Objectives

- Introduction to JavaScript
- Variables and Data Types
- Working with Numbers and String
- Working with Arrays and Objects

Intoduction to JavaScript

Javascript

JavaScript is an object-oriented programming language created in 1995 by Brendan Eich. As the main programming language for the web, it transformed the internet into what it is today. Before JavaScript, web pages were static, only displaying content. Now, they're dynamic, interactive, and engaging capable of responding to users, performing calculations, saving data, and much more.
JavaScript code is written in files with the **.js** extension. Thanks to its powerful capabilities and simplicity, JavaScript has grown far beyond just web browsers. With Node.js, we can now run JavaScript in a server-side environment, enabling it to execute outside the browser opening doors to backend development, scripting, and more.

Adding JavaScript to a Webpage

There are two ways to add JavaScript to a webpage: internal (embedded) or external scripts.

Internal Script

This method involves placing JavaScript code directly within the HTML file using the `<script>` tag.
Example

```
<script>
  // Your JavaScript code here
  alert("Hello from an internal script!");
</script>
```

External Script

For larger scripts, embedding them in HTML can make the code hard to read and maintain. Instead, we can store JavaScript in external `.js` files and link them to the HTML using the `<script>` tag with the `src` attribute.
Example

```
<script src="scripts/main.js"></script>
```

The Browser Console

The browser console is a powerful built-in tool that comes with every modern web browser. Its primary purpose is to help developers debug web pages when errors occur in your JavaScript code, they appear here with helpful details to guide your troubleshooting.
But that's not all the Console can do! It can also execute JavaScript code and instructions, making it the perfect environment to run scripts and test code while you're learning this new programming language.

Opening The Console

The way you open the console changes depending on which browser you're using. If you're on Windows/Linux (Chrome/Edge/Firefox), you can press `Ctrl+Shift+J` to open the console. If you're on Mac (Chrome/Safari), you'll use `Cmd+Option+J`.
Task
Open the browser console and write the following script:

```
alert("Hello !");
```

Variables and Data Types

Introduction

We can think of variables as boxes that store data and information. JavaScript provides two keywords to help us create variables: `let` and `var`.

- `let` is used to create **block-scoped variables**. Variables declared with `let` are accessible only within the block (like inside `{}`) in which they are defined, including any nested blocks.
- `var` is used to create **function-scoped variables**. Variables declared with `var` are accessible throughout the entire function in which they are defined. If declared outside a function, they become globally scoped.
  To create a variable in JavaScript, we use either `let` or `var`, followed by a descriptive name without spaces. The variable name can include letters, numbers, underscores (`_`),

and dollar signs (`$`), but it cannot start with a number. For example, `userName` and `_count` are valid names, while `2ndPlace` is invalid.

Example

```
var name = "Ali";
let age = 23;
let m;
```

We can see that we ended our lines with semicolons. In JavaScript, semicolons are not required we can omit them but the standard and best practice is to use a semicolon at the end of each statement.

Constant

Constants are special variables that maintain a fixed value throughout program execution. In JavaScript, we declare constants using the `const` keyword. Once assigned, their value cannot be changed, attempting to do so will trigger a TypeError.
Constants follow the same naming rules as regular variables, but with one important stylistic difference. By convention, we write constant names in `UPPER_CASE` with underscores separating words.

```
const PI = 3.14;
```

Remark

Objects and arrays declared with `const` can still be modified.

Data Types

We use variables to store data, including numeric values, text (strings), lists (arrays), objects, and dates. JavaScript provides different data types to efficiently store and work with various kinds of information.
JavaScript data types are divided into two main categories Primitive Data Types and Non-Primitive Data Types

Primitive Data Types

In JavaScript, we work with primitive data types as our most basic building blocks. These types have three crucial behaviors we need to understand:

1. **Direct Memory Storage** - When we create primitive values, they're stored directly in the variable's memory location
2. **Immutability** - Once we create them, we can't modify primitive values themselves
3. **Value-based Operations** - When we try to "change" a primitive, we're actually creating a brand new value in memory

Number

The first primitive type we encounter in JavaScript is the `number` type. We use this type to represent all numeric values, including both integers and floating-point numbers.

```
let age = 23;
let pi = 3.14;
let negative_number = -5;
```

