

# WORKSHOP

2022

MARIA ALEJANDRA BUQUETE

λ

# INDICE

- LAMBDAS
- STREAMS
- OPTIONAL

# MOTIVACION

- SWING -> Y SUS LISTENERS

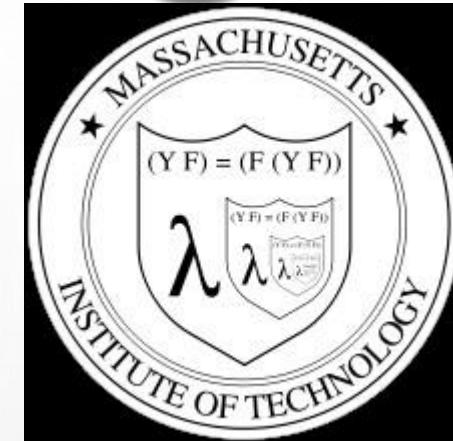
```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("button clicked");  
    }  
});
```

- LAMBDAS

```
button.addActionListener(event -> System.out.println("button clicked"));
```

# Cálculo Lambda

Sistema lógico formal para expresar computos basados en abstracción y aplicación de funciones  
Usando sustitución y enlace (binding) de variables



- Introducido por Alonzo Church (1903-1995)
- Implementado entre otros en Lisp y Scheme

# Cálculo Lambda

`Java.util.function`

Son "functional interfaces"

- . Sirven como "variables" para expresiones lambda.
- . T es el tipo del parámetro, R del resultado
  - Function ( $T \rightarrow R$ )
  - Consumers ( $T \rightarrow void$ )
  - Predicate ( $T \rightarrow boolean$ )
  - Supplier ( $nil \rightarrow R$ )

# LAMBDA

Interface name	Arguments	Returns	Example
Predicate<T>	T	boolean	Has this album been released yet?
Consumer<T>	T	void	Printing out a value
Function<T, R>	T	R	Get the name from an <code>Artist</code> object
Supplier<T>	None	T	A factory method
UnaryOperator<T>	T	T	Logical not (!)
BinaryOperator<T>	(T, T)	T	Multiplying two numbers (*)

La expresión lambda más simple contiene un solo parámetro y una expresión:

*parameter -> expression*

Para usar más de un parámetro, envuélvalos entre paréntesis:

*(parameter1, parameter2) -> expression*

# EXPRESIONES LAMBDA

- En un lenguaje funcional (Mit Scheme) una expresión lambda se puede escribir
  - (lambda (x) (\* x x))
    - Que significa que una variable libre x se multiplica por si misma (\*) y se produce el resultado de esta operación.
- Y se aplica luego a un valor
  - ((lambda (x) (\* x x)) 3)
    - Produciendo un 9

# LAMBDAS EN JAVA

- En Java se escribe
  - `(Integer x) -> {return x*x; }`
  - `(Integer x) -> x*x;`
- O si el tipo puede inferirse
  - `( x) -> x*x`
- Y se lo usa donde que java espera una función.
  - Pero en Java no hay objetos-función...
  - ...y la expresión lambda ni siquiera es un objeto.

The function body may or  
may not contain a return  
statement.

(arguments) -> {function body}

A lambda function can take in  
multiple arguments separated by  
commas.

Las expresiones son limitadas. Deben devolver inmediatamente un valor y no pueden contener variables, asignaciones o declaraciones como `if` o `for`. Para realizar operaciones más complejas, se puede usar un bloque de código con llaves. Si la expresión lambda necesita devolver un valor, entonces el bloque de código debe tener una `return` declaración.

```
(parameter1, parameter2) -> { code block }
```

# Vemos ejemplos →

```
Runnable noArguments = () -> System.out.println("Hello World"); ①
```

```
ActionListener oneArgument = event -> System.out.println("button clicked"); ②
```

```
Runnable multiStatement = () -> { ③  
    System.out.print("Hello");  
    System.out.println(" World");  
};
```

```
BinaryOperator<Long> add = (x, y) -> x + y; ④
```

```
BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y; ⑤
```

