Ministry of SMEs and Startups

KOSME Korea SMEs and Startups Agency

LIKELION

JOBKOREA

LESSON 02
JavaScript Fundamentals

WEEK 01

Objectives

- Understand the Basics of JavaScript
- Variables and Data Types
- Control Flow
- Functions and Scope
- Manipulating Objects and Arrays
- Error Handling and Debugging
- Asynchronous JavaScript
- DOM Manipulation
- Event Handling

Understand the Basics of JavaScript

Understand the Basics of JavaScript

❖ What is JavaScript?

JavaScript is a high-level, interpreted programming language used to create interactive effects within web browsers.

❖ Core Concepts of JavaScript

It is primarily used for web development to enhance user interaction through features like dynamic content, form validation, and event handling.

❖ JavaScript in the Web Development Stack

JavaScript is part of the frontend stack, alongside HTML and CSS, to create dynamic, interactive websites.

Work with Variables and Data Types

Work with Variables and Data Types

❖ What are Variables?

Variables are used to store data in memory. In JavaScript, variables are containers for data.

❖ Declaring Variables

Variables are declared using let, const, or var.

Example: let name = "Alice";

❖ Data Types in JavaScript

JavaScript supports various data types: string, number, boolean, undefined, null, object, symbol.

Work with Variables and Data Types

❖ What are Variables?

Variables are used to store data in memory. In JavaScript, variables are containers for data.

❖ Declaring Variables

Variables are declared using let, const, or var.

Example: let name = "Alice";

❖ Data Types in JavaScript

JavaScript supports various data types: string, number, boolean, undefined, null, object, symbol.

JavaScript Data Types

- **String**
  Represents text. Enclosed in single or double quotes.
  Example: let message = "Hello, world!";
- **Number**
  Represents both integer and floating-point numbers.
  Example: let age = 25; let price = 19.99;
- **Boolean**
  Represents true or false values.
  Example: let isAdult = true;

Other Data Types

- **Undefined**

  A variable declared but not assigned a value. Default value is undefined.

  Example: let x; console.log(x); // undefined

- **Null**

  Represents the intentional absence of any value or object.

  Example: let car = null;

- **Object**

  A collection of key-value pairs.

Working with Constants

❖ Using const

const is used to declare variables that should not be reassigned after initialization.

Example: const birthYear = 1990;

❖ Reassigning Variables

let allows reassignment but **const** does not.

```
1  let a = 25;
2  a = 30; // Reassignable
3
4  const birthYear = 1990;
5  birthYear = 1995; // Error
```

Control Flow

Control Flow

❖ Conditional Statements

Use if, else, and switch to execute code based on conditions.

```
1  // if statement example:
2  let age = 20;
3  if (age ⩾ 18) {
4    console.log('You are an adult.');
5  }
```

Control Flow

❖ Conditional Statements

Use if, else, and switch to execute code based on conditions.

```
1  // if-else statement example:
2  if (age < 18) {
3    console.log('You are a minor.');
4  } else {
5    console.log('You are an adult.');
6  }
```

Control Flow

❖ Conditional Statements

Use if, else, and switch to execute code based on conditions.

```
1  // switch statement example:
2  let fruit = 'apple';
3  switch (fruit) {
4    case 'banana':
5      console.log('Banana is yellow.');
6      break;
7    case 'apple':
8      console.log('Apple is red.');
9      break;
10   case 'grape':
11     console.log('Grape is purple.');
12     break;
13   default:
14     console.log('Unknown fruit.');
15 }
```

Control Flow

❖ Loops: for

JavaScript supports loops like for, while, and do...while to repeat code multiple times.

```
1  // loop, for example
2  for (let i = 0; i < 5; i++) {
3    console.log('Iteration:', i);
4  }
```

Control Flow

❖ Loops: foreach

JavaScript supports loops like for, while, and do...while to repeat code multiple times.

```
1  // loop, foreach example
2  let fruits = ['banana', 'apple', 'grape'];
3  fruits.forEach(function (fruit) {
4    console.log('Fruit:', fruit);
5  });
6
```

Control Flow

❖ Loops: while

JavaScript supports loops like for, while, and do...while to repeat code multiple times.

```
1  // while loop example
2  let i = 0;
3  while (i < 5) {
4    console.log('Iteration:', i);
5    i++;
6  }
```

Control Flow

❖ Loops: do-while

JavaScript supports loops like for, while, and do...while to repeat code multiple times.

```
1  // do-while loop example
2  let j = 0;
3  do {
4    console.log('Iteration:', j);
5    j++;
6  } while (j < 5);
7
```

Functions and Scope in JavaScript

Functions and Scope in JavaScript

❖ What is a Function?