Bigint

BigInt is a primitive data type we use to represent integer values that are too large to be stored safely as regular Numbers. We create BigInts by appending `n` to the end of an integer or using the `BigInt()` constructor.

```
let number1 = 12345678901234567890n
let number2 = BigInt("9007199254740991")
```

String

The `string` type is used to store and manipulate textual data. We can create strings by enclosing our text in either single quotes **(')**, double quotes **("")**, or backticks **(``)** for template literals.

```
let greeting = "Hello World";
let name = 'Alice';
```

Template literals (enclosed in backticks `` ` ``) allow us to embed expressions and variables directly into strings,To inject values we create string using `` ` `` then we insert variables or expressions with `${}`

```
let name = "Alice";
let greeting = `Hello, ${name}!`;
```

The greeting variable will have as value `Hello, Alice`

Boolean

Booleans are used to store the results of logical operations, with only two possible values: `true` or `false`.

```
let is_adult = false;
```

JavaScript automatically converts any value to a boolean when used in a logical context. The result depends on whether the value is "truthy" or "falsy".

Truthy Values

```
true, "text", 1, [], {}, 3.14, Infinity, -1, "false"
```

Falsy Values

```
false, 0, "", null, undefined, NaN, 0n
```

Undefined

`undefined` represents a variable that has been declared but not yet assigned a value. It is JavaScript's default state for uninitialized variables.

```
let number;
```

Here type of number will be `undefined`

Null

`null` is a special primitive value that represents the intentional absence of any object value. We use it to explicitly indicate that a variable should have "no value" or "empty value."

```
let user = null;
```

Non-Primitive Data Types

Unlike primitive types, non-primitive types (objects) are mutable and store references to memory locations rather than direct values. They share these key behaviors:

1. **Reference Storage** - Variables store pointers to memory locations
2. **Mutability** - We can modify their contents without creating new references
3. **Pass-by-Reference** - Assignments/copies share the same underlying data

Object

We use objects to store collections of key-value pairs, where:

- **Keys** are always strings.
- **Values** can be any data type including other objects, functions, or primitives

To create an object, we place key-value pairs inside curly braces `{}`. We separate each key from its value with a colon (`:`), and multiple pairs with commas (`,`):

```
const user = {
  name: "Alex",
  age: 30,
  isAdmin: true,
  address: {
    city: "Berlin",
    country: "Germany"
  }
}
```

There are two ways to access elements inside an object:

1. **Dot Notation**
   We start with the object name, followed by a dot (`.`), then the key:

```
user.name;
```

2. **Bracket Notation**
   We start with the object name, followed by square brackets `[]` containing the key (as a string):

```
user["name"];
```

Arrays

We use arrays to store ordered collections of data where:

- **Elements** can be any data type (including other arrays/objects)
- **Indexes** are zero-based integer positions
- **Length** dynamically adjusts as we add/remove items

To create an array, we place values inside **square brackets** `[]`. We separate the values with commas (`,`):

```
const mixedArray = [
  "text",
  42,
  true,
  { id: 1 },
  ["nested"]
];
```

We access values inside an array by using indexes. The first element has index 0. For example, to access the value 42 we use `mixedArray[1]`.

Functions

The final non-primitive data type is the **function**. We use functions to create reusable blocks of code that execute only when called.

```
function greet(name) {
  return `Hello, ${name}!`;
}
greet("Ali");
```

Using console.log()

To display values or print messages to the console, we use the `console.log()` function. Inside the parentheses `()`, we specify what we want to output:

```
console.log("Hello World");
console.log(42);
console.log(myVariable);
```

Comments

Comments are special lines of code that the browser won't execute, but they're useful for us programmers to leave hints or mark tasks for later. There are two types of comments:

- **Single-line comments**: We create them using `//` - everything after these slashes on the same line becomes a comment and gets ignored.

```
// this is single line comment
```

- **Multi-line comments**: We create them using `/* */` - all text between these symbols, even across multiple lines, will be ignored and not executed.

```
/*
    this
    is
    multi-line
    comment
*/
```

Working with Numbers and String

Arithmetic Operations

JavaScript allows us to do more than just store numbers, it provides arithmetic operators to perform calculations and manipulate numeric values.

