classmate
Date
Page

Complete Notes On

JavaScript ...

Instagram : @Curious_.programmer

Telegram : @Curious-coder

Written By :
Divya (CSE Student)

Index

Sr.no	Chapters	Pg.no
1.	Introduction to JavaScript 1.1. What is JavaScript and its role in web development 1.2. JavaScript History and Evolution 1.3. JavaScript Syntax and basic concepts 1.4. Integrating JavaScript into HTML.	1-7
2.	Variables, Data Types and Operators 2.1. Declaring and using variables in JavaScript 2.2. Primitive Data types 2.3. Complex Data types 2.4. Operators and expressions	8-13
3.	Control Flow and Loops 3.1. Conditional statements 3.2. Comparison operators and logical operators 3.3. Loops 3.4. Breaking and Continuing loop execution	14-20

Sr.no.	Chapters	Pg. no.
4.	Functions and Scope 4.1. Defining and invoking functions 4.2. Function parameters and return values 4.3. Function expression and arrow functions. 4.4. Scope and variable visibility	21-26
5.	Arrays and Array Methods 5.1. Creating and manipulating arrays 5.2. Accessing Array elements 5.3. Array methods 5.4. Iterating over arrays	27-31
6.	Object and Object-oriented Programming 6.1. Creating objects and object literals 6.2. Object properties and methods 6.3. Constructor functions and prototypes 6.4. Inheritance and object composition.	32-38
7.	Working with Document Object Model (DOM)	

Sr.no	Chapters	Pg.no.
	7.1. Introduction to the DOM 7.2. Accessing and manipulating HTML elements with JavaScript 7.3. Modify element styles and attributes 7.4. Handling events with event listeners.	39-44
8.	Debugging and Error-Handling 8.1. Understanding JavaScript errors. 8.2. Error handling with try-catch blocks 8.3. Debugging techniques 8.4. Handling asynchronous errors with promises	46-50
9.	Working with Dates and Times 9.1. ~~Formatting~~ Creating and manipulating data objects 9.2. Formatting and displaying dates 9.3. Performing date calculations and comparisons 9.4. Working with time zones.	51-55
10.	Introduction to JavaScript	

Sr.no	Chapters	Pg.no.
	7. Frameworks and Libraries	
	10.1. Overview of popular JavaScript Frameworks	
	10.2. Introduction to libraries	
	10.3. Using Frameworks and libraries for building interactive web applications	56-60
	10.4. Pros and Cons of using JavaScript frameworks.	

1. Introduction to JavaScript

JavaScript is a crucial programming language for web development.

In this chapter, we will delve into the basics of JavaScript, its history, syntax and how to integrate it into HTML documents:

1.1. What is JavaScript and its role in Web Development?

JavaScript is a versatile and widely-used programming language primarily used for web development. It plays a vital role in creating dynamic and interactive web applications.

Here are some key aspects of JavaScript's role in web development:

1. Client-side Scripting:

JavaScript is primarily executed on client-side (in the user's web browser). It allows web developers to enhance the functionality and interactivity of websites without relying solely on server-side technologies.

2. Interactivity:

JavaScript enables developers to add interactive features like form validation, animations, image sliders, and real-time updates to web pages. This interactivity enhances user experience and engagement.

3. DOM manipulation :

JavaScript can manipulate the Document Object Model (DOM), which represents the structure and content of a web page. Developers can use JavaScript to add, remove, or modify elements on a webpage dynamically.

4. AJAX :

Asynchronous JavaScript and XML (AJAX) allows web application to send and receive data from the server without requiring a page refresh. This technology is instrumental in creating responsive and seamless user interfaces.

5. Frameworks and Libraries :

JavaScript has a vast ecosystems of frameworks and libraries, such as react, Angular and vue.js, which simplify the development of complex

web applications.

1.2. JavaScript History and Evolution :

JavaScript was created by Brendan Eich at Netscape Communications Corporation in 1995. Originally called "LiveScript" it was later renamed JavaScript due to a marketing partnership with Sun MicroSystems.

Key points in JavaScript's history include:

- **Netscape Navigator :**

JavaScript was first introduced in Netscape Navigator 2.0, making it one of the first programming languages for web development.