A function is a reusable block of code that performs a specific task.

```
1  function greet() {
2    console.log('Hello, World!');
3  }
4  greet();
```

Function Parameters and Return Values

❖ Parameters

Functions can accept parameters, which are values passed into the function.

```
1  // function example
2  function greet(name) {
3    console.log('Hello, ' + name + '!');
4  }
5
6  // Call the function
7  greet('Alice');
8  greet('Bob');
9
```

Function Parameters and Return Values

❖ Return Values

Functions can return a value using the return keyword.

```
1  function square(x) {
2    return x * x;
3  }
4  console.log(square(4)); // 16
```

Scope in JavaScript

❖ What is Scope?

- ➢ Scope defines the accessibility of variables and functions in different parts of the code.
- ➢ There are two main types: Global scope and Local scope.

❖ Global Scope

Variables declared outside of any function or block are in the global scope and can be accessed anywhere in the program.

Scope in JavaScript

❖ Global Scope

Variables declared outside of any function or block are in the global scope and can be accessed anywhere in the program.

```
1  let name = 'John'; // Global scope
2  function greet() {
3    console.log(name);
4  }
5  greet(); // John
```

Local Scope and Function Scope

❖ Local Scope

Variables declared inside a function are only accessible within that function.

```
1  function test() {
2    let num = 10; // Local scope
3    console.log(num); // Accessible here
4  }
5  test();
6  console.log(num); // Error: num is not defined
7
```

Local Scope and Function Scope

❖ Function Scope

Functions have their own scope, meaning variables declared inside a function are not accessible outside of it.

Manipulating Objects and Arrays

Manipulating Objects in JavaScript

❖ What are Objects in JavaScript?

An object is a collection of key-value pairs used to store multiple values in a single variable.

```
let person = {
  name: 'Alice',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'Wonderland',
  },
  hobbies: ['reading', 'traveling'],
  alive: true,
  greet: function () {
    console.log('Hello, ' + this.name);
  },
};
```

Manipulating Objects in JavaScript

❖ Accessing Object Properties

Access object properties using dot notation or bracket notation.

```
person.greet(); // Hello, Alice
console.log(person.name); // Dot notation
console.log(person['age']); // Bracket notation
```

Manipulating Objects in JavaScript

❖ Adding and Modifying Object Properties

You can add or modify properties of an object.

```
1  person.age = 31; // Modifying an existing property
2  person.address.city = 'New York'; // Modifying a nested property
3  person.hobbies.push('cooking'); // Adding a new hobby
```

Manipulating Objects in JavaScript

❖ Adding and Modifying Object Properties

You can add or modify properties of an object.

```
1  person.age = 31; // Modifying an existing property
2  person.address.city = 'New York'; // Modifying a nested property
3  person.hobbies.push('cooking'); // Adding a new hobby
```

Working with Arrays

❖ What is an Array?

An array is an ordered collection of values. The values can be of any data type.

```
1  let numbers = [1, 2, 3, 4];
2  const fruits = ['banana', 'apple', 'grape'];
3  let mixed = [1, 'banana', true];
4  let empty = [];
```

Working with Arrays

❖ Accessing Array Elements

Access array elements using index positions (starting from 0).

```
1  let numbers = [1, 2, 3, 4];
2  const fruits = ['banana', 'apple', 'grape'];
3  let mixed = [1, 'banana', true];
4  let empty = [];
5  // Access array elements using index positions (starting from 0).
6  console.log(numbers[0]); // 1
7  console.log(fruits[1]); // apple
8  console.log(mixed[2]); // true
```

Working with Arrays

❖ Adding and Removing Array Elements

You can add elements using push() and remove elements using pop().

```
1   numbers.push(5); // Add 5 to the end
2   fruits.pop(); // Remove the last element (grape)
3   mixed.push('orange'); // Add 'orange' to the end
4   empty.push(1); // Add 1 to the empty array
5
```

