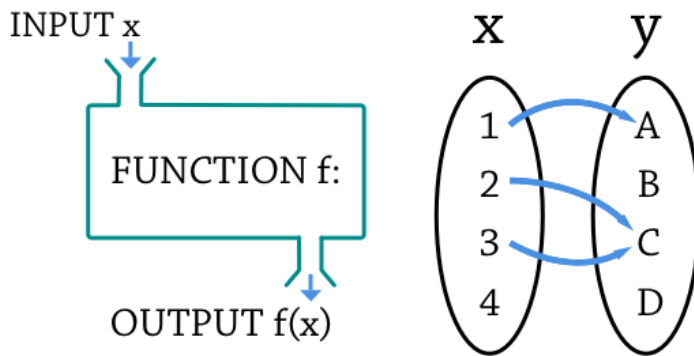


Programación Funcional

¿Qué es una **función**?

Una correspondencia entre dos conjuntos de elementos cualesquiera, no necesariamente numéricos, que asocia a cada elemento en el primer conjunto un único elemento del segundo.

(caja negra)



IMPORTANTE: Diferencia entre paradigmas de programación

En la Programación Funcional las definiciones de funciones son árboles de expresiones que asocian valores a otros valores, en lugar de una secuencia de instrucciones que actualizan el estado de ejecución del programa.

Paradigma Imperativo (énfasis en la ejecución de instrucciones/estado de variables) (POO)

Paradigma Declarativo (énfasis en la evaluación de expresiones/muchas expresiones relacionadas) (Programación Funcional)

Características de la Programación Funcional

Programas como Funciones

- Estructura de un programa: un programa consiste en una lista de definiciones de funciones.
- Ejecución de un programa: consiste en aplicar las funciones a sus argumentos.

Funciones Puras

- Determinismo: Ante los mismos argumentos, una función pura retorna el mismo valor. Esto conlleva la transparencia referencial (una función pura puede reemplazarse por su valor de retorno).
- Ausencia de efectos colaterales (side effects): Además del valor de retorno, una función pura no tiene ningún efecto visible sobre el ambiente desde el cual se la invoca.

→ Consecuencia práctica: Se hacen más fáciles las Pruebas Unitarias

```
1 package impura1;
2
3 public class Main {
4
5     private static final int UNO = 1;
6
7     public static void main(String[] args) {
8         int x = 5;
9         System.out.println("El sucesor de " + x + " es " + sucesor(x));
10    }
11
12    private static int sucesor(int x) {
13        return x + UNO;
14    }
15 }
```

Console X
<terminated> Main (1) [Java Application]
El sucesor de 5 es 6

Impura

Esta función es impura, pues es no determinística (depende de la constante UNO que se encuentra fuera de la función)

