

Facultad de Informática - UNLP

## CONCEPTOS Y PARADIGMAS DE LENGUAJES DE PROGRAMACIÓN

### TRABAJO INTEGRADOR 2025

Integrantes:

- Acosta Alfano Juan Francisco – Legajo 24077/4
- Guaymas Matías Julián– Legajo 23061/0
- Lamiral Matheo Joaquín – Legajo 23466/9
- Lima Francisco – Legajo 23421/6

Grupo: N° 6

Lenguajes asignados: Java, JavaScript

Fecha de entrega: 21 de Junio de 2025

## Introducción General

Este trabajo compara Java y JavaScript en tres aspectos: parámetros, sistema de tipos y manejo de excepciones. Con ejemplos, se mostrará cómo cada lenguaje define y ejecuta estos mecanismos.

### A. Parámetros

En esta sección se hará una explicación sobre parámetros de Java y JavaScript. Los parámetros son variables que se utilizan para pasar información a funciones o métodos.

#### Java:

La ligadura de los parámetros es estrictamente posicional y son de tipado estático y fuerte, todos los parámetros deben ser declarados con un tipo específico y el mismo no puede cambiar durante la ejecución. Además, no soporta parámetros por defecto. Sintaxis para declarar parámetros: tipo\_parametro1 nombre\_parametro1

Ejemplos:

- Los parámetros primitivos corresponden a los tipos de datos básicos en Java, como int, double, char y boolean. Cuando pasamos un parámetro primitivo a una función o método, estamos pasando el valor directamente. Esto se conoce como paso por valor.
- En Java, todo se pasa por valor, incluso los objetos. Lo que se pasa es una copia de la referencia, y eso puede llevar a cambios internos del objeto. Cualquier cambio realizado al objeto o arreglo dentro de la función o método tendrá incidencia.

```
3 public class Main {
4
5     public static void main(String[] args) {
6         int[] numeros = {1, 2, 3};
7         incrementarPrimerElemento(numeros);
8         System.out.println("El valor del primer elemento después de llamar a incrementarPrimerElemento: " + numeros[0]);
9     }
10
11     public static void incrementarPrimerElemento(int[] arr) {
12         arr[0]++;
13         System.out.println("El valor del primer elemento dentro de la función incrementarPrimerElemento: " + arr[0]);
14     }
15 }
16
17
18
19
```

Console X

<terminated> Main [Java Application] C:\Users\ThinkPad\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64.21.0.6.v20230130-0529\jre\bin\java.exe  
El valor del primer elemento dentro de la función incrementarPrimerElemento: 2  
El valor del primer elemento después de llamar a incrementarPrimerElemento: 2

```
3 public class Main {
4
5     public static void main(String[] args) {
6         int numero = 5;
7         incrementarNumero(numero);
8         System.out.println("El valor de numero después de llamar a incrementarNumero: " + numero);
9     }
10
11     public static void incrementarNumero(int n) {
12         n++;
13         System.out.println("El valor de n dentro de la función incrementarNumero: " + n);
14     }
15 }
16
17
18
19
```

Console X

<terminated> Main [Java Application] C:\Users\ThinkPad\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64.21.0.6.v2  
El valor de n dentro de la función incrementarNumero: 6  
El valor de numero después de llamar a incrementarNumero: 5

- En Java, es posible utilizar la palabra clave final en la declaración de un parámetro. Un parámetro marcado como final es inmutable, lo que significa que su valor no puede ser modificado una vez asignado.

```
5 public static void main(String[] args) {
6     int resultado = sumaCuadrados(3, 4);
7     System.out.println("La suma de los cuadrados de 3 y 4 es: " + resultado);
8 }
9
10 public static int sumaCuadrados(final int num1, final int num2) {
11     num1 = num1 * num1;
12     num2 = num2 * num2;
13
14     int cuadrado1 = num1 * num1;
15     int cuadrado2 = num2 * num2;
16
17     return cuadrado1 + cuadrado2;
18 }
19
20
21
22 }
23 }
```

Console

<terminated> Main [Java Application] C:\Users\ThinkPad\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.6.v20250130-  
Exception in thread "main" java.lang.Error: Unresolved compilation problems:  
The final local variable num1 cannot be assigned. It must be blank and not using a compound assignment  
The final local variable num2 cannot be assigned. It must be blank and not using a compound assignment  
at TpCplp/TPcplp.Main.sumaCuadrados(Main.java:11)  
at TpCplp/TPcplp.Main.main(Main.java:6)

## JavaScript:

JavaScript es un lenguaje de tipado dinámico y débil, por lo que no es obligatorio declarar tipos. Soporta parámetros posicionales, parámetros opcionales con valores por defecto, parámetros nombrados usando objetos y parámetros variables.

