

TRABAJO INTEGRADOR 2025

CONCEPTOS Y PARADIGMAS DE LENGUAJES DE PROGRAMACIÓN

Integrantes *Grupo 25*:

- Ajala Mariela — 01959/7
- Escobares Conrado — 25067/6
- Bieta Juan Ignacio — 23712/4
- Nuñez Maximiliano — 23980/4

Lenguajes asignados:

- Java
- Javascript



INTRODUCCIÓN GENERAL:

El presente trabajo tiene como objetivo desarrollar un análisis sobre los conceptos dictados por la cátedra “*Conceptos y Paradigmas de Lenguajes de Programación*”, donde se pueda observar la integración de los mismos sobre los lenguajes asignados: **Java** y **JavaScript**. Esto a través del estudio de ciertos aspectos de cada lenguaje con el fin de lograr una mejor comprensión de los mismos, junto a una explicación clara y concisa basada en nuestros aprendizajes adquiridos.

TIPOS DE PARÁMETROS SOPORTADOS Y MODO DE LIGADURA:

En esta sección se desarrollará una explicación y se ejemplificarán los distintos *tipos de parámetros* y *modos de ligadura* soportados por los lenguajes asignados, comprendiendo así, no sólo el modo en el que estos conceptos son aplicados, sino también los fundamentos de los lenguajes y su funcionamiento.

• Java



| | |
|-------------------------|---------------------------------------------------------------------------------------------|
| Características | Tipado obligatorio: Sí (estático y fuerte) |
| Ligadura: Posicional | Modos de pasaje: Solo por valor (aunque con objetos se pasa su <i>referencia</i> por valor) |

```
public class Alumno
{
    private String apellido;
    private String nombre;
    private int edad;
    private double promedio;

    public Alumno(String apellido, String nombre, int edad, double promedio)
    {
        this.apellido = apellido;
        this.nombre = nombre;
        this.edad = edad;
        this.promedio = promedio;
    }

    public void setPromedio(double p)
    {
        this.promedio = p;
    }
}
```

Para comenzar, se crea una clase *Alumno* con sus variables, un setter, y su constructor

Luego, en el cuerpo del **main**, se crea una instancia de esta clase, además de un *entero* **n**, ambos con valores iniciales. Se **imprime** en pantalla el valor inicial del promedio del alumno (**7.32**) y del entero (**0**). Luego se invoca la función **modificarDato**, enviando los parámetros

```

public class CPLP {
    public static void main(String[] args)
    {
        int n = 0;
        Alumno a = new Alumno("Juan", "Pérez", 18, 7.32);
        System.out.println("Dentro del main: " + a.promedio);
        System.out.println("Dentro del main: " + n);
        modificarDato(a, n);
        System.out.println("Dentro del main: " + a.promedio);
        System.out.println("Dentro del main: " + n);
        System.out.println("Dentro del main: " + a.nombre);
        modificarConexion(a);
        System.out.println("Dentro del main: " + a.nombre);
    }

    public static void modificarDato(Alumno alumno, int numero)
    {
        alumno.setPromedio(7.39);
        System.out.println("Dentro del modificar: " + alumno.promedio);
        numero = 1;
        System.out.println("Dentro del modificar: " + numero);
    }

    public static void modificarConexion(Alumno a)
    {
        a = new Alumno ("Juana", "Fernández", 19, 8.37);
        System.out.println("Dentro del modificar: " + a.nombre);
    }
}

```

a y *n*, dentro de la cual se setea el promedio en **7.39**, y *n* en **1**. Se incluyen impresiones de ambos para mostrar que mientras nos encontremos **dentro de la unidad**, los valores son los asignados en esta.

```

Dentro del main: 7.32
Dentro del main: 0
Dentro del modificar: 7.39
Dentro del modificar: 1
Dentro del main: 7.39
Dentro del main: 0
Dentro del main: Juan
Dentro del modificar: Juana
Dentro del main: Juan

```

Sin embargo, al **salir de la función**, se ven las diferencias: el valor de *n* es el mismo que al entrar porque es un **primitivo** y su pasaje es por **valor**. En cambio, **alumno** sí refleja la **modificación** porque más allá de que el pasaje es por **valor**, lo que se copia es la **referencia** al **objeto** que está alojado en la **heap**. Al

imprimir *n* luego de invocar la función modificar se puede observar la **no** modificación de éste.

