHello();
}
```

```java
import pe.com.java.bootcamp.interfaces.IGreeting;

public class Ejercicios07 {

    public static void main(String[] args) {
        IGreeting greeting = () -> {System.out.println("Hello World");};
        greeting.SayHello();
    }
}
```

Interfaces Funcionales (Parámetros)

```java
@FunctionalInterface
public interface IGreetingV2 {
    void SayHello(String usuario);
}
```

```java
6
7       public static void main(String[] args) {
8
9           IGreetingV2 greeting = usuario -> {
10              if (usuario == "Bootcamp")
11                  System.out.println("Hello " + usuario);
12              else
13                  System.out.println("Hello World " + usuario);
14          };
15          greeting.SayHello("Bootcamp");
16      }
17  }
```

Interfaces Funcionales (Retornar Valores)

```java
@FunctionalInterface
public interface IAddition {
    int add(int a , int b);
}
```

```java
import pe.com.java.bootcamp.interfaces.IAddition;

public class Ejercicios10 {

    public static void main(String[] args) {

        IAddition addition =  (a, b) -> {
            int c = a * 2;
            return c + b;
        };
        int resultado = addition.add(4, 5);
        System.out.println(resultado);

    }
}
```

Interfaces Funcionales Genéricas

- Supplier
- Consumer y BiConsumer
- Predicate y BiPredicate
- Function y Bifunction
- UnaryOperator y BinaryOperator

Supplier

```
Supplier<LocalDateTime> s = () -> LocalDateTime.now();
LocalDateTime time = s.get();

System.out.println(time);
```

*Contiene el método **get** que no recibe argumentos y produce un valor de tipo <T>. A menudo se usa para crear un objeto colección en donde se colocan los resultados de la operación de un flujo.*

Consumer y BiConsumer

```java
// Consumer to display a number
Consumer<Integer> display = a -> System.out.println(a);
display.accept(10);

// BiConsumer to add two numbers
BiConsumer<Integer, Integer> addTwo = (x, y) -> System.out.println(x + y);
addTwo.accept(1, 2);
```

Contiene el método **accept** que recibe como argumento <T> y devuelve void. Realiza una tarea con su argumento <T>, como mostrar el objeto en pantalla, invocar a un método del objeto, etc

Predicate y BiPredicate

```java
 // Creating predicate
Predicate<Integer> lesserthan = i -> (i < 18);

// Calling Predicate method
System.out.println(lesserthan.test(10));

// Creating biPredicate
BiPredicate<String, Integer> filter = (x, y) -> {
    return x.length() == y;
};

// Calling BiPredicate method
boolean result = filter.test("mkyong", 6);
System.out.println(result);  // true
```

*Contiene el método **test** que recibe un argumento T y devuelve un boolean.*

UnaryOperator y BinaryOperator

```java
public static void main(String args[]) {

    // Creating UnaryOperatos
    UnaryOperator<Integer> func2 = x -> x * 2;

    // Calling  method
    Integer result2 = func2.apply(2);
    System.out.println(result2);

    // Creating BinaryOperator
    BinaryOperator<Integer> func3 = (x1, x2) -> x1 + x2;

    // Calling  method
    Integer result3 = func3.apply(2, 3);
    System.out.println(result3);

}
```

*Contiene el método **apply** que recibe un argumento T y devuelve un valor del mismo tipo (T).*

Function y BiFunction

```java
public class Ejercicios15 {

    public static void main(String args[]) {

        // Creating Function
        Function<Integer, Integer> func2 = x -> x * 2;

        // Calling  method
        Integer result2 = func2.apply(2);
        System.out.println(result2);

        // Creating BinaryOperator
        BiFunction<Integer, Integer, Integer> func3 = (x1, x2) -> x1 + x2;

        // Calling  method
        Integer result3 = func3.apply(2, 3);
        System.out.println(result3);

    }
}
```

*Contiene el método **apply** que recibe un argumento T y devuelve un valor del mismo tipo K.*

NTT DATA

API Stream

Stream

- ✓ *Es una abstracción de interface disponible a partir de Java 8, que nos permite procesar información de una **colección**.*
- ✓ *Un Stream consta de un **origen**, cero o más **operaciones intermedias** y una **operación terminal.***
- ✓ *Los Streams admiten operaciones estilo SQL*
- ✓ *Trabaja de dos maneras: por estructura de proceso (un Stream puede generar otro Stream de manera que se puede estructurar) o por iteración interna (internamente realiza acciones hasta obtener el resultado en concreto).*
- ✓ *Son de tipo Lazy cálculo de los datos del origen solo se realiza cuando comienza la operación final.*

Source (Origen)

```java
public static void main(String[] args) {

    List<User> users = new ArrayList<>();
    users.add(new User("Angelo", 21));
    users.add(new User("Rafael", 27));
    users.add(new User("Tania", 20));
    users.add(new User("Jose", 33));

    //A partir de una coleccion
    Stream<User> stream = users.stream();

    //A partir de otro stream
    Stream<User> secondStream = users.stream().filter((user) -> user.getAge() > 25);

    //A partir de un arreglo
    int [] numbers = {1,2,3,4,5};
    Stream<int[]> thirdStream = Stream.of(numbers);

