



# PARADIGMAS II

## PROGRAMACION FUNCIONAL

GODOY JUAN MANUEL

.....

# ¿QUÉ ES UN PARADIGMA DE PROGRAMACIÓN?



# PARADIGMAS DE PROGRAMACIÓN

Pueden ser entendidos como patrones de pensamiento para la construcción de programas y resolución de problemas.

Un paradigma de programación es básicamente un marco de trabajo que contiene un conjunto de normas, conceptos y comportamiento a seguir.



for what is to be best in point of view

**Paradigm**

typical examples accepted per discipline a

¿CUANTOS  
PARADIGMAS  
HAY?

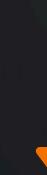
¿COMO LOS  
CLASIFICAMOS?



# PARADIGMAS IMPERATIVOS



Programación  
Procedural



Organiza el código en procedimientos o funciones. Es una de las formas más básicas de programación imperativa.



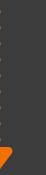
Programación Orientada a  
Objetos



Organiza el código en objetos que encapsulan estado (datos) y comportamiento (métodos). Usa conceptos como herencia, polimorfismo, y encapsulamiento.



Programación Basada en  
Eventos



Las acciones se disparan como respuesta a eventos, como clics de usuario o mensajes del sistema.

se centra en **cómo** se deben realizar las tareas, mediante instrucciones secuenciales que alteran el estado del programa.

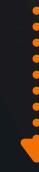


# PARADIGMAS DECLARATIVOS

se centra en **qué** debe hacerse, en lugar de cómo hacerlo. El programador describe el problema y el lenguaje de programación o sistema se encarga de encontrar la solución.



Programación  
Funcional



Se basa en funciones puras, inmutabilidad, y evaluación perezosa. Evita cambiar el estado y usa funciones de orden superior.



Programación  
Lógica



Se basa en reglas y hechos para definir relaciones entre los datos, y el motor de inferencia del lenguaje resuelve las consultas.



Programación  
Relacional



Especifica relaciones entre datos, y el sistema gestiona las consultas para extraer o manipular estos datos.



Programación  
Funcional Reactiva



Extiende la programación funcional con la capacidad de manejar flujos de datos y la propagación del cambio, útil en interfaces de usuario y sistemas reactivos.

# QUÉ ES UNA FUNCIÓN?

RELACIÓN

$f(x)$

Valor de  
Entrada

ENTRE DOS NÚMEROS

Valor de  
Salida



EN LA INFORMÁTICA

# PROGRAMACIÓN FUNCIONAL



ES UN PARADIGMA DE PROGRAMACIÓN QUE DESCRIBE LA CONSTRUCCIÓN DE UN PROGRAMA BASADO EN **FUNCIONES MATEMÁTICAS** Y EN LA COMPOSICIÓN DE ESTAS PARA RESOLVER PROBLEMAS. LA PROGRAMACIÓN FUNCIONAL PONE EN EL CENTRO LAS “FUNCIONES” COMO ELEMENTOS FUNDAMENTALES Y SE CONSIDERAN EL PILAR CENTRAL DEL PARADIGMA.

A DIFERENCIA DE LA PROGRAMACIÓN IMPERATIVA, QUE SE ENFOCA EN CÓMO SE DEBEN REALIZAR LAS ACCIONES PASO A PASO, LA PROGRAMACIÓN FUNCIONAL SE BASA EN **DECLARAR QUÉ OPERACIONES DEBEN REALIZARSE Y CONFÍA EN QUE EL SISTEMA DETERMINE CÓMO HACERLO DE MANERA INTERNA**.



# CIUDADANOS DE PRIMERA CLASE



CUANDO DECIMOS QUE ALGO ES UN "CIUDADANO DE PRIMERA CLASE", SIGNIFICA QUE PUEDE SER TRATADO DE LA MISMA MANERA QUE OTROS ELEMENTOS EN EL LENGUAJE. POR EJEMPLO, EN LENGUAJES QUE CONSIDERAN LAS FUNCIONES COMO CIUDADANOS DE PRIMERA CLASE, LAS FUNCIONES PUEDEN SER:

**ASIGNADAS A VARIABLES:** PUEDES ALMACENAR UNA FUNCIÓN EN UNA VARIABLE.

**PASADAS COMO ARGUMENTOS:** PUEDES PASAR UNA FUNCIÓN COMO ARGUMENTO A OTRA FUNCIÓN.

**RETORNADAS DESDE OTRAS FUNCIONES:** PUEDES DEVOLVER UNA FUNCIÓN DESDE OTRA FUNCIÓN.

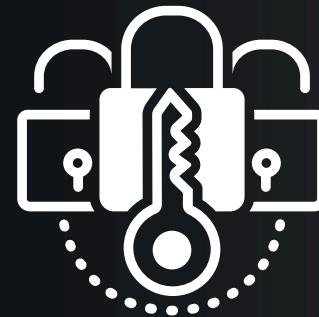


# PILARES DE PARADIGMA FUNCIONAL

---

 $f(x)$ 

Funciones de  
Orden Superior



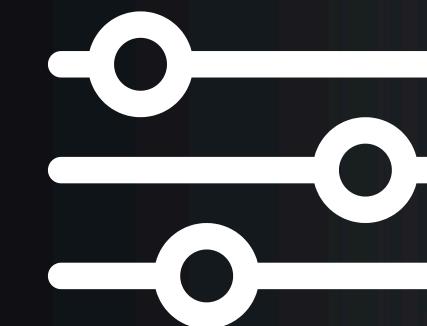
Immutabilidad



Lambdas

 $f(x)$ 

Funciones Puras



Streams



# INMUTABILIDAD

---

**Problema:** En la programación imperativa, el estado mutable (como las variables que cambian de valor) puede causar errores difíciles de rastrear, especialmente en programas concurrentes.

```
class CuentaBancaria {  
    private double saldo;  
  
    public void depositar(double cantidad) {  
        saldo += cantidad;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}
```

```
final class CuentaBancaria {  
    private final double saldo;  
  
    public CuentaBancaria(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public CuentaBancaria depositar(double cantidad) {  
        return new CuentaBancaria(this.saldo + cantidad);  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}
```



# INMUTABILIDAD

---

**En la programación funcional, los datos no cambian su estado una vez creados.  
Esto reduce errores relacionados con la modificación del estado compartido.**

```
//INMUTABILIDAD
final List<String> nombres = List.of("Juan", "Ana", "Luis");
nombres.add("Maria"); // Esto funciona o no?
```



# FUNCIONES PURAS

---

**Problema: Las funciones con efectos secundarios pueden llevar a resultados inesperados, complicando la depuración y la prueba del código.**

```
int total = 0;

public int agregar(int valor) {
    total += valor;
    return total;
}
```

**FUNCIÓN NO SEA PREDECIBLE**



# FUNCIONES PURAS

---

Son funciones que siempre devuelven el mismo resultado para los mismos argumentos y no tienen efectos secundarios, lo que facilita la previsibilidad y prueba del código.

```
/** Predicate: Evalúa una condición booleana para un argumento. Ejemplo:  
 *  
 */  
Predicate<Integer> isEven = n -> n % 2 == 0;  
  
System.out.println(isEven.test( t: 4)); // true o false?
```



# LAMBDAS

---

**Problema: La sintaxis de las funciones anónimas en lenguajes tradicionales puede ser verbosa y poco clara.**

```
List<String> nombres = Arrays.asList("Ana", "Carlos", "Beatriz");
Collections.sort(nombres, new Comparator<String>() {
    public int compare(String a, String b) {
        return a.compareTo(b);
    }
});
```

ESTE CÓDIGO ES INNECESARIAMENTE  
LARGO Y COMPLICADO PARA UNA  
SIMPLE COMPARACIÓN.

```
nombres.sort((a, b) -> a.compareTo(b));
```



# LAMBDAS

---

Son funciones anónimas que pueden definirse de manera concisa y pasarse como argumentos a otras funciones. Simplifican el código y son útiles para operaciones breves.

LambdaParameters -> LambdaBody

Método abstract	lambda expression
int m (int a, int b)	(a, b) -> a (int a, int b) -> a
int m (int a)	(a) -> a a -> a a -> { return a; }
int m ()	() -> 5
void m (int b)	(a) -> {}



# FUNCIONES DE ORDEN SUPERIOR

---

**Problema: El código repetitivo y poco modular es difícil de mantener y escalar.**

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
for (int n : numeros) {
    System.out.println(n * n); // Imprime el cuadrado de cada número
}
```

**EL CÓDIGO ES REPETITIVO Y DIFÍCIL DE MANTENER.**

```
for (int n : numeros) {
    System.out.println(n + 10); // Imprime cada número más 10
}
```

```
public static void aplicarOperacion(List<Integer> numeros, Function<Integer, Integer> operacion) {
    numeros.forEach(n -> System.out.println(operacion.apply(n)));
}

aplicarOperacion(numeros, n -> n * n); // Cuadrado
aplicarOperacion(numeros, n -> n + 10); // Suma 10
```



# FUNCIONES DE ORDEN SUPERIOR

---

Son funciones que toman otras funciones como argumentos o las devuelven como resultados, permitiendo construir funciones más complejas.

```
ejectFunction(s -> System.out.println(s.toUpperCase()), value: "imprime orden superior");

}

1 usage
public static void ejectFunction(Consumer<String> function, String value) {
    function.accept(value);
}
```



# STREAMS

---

**Problema: Procesar grandes cantidades de datos de manera eficiente y legible es difícil con bucles tradicionales.**

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> cuadrados = new ArrayList<>();
for (int n : numeros) {
    if (n % 2 == 0) {
        cuadrados.add(n * n);
    }
}
```

**ESTE ENFOQUE ES PROPENSO A ERRORES Y MENOS EXPRESIVO.**

```
List<Integer> cuadrados = numeros.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
```



# STREAMS

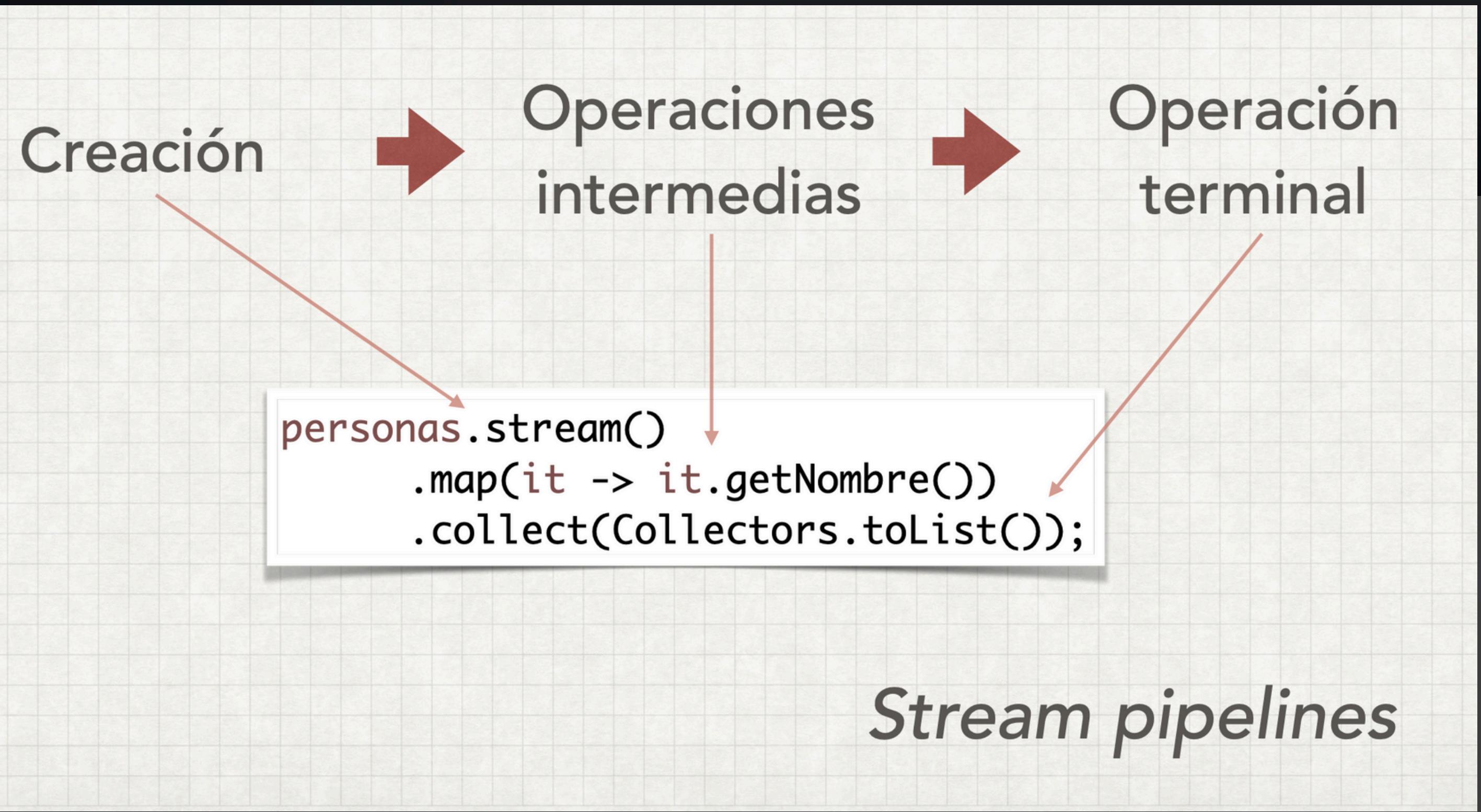
---

Son secuencias de datos que se pueden procesar de manera funcional. Permiten realizar operaciones como map, filter y reduce de manera eficiente, a menudo en un estilo perezoso, donde los elementos se procesan bajo demanda.

```
List<String> palabras = Arrays.asList("rojo", "verde", "azul", "Naranja");
palabras.stream()
    .filter(p -> p.length() > 4)
    .forEach(System.out::println); // que colores imprime?
```



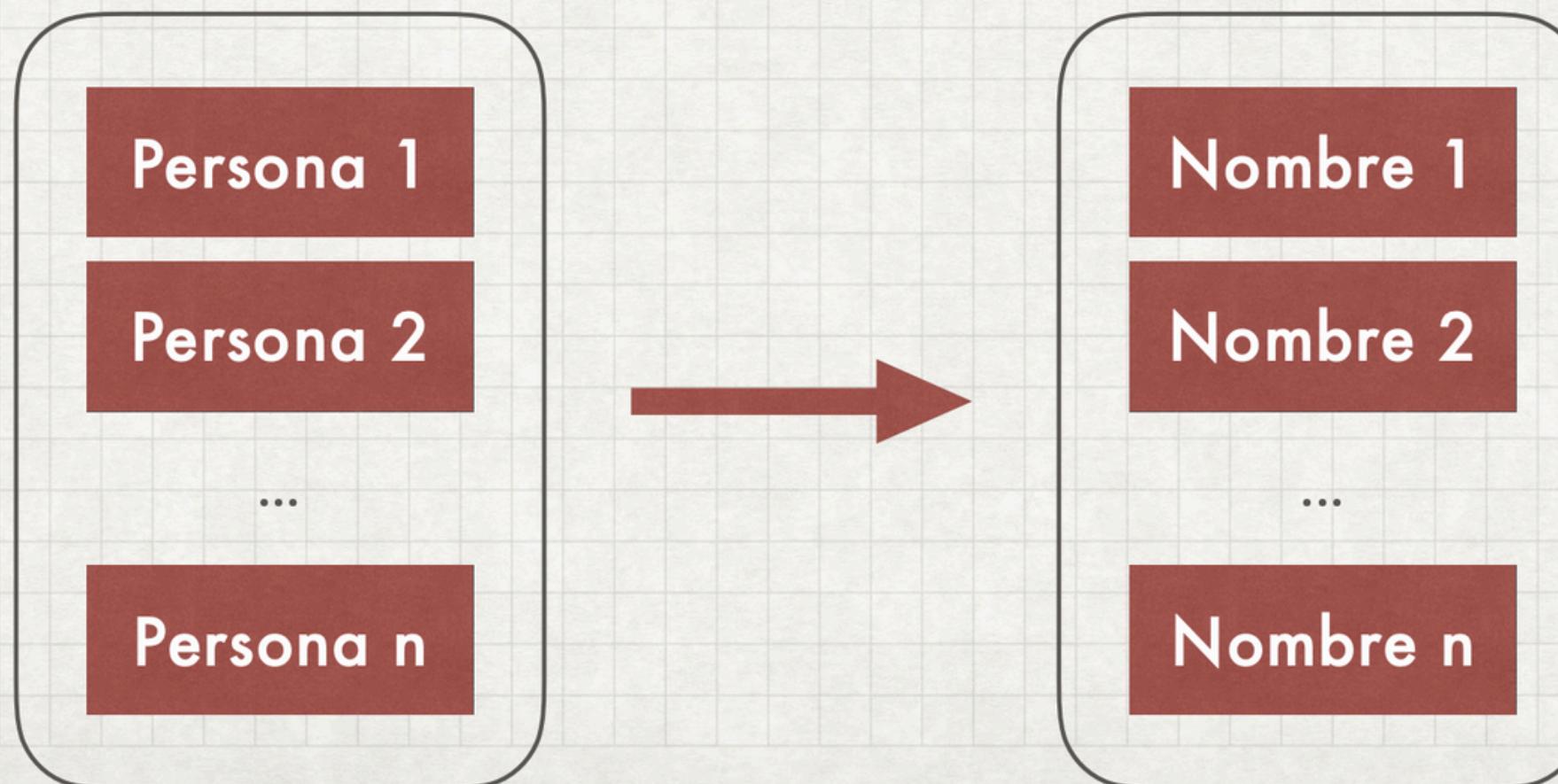
# STREAMS



# STREAMS

---

```
Stream<String> nombres = personas.map(it -> it.getNombre());
```



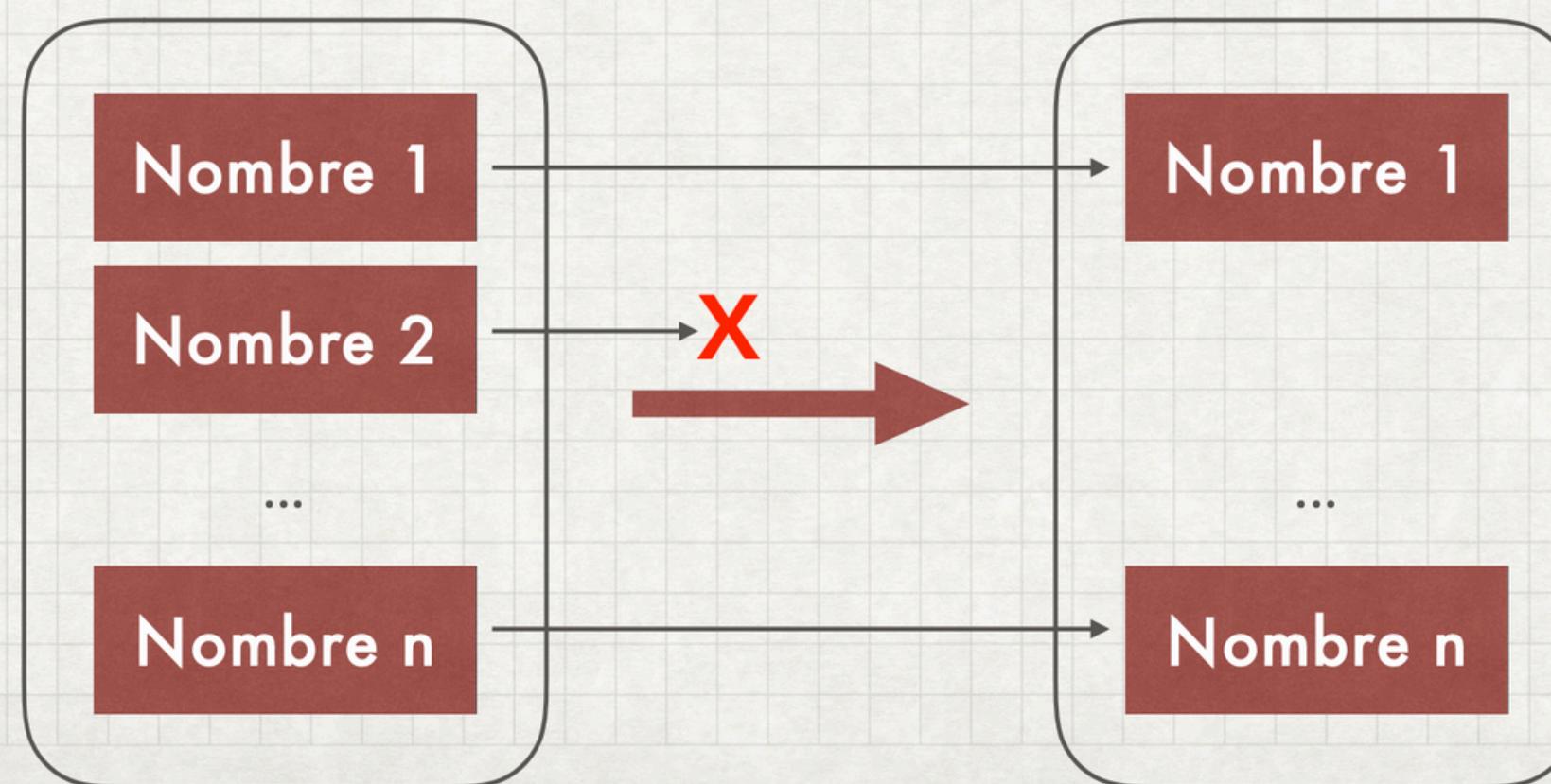
- **Mismo número**
- **Mismo orden**
- **Distinto tipo**



# STREAMS

---

```
Stream<String> nombresA = nombres.filter(it -> it.startsWith("A"));
```



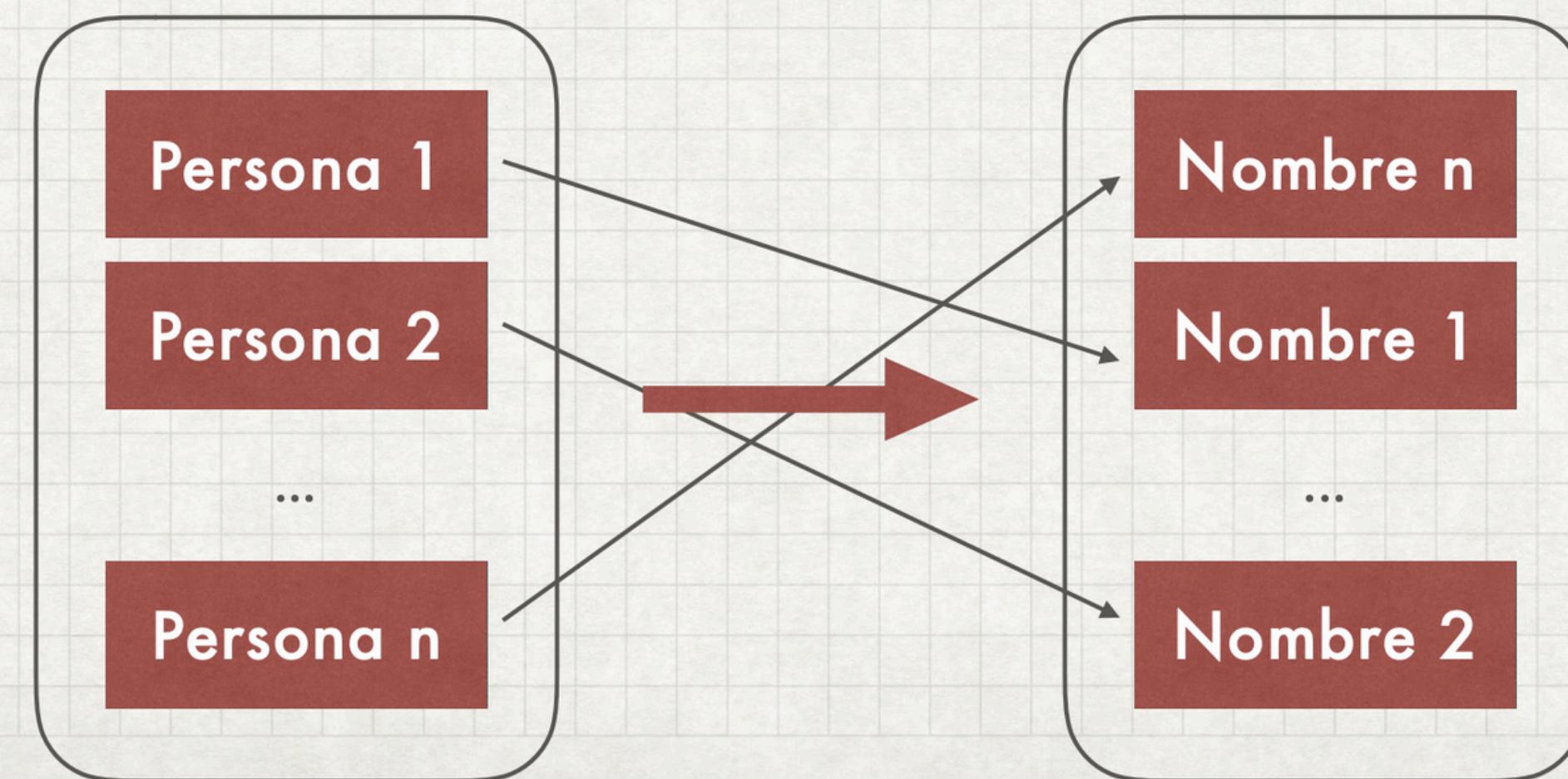
- Distinto número
- Mismo orden
- Mismo tipo



# STREAMS

---

```
nombres.sorted((o1, o2) -> o2.length() - o1.length())
```



- **Mismo número**
- **Distinto orden**
- **Mismo tipo**



# VENTAJAS

.....

01

## Inmutabilidad

Las variables no cambian su valor, lo que reduce errores y facilita la depuración.

02

## Facilidad de Paralelismo

Sin estado compartido, es más fácil ejecutar código en paralelo, mejorando el rendimiento.

05

## Pruebas Simplificadas

Las funciones puras son predecibles y fáciles de probar, mejorando la calidad del software.

03

## Funciones como Ciudadanos de Primera Clase

Las funciones se pueden pasar como argumentos o devolver, lo que promueve la modularidad y reutilización.

04

## Código Conciso y Expresivo

Permite escribir menos código, haciéndolo más legible y fácil de mantener.





FEEDBACK



LOONEY TUNES



*¡Eso es todo amigos!*