Tarea Independiente 28/08/2025

David Núñez Franco

August 31, 2025

Inventario de Conceptos Claves

- Rol de un linker
- Idea de un jlink/jpackage en Java
- Ventajas
- Customizar un entorno para una app (menos tamaño de la app)
- Combinadores en FP
- forEach

Explicación breve

```
a.forEach(f) es como si fuera un bloque de código imperativo:

{
 f(a[0]);
 f(a[1]);
 ...
 f(a[a.length - 1]);
}
Por ejemplo:
[..."abcd"].forEach(e => console.log(e))
Imprime las letras a, b, c, d
```

- apply (@) combinador 'invisible' en métodos pero no en lambdas
- Diferencias entre JS versus Java: @ es variable, depende del tipo de lambda
- Function (@ = apply)
- Predicate (@ = test)

- UnaryOperator (@ = apply)
- zip
- unzip
- every
- some
- Caso de estudio: un modelo de objetos OOP en JS para un lenguaje
- ES6: estándar que permite class en JS
- class
- constructor
- rest (análodo de spread)
- super
- this
- get
- toString
- Modelo OOP
- ast

Ejercicio 1

```
1) ¿Qué tipo y @ serían válidos en Java para cada una de las siguientes declaraciones
a) ??? foo = () -> 666
print( foo.???() )

b) ??? goo = x -> print(x)
goo.???(666)

c) ??? hoo = (x, y) -> x*x - 2*x*y + y*y
print( hoo.???(666, 0) )
```

Solución

```java
import java.util.function.BiFunction;
import java.util.function.Consumer;
import java.util.function.IntBinaryOperator;
import java.util.function.Supplier;

public class Main {
    public static void main(String[] args) {
        /*
        a) ??? foo = () -> 666
        print( foo.???() )
        * */
        Supplier<Integer> foo = () -> 666;
        System.out.println(foo.get());

        /*
        * ??? goo = x -> print(x)
        goo.???(666)
        * */
        Consumer<Integer> goo = x -> System.out.println(x);
           // NOTA: PUEDO COLOCAR IntConsumer insteadOf
           Consumer
        goo.accept(666);

        /*
        * ??? hoo = (x, y) -> x*x - 2*x*y + y*y
        print( hoo.???(666, 0) )
        * */
        IntBinaryOperator hoo = (x, y) -> x*x - 2*x*y + y*y;
        System.out.println(hoo.applyAsInt(666, 0));

        /*Alternativa ejercicio c*/
        BiFunction<Integer, Integer, Integer> hoo2 = (x, y)
           -> x*x - 2*x*y + y*y;
        System.out.println(hoo2.apply(666, 0));
    }
}
```

Listing 1: Sol. en Java

Ejercicio 02

Considere el modelo ast en JS. Añada:
a) Un método same(other) a la classe Node que permita saber (falso o verdadero) si el
b) Un método que diga si Node es o no una hoja (no tiene head ni children)

c) Un método que calcule la altura de un Node (largo del camino más largo desde el nod

Solución

Este es nuestro modelo de AST, previo al desarrollo de los ejercicios:

```
        // No puede haber mas de un constructor, no hay sobrecarga
        export class Node {
            // ... = rest, queremos tener 0, 1, 2,...., n childrens
            constructor(head, ...children) {
                this.head = head;
                this.children = children;
            }
        }

        export class Num extends Node {
            constructor(value) {
                super(value);
            }

            get value() {
                return this.head;
            }

            toString() {
                return `${this.value}`;
            }
        }
```

Listing 2: ast.mjs

```
        import {Node, Num} from  './ast.mjs';

        function test_case_0() {
            const n = new Node("add", 1, 1, 2, 3)
            console.log("Node=", n)

            const num = new Num(666)
            console.log("Num as Node=", num)
            console.log("Num.value()=", num.value)
            console.log("Num.toString()=", num.toString())

        }

        function main() {
            test_case_0()
        }
```

```
17
18          main()
```

Listing 3: main.mjs

```
1           /* Funciones relacionadas al ejercicio 2.a:
2           Metodo same(other): retorna true si el nodo actual y "other"
                son equivalentes
3           (igual head y mismos children recursivamente). Retorna false
                si "other" no es Node.
4           */
5
6           // Verifica si un valor es instancia de Node
7           const isNode = (x) => x instanceof Node;
8
9           // Compara dos hijos:
10          // - Si ambos son Node, llama recursivamente a same
11          // - Si no, compara valores directamente
12          const sameChild = (a, b) =>
13          isNode(a) && isNode(b) ? a.same(b) : Object.is(a, b);
14
15          // Compara dos arrays de children:
16          // - Deben tener la misma longitud
17          // - Cada elemento debe ser igual en la misma posicion
18          const sameArray = (xs, ys) =>
19          xs.length === ys.length && xs.every((x, i) => sameChild(x,
                ys[i]));
20
21          /* 2.a: Compara el nodo actual con "other" */
22          same(other) {
23              return (
24              isNode(other) && // 1. Debe ser Node
25              Object.is(this.head, other.head) && // 2. head iguales
26              sameArray(this.children, other.children) // 3. children
                    iguales
27              );
28          }
29
30          function test_case_1() {
31              const a = new Node("add", new Num(1), new Num(2));
32              const b = new Node("add", new Num(1), new Num(2));
33              const c = new Node("sub", new Num(2), new Num(3));
34
35              console.log(a.same(b)); // true
36              console.log(a.same(c)); // false
37              console.log(a.same(42)); // false
38          }
```

Listing 4: 2a

```
        /* 2.b: Retorna true si el nodo es hoja.
        Un nodo es hoja cuando no tiene head y no tiene children */
        isLeaf() {
            return this.head == null && this.children.length === 0;
        }

        function test_case_2() {
            const a = new Node(null);
            const b = new Node("x");
            const c = new Node(null, 1, 2);

            console.log(a.isLeaf()); // true
            console.log(b.isLeaf()); // false
            console.log(c.isLeaf()); // false
        }
```

Listing 5: 2b

```
        /* Funciones relacionadas al ejercicio 2.c:
        Metodo height(): calcula la altura de un Node
        (camino mas largo desde el nodo hasta una hoja).
        */

        // Retorna la altura de un hijo:
        // - Si es Node, usa su metodo height
        // - Si no lo es, la altura se toma como 0
        const childHeight = (c) => (isNode(c) ? c.height() : 0);

        // Funcion auxiliar para quedarse con el mayor de dos
           valores
        const max = (a, b) => (a > b ? a : b);

        // Calcula la altura de un Node n:
        // - Si no tiene hijos -> altura 0
        // - Si tiene hijos -> 1 + maximo de las alturas de sus
           hijos
        const heightOf = (n) =>
        n.children.length === 0 ? 0 : 1 + n.children.map(childHeight
           ).reduce(max);

        /* 2.c: Retorna la altura del nodo actual */
        height() {
            return heightOf(this);
        }

        function test_case_3() {
            const leaf = new Num(7);
```

```
        const tree = new Node("add", leaf, new Node("null", new
            Num(1), new Num(2)));

        console.log(leaf.height()); // 0
        console.log(tree.height()); // 2
    }
```

Listing 6: 2c

Ejercicio 3

Añada un AST para identificador (Ident)

Solución

```
    // Clase Ident: modela un identificador dentro del AST
    export class Ident extends Node {
        // name = lexema (ej. "x"); no tiene children
        constructor(name) {
            super(name);
        }

        // Devuelve el lexema del identificador
        get name() {
            return this.head;
        }

        // Representacion como string (solo el nombre)
        toString() {
            return `${this.name}`;
        }
    }

    function test_case_4() {
        const x = new Ident("x");
        const y = new Ident("y");

        console.log(String(x)); // "x"
        console.log(x.same(new Ident("x"))); // true
        console.log(x.same(y)); // false
    }
```

(

Ejercicio 4)

Añada un AST Operation(operator, ...args) que modele una operación como una suma o una multiplicación.

Solución

```
        // Modela una operacion n-aria ('+', '-', '*', '/')
        export class Operation extends Node {
            constructor(operator, ...args) {
                super(operator, ...args);
                this.operator = operator;
            }

            // Lista de argumentos de la operacion
            get args() {
                return this.children;
            }

            // Cantidad de argumentos
            arity() {
                return this.children.length;
            }

            toString() {
                const argsStr = this.args.map(String).join(", ");
                return `${this.operator}(${argsStr})`;
            }
        }

        function test_case_5() {
            const x = new Ident("x");
            const expr = new Operation(
            "+",
            x,
            new Num(2),
            new Operation("*", new Num(3), x)
            );

            console.log(String(expr)); // "+(x, 2, *(3, x))"
            console.log(expr.arity()); // 3
        }
```