Array Methods

❖ Other Useful Array Methods

shift() and unshift() remove or add elements from the beginning of an array.

```
1  fruits.shift(); // Remove the first element (banana)
2  mixed.unshift('kiwi'); // Add 'kiwi' to the beginning
3
```

Array Methods

❖ Iterating Over Arrays

- ➢ You can use loops like for, forEach(), or map() to iterate over array elements.

```
1  // for loop example
2  for (let i = 0; i < numbers.length; i++) {
3    console.log('Number:', numbers[i]);
4  }
```

Array Methods

❖ Iterating Over Arrays

- ➢ You can use loops like for, forEach(), or map() to iterate over array elements.

```
1  // loop, foreach example
2  let fruits = ['banana', 'apple', 'grape'];
3  fruits.forEach(function (fruit) {
4    console.log('Fruit:', fruit);
5  });
```

Error Handling and Debugging

Error Handling and Debugging in JavaScript

❖ What is Error Handling?

Error handling is the process of anticipating, detecting, and responding to errors in a program to prevent crashes and unexpected behavior.

❖ Types of Errors

- ➢ Syntax Errors: Mistakes in the code structure.
- ➢ Runtime Errors: Errors that occur while the program is running.
- ➢ Logical Errors: Incorrect results due to wrong logic.

Try-Catch Statement

❖ Using Try-Catch

The **try** block contains code that may throw an error. The **catch** block handles the error.

```
1  try {
2    let result = riskyFunction();
3  } catch (error) {
4    console.log('Error: ' + error.message);
5  }
6
```

Try-Catch Statement

❖ Finally Block

The finally block will always execute after try and catch, regardless of whether an error occurred or not.

```
1  try {
2    let result = riskyFunction();
3  } catch (error) {
4    console.log('Error: ' + error.message);
5  } finally {
6    console.log('Always runs');
7  }
```

Throwing Custom Errors

❖ Throwing Errors

You can manually throw an error using the throw keyword.

```
1  function validateAge(age) {
2    if (age < 18) {
3      throw new Error('Age must be 18 or older.');
4    }
5  }
6
7  // usage:
8  try {
9    validateAge(16);
10 } catch (error) {
11   console.log('Error: ' + error.message);
12 }
```

Debugging Techniques

❖ **Using Console Methods**

console.log(), console.error(), console.warn(), and console.table() are useful for outputting data for debugging.

❖ **Using Browser Developer Tools**

Modern browsers have built-in developer tools (DevTools) for inspecting, debugging, and testing JavaScript code.

❖ **Setting Breakpoints**

Breakpoints can be set in the browser's DevTools to pause code execution at a specific line, allowing step-by-step inspection.

Asynchronous JavaScript

Asynchronous JavaScript

❖ What is Asynchronous JavaScript?

Asynchronous JavaScript allows tasks to run in the background without blocking the main execution thread, improving the performance of web applications.

❖ Why is Asynchronous Programming Important?

It helps handle tasks like fetching data from a server, loading images, and user input without freezing the user interface.

Callback Functions

❖ What is a Callback Function?

A callback is a function passed as an argument to another function, to be executed later after a task is completed.

```
1  function fetchData(callback) {
2    setTimeout(() ⇒ {
3      const data = { id: 1, name: 'John Doe' };
4      callback(data);
5    }, 1000);
6  }
7
8  fetchData(function (user) {
9    console.log('User fetched:', user);
10 });
```

Promises

❖ What is a Promise?

- ➢ A promise is an object that represents the eventual completion (or failure) of an asynchronous operation.

```
1  function fetchData() {
2    return new Promise((resolve) ⇒ {
3      setTimeout(() ⇒ {
4        const data = { id: 1, name: 'John Doe' };
5        resolve(data);
6      }, 1000);
7    });
8  }
9
10 fetchData().then((user) ⇒ {
11   console.log('User fetched:', user);
12 });
```

Async/Await

❖ What is Async/Await?