Addition

Add two numbers using the `+` operator:

```
let a = 4;
let b = 5;
let c = a + b; // 9
```

Subtraction

Subtract numbers using the `-` operator:

```
let a = 4;
let b = 5;
let c = a - b; // -1
```

Multiplication

Multiply numbers using the `*` operator:

```
let a = 4;
let b = 5;
let c = a * b; // 20
```

Division

Divide numbers using the `/` operator:

```
let a = 4;
let b = 5;
let c = a / b; // 0.8
```

Remainder

Calculate division remainders using the `%` operator:

```
let a = 5;
let b = 4;
let c = a % b; // 1
```

Exponentiation

Raise numbers to powers using the `**` operator:

```
let a = 2;
let b = 3;
let c = a ** b;  // 8
```

Number Methods

The Number data type includes built-in methods that help us manipulate and work with numeric values more effectively. These methods provide functionality for formatting, conversion, and precision control.

Example

```
let num = 123.4567;

num.toFixed(2);      // "123.46" (Rounds to 2 decimal places)
num.toPrecision(4);   // "123.5" (Formats to 4 significant digits)
num.toString(16);     // "7b.74bc6a7ef9" (Converts to hexadecimal)
Number.isInteger(num); // false (Checks if value is integer)
```

Math Object

For more complex mathematical operations beyond basic arithmetic, JavaScript provides the built-in `Math` object. This object contains methods and constants for advanced calculations including:

- Trigonometric functions (`sin` , `cos` , `tan`)
- Rounding operations (`round` , `ceil` , `floor`)
- Logarithmic and exponential calculations
- Random number generation
- Mathematical constants (π, e, etc.)

Example

```
Math.sin(45 * Math.PI/180);  // Calculate sine (0.7071)
Math.round(3.7);             // Round to nearest integer (4)
Math.random();               // Generate random number (0-1)
```

String Operation

The `+` operator is the primary operator for string operations in JavaScript. When used with strings, it performs concatenation (combining them together):

```
let greeting = "Hello" + " " + "World";  // "Hello World"
```

String Indexing

We can treat strings as arrays of characters, accessing individual characters using zero-based indexing

```
const word = "hello";
word[0];  // 'h' (first character)
```

```
word[4];  // 'o' (last character)
```

String Methods

Case Conversion

We can use case conversion methods to modify the letter casing of strings in JavaScript.

```
'hello'.toUpperCase();  // 'HELLO'
'HELLO'.toLowerCase();  // 'hello'
```

Searching & Checking

JavaScript provides methods to search and check strings, allowing us to:

1. **Verify if a substring exists**

```
'JavaScript'.includes('Script');  // true
```

2. **Find character/word positions**

```
'hello'.indexOf('e');     // 1 (first occurrence)
'hello'.lastIndexOf('l'); // 3 (last occurrence)
```

3. **Check starting/ending patterns**

```
'file.txt'.startsWith('file'); // true
'image.png'.endsWith('.png');  // true
```

Extraction & Slicing

JavaScript provides several methods to extract portions of strings:

1. **slice()**: Extracts a section between two indices:

```
'JavaScript'.slice(0, 4);  // 'Java'
'JavaScript'.slice(4);     // 'Script' (to end)
'JavaScript'.slice(-6);    // 'Script' (from end)
```

2. **substring()**: Similar to slice() but doesn't accept negative indices:

```
'JavaScript'.substring(0, 4);  // 'Java'
```

3. **substr()**: Uses start index and length:

```
'JavaScript'.substr(4, 3);  // 'Scr'
```

4. **charAt()**: Gets a specific character:

```
'Hello'.charAt(1);  // 'e'
```

Modification

JavaScript provides methods to create modified versions of strings (remembering that strings themselves are immutable).

1. **Trimming Whitespace**
   - `trim()` : Removes whitespace from both ends
   - `trimStart()` : Removes only leading whitespace
   - `trimEnd()` : Removes only trailing whitespace

```
'  hello  '.trim();       // 'hello'
'  hello  '.trimStart();  // 'hello  '
'  hello  '.trimEnd();    //  '  hello'
```

2. **Replacing Text**
   - `replace()` : Replaces the first matching pattern
   - `replaceAll()` : Replaces all matching patterns

```
'apple orange apple'.replace('apple', 'banana');    // 'banana orange apple'
'apple orange apple'.replaceAll('apple', 'banana'); // 'banana orange banana'
```