- **ECMAScript :**

JavaScript's standardization is overseen by the Ecma international organisation. The official name of the language specification is ECMAScript, with different versions (ES5, ES6 e.t.c.) released over the years.

- **Browser Compatibility :**

JavaScript had compatibility issues between different web browsers, leading to the need of libraries like jQuery to abstract these differences.

- Node.js :

In 2009, Node.js was released, allowing developers to run JavaScript on the server-side. This marked a significant expansion of JavaScript's use beyond the browser.

1.3. JavaScript Syntax and Basic Concepts:

JavaScript has a C-style syntax and shares similarities with other programming languages like Java and C++.

Some basic concepts include:

- Variables :

Variables are used to store data values. They can hold numbers, strings, objects or other data types.

- Data-types :

JavaScript has various data types, including numbers, strings, booleans, objects and more.

- Operators:

Used to perform operations on variables and values, such as arithmetic, comparision and logical operations.

- Functions:

Functions are blocks of reusable code that can be called with specific arguments. They allow us to encapsulate logic and make our code more modular.

- Control Structures:

JavaScript supports conditional statements (if, else), loops (for, while), and switch statements for controlling program flow.

- Objects and arrays:

JavaScript uses objects and arrays to store and manipulate data. Objects are collections of key-value pairs, while arrays are ordered lists of values.

1.4. Integrating JavaScript into HTML:

- Inline Script :

You can embed JavaScript code directly in an HTML document using the '<script>' tag within the '<head>' or '<body>' of your HTML file.

- External Script :

You can also link to an external JavaScript using the '<script>' tag's 'src' attribute. This approach is preferred for maintaining clean and organised code.

- Event Handlers :

JavaScript can be used to handle events like clicks, keypresses, and form submissions. You can attach event handlers to HTML elements to trigger specific JavaScript functions.

Here is an example of embedding JavaScript in an HTML document:

```
<!DOCTYPE html>
<html>
<head>
        <title> JavaScript Introduction </title>
</head>
```

```
<body>
    <h1> Hello , World </h1>
<script>
    // Inline JavaScript code
  alert ('Welcome to JavaScript');
</script>
</body>
</html>
```

This chapter provides a foundation for understanding JavaScript's role in web development, its history, basic syntax, and how to integrate it into HTML documents. Subsequent chapters will delve into more advanced JavaScript topics and techniques.

2. Variables, Data Types and Operators

In this chapter, we will explore the fundamental building blocks of JavaScript including how to declare and use variables, the various data types available, and how to work with operators and experssions.

2.1. Declaring and Using Variable in JavaScript:

Variables in JavaScript are used to store and manage data. They provide a way to give a name to a value so that you can reference and manipulate it in your code.

To declare a variable, we use the 'var', 'let' or 'const' keyword, followed by the variable name and, optionally an initial value.

- `var age = 25;`
- `let name = "John";`
- `const PI = 3.14;`
- 'var': Used for variable declaration globally or within a function scope. It has some scoping

issues and is less commonly used in modern javaScript.

- 'let' : introduced in ES6, it allows block scoped variables. It's commonly used for variables that may change their value.

- 'const' : Also, introduced in ES6, it declares a constant variable with a value that cannot be reassigned. Use it for values that should not change.

To use a variable, you simply reference it by its name.

For Example:

```
console.log(name);    // Outputs: "John"
```

2.2. Primitive Data Types (Numbers, strings, Booleans);

JavaScript has several primitive data types that represent simple values:

- Numbers:
Used for numeric values. JavaScript supports both integer and floating-point

numbers. For example:

```
let num = 42 ;   // Integer
let price = 12.99  // Floating-point
```

- **Strings:**
Used for text data. Strings are enclosed in single or double quotes.
For example:

```
let greeting = "Hello, World!" ;
```

- **Booleans:**
Represent binary values, 'true' or 'false' often used for conditions and logic.

For example:

```
let isStudent = true ;
```


2.3. Complex Data Types (Arrays, Objects):

JavaScript also has complex data types that allow you to group multiple values together.