- Ej. de parámetros posicionales y parámetros opcionales con valores por defecto:

```
1 function multiply(a, b = 1) {
2     return a * b;
3 }
4
5 multiply(5, 2); // 10
6 multiply(5); // 5
7 multiply(5, undefined); // 5
```

- En cuanto a los parámetros predeterminados, los definidos anteriormente (a la izquierda) están disponibles para los parámetros predeterminados posteriores:

```
1 function greet(name, greeting, message = greeting + " " + name) {
2     return [name, greeting, message];
3 }
4
5 greet("David", "Hi"); // ["David", "Hi", "Hi David"]
6 greet("David", "Hi", "Happy Birthday!"); // ["David", "Hi", "Happy Birthday!"]
```

El último parámetro de una función se puede prefijar con "...," lo que hará que todos los argumentos restantes se coloquen dentro de un array de JavaScript "estándar". Lo que se conoce como un parámetro "rest".

Los parámetros rest pueden ser desestructurados, eso significa que sus datos pueden ser desempaquetados dentro de distintas variables. Simulando una ligadura por nombre.

```

1 function myFun(a, b, ...manyMoreArgs) {
2   console.log("a", a);
3   console.log("b", b);
4   console.log("manyMoreArgs", manyMoreArgs);
5 }
6
7 myFun("one", "two", "three", "four", "five", "six");
8
9 // Console Output:
10 // a, one
11 // b, two
12 // manyMoreArgs, [three, four, five, six]

```

```

1 function f(...[a, b, c]) {
2   return a + b + c;
3 }
4
5 f(1); // NaN (b y c son indefinidos)
6 f(1, 2, 3); // 6
7 f(1, 2, 3, 4); // 6 (el cuarto parámetro no está desestructurado)

```

## B. Fortaleza del sistema de tipos.

Se detallará sobre el sistema de tipos: este es un conjunto de reglas usadas por un lenguaje para estructurar y organizar sus tipos. El sistema de tipos impone reglas que previenen errores, ya sea en compilación (Java) o en tiempo de ejecución (JavaScript).

### Java:

Java es un lenguaje estáticamente tipado, lo cual indica que cada variable o expresión tiene un tipo conocido en tiempo de ejecución. A su vez, Java es un lenguaje fuertemente tipado ya que impone restricciones estrictas sobre cómo se utilizan los tipos de datos.

Ejemplos:

```

List<String> lista = new ArrayList<>();
lista.add("hola");
// lista.add(10); // ERROR en compilación: incompatible types

```

Estáticamente tipado:

```
nombre = "Juan";
```

*Este código dará error en compilación ya que no se definió ningún tipo a la variable "nombre".*

```

1 package ar.edu.unlp.info.oo2.rw.example;
2
3 public class Main {
4
5   static int funcion() {
6     return true;
7   }
8
9   public static void main(String[] args) {
10    System.out.print(funcion());
11  }
12 }
13

```

Console X

terminated> Main [Java Application] /Applications/Eclipse.app/Contents/Eclipse/plugins/org.eclipse.justi.openjdk.hotspot.jre.full.macos.x86\_64\_17.0.9.v20230918

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Type mismatch: cannot convert from boolean to int

at ar.edu.unlp.info.oo2.rw.example.Main.funcion(Main.java:6)

at ar.edu.unlp.info.oo2.rw.example.Main.main(Main.java:10)

En este ejemplo, se puede observar cómo java detecta nuevamente en compilación que un método está retornando un valor que no coincide con el tipo especificado.

```
1 package ar.edu.unlp.info.oo2.rw.example;
2
3 public class Main {
4
5
6     public static void main(String[] args) {
7         int num = 1;
8         boolean ok = false;
9
10        int aux = num - ok;
11    }
12 }
13
```

```
terminated> Main [Java Application] /Applications/Eclipse.app/Contents/Eclipse/plugins/org.eclipse.justi.openjdk.hotspot.jre.full.macosx.x86_64_17.0.0.v20230314-1.0
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The operator - is undefined for the argument type(s) int, boolean
    at ar.edu.unlp.info.oo2.rw.example.Main.main(Main.java:10)
```

Acá se puede observar cómo al intentar restar una variable entera con una booleana, java nos retorna que el operador - no está definido para los booleanos. Esto debido a que las operaciones aritméticas no están soportadas para los booleanos.