3. **Repeating Strings**
   - `repeat()` Creates repeated copies:

```
'ha'.repeat(3);  // 'hahaha'
```

4. **Padding Strings**
    - `padStart()` Adds characters to the **beginning** (left side) of a string until it reaches the specified length.
    - `padEnd()` Adds characters to the **end** (right side) of a string until it reaches the specified length.

```
'5'.padStart(3, '0');    // '005'
'5'.padEnd(3, '!');      // '5!!'
```

Splitting & Combining

1. **Splitting Strings**: `split(separator)` divide a string into an array:

```
'a,b,c'.split(',');      // ['a', 'b', 'c']
'hello'.split('');       // ['h', 'e', 'l', 'l', 'o']
```

2. **Combining Arrays**: `join(separator)` merge an array into a string:

```
['a', 'b', 'c'].join('-');  // 'a-b-c'
['2023', '10', '05'].join('/'); // '2023/10/05'
```

Working with Arrays and Objects

Arrays

We can do much more than simply access arrays and their elements using array methods, we can:

- **Add** or **remove** elements
- **Search** for specific items
- **Transform** array contents
- **Sort** and **reorganize** data
- **Combine** or **split** arrays

Getting Array Length

We can determine the number of elements in an array using the `length` property:

```
let fruits = ['apple', 'banana', 'orange'];
console.log(fruits.length);  // 3
```

Searching Elements

we can search for elements insdie array using one of he following methods

- `includes()` : Checks if an element exists in the array

```
[1, 2, 3].includes(2);  // true
```

- `indexOf()` : Finds the **first occurrence** of an element

```
['a', 'b', 'a'].indexOf('a');  // 0
```

- `lastIndexOf()` : Finds the **last occurrence** of an element (searches from the end)

```
['a', 'b', 'a'].lastIndexOf('a');  // 2
```

- `find()` : Returns the **first element** matching a condition

```
let users = [{id: 1}, {id: 2}];
users.find(user => user.id === 2);  // {id: 2}
```

Adding and Remove Elements

1. `push()` : Add to End

```
let fruits = ['apple', 'banana'];
fruits.push('orange');
// ['apple', 'banana', 'orange']
```

2. `unshift()` : Add to Start

```
let fruits = ['apple', 'banana'];
fruits.unshift('orange');
// ['orange', 'apple', 'banana']
```

3. `pop()` : Remove Last Element

```
let fruits = ['apple', 'banana', 'orange'];
const last = fruits.pop();
// fruits = ['apple', 'banana']
// last = 'orange'
```

4. `shift()` : Remove First Element

```
let fruits = ['apple', 'banana', 'orange'];
const first = fruits.shift();
// fruits = ['banana', 'orange']
// first = 'apple'
```

5. `splice()` : Performs three key operations on arrays **Adds elements** at any specified position, **Removes elements** from any position, **Replaces elements** by combining addition and removal

```
let colors = ['red', 'green', 'blue'];

// 1. Add without removing (deleteCount = 0)
colors.splice(1, 0, 'yellow');
// ['red', 'yellow', 'green', 'blue']

// 2. Remove without adding
colors.splice(2, 1);
// ['red', 'yellow', 'blue']

// 3. Replace elements
colors.splice(1, 1, 'purple');
// ['red', 'purple', 'blue']
```

- **First Argument (Required)** The index position where the operation should be performed
- **Second Argument (Required)** The number of existing elements to remove
- **Third Argument (Optional)** New element(s) to insert at the specified position

Combining, Spreading and Slicing

1. `concat()` concatenates and merges two or more arrays into a single new array

```
let arr1 = [1, 2];
let arr2 = [3, 4];
let combined = arr1.concat(arr2); // [1, 2, 3, 4]
```

2. (`...`) The spread operator expands an array into its individual elements

```
// Combine arrays
let merged = [...arr1, ...arr2]; // [1, 2, 3, 4]

// Insert additional elements
let withExtras = [...arr1, 'a', ...arr2, 'b']; // [1, 2, 'a', 3, 4, 'b']
```

3. `slice()` creates a shallow copy of a portion of an array without modifying the original array.

```
let original = [1, 2, 3];
let copy1 = original.slice(); // copy1 = [1, 2, 3]
let copy2 = original.slice(1); // copy2 = [2, 3]
```