1. SIN ARGUMENTOS
2. UN ÚNICO ARGUMENTO, PUEDE IR SIN PARÉNTESIS
3. UN BLOQUE DE CÓDIGO A EJECUTAR, ENTRE { }
4. ADD, RECOGE EL RESULTADO DE SUMAR DOS NÚMEROS, E INFIERE EL TIPO DE LOS ARGUMENTOS
5. HACE LO MISMO QUE EL ANTERIOR, PERO CON LOS TIPOS EXPLICITOS DE LOS ARGUMENTOS

# INTERFACES FUNCIONALES

- Son interfaces con un solo método abstracto.

```
@FunctionalInterface  
interface Calc {  
    Integer op(Integer a, Integer b);  
}
```

- La annotation es optativa
- Pueden tener mas métodos concretos (default).
- Y ahora se puede asignar la expresion lambda

```
Calc sum =(Integer x, Integer y)->{return x+y;};  
Calc mult = (Integer x, Integer y) -> x * y;  
Calc rest =(x,y)->x-y;
```

# EJEMPLO COMPLETO

```
public class Lambda01 {  
    /*      *cálculo lambda      */  
  
    @FunctionalInterface  
    interface Calc {  
        Integer op(Integer a, Integer b);  
    }  
  
    public static void main(String[] args) {  
        System.out.print(" suma lambda on site 4+2=");  
        Calc sum = (Integer x, Integer y) -> {  
            return x + y;  
        };  
        Calc mult = (Integer x, Integer y) -> x * y;  
        Calc rest = (x, y) -> x - y;  
        System.out.println("sum " + sum.op(6, 4));  
        System.out.println("rest " + rest.op(6, 4));  
        System.out.println("mult " + mult.op(6, 4));  
    }  
}
```

sum 10  
rest 2  
mult 24

# TIPOS DE FUNCTIONAL INTERFACES

Functional Interface	Parameter Types	Return Type	Description
<code>Supplier&lt;T&gt;</code>	None	<code>T</code>	Supplies a value of type <code>T</code>
<code>Consumer&lt;T&gt;</code>	<code>T</code>	<code>void</code>	Consumes a value of type <code>T</code>
<code>BiConsumer&lt;T, U&gt;</code>	<code>T, U</code>	<code>void</code>	Consumes values of types <code>T</code> and <code>U</code>
<code>Predicate&lt;T&gt;</code>	<code>T</code>	<code>boolean</code>	A Boolean-valued function
<code>ToIntFunction&lt;T&gt;</code>	<code>T</code>	<code>int</code>	An int-, long-, or double-valued function
<code>ToLongFunction&lt;T&gt;</code>		<code>long</code>	
<code>ToDoubleFunction&lt;T&gt;</code>		<code>double</code>	
<code>IntFunction&lt;R&gt;</code>	<code>int</code>	<code>R</code>	A function with argument of type int, long, or double
<code>LongFunction&lt;R&gt;</code>	<code>long</code>		
<code>DoubleFunction&lt;R&gt;</code>	<code>double</code>		
<code>Function&lt;T, R&gt;</code>	<code>T</code>	<code>R</code>	A function with argument of type <code>T</code>
<code>BiFunction&lt;T, U, R&gt;</code>	<code>T, U</code>	<code>R</code>	A function with arguments of types <code>T</code> and <code>U</code>
<code>UnaryOperator&lt;T&gt;</code>	<code>T</code>	<code>T</code>	A unary operator on the type <code>T</code>
<code>BinaryOperator&lt;T&gt;</code>	<code>T, T</code>	<code>T</code>	A binary operator on the type <code>T</code>

# AGGREGATE OPERATIONS

- Son operaciones sobre un conjunto de datos (una collection por ejemplo).
  - Como resultado puede dar un valor u otro conjunto de datos
- En java se pueden componer.
  - Como los pipes del shell
- Están en el paquete Java.util.stream

# Stream

- Un Stream no almacena sus elementos.

Un Stream no cambia sus elementos, crea un nuevo Stream a partir de ellos.