## JavaScript:

JavaScript es un lenguaje dinámico y débilmente tipado, esto último hace referencia a que JavaScript permite generar una conversión implícita de tipos cuando una operación incluye dos tipos diferentes, en vez de generar un error de tipos

Ejemplo:

```
const foo = 42; // foo es un numero
const result = foo + "1"; // JavaScript convierte foo en un string, para que pueda ser concatenado con el
otro operando.
console.log(result); // 421
```

Debido a esto, podríamos considerar al sistema de tipos de JavaScript como débil.

Otro ejemplo:

```
let variable = 42; // number
variable = "Hola"; // ahora es string
console.log(variable); // "Hola"
console.log("50" + 10); // "5010" (concatenación)
console.log("50" * 10); // 500 (multiplicación, convierte string a number)
```

En resumen, el tipado de JavaScript tiene ventajas como en el ejemplo de “variable” en el cual cambiar el tipo de una variable no genera problemas, pero esta flexibilidad en el tipado podría llevar a errores de tipo graves si no se maneja correctamente.

## C. Manejo de excepciones y sus variantes de manejo.

En esta sección se desarrollará sobre excepciones y su manejo en los dos lenguajes asignados en el trabajo. Las excepciones son situaciones anómalas poco frecuentes en tiempo de ejecución. Un lenguaje que las maneja debe permitir definir las, lanzarlas, capturarlas, tratarlas y establecer cómo continúa la ejecución.

Tanto Java como JavaScript utilizan manejo de excepciones por terminación: la unidad que generó la excepción busca manejar la excepción y termina.

#### Java:

Posee propagación dinámica (stack trace): una vez lanzadas, en caso de no ser tratadas por la unidad que las generó, se propagan dinámicamente. Si un método puede generar excepciones, pero decide no manejarlas localmente, debe enunciarlas en su encabezado (throws). Los bloques de código que manejan excepciones se distinguen por las palabras clave try y catch. Las excepciones se pueden lanzar explícitamente (throw), o se alcanzan por una condición de error (en el caso de las provistas por el lenguaje).

Consideraciones: Puede haber tantos catch como excepciones maneje el método. Se pueden anidar los bloques try/catch. Si se omite el bloque catch, pero hay un bloque finally, este último se ejecuta siempre antes de propagar la excepción en forma dinámica.



```
import java.io.*;

public class EjemploExcepciones {
    public static void main(String[] args) {
        try {
            procesaArchivo("datos.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("Ejecutando tras manejo en main.");
    }

    static void procesaArchivo(String nombre) throws IOException {
        try {
            try (FileReader fr = new FileReader(nombre)) {
                fr.read();
            } // Inner finally implícito en try-with-resources
        } catch (FileNotFoundException e) {
            System.out.println("No existe: " + e.getMessage());
            throw e;
        } catch (IOException e) {
            System.out.println("Error I/O: " + e.getMessage());
        } finally {
            System.out.println("Siempre limpio recursos");
        }
    }
}
```

Este código abre un archivo usando un bloque try-with-resources que garantiza el cierre automático del FileReader; si el fichero no existe, salta un FileNotFoundException, se imprime un mensaje y se relanza para que el main también lo capture, mientras que cualquier otro error de E/S se atrapa en un segundo catch y se maneja localmente sin relanzar; finalmente, el bloque finally siempre ejecuta una limpieza (o un mensaje) antes de salir, y en el main, si llega una excepción no gestionada se imprime su stack trace, y programa continúa su ejecución normalmente.

## JavaScript:

JavaScript también utiliza manejo de excepciones por terminación. Se pueden lanzar excepciones usando la instrucción `throw` y manejarlas usando las declaraciones `try...catch`.

- `throw`: Permite lanzar excepciones con cualquier valor u objeto (e.g., `Error`, `CustomError`).
- `try...catch`: El bloque `try` contiene código que puede arrojar una excepción, y el `catch` maneja la excepción si ocurre, usando un identificador para acceder a la información del error.
- `finally`: Se ejecuta siempre después de `try` y `catch` independientemente de que se produzca una excepción.
- Anidamiento: Los bloques `try...catch` pueden anidarse y si un `try` interno no tiene `catch`, debe tener `finally`, y se busca un `catch` externo.
- Propagación: Las excepciones se buscan en la pila de llamadas. Si no hay un `catch` en el ámbito actual, la excepción se propaga al método o función que llamó al código, y así sucesivamente. Si no se captura en ningún nivel, el programa termina.



```
function processInner() {
    throw new Error("Error interno");
}

function processOuter() {
    try {
        processInner();
    } catch (innerError) {
        console.log("Error interno capturado:", innerError.message);
        throw innerError; // Propagación dinámica
    }
}

try {
    processOuter();
} catch (outerError) {
    console.log("Error externo capturado:", outerError.message);
} finally {
    console.log("Siempre me ejecuto");
}
```

- Ruta 1: Excepción capturada y propagada
  - Excepción en `processInner()` capturada por `catch` interno, propagada al `catch` externo.
  - Salida: "Error interno capturado: Error interno", "Error externo capturado: Error interno", "Siempre me ejecuto".
- Ruta 2: Excepción no capturada

- Excepción propagada fuera del ámbito si no hay try...catch externo, termina el programa.
- Salida: Error en consola (e.g., "Uncaught Error: Error interno"), sin finally.
- Ruta 3: Ejecución exitosa (hipotética)
  - Sin excepción en processInner() (no aplicable a este código), control pasa a finally.
  - Salida: Solo "Siempre me ejecuto".

Con el código actual, la ruta 1 es la que se termina ejecutando, pero las otras dos rutas mencionadas sirven a modo de ejemplo.

#### **D. Conclusión sobre el trabajo completo**

Por mi parte (Matías), ya había trabajado con dichos lenguajes para proyectos personales, y el trabajo realizado me permitió entender por completo cómo funciona el manejo de excepciones en ambos lenguajes, y comprender cómo funciona el tipado de JavaScript, además de reforzar los contenidos vistos en la materia.

Personalmente (Acosta) aprendí mucho al realizar este trabajo, principalmente de JavaScript, ya que es un lenguaje el cual no usé mucho. En cuanto a Java, me ayudó a entender varias cosas las cuales no tenía mucho conocimiento como el manejo de excepciones y la palabra clave "final".

En mi opinión (Lima), ya conocía y tenía experiencia con ambos lenguajes, pero nunca había mirado más profundo en su diseño y por qué funcionan como lo hacen. Siento que el trabajo me dio un mayor entendimiento en cómo ambos lenguajes manejan sus tipos y la lógica detrás de estos, además de permitirle discernir entre las ventajas y desventajas de las distintas formas de tipado.

En mi caso (Matheo), este trabajo me permitió entender mejor cómo se comportan los parámetros en ambos lenguajes y cómo se relacionan con el sistema de tipos. Si bien ya tenía una idea general sobre Java por la facultad, ver sus diferencias con JavaScript me ayudó profundizar conceptos sobre Java y aprender un lenguaje nuevo para mí, JavaScript. Además, me sirvió a modo de repaso de cómo se manejan las excepciones.

Como grupo, este trabajo nos sirvió no solo para aprender más sobre Java y JavaScript, sino también para organizarnos y trabajar en conjunto. Nos ayudó a repartir tareas y compartir conocimientos para entender mejor los temas. Fue una buena experiencia para fortalecer el trabajo en equipo y aprovechar lo que cada uno podía aportar.



## **Bibliografía:**

Oracle. (s.f.). Generics: Type Parameters. Oracle.

<https://docs.oracle.com/javase/tutorial/java/generics/types.html>

Oracle. (2023). The Java® Language Specification – SE 21 Edition, Chapter 4: Types, Values, and Variables. Oracle. <https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.11>

Oracle. (s.f.). Arrays. Oracle.

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

MDN Web Docs. (s.f.). Estructuras de datos en JavaScript. Mozilla Foundation.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Data\\_structures](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Data_structures)

Oregoom. (s.f.). Parámetros en Java. Oregoom. <https://oregoom.com/java/parametros>

Oracle. (s.f.). Passing Information to a Method or a Constructor. Oracle.

<https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html>

MDN Web Docs. (s.f.). Funciones en JavaScript. Mozilla Foundation.

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/>

W3Schools. (s.f.). Java Try and Catch. W3Schools.

[https://www.w3schools.com/java/java\\_try\\_catch.asp](https://www.w3schools.com/java/java_try_catch.asp)

MDN Web Docs. (s.f.). Control de flujo y manejo de errores en JavaScript. Mozilla Foundation.

[https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Control\\_flow\\_and\\_error\\_handling](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Control_flow_and_error_handling)

Teoría clase 07-08-2025: Conceptos y paradigmas de lenguajes de programación

Explicación Práctica Excepciones: Conceptos y paradigmas de lenguajes de programación