- First argument is **start index** (inclusive)
- Second argument is **end index** (exclusive)
- Using 1 argument: extracts from start index to end
- Using no arguments: extracts entire array (full copy)

Sorting Array

- `sort()` It arranges array elements **strings alphabetically** and for number it require compare function

```
let fruits = ['banana', 'apple', 'orange'];
fruits.sort(); //  ['apple', 'banana', 'orange']
[40, 1, 5, 200].sort((a, b) => a - b); // small to big
[40, 1, 5, 200].sort((a, b) => b - a); // big to small
```

- `reverse()` : It flips the current order of elements:

```
let letters = ['a', 'b', 'c'];
letters.reverse(); // ['c', 'b', 'a']
```

Objects:

Same as array we can do more then just creating and reading objects,we can

- **Add** or **remove** properties
- **Search** for specific values
- **Transform** object structure
- **Combine** or **split** objects
- **Freeze/seal** objects for protection

Getting Values And keys

- `Object.keys()` : Get all keys inside our object

```
let person = {"name": 'Alex', "age": 30};
Object.keys(person); // ["name", "age"]
```

- `Object.values()` : Get all values inside object

```
let person = {"name": 'Alex', "age": 30};
Object.values(person); // ["Alex", 30]
```

Object Lenght

When we retrieve an object's keys using `Object.keys()`, it returns an **array of property names**. We can then use `.length` on this array to determine the object's property count

```
let person = {"name": 'Alex', "age": 30};
Object.keys(person).length; // 2
```

Searching Object

- `hasOwnProperty()` : Checks if an object **directly contains** a specified property

```
let person = {"name": 'Alex', "age": 30};
person.hasOwnProperty('age');  // true
person.hasOwnProperty('id'); // false
```

- `in` Checks if a property exists **anywhere** in the object

```
let person = {"name": 'Alex', "age": 30};
"name" in person: // true
```

- `Object.values()` + `includes()` Check if value exist in the object

```
let person = {"name": 'Alex', "age": 30};
Object.values(person).includes('Alex'); // true
```

Adding and Removing Properties

- We can add or modify object properties using dot or bracket notation

```
const user = {};
user.name = 'Alex';    // Adds new property
user.name = 'Sarah';   // Modifies existing property
user["age"] = 20;
```

- We can remove properties from objects using the `delete` keyword

```
let person = {"name": 'Alex', "age": 30};
delete person["age"]
```

Combining and Spreading Object

- We can combine multiple objects into a single object using `Object.assign()`

```
let obj1 = { a: 1 };
let obj2 = { b: 2 };
let merged = Object.assign({}, obj1, obj2); // { a: 1, b: 2 }
```

- We can use the spread operator (`...`) to work with objects

```
let obj1 = { a: 1 };
let obj2 = { b: 2 };
let merged = {...obj1, ...obj2}; // { a: 1, b: 2 }
```

Freeze and Seal

These methods help lock objects to prevent changes:

- **`Object.freeze()`**
  Prevents all modifications:
  - Cannot change existing values
  - Cannot add new properties
  - Cannot remove properties

```
let person = {"name": 'Alex', "age": 30};
Object.freeze(person);
person.name = 'Max'; // Fails silently
```

- **`Object.seal()`**
  Allows only value changes:
  - Can modify existing values
  - Cannot add new properties
  - Cannot remove properties

```
let person = {"name": 'Alex', "age": 30};
Object.seal(person);
person.id = '49416548'; // Fails silently
```

Task

Task 1

You have the following data:

```
let firstName = "luna";
let birthYear = "2000";
let currentYear = 2025;
```

- Capitalize the first letter of `firstName` using string methods.
- Convert `birthYear` to a number and calculate the age.
- Build a sentence like: `"Luna is 25 years old."`

Task 2

You have the following array:

```
let fruits = ["apple", "banana", "mango"];
```

- Add `"orange"` to the **start** of the array.
- Replace `"banana"` with `"kiwi"` using `indexOf()` and direct assignment.
- Use `.join(", ")` to turn the array into a single string.

Task 3

You have the following object:

```
let user = {
  username: "skywalker",
  hobbies: ["coding", "gaming", "reading"]
};
```

- Add a new hobby `"traveling"` using `.push()`.
- Convert all hobbies to **uppercase**.
- Create a summary like: `"SKYWALKER enjoys CODING, GAMING, READING, TRAVELING."`