- async and await are modern ways to work with promises, making asynchronous code look synchronous.

```
async function fetchData() {
  try {
    let result = await fetch('https://api.example.com/data');
    if (!result.ok) {
      throw new Error('Network response was not ok');
    }
    let data = await result.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}
```

DOM Manipulation in JavaScript

DOM Manipulation in JavaScript

❖ What is DOM Manipulation?

DOM manipulation refers to changing the structure, content, and style of a webpage using JavaScript by interacting with the Document Object Model (DOM).

❖ What is the DOM?

The DOM is an object-oriented representation of the web page, which allows JavaScript to access and modify HTML and CSS dynamically.

❖ Why is DOM Manipulation Important?

It allows you to make web pages interactive and dynamic by changing elements in response to user actions (e.g., clicks, form submissions).

Accessing Elements in the DOM

- **getElementById()**
  Selects an element by its id attribute.
  let element = document.getElementById("myElement");
- **getElementsByClassName()**
  Selects elements by their class attribute. Returns a collection of elements.
  let elements = document.getElementsByClassName("myClass");
- **querySelector()**
  Selects the first matching element using CSS selectors.
  let element = document.querySelector(".myClass");

Modifying Element Content and Attributes

❖ Changing Text Content

Use **textContent** or **innerText** to change the text of an element.

```
document.getElementById("myElement").textContent = "New Text!";
```

❖ Changing HTML Content

Use **innerHTML** to change the HTML inside an element.

```
document.getElementById("myElement").innerHTML = "<b>New HTML!</b>";
```

❖ Changing Attributes

Use **setAttribute()** to change attributes such as src, href, etc.

```
document.getElementById("myImage").setAttribute("src", "newImage.jpg");
```

Manipulating Element Styles

❖ **Changing Styles Directly**

Modify inline styles using the style property.

```
document.getElementById("myElement").style.color = "blue";
```

❖ **Adding/Removing Classes**

Use **classList** to add, remove, or toggle classes.

```
document.getElementById("myElement").classList.add("newClass");
```

❖ **Changing Attributes**

Use **setAttribute()** to change attributes such as src, href, etc.

```
document.getElementById("myImage").setAttribute("src", "newImage.jpg");
```

Event Handling in JavaScript

Manipulating Element Styles

❖ What is Event Handling?

Event handling refers to the process of detecting user actions (like clicks, key presses, etc.) and executing code in response to those actions.

❖ Common Types of Events

Examples include **click**, **keypress**, **mouseover**, **submit**, **focus**, and **blur**.

Event Listeners

❖ What is an Event Listener?

An event listener is a function that waits for an event to occur on a specific element and executes code when the event is triggered.

```
1  document.getElementById('myButton').addEventListener('click', function () {
2    alert('Button clicked!');
3  });
4
```

Use addEventListener() to bind an event to an element.

Event Listeners

❖ Adding an Event Listener

Use **addEventListener()** to bind an event to an element.

Syntax: element.addEventListener(event, function, useCapture)

useCapture is a boolean parameter in addEventListener() that determines whether the event is handled during the capturing phase (true) or bubbling phase (false). Default is false (bubbling).

- Capturing: Event travels from the root to the target element.
- Bubbling: Event travels from the target element back to the root.

It controls the order of event handling when multiple listeners are attached to nested elements.

Use addEventListener() to bind an event to an element.

Event Object

❖ What is the Event Object?

The event object contains information about the event, such as the type of event, the target element, and other relevant details.

```
1  document.getElementById('myButton').addEventListener('click', function (event) {
2    console.log(event.target); // Logs the clicked element
3  });
4
```

Use addEventListener() to bind an event to an element.

Event Object

❖ Prevent Default Action

Use event.preventDefault() to prevent the default behavior of an event (e.g., preventing a form from submitting).

```
1  form.addEventListener('submit', function (event) {
2    // Check if the form is valid
3    if (!form.checkValidity()) {
4      event.preventDefault();
5      alert('Form submission prevented!');
6    }
7  });
```

Use addEventListener() to bind an event to an element.

Removing Event Listeners

❖ Removing an Event Listener

Use removeEventListener() to remove an event listener that was previously added.

```
1  function showMessage() {
2      alert("Event triggered!");
3  }
4
5  document.getElementById("myButton").addEventListener("click", showMessage);
6  // To remove it:
7  document.getElementById("myButton").removeEventListener("click", showMessage);
```