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.

Parámetros

En esta sección se hará una explicación sobre parámetros de Java y JavaScript. Los parámetros son variables que se utilizan para compartir 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

```java
import java.util.*;

public class EjemploPasaje {
    // Pasaje por valor: "x" de tipo primitivo: su valor original no cambia
    static void cambiarEntero(int x) {
        x = 100;
    }

    // Pasaje por valor que modifica el objeto
    static void cambiarNombre(Persona p) {
        p.nombre = "NuevoNombre";
    }

    // Pasaje de lista de valores (varargs)
    static void imprimirNombres(String... nombres) {
        for (String n : nombres) System.out.println(n);
    }

    // Pasaje por resultado de función
    static int obtenerEdad() {
        return 30;
    }

    public static void main(String[] args) {
        int numero = 50;
        cambiarEntero(numero);
        System.out.println("Número después: " + numero); // No cambia

        Persona persona = new Persona("Juan");
        cambiarNombre(persona);
        System.out.println("Nombre después: " + persona.nombre); // Cambia

        imprimirNombres("Ana", "Luis", "Pedro"); // Lista de valores

        int edad = obtenerEdad(); // Resultado de función
        System.out.println("Edad obtenida: " + edad);
    }
}

class Persona {
    String nombre;
    Persona(String nombre) {
        this.nombre = nombre;
    }
}
```

- 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. Por lo que en este caso funciona como el pasaje por referencia.

- Java ofrece una característica llamada argumentos variables, conocidos como "*varargs*", que nos permite pasar un número variable de argumentos a una función o método.

- 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. Por lo que funciona como una constante.

```java
void modificar(final int x) {
    x = 10; // Error: cannot assign a value to final variable 'x'
}
```

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. El paso de parámetros a las funciones se realiza por valor para tipos primitivos y por referencia para objetos.

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

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5, 2); // 10
multiply(5); // 5
multiply(5, undefined); // 5
```

En JavaScript, los parámetros pueden tener valores por defecto, es decir que, si al invocar una función no se pasa un argumento para ese parámetro, no se produce un error, sino que se usa el valor por defecto especificado.

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

```
function greet(name, greeting, message = greeting + " " + name) {
  return [name, greeting, message];
}

greet("David", "Hi"); // ["David", "Hi", "Hi David"]
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 recibir datos heterogéneos.

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");

// Console Output:
// a, one
// b, two
// manyMoreArgs, [three, four, five, six]
```

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

```
function f(...[a, b, c]) {
  return a + b + c;
}

f(1); // NaN (b y c son indefinidos)
f(1, 2, 3); // 6
f(1, 2, 3, 4); // 6 (el cuarto parámetro no está desestructurado)
```

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.

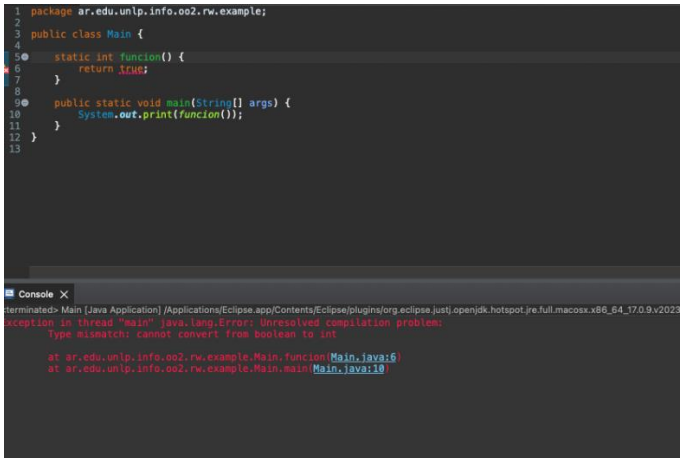
Ejemplos:

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

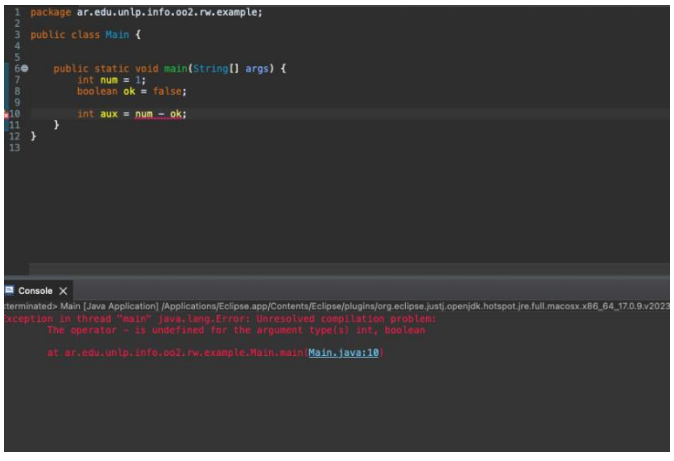
Estáticamente tipado:

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

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



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



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.

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 definirlas, 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 con la palabra clave “*throws”*.

Los bloques de código que manejan excepciones se distinguen por las palabras clave try y catch. Las excepciones se pueden lanzar explícitamente con la palabra clave "*throw*", o se alcanzan por una condición de error (en el caso de las provistas por el lenguaje).

Los bloques catch deben ordenarse de excepciones más específicas a más generales, ya que un catch de Exception antes de IOException capturaría todas las excepciones, causando un error de compilación. Se pueden lanzar ("*throw*") tanto excepciones del sistema como personalizadas (heredando de Exception). Los bloques try pueden anidarse, y una excepción no capturada se propaga al try externo o al método llamador. Un bloque finally, si existe, se ejecuta siempre, con o sin catch, antes de propagar la excepción o finalizar el método.

```java
import java.io.*;

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

    static void processArchivo(String nombre) throws IOException {
        try (FileReader fr = new FileReader(nombre)) {
            // Leer archivo
        } catch (FileNotFoundException | IOException e) {
            System.out.println("No existe: " + e.getMessage());
            throw e;
        } catch (Exception 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, se lanza una excepción FileNotFoundException, se imprime un mensaje y luego se relanza para que el método main también pueda capturarla. En cambio, cualquier otro error de entrada/salida se atrapa en un segundo bloque catch y se maneja localmente sin volver a lanzar la excepción.

Finalmente, el bloque finally se ejecuta siempre, independientemente de si hubo o no una excepción, permitiendo realizar tareas de limpieza o imprimir un mensaje antes de salir del bloque. En el método main, si llega una excepción no gestionada, se imprime su stack trace y el programa continúa con su ejecución normalmente. Sin embargo, si la excepción no puede ser capturada allí, el programa termina en falla y no sigue ejecutándose.

JavaScript:

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

- 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.
- throw: Lanza manualmente una excepción (puede ser Error, Exception o clase personalizada como CustomError). Throw activa el manejo de excepciones.

- throw new: Crea e inmediatamente lanza una nueva instancia de una clase de excepción (ej. throw new Exception()). New solo instancia.
- 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 para capturar la excepción.
- 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 (propagación dinámica) 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.

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 habia 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 logica 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

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

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

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

Explicación Práctica Excepciones: Conceptos y paradigmas de lenguajes de programación