También se puede apreciar que Java se trata de un lenguaje con **ligadura posicional** ya que a pesar de la diferencia de nombre entre los parámetros reales y formales, estos se asocian por la posición en la que se escriben. Si se intenta hacer por nombre da error.

Para confirmar que los parámetros **no primitivos** también son pasados **por valor**, se llama a **modificarConexion**, donde se cambia la **referencia de la variable a**. Dentro del proceso, *a* apunta al nuevo objeto (con nombre "*Juana*"), mientras que al salir del mismo, la referencia **se mantiene igual** (al alumno "*Juan*").

- JavaScript



| Características | Tipado obligatorio: No (dinámico y débil) |
|----------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ligadura: <i>Posicional</i> (por defecto), por <i>nombre</i> (simulado con <i>destructuring</i>) | Modos de pasaje: Solo por <i>valor</i> (aunque con <i>objetos</i> , colecciones como <i>arrays</i> y <i>funciones</i> se pasa la <i>referencia</i> por valor). Además admite valores por defecto. |

```
let numero = 5 // primitivo number
let miObjeto = { nombre : "Pepe" } // objeto literal {}
let miArray = [1,2,3] // array literal []
let miFuncion = (a) => {console.log(a)} // función flecha

// Se puede llamar a funciones declaradas más abajo en JavaScript (hoisting)
modificarParametros(numero, miObjeto, miArray, miFuncion)
console.log("Fuera de la función:", numero, miObjeto, miArray)
// Fuera de la función: 5 { nombre: 'Pepe' } [ 1, 2, 3, 4 ]
// No se especifican los tipos de parámetros JavaScript
function modificarParametros(unNumero, unObjeto, unArray, unaFuncion) {
  unNumero = 10 // no afecta al número original
  unObjeto.nombre = "Pedro" // modifica el objeto original
  unObjeto = { otroValor : 50 } // no afecta al objeto original
  unArray.push(4) // modifica el array original
  // nueva función con operador spread (...) para mostrar todos los parámetros
  unaFuncion = (...a) => {console.log(...a)} // no afecta a la original
  unaFuncion("Dentro de la función:", unNumero, unObjeto, unArray)
  // Dentro de la función: 10 { otroValor: 50 } [ 1, 2, 3, 4 ]
}
// Se puede simular el pasaje de parámetros por nombre utilizando objetos
function ejemploPorNombre({nombre, edad}){
  // con `` se pueden incluir variables dentro de cadenas de texto
  console.log(`Soy ${nombre}, tengo ${edad} años`)
  // Soy Alfonso, tengo 30 años
}
// Se utiliza destructuring para acceder a los parámetros
ejemploPorNombre({edad: 30, nombre: "Alfonso"})
```

Al igual que en *Java*, en **JavaScript** el pasaje es por **valor** y en *objetos*, *arrays* y *funciones* por **referencia**.

Las **funciones** tienen mucha plasticidad en JS, ya que se pueden pasar como argumentos, retornar, guardar en variables, usar antes de su declaración (**Hoisting**), no tener nombres, ser resumidas y usadas como callbacks (**Arrow functions**), etc.

FORTALEZA DEL SISTEMA DE TIPOS:

A continuación, se abordará el tema de la fortaleza del sistema de tipos de ambos lenguajes, desarrollando cómo funciona en cada uno de ellos, y ejemplificando con código que respalden lo afirmado. Estos fragmentos permitirán observar de manera concreta las reglas que cada lenguaje impone, así como las similitudes y diferencias entre ellos.

Java es un lenguaje **estáticamente tipado**, es decir que cada variable y expresión deben definirse con un tipo, el cual es *conocido* y *verificado* en tiempo de *compilación*.

Java es, además, un lenguaje **fuertemente tipado**, lo que implica que se limitan tanto los *valores* asignables a una variable como las *operaciones* que se pueden hacer con ella. Esto permite que los errores de tipo sean *detectados* también en tiempo de *compilación*.

En este ejemplo se puede observar que es necesario incluir el tipo de la variable `x` (`int`) al declararla, así demostrando que Java se trata de un lenguaje **estáticamente tipado**. Además, al intentar asignarle un valor de tipo `String` a esta variable, se produce un error en *compilación*, así confirmando que Java se trata de un lenguaje **fuertemente tipado**.

```
public class SistemaTipos
{
    public static void main(String[] args)
    {
        int x = 5;
        x = "Cinco";
        System.out.println(x);
    }
}
```

Por otro lado, *JavaScript* se trata de un lenguaje muy distinto en este aspecto.

En primera instancia, se trata de un lenguaje **dinámicamente tipado**. Esto quiere decir que las variables *no necesitan* un tipo asociado a ellas al momento de declararlas, ya que esto se verifica en tiempo de *ejecución*.

Además, *JavaScript* es un lenguaje **débilmente tipado**, por lo que esta *limitación* presente en *Java* no existe, sino que se hacen *conversiones automáticas* entre tipos, permitiendo un rango mayor de operaciones.

```
let x = 5;
console.log(typeof x, x);
x = "Cinco";
console.log(typeof x, x);
console.log(5 + "5");
```

Este fragmento de código ejemplifica lo afirmado previamente, y contrasta con lo explicado en la sección de *Java*. Primero, la variable `x` se declara sin especificar ningún tipo, lo cual refleja el comportamiento de un lenguaje **dinámicamente tipado**.

```
number 5
string Cinco
55
```

Luego, la reasignación de un valor de tipo *String* a una variable previamente *numérica*; además de una operación de suma entre un número y una cadena, demuestran el **tipado débil** de *JavaScript*, permitiendo la conversión automática entre tipos.

EXCEPCIONES:

En esta sección se explicará y demostrará qué son las excepciones y cómo son gestionadas en ambos lenguajes. Se dará una explicación básica para el entendimiento del tema, y se detallarán todos los conceptos vitales para el funcionamiento correcto de los mismos.

```
public static int division (int a, int b) throws ArithmeticException
{
    if (b == 0)
    {
        throw new ArithmeticException("Error: no se puede dividir por 0.");
    }
    return a / b;
}
```

En este código se puede apreciar cómo funcionan las excepciones en **Java**.

En el cuerpo del *main* se incluye una cláusula **try-catch**, que intenta ejecutar cierto código y

```
public class CPLP_Excepciones {  
    public static void main(String[] args)  
    {  
        try  
        {  
            int numero = division(5, 0);  
            System.out.println("No se llega acá");  
        }  
        catch (ArithmeticException e)  
        {  
            System.out.println(e.getMessage());  
        }  
        finally  
        {  
            System.out.println("Esto se ejecuta siempre :)");  
        }  
  
        System.out.println("La ejecución del programa continúa luego de la excepción.");  
    }  
}
```

actuar en consecuencia. En este, se le asigna a la variable *numero* el valor resultado del proceso *division*, pasando como parámetros el 5 y el 0.

Dentro de la unidad se verifica que el segundo parámetro no sea 0; como en este caso sí lo es –y la división por cero no está permitida– se lanza una *ArithmeticException*, la cual es gestionada por el bloque **catch**, que imprime el mensaje definido al lanzarla. Una vez que se dio la excepción, se ejecuta la cláusula **finally**, que es accedida siempre independientemente de la existencia de excepciones.

Finalmente, se continúa el flujo del programa por fuera del bloque **try-catch-finally**, así mostrando el modelo de **terminación** que posee **Java**, nunca accediendo a la línea siguiente a la asignación de *numero*.

JavaScript, al igual que *Java*, replica el modelo de **terminación** para la gestión de excepciones. Sin embargo, *JavaScript* no requiere **declarar** explícitamente las excepciones que puede lanzar una función ya que es un lenguaje **dinámicamente tipado**, a diferencia de *Java*, y además permite lanzar **cualquier tipo** de valor como excepción. La gestión de excepciones se lleva a cabo mediante cuatro palabras clave: **try**, **catch**, **throw**, **finally**.

```
function division(a, b) {  
    if (b === 0) {  
        throw "División por cero no permitida";  
    }  
    return a / b;  
}  
  
function main() {  
    try {  
        console.log("Iniciando cálculo...");  
        let numero = division(5, 0);  
        console.log("Esta línea nunca se ejecutará");  
        console.log("Resultado: " + numero);  
    } catch (excepcion) {  
        console.log("Excepción capturada: " + excepcion);  
    } finally {  
        console.log("Bloque finally ejecutado");  
    }  
    console.log("Continuando ejecución después del try-catch-finally");  
}
```

En este código se puede apreciar cómo funcionan las excepciones en **JavaScript**. Dentro de la función *main* se incluye una cláusula **try-catch**, que intenta asignar a la variable “*numero*” el resultado del proceso “*division*”, pasando como parámetros el 5 y el 0.

Dentro de la función se verifica que el segundo parámetro no sea 0; como en este caso sí lo es, y la división por cero no está permitida, se lanza un valor de tipo *String* como **excepción** (JavaScript también permite lanzar **objetos Error** más estructurados), el cual es gestionado por el bloque **catch**, que imprime el mensaje definido al lanzarla.

Una vez que se dio la excepción, se ejecuta la cláusula **finally**, que siempre se ejecuta, incluso si no hay excepciones.

Finalmente, se continúa el flujo del programa por fuera del bloque **try-catch-finally**, así demostrando el modelo de **terminación** que posee **JavaScript**, nunca accediendo a la línea siguiente de la asignación de “numero”.

CONCLUSIÓN:

Este trabajo nos permitió **profundizar** en ciertos **conceptos** que ya poseíamos de **Java** por utilizar el lenguaje en otras instancias, pero no de una forma tan **estructurada** y **exhaustiva** como fue necesario para realizar este trabajo. Esto se pudo ver, por ejemplo, a la hora del **pasaje de parámetros**: sabíamos que se podían modificar objetos, pero no que era debido al pasaje de una **copia** de la **referencia**, y no la referencia en sí. También nos ayudó a ahondar en temas como el **tipado del lenguaje** o el **manejo de excepciones**, que entendíamos por el uso de **Java**, pero no desde una perspectiva tan **teórica**.

Por el contrario, **JavaScript** fue un terreno nuevo para nosotros. Conocíamos ciertos aspectos básicos, pero fue un aprendizaje mucho más **profundo**. Comenzando por conceptos simples como su **sintaxis**, hasta su sistema de **tipos** o **manejo de excepciones**; pudimos **analizar** un lenguaje que resultó muy **distinto** a lo que nos era familiar.

En general, este trabajo nos permitió **contrastar** ambos lenguajes, **interiorizando** sus características principales, así como las **similitudes** y **diferencias** entre ellos.

Además de las **herramientas** para poder realizar este **proceso** con cualquier otro lenguaje.

BIBLIOGRAFÍA

| | | |
|--------------|-----------------------------------------------------------|----------------------------------------------------|
| Java: | Java SE 24 Documentación Oracle | Documentación Java SE 24 (Oracle) |
| | Clase 10 CPLP (excepciones en Java) | Clase 6 CPLP (parámetros en Java) |
| | Página oficial java, parámetros, ligadura | Parámetros en Java (Open Webinars) |

| | | |
|--------------------|-----------------------------------------------|-------------------------------------------------------|
| JavaScript: | Documentación ECMAScript 262 | Documentación JavaScript (MDN) |
| | Funciones en JavaScript (MDN) | Clase 6 CPLP (valores por defecto JS) |