- **Arrays:**

Ordered collections of values, Arrays are defined as by using square brackets '[]' and can contain

elements of different data types:

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

- Objects:

Unordered collects of key-value pairs. Objects are defined using curly braces '{}'.

```
let person = {
        name : "Alice",
        age : 30,
        isStudent : false
    };
```


2.4. Operators and Expressions:

Operators in JavaScript are used to perform operations on values and variables. JavaScript supports a wide range of operations, including:

- Arithmetic Operators:

Used for basic mathematical operations such as addition, subtraction, multiplication, division, and modulus.

```
let a =10 ;
let b = 5 ;
let sum = a + b ; // 15
let product = a * b ; // 50
```

- **Comparison Operators :**

Used to compare values and return boolean results.

```
let y = 5 ;
let z = 10 ;
let isEqual = y === z ; // false
let isGreater = y > z ; // false
```


- **Logical Operator :**

Used for logical operations like AND ('&&'), OR ('||'), and NOT ('!').

```
let isStudent = true ;
let isTeenager = false ;
let canVote = isStudent && ! isTenager ;// true
```

- **Assignment Operators :**

Used to assign values to variables.

```
let count = 0 ;
Count += 5 ; // Equivalent to count = count+5
```

- Conditional (Ternary) Operator:

Allows you to write concise conditional statement.

```
let age = 20;
let canDrink = age >= 21 ? "Yes" : "No";
                           // "No"
```

Understanding variables, datatypes, and operators is fundamental to writing JavaScript code. Those concepts lay the groundwork for more complex programming tasks, allowing you to create dynamic and interactive web applications.

3. Control flow and loops

In this chapter, we will explore the control flow mechanisms in JavaScript, which allow us to make decisions and control the flow of our code using conditional statements and loops.

3.1. Conditional Statements (if, else if, switch):

Conditional Statements in JavaScript are used to execute different blocks of code based on certain conditions. The primary conditional statements are 'if', 'else if', and 'else'.

Additionally, JavaScript provides the 'Switch' statement for more complex branching logic.

- **If Statement:**

The 'if' statement allows you to execute a block of code if a specified condition is true.

- ```
  let age = 18;
  if (age >= 18) {
      console.log("You can Vote")
  }
  ```
  ```

- **else Statement:**

You can use the 'else' statement to specify a block of code to be executed if the condition in the 'if' statement is false.


```
let age = 15;
if (age >= 18) {
    console.log("You can vote!");
} else {
    console.log("You cannot vote.");
    // This code block is executed.
}
```

- **else if statement:**

When you have multiple conditions to check, you can use the 'else if' statement to add more conditions.

```
let age = 25;
if (age < 18) {
    console.log("You cannot vote!");
} else if (age >= 18 && age < 25) {
    console.log("You can vote, but not
        run for office.");
} else {
    console.log("You can vote and run
        for office.");
}
```

- **Switch Statement:**

The 'switch' statement is used when you have a value that can have multiple possible outcomes. It is often more concise than a series of 'if' and 'else if' statements.

```
let day = "Monday";
switch (day) {
        case "Monday":
    console.log ("It's the start of
                the week.");
    case "Friday":
    console.log ("It's almost the
                weekend.");
    break;
  default:
    console.log ("It's a regular day");
}
```


3.2. Comparison Operators and Logical Operators:

Conditional Statements often rely on comparison operators and logical operators to evaluate conditions.

Here are some common operations:

- **Comparison Operators:**

Used to compare values and return boolean results.

- '==' (equality): checks if two values are equal.
- '===' (strict equality): checks if two values are equal in both value and datatype.
- '!==' (strict inequality): checks if two values are not equal in both value and datatype.
- '!=' (inequality): Checks if two values are not equal.
- '>' (greater than), '<' (less than), '>=' (greater than or equal to), '<=' (less than or equal to): Compare numerical values.

- **Logical Operators:**

Used to perform logical operations that are on boolean values.

- '&&' (logical and): Returns true if both operands are true.
- '||' (logical OR): Returns true if atleast one operand is true.

- '!' (logical NOT) : Negates the boolean value of an operand.

3.3. Loops (for, while, do-while):

Loops are used in JavaScript to repeatedly execute a block of code as long as a specified condition is true.

JavaScript offers several loop constructs:

- **For Loop:**

A 'for' loop is used when you know the number of iterations you need in advance.

```
for (let i=0