Los Streams son lazy. Solo actúan cuando se los pide desde la salida.

```
static void imp(int n) {  
    System.out.print(" " + n);}
```

```
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140  
0 10 20 30 40
```

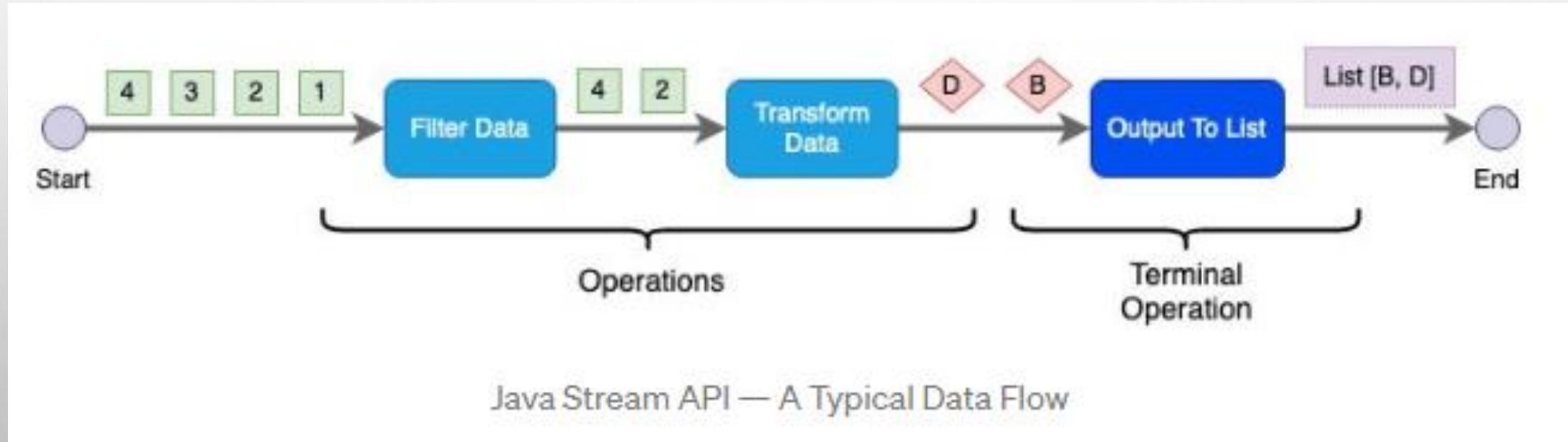
```
Stream.iterate(0, n -> n +10).limit(15).forEach(Stream_examples::imp);  
System.out.println("");
```

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);  
numbers.limit(5).forEach(Stream_examples::imp);  
System.out.println("");
```

ReferenciaAlObjeto::nombreDelMetodo

# JAVA.UTIL.STREAM

- El método `stream()` de `Collection` crea un stream a partir de una Collection.
  - Hay otras formas de crearlo, por ejemplo `generate` (`Supplier s`)
- Hay métodos intermedios que crean un stream a partir de otro.
- Hay métodos que recorren el stream operando sobre cada elemento.
- Hay métodos finales que reducen el stream a un valor
- `Object [] toArray()` devuelve el stream en un array .



# Ejemplo de grupos (Stream)

```
List <Integer> li= Arrays.asList(5, 7, 10, 25, 74);  
int suma=li.stream()  
    .mapToInt(x->x.intValue())  
    .sum();  
  
System.out.println("Suma de la lista "+suma);
```

```
long pares=li  
    .stream().filter(x->x % 2 ==0)  
    .count();  
System.out.println("Cantidad de pares "+pares);
```

```
Object[] arrInt=Stream.of(23,45,67,88,2,27)  
    .filter(x->x % 2 ==0)  
    .toArray();
```

Suma de la lista 121  
Cantidad de pares 2  
88  
2

# Operaciones sobre un conjunto de datos Stream

<R,A> R	<b>collect(Collector&lt;? super T,A,R&gt; collector)</b> Performs a <b>mutable reduction</b> operation on the elements of this stream using a Collector.
<R> R	<b>collect(Supplier&lt;R&gt; supplier, BiConsumer&lt;R,&gt; accumulator, BiConsumer&lt;R,R&gt; combiner)</b> Performs a <b>mutable reduction</b> operation on the elements of this stream.
static <T> Stream<T>	<b>concat(Stream&lt;? extends T&gt; a, Stream&lt;? extends T&gt; b)</b> Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	<b>count()</b> Returns the count of elements in this stream.
Stream<T>	<b>distinct()</b> Returns a stream consisting of the distinct elements (according to <code>Object.equals(Object)</code> ) of this stream.
<R> Stream<R>	<b>flatMap(Function&lt;? super T,&gt; mapper)</b> Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
DoubleStream	<b>flatMapToDouble(Function&lt;? super T,&gt; mapper)</b> Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	<b>flatMapToInt(Function&lt;? super T,&gt; mapper)</b> Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
LongStream	<b>flatMapToLong(Function&lt;? super T,&gt; mapper)</b> Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
void	<b>forEach(Consumer&lt;? super T&gt; action)</b> Performs an action for each element of this stream.

# OPTIONAL

- Es un container que puede o no tener un valor.
- Ver en el ejemplo que los Stream no pueden reusarse

```
List <String> qacL=Arrays.asList("Arriba", "Quilmes");
Stream <String> qacSt=qacL.stream();
List <String> cerveL=Arrays.asList("Arriba", "Cerveceros");
```

```
Optional <String> conQ=qacSt.filter(s->s.startsWith("Q")).findFirst();
System.out.println(qacL.stream().reduce(" ",String::concat)+" tiene empezando con Q "+conQ);

conQ=cerveL.stream().filter(s->s.startsWith("Q")).findFirst();
System.out.println(cerveL.stream().reduce(" ",String::concat)+" tiene empezando con Q "+conQ);
```



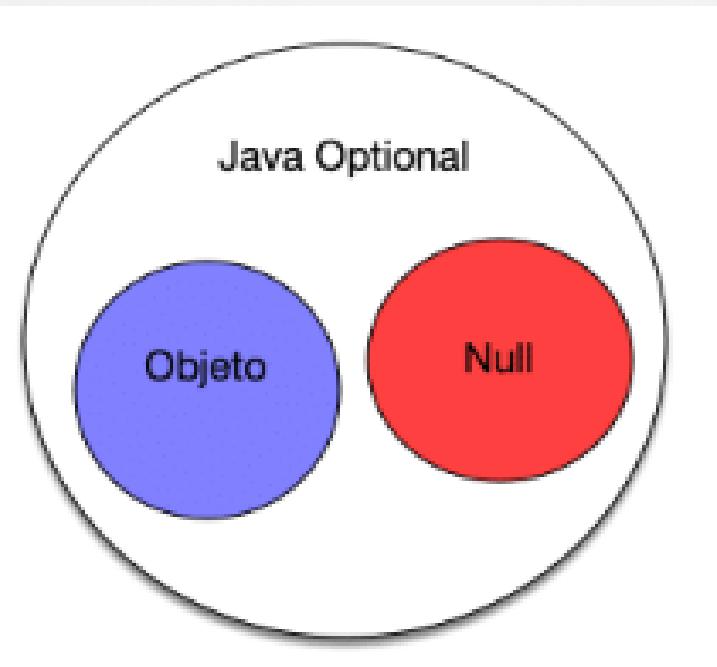
ArribaQuilmes tiene empezando con Q Optional[Quilmes]  
ArribaCerveceros tiene empezando con Q Optional.empty

# OPTIONAL

- El concepto de **java optional** hace referencia a una variable que puede tener un valor asignado o que puede contener un valor **null**.
- En muchas casuísticas nos encontramos con situaciones en las que un valor puede devolver nulo .
- Ante esta situación los programadores están obligados a comprobar si la variable es **null** antes de acceder a su valor.
- Ya que en el caso de ser nula e intentar acceder a algunas de sus propiedades el programa falla y lanza una excepción de **java.Lang.Nullpointerexpcion**

# OPTIONAL

- El primer valor es un objeto que nosotros necesitamos utilizar .
- El segundo valor es un valor “vacio” o empty en el caso de que nos encontremos con una ausencia de un valor concreto y queramos informar al programador de que estamos ante dicha situación.



# OPTIONAL

- Con una estructura if , podemos orientar al programador a que accede al valor del optional pero primero pregunte por él utilizando el método isPresent(). Este método nos devuelve true o false dependiendo si el optional contiene un valor o esta vacío:

