METRICA

CONSULTING

Programación funcional

Paradigmas de programación 01

Imperativo

- ordenes explicitas

Declarativo

- definición de qué deseamos lograr

Orientada a objetos

- abstracción
- herencia

Funcional

- define que hace y no como

Reactiva

- escucha y responde a eventos

02 Ejemplo imperativo

- Contar cuántos elementos de una lista son mayores de 10

```
List<Integer> numeros = Arrays.asList(18, 6, 4, 15, 55, 78, 12, 9, 8);

int contador = 0;
for(int numero : numeros) {
    if(numero > 10) {
        contador ++;
    }
}
System.out.println(contador);
```

02 Ejemplo funcional

- Contar cuántos elementos de una lista son mayores de 10

```
List<Integer> numeros = Arrays.asList(18, 6, 4, 15, 55, 78, 12, 9, 8);

long contador = numeros.stream().filter(n->n > 10).count();
System.out.println(contador);
```

03

Expresiones Lambda

- Función anónima
- Implementan interfaces funcionales

parámetros -> devuelve la operación

04 Interfaz funcional

- @FunctionalInterface

Interface IntUnaryOperator

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface IntUnaryOperator
```

Represents an operation on a single `int`-valued operand that produces an `int`-valued result. This is the primitive type specialization of `UnaryOperator` for `int`.

This is a functional interface whose functional method is `applyAsInt(int)`.

Since:
1.8

See Also:
`UnaryOperator`

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
`default IntUnaryOperator`	`andThen(IntUnaryOperator after)`	Returns a composed operator that first applies this operator to its input, and then applies the `after` operator to the result.
`int`	`applyAsInt(int operand)`	Applies this operator to the given operand.
`default IntUnaryOperator`	`compose(IntUnaryOperator before)`	Returns a composed operator that first applies the `before` operator to its input, and then applies this operator to the result.
`static IntUnaryOperator`	`identity()`	Returns a unary operator that always returns its input argument.

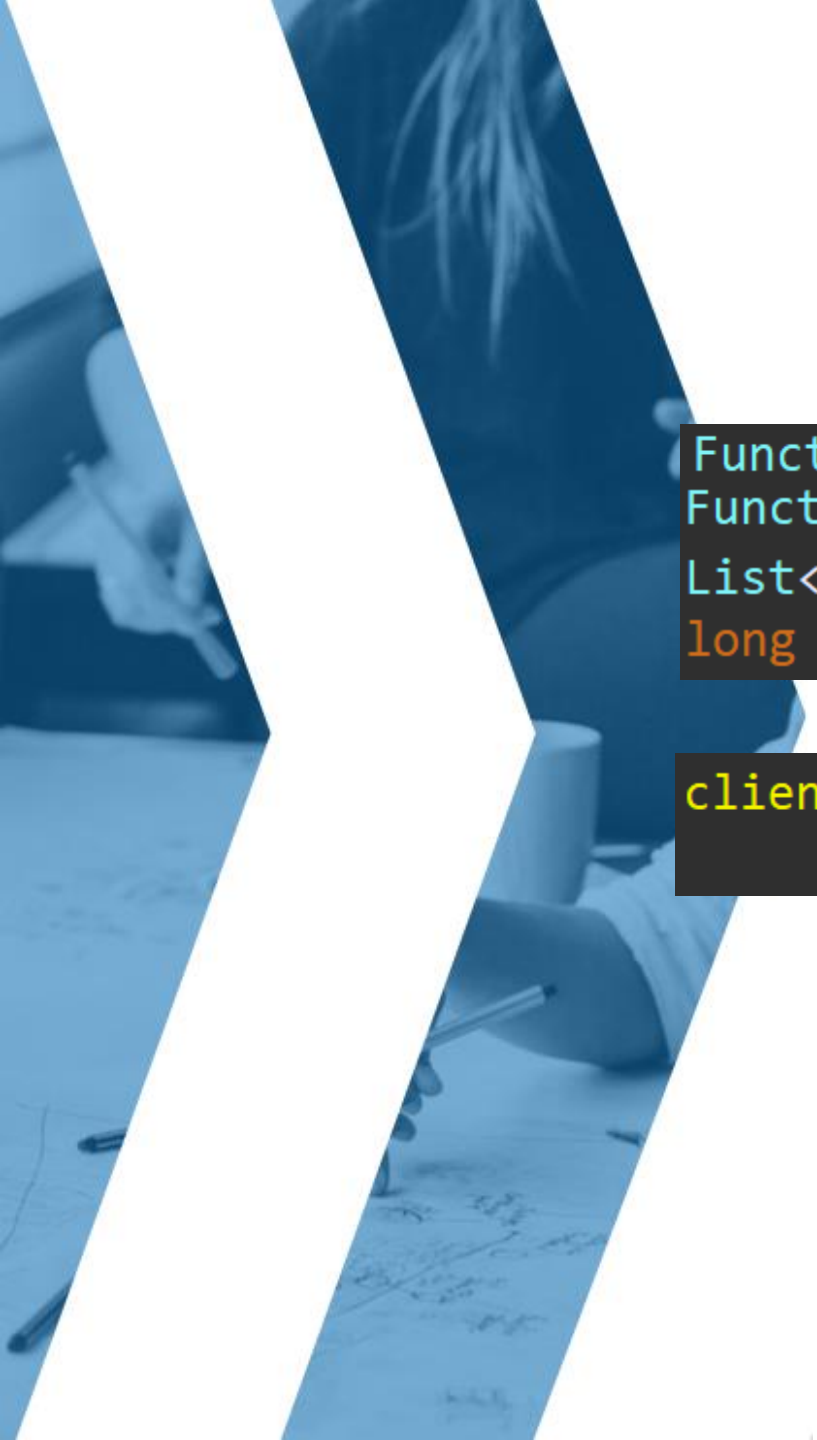
RICA CONSULTING

```
IntUnaryOperator function = new IntUnaryOperator() {
    @Override
    public int applyAsInt(int operand) {
        return operand*2;
    }
};

int value = function.applyAsInt(2);
```

```
IntUnaryOperator function = n -> n*2;

int value = function.applyAsInt(2);
```

Mas Functional Interface

Consumer<T>	Predicate<T>	Supplier<T>
Function<T,R>	UnaryOperator<T>	BinaryOperator<T>
ToXXXFunction<T>	Comparator<T>	. . .



standalone vs poly expresion

```
Function<String, Integer> s2i = (String str) -> str.length();
Function<String, Integer> s2inoType =  str -> str.length();
List<String> list =  Arrays.asList("a","b","c");
long size = list.stream().map(str -> str.length()).count();
```

```
client -> {if (client.getCredit()<50) return "Refuse";
                                      else return "Valid";};
```

standalone vs poly expresion

standalone

new String(“Hola”);

new ArrayList<String>();

poly

new ArrayList<>();

(x,y) -> x+y;

Posibles opciones

```
(int x, int y) -> {return x + y;};
```

```
(x, y) -> {return x + y;};
```

```
(x, y) -> x + y;
```

```
BiFunction<Integer, Integer> f =(int x,int y) -> x + y;
```

Restricciones @FunctionalInterface

Para asignar una lambda a una interfaz I:

- I debe ser una interfaz funcional
- mismo número de parámetros
- tipo de los parámetros compatibles
- el tipo de retorno debe ser compatible
- si se lanzan excepciones, I debe lanzar excepciones compatibles

```java
@FunctionalInterface
interface Adder {
    double add(double n1, double n2);
}
```

```java
@FunctionalInterface
interface Joiner {
    String join(String n1, String n2);
}
```

Dadas las dos interfaces anteriores, ¿cuáles de las siguientes instrucciones son correctas?

- (x + y) -> x+ y;
- Adder a = (x, y) -> x + y;
- Adder b = (x, y) -> x – y;
- Adder c = (int a, int b) -> a + b;

```java
@FunctionalInterface
interface Adder {
    double add(double n1, double n2);
}
```

```java
@FunctionalInterface
interface Joiner {
    String join(String n1, String n2);
}
```

Dadas las dos interfaces anteriores, ¿cuáles de las siguientes instrucciones son correctas?

- Adder a = (double x, double y) -> x + y;
- Adder b = (x, y) -> {return x – y;};
- var d = (a, b) -> a + b;
- Joiner j = (x, y) -> x + y;

```java
@FunctionalInterface
interface Adder {
    double add(double n1, double n2);
}
```

```java
@FunctionalInterface
interface Joiner {
    String join(String n1, String n2);
}
```

Dadas las dos interfaces anteriores, ¿cuáles de las siguientes instrucciones son correctas?

- Adder a = new Adder() {
  @Override
  public double add(double n1, double n2) {
  return n1 + n2;
  } };
- Joiner b = (x, y) -> x – y;
- Joiner c = (a, b) -> a.trim() + b.charAt(0);
- Joiner d = (double x,double y) -> String.valueOf(x) + y;

```java
@FunctionalInterface
interface Adder {
    double add(double n1, double n2);
}
```

```java
public void test(Adder adder) {
    System.out.println(adder.add(3, 2));
}
```

Dadas la interfaz y el método anteriores, ¿cuáles de las siguientes instrucciones son correctas?

- test((double x, double y) -> x + y);
- test((Adder)(x, y) -> x + y);
- Adder adder = (x, y) -> x + y;
  test(adder);
- test((x, y) -> x + y);

Funciones de orden superior

Formas de ordenar una lista

```
List<Integer> listaEnteros = …
Collections.sort(listaEnteros);
```

```
List<String> listaPalabras= …
listaPalabras.sort( (a,b) -> a.length()-b.length());
```

Funciones de orden superior

Formas de ordenar una lista

```
List<Empleados> listaEmpleados= …
listaEmpleados.sort(
(a,b) -> a.getNombre().compareTo(b.getNombre()));
```

```
List<Empleados> listaempleados= …
listaPalabras.sort(
Comparator.comparing(Empleado::getNombre));
```

Variable Capture

Una lambda puede acceder a variables externas sólo si

- son finales
- no son modificadas después de inicializar

```
public Adder increment() {
    double value = 1;
    Adder adder = (x, y) -> x+y + value;
    return adder;
}
```

```
public Adder increment() {
    double value = 1;
    Adder adder = (x, y) -> x+y + value;
    value++;
    return adder;
}
```