    //A partir de una secuencia
    Stream fourthStream = Stream.of("Angelo", "Rafael", "Tania","Jose");
}
```

Operaciones Intermedias

Operación	Descripción
filter	Devuelve los elementos que coinciden con el predicado dado
map	Aplica la función dada a cada elemento del stream
flatMap	Aplana un stream
distinct	Devuelve elementos únicos del stream
sorted	Devuelve los elementos del stream en orden
peek	Examina solo el primer elemento del stream
limit	Devuelve un número limitado de elementos del stream
skip	Omite un número de elementos del stream

Ejemplos Operaciones Intermedias

```
//Filter
var numbers = num.stream()
            .filter(n -> n > 1)
            .collect(Collectors.toList());
System.out.println(numbers);

//Map
var numbers2 = num.stream()
            .map(n -> n * 2)
            .collect(Collectors.toList());
System.out.println(numbers2);

//Distinct
var numbers3 = num.stream()
            .distinct()
            .collect(Collectors.toList());
System.out.println(numbers3);
```

```
//Distinct
var numbers3 = num.stream()
            .distinct()
            .collect(Collectors.toList());
System.out.println(numbers3);

//Sorted
var numbers4 = num.stream()
        .sorted()
        .collect(Collectors.toList());
System.out.println(numbers4);

//All
var numbers5 = num.stream()
        .filter(n -> n > 1)
        .filter (n -> n < 5)
        .map(n -> n*2)
        .sorted()
        .distinct()
        .collect(Collectors.toList());
System.out.println(numbers5);
```

Operaciones Terminales

Operación	Descripción
forEach	Realiza una acción en todos los elementos del stream
toArray	Regresa un arreglo conteniendo los elementos del stream
reduce	Realiza una operación de reducción en los elementos del stream usando un valor inicial y una operación binaria
collect	Regresa un contenedor mutable como List o Set
min	Regresa el elemento mínimo en el stream
max	Regresa el elemento máximo en el stream
count	Regresa el número de elementos del stream
anyMatch	Regresa true si alguno de los elementos del stream coincide con el predicado
allMatch	Regresa true si todos los elementos del stream coinciden con el predicado
noneMatch	Regresa true si ninguno de los elementos del stream coincide con el predicado
findFirst	Regresa el primer elemento del stream
findAny	Regresa un elemento aleatorio del stream

Ejemplo Operaciones Terminales

```
List<Integer> num = Arrays.asList(5,2,2,1,3);

//ForEach
num.stream().forEach((x) -> System.out.println(x));

//Collect
var numbers2 = num.stream().collect(Collectors.toList());
System.out.println(numbers2);

//Min
var numbers3 = num.stream().min(Integer::compare);
System.out.println(numbers3.get());

//Max
var numbers4 = num.stream().max(Integer::compare);
System.out.println(numbers4);
```

Métodos por Referencia

Métodos por referencia

- ✓ *Los métodos referenciados son una de las nuevas características de Java 8 que nos permite **hacer referencia a los métodos y constructores por medio de una interface funcional**, dicho de otra manera, podemos implementar la funcionalidad de un método abstracto por medio de la implementación de un método ya implementado, asignando el método implementado al método abstracto.*

Referencia a Métodos estáticos

```
public static void main(String[] args) {
    List<Double> num = Arrays.asList(1.0,2.0,3.0,4.0,5.0);

    num.stream()
     .map(Ejercicios04::toCube)
     .forEach(System.out::println);

}

public static double toCube(double numero) {
    return Math.pow(numero, 3);
}
```

Referencia a Métodos de instancia

```
public static void main(String[] args) {
    List<Double> num = Arrays.asList(1.0,2.0,3.0,4.0,5.0);
    Calculator calculator = new Calculator();

    num.stream()
     .map(calculator::toCube)
     .forEach(System.out::println);

}
```

Referencia a Métodos de instancia de un objeto arbitrario

```java
public static void main(String[] args) {

    List<User> users = new ArrayList<>();
    users.add(new User("Angelo", 21));
    users.add(new User("Rafael", 27));
    users.add(new User("Tania", 20));
    users.add(new User("Jose", 33));

    users.stream()
     .map(User::getName)
     .forEach(System.out::println);

}
```

Referencia a un constructor

```
public static void main(String[] args) {

    List<User> users = new ArrayList<>();
    IUser iUser = User::new;
    users.add(iUser.Create("Angelo", 30));
    users.add(iUser.Create("Rafael", 27));
    users.add(iUser.Create("Tania", 20));
    users.add(iUser.Create("Jose", 33));

    users.stream()
     .map(User::getName)
     .forEach(System.out::println);

}
```

Ejercicios

1. Crear una función anónima o stream que nos permita calcular el promedio de un arreglo de 10 números enteros.

2. Dado un arreglo de 10 números enteros, crear una función anónima o stream que retorne el número mayor.

3. Dado un arreglo de 10 números enteros, crear una función anónima o stream que retorne el número menor.

4. Crear una función anónima o stream que reciba como parámetro 3 números enteros. La función retorna el número mayor.

5. Crear una función anónima o stream que reciba dos parámetros, un string y un numero entero. La función retorna un string el cual será de repetir la cadena tantas veces el número entero

Gracias