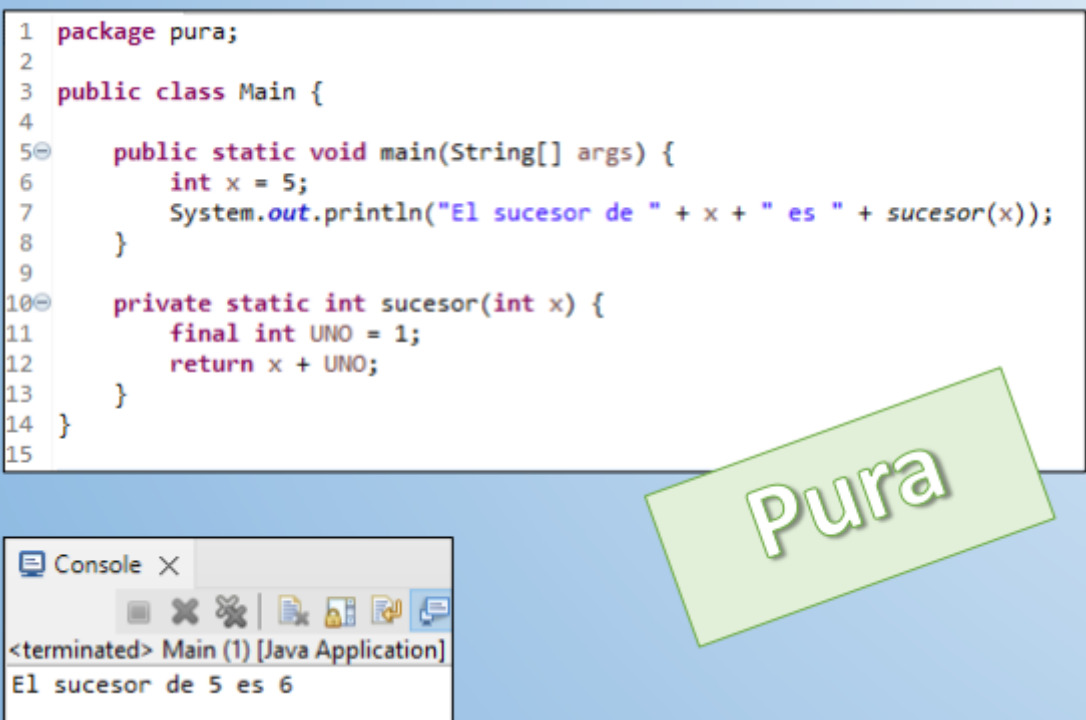
```
1 package impura2;
2
3 public class Main {
4
5     private static boolean yaUsada;
6
7     public static void main(String[] args) {
8         yaUsada = false;
9
10        if (yaUsada)
11            System.out.println("La funcion ya fue usada.");
12        else
13            System.out.println("La funcion aun no fue usada.");
14
15        int x = 5;
16        System.out.println("El sucesor de " + x + " es " + sucesor(x));
17
18        if (yaUsada)
19            System.out.println("La funcion ya fue usada.");
20        else
21            System.out.println("La funcion aun no fue usada.");
22    }
23
24    private static int sucesor(int x) {
25        final int UNO = 1;
26        yaUsada = true;
27        return x + UNO;
28    }
29 }
```

Console X
<terminated> Main (3) [Java Application]
La funcion aun no fue usada.
El sucesor de 5 es 6
La funcion ya fue usada.

Impura

Está modificando dentro de la función la variable yaUsada que forma parte del ambiente y no solo de la función misma (efecto sorpresa).

```
1 package pura;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         int x = 5;
7         System.out.println("El sucesor de " + x + " es " + sucesor(x));
8     }
9
10    private static int sucesor(int x) {
11        final int UNO = 1;
12        return x + UNO;
13    }
14 }
15
```



Pura

Es pura, pues la constante UNO vive dentro de la misma función.

Que una función sea impura no quiere decir que no funcione bien.

Datos inmutables

- Después de creado un dato, su estado no cambia más.
- Las funciones no modifican los datos recibidos como argumentos: los valores retornados siempre están alojados en otras ubicaciones de memoria (las estructuras de datos son persistentes).

Que las estructuras de datos sean persistentes significa que no se modifican las estructuras originales, sino que al sufrir algún cambio se genera una copia y agrega lo nuevo.

```
1 package mutable;
2
3 import java.util.ArrayList;
4
5 public class Main {
6
7     public static void main(String[] args) {
8
9         ArrayList<Integer> numeros = new ArrayList<>();
10        numeros.add(1);
11        numeros.add(2);
12        numeros.add(3);
13        numeros.add(4);
14        numeros.add(5);
15
16        for (int i = 0; i < numeros.size(); i++) {
17            numeros.set(i, numeros.get(i) + 1);
18        }
19
20        System.out.println(numeros);
21    }
22 }
```

Console X
<terminated> Main (4)
[2, 3, 4, 5, 6]

Mutable

Vemos que se trabaja y se modifica el mismo array.

```
1 package immutable;
2
3 import java.util.ArrayList;
4
5 public class Main {
6
7     public static void main(String[] args) {
8
9         final ArrayList<Integer> numeros = new ArrayList<>();
10        numeros.add(1);
11        numeros.add(2);
12        numeros.add(3);
13        numeros.add(4);
14        numeros.add(5);
15
16        ArrayList<Integer> nuevosNumeros = (ArrayList<Integer>) numeros.stream().map(value -> value + 1)
17            .collect(Collectors.toList());
18
19        System.out.println(numeros);
20        System.out.println(nuevosNumeros);
21    }
22 }
23 }
```

Console X
<terminated> Main (5)
[1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]

Immutable

Observamos que el array original no se modifica.

Explicación del código en POO:

Al objeto numeros le pasamos el mensaje stream() y nos retorna un objeto stream(). A ese objeto le pasamos un mensaje map, al que le pasamos un objeto que representa una función. Esto devuelve otro stream y le enviamos el mensaje Collect al que le pasamos el objeto que representa a la función Collectors.toList.

Explicación del código en Programación Funcional:

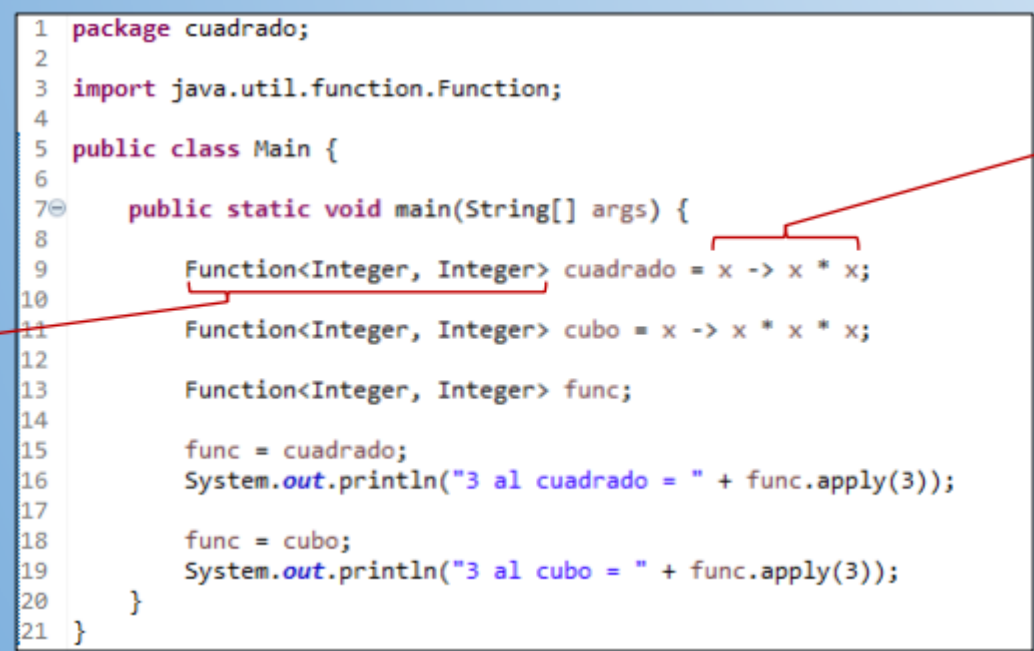
La función stream() recibe el array números y devuelve un stream(). La función de orden superior map recibe ese stream y una función sucesor (de primera clase) y devuelve un stream(). La función de orden superior Collect recibe ese resultado del map y una función sucesor. Y devuelve el retorno final que se guarda en nuevosNumeros

Con ambos paradigmas obtenemos el mismo resultado:

A partir de un array de números, generamos un stream (flujo de datos/bytes) y hacemos que en cada posición aparezca el siguiente de c/u y enlistamos en una colección cada uno.

Funciones de Primera Clase

- Soportan las operaciones posibles para las otras entidades del lenguaje
- Pueden:
 - ser asignadas a una variable,
 - ser pasadas como argumento y
 - ser retornadas por una función.



```
1 package cuadrado;
2
3 import java.util.function.Function;
4
5 public class Main {
6
7     public static void main(String[] args) {
8
9         Function<Integer, Integer> cuadrado = x -> x * x;
10
11         Function<Integer, Integer> cubo = x -> x * x * x;
12
13         Function<Integer, Integer> func;
14
15         func = cuadrado;
16         System.out.println("3 al cuadrado = " + func.apply(3));
17
18         func = cubo;
19         System.out.println("3 al cubo = " + func.apply(3));
20     }
21 }
```

Declaración de función:

Function<Entrada, Salida> Nombre = dfsgdfhdh

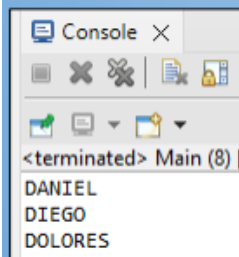
func.apply(3) aplica la entrada 3 a la función func

Funciones de Orden Superior

Tienen alguna de las siguientes capacidades (o ambas):

- Recibir funciones como argumento
- Retornar una función

```
1 package nombres;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 public class Main {
7
8     public static void main(String[] args) {
9
10         List<String> lista = Arrays.asList("Dolores", "Diego", "Santiago", "Mario", "Daniel", "Silvia", "Rita");
11
12         lista.stream().filter(s -> s.startsWith("D")).map(String::toUpperCase).sorted().forEach(System.out::println);
13     }
14 }
```



```
<terminated> Main (8) [
DANIEL
DIEGO
DOLORES
```

Method references

Predicado: funciones de programación que devuelven verdadero o falso según si el parámetro de entrada cumple una condición o no.

Analizo la función como POO

Al objeto lista se le pasa el mensaje stream() que devuelve un stream con el contenido de lista. A ese resultado se le pasa el mensaje filter y a ese mensaje un objeto que representa una función para filtrar y devolver un stream con solo con los elementos que comienzan con D. A ese objeto resultado, se le manda el mensaje map y este recibe un objeto que representa a una función (con los :: se hace una referencia) para que el objeto stream que devuelve sea con los elementos todos en mayúscula. A ese objeto resultado se le envía el mensaje sorted() que devuelve el stream ordenado y por último se le envía el mensaje forEach y a ese un objeto que representa una función que imprime elementos por pantalla, para imprimir cada uno de los elementos del steam final por pantalla.

Analizo la función como Programación Funcional

A la función stream se le pasa como entrada la lista y devuelve como salida un steam con los elementos de la lista. La función filter recibe el steam y una función sucesor (predicado) que devolverá un steam con los elementos que comienzan con D. La función map recibe el steam resultante anterior y una referencia a una función que pasa las letras de un string a mayúsculas. Se mapean con esta función cada elemento del steam. La función sorted lo

recibe y lo devuelve ordenado. Por último, la función `forEach` recibe el stream ordenado y una función sucesor para imprimir por pantalla. Se imprime cada elemento del stream por pantalla.

Composición de Funciones

- Consiste en combinar funciones para formar otra:

El valor resultante se calcula aplicando una función a los argumentos. Luego se aplica otra al resultado, y así sucesivamente.

```
1 package compose;
2
3 import java.util.function.BiFunction;
4
5 public class Main {
6
7     public static void main(String[] args) {
8
9         BiFunction<Double, Double, Double> suma = (x1, x2) -> x1 + x2;
10
11         BiFunction<Double, Double, Double> potencia = (x1, x2) -> Math.pow(x1, x2);
12
13         BiFunction<Double, Double, Double> sumaDePotencias = (x1, x2) -> suma.apply(potencia.apply(x1, x2),
14             potencia.apply(x1, x2));
15
16         BiFunction<Double, Double, Double> potenciaDeSuma = (x1, x2) -> potencia.apply(suma.apply(x1, x2),
17             suma.apply(x1, x2));
18
19         System.out.println("(2.0 ^ 3.0) + (2.0 ^ 3.0) = " + sumaDePotencias.apply(2.0, 3.0));
20         System.out.println("(2.0 + 3.0) ^ (2.0 + 3.0) = " + potenciaDeSuma.apply(2.0, 3.0));
21     }
22 }
```

Recursividad

Sin asignaciones de variables y ni construcciones estructuradas como la secuencia y la iteración, la repetición de instrucciones se logra mediante funciones de orden superior o funciones recursivas:

- Una función recursiva es aquella que contiene, en el bloque de instrucciones que la definen, una llamada a la propia función, permitiendo que una operación se realice una y otra vez, hasta alcanzar el caso base.

```
1 package fibo;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         int fibonacciNumber = getFibonacciNumberAt(6);
7         System.out.println(fibonacciNumber);
8     }
9
10    public static int getFibonacciNumberAt(int n) {
11        if (n < 2)
12            return n;
13        else
14            return getFibonacciNumberAt(n - 1) + getFibonacciNumberAt(n - 2);
15    }
16 }
```

Ejemplo de composición

```
>>> ns = list(range(100))
>>> ns
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
>>> def es_divisible_por_3(x): return x % 3 == 0
...
>>> es_divisible_por_3(8)
False
>>> es_divisible_por_3(15)
True
>>> list(filter(es_divisible_por_3, ns))
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
>>> list(filter(es_divisible_por_3, ns))
KeyboardInterrupt
>>> def cuad(x): return x * x
...
>>> list(map(cuad, ns))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
```

Ejemplo práctica

```
public class Main {
    IntStream.range(0, 101).filter(n -> n % 3 == 0).forEach(System.out::print(n));
}
```