Tarea Independiente 29/09/2025

David Núñez Franco

October 12, 2025

Inventario de Conceptos Claves

- JS: nulidades y accesos con punto seguro
  - Operador de acceso a propiedades (elvis) ?.
  - ?? (nullish coalescing)
- function versus arrow
  - Clausura: objeto que tiene las ids locales de una función (incluyendo a this) y que apunta a la Clausura de su padre (función en donde se creó la función, que a su vez tiene su propia clausura y así hasta llegar a global si es el caso.
  - this. Dinámico versus estático. Relación con clausuras. En function this es cambiable (call, apply, bind) y se calcula en runtime dinámicamente. En Arrow this es fijo y se calcula por el compilador. Y no se puede cambiar.
  - Property 'name' de las funciones y arrow. Function versus arrow
- FP combinadores map, filter, reduce, forEch: lambda recibe elemento, posición y objeto
- Java:
  - List y su método stream (conecta objetos con streams y estos tienen los combinadores de FP. Son lazy.
  - Patrón collect(Collectors.toList()). Nota: hay toSet(), toMap() entre muchos otros colectores en Collectors (ver ejercicio adelante opcional). Use collect(Collectors.toList()) cuando ocupe una lista y el stream no tenga toList
- Cálculo lambda
  - Pruebas de identidades usando el Cálculo (no hubo tiempo en todos)

```
// Ejemplo en JS.

// Definamos:
const id = x => x
const compose = (f ,g) => x => f(g(x))
```

```
6
7                   // Probar: Para todo f se cumple -> compose(
                       f, id) = f
8                   // Prueba:
9                   compose(f, id) = ((f, g) => x => f(g(x)))(f,
                        id) // def de compose
10                  = x => f(id(x)) // beta reduccion
11                  = x => f((x => x)(x)) // def de id
12                  = x => f(x) // beta reduccion
13                  = f // x => f(x) y f hacen funcionalmente lo
                        mismo (eta-reduccion)
14
15                  // Por lo tanto, compose(f, id) = f
```

- Java: interfaces y tipos de lambdas
  - default methods en interfaces

```
1                   interface Saluter{
2                       default void salute(String msg) {
3                           System.out.println(msg);
4                       }
5                   }
6
7                   class Hello implements Saluter{} // no
                       implementa salute
8                   var hello = new Hello();
9                   hello.salute("Hola Mundo"); // usa el de la
                       interface --- IMPRIME Hola Mundo ---
```

  - SAM: single abstract method. Tiene uno y solo un método abstracto

```
1                   interface Saluter{
2                       void salute(String msg);
3                   }
4
5                   Saluter hello = msg -> System.out.println(
                       msg); // Saluter sirve como tipo para la
                       lambda
6                   hello,salute("Hola Mundo"); // --- IMPRIME
                       Hola Mundo ---
7
8                   // Tambien se pudo hacer asi
9                   Consumer<String> = msg -> System.out.println
                       (msg);
10                  msg.accept("Hola Mundo"); // --- IMPRIME
                       Hola Mundo ---
```

Ejercicio 1

```
record Tuple<X, Y>(X x, Y y){}
record Person(String name, int age){}
¿Qué tipo se necesita para que la siguiente lambda compile (sustituya ??? y ???? por l
??? personToTuple(???? p){
 return switch (p) {
  case Person(var name, var age) -> new Tuple<>(name, age);
  default -> throw new RuntimeException("not a person");
 };}
```

Solución

```java
// Punto 5: Convertir un Person en Tuple<String, Integer>
   usando Pattern-Matching en Switch

import java.util.function.*;

public class Ejercicio01 {
    // record generico que almacena dos valores de tipo
       arbitrario
    public record Tuple<X, Y> (X x, Y y) {}
    // recorde que representa una persona con nomre y edad
    public record Person(String name, int age) {}

    // convierte un objeto en una tupla (nombre, edad) si es
        de tipo Person. Usamos PM en Switch
    public static Tuple<String, Integer> personToTuple(
       Object p) {
        return switch (p) {
            case Person(var name, var age) -> new Tuple<>(
               name, age);
            default -> throw new RuntimeException("not a
               person");
        };
    }

    public static void main(String[] args) {
        // Instanciacion de prueba
        var p = new Person("David", 21);

        // Conversion aplicando PM
        var t1 = personToTuple(p);
        System.out.println("t1 = (" + t1.x() + ", " +  t1.y
           () + ")");

```

```
27                  // Equivalente pero en lambda function
28                  Function<Person, Tuple<String, Integer>> f = x ->
                        new Tuple<>(x.name(), x.age());
29                  var t2 = f.apply(p);
30                  System.out.println("t2 = (" + t2.x() + ", " +  t2.y
                        () + ")");
31              }
32          }
```

Listing 1: Soluc. en Java aplicando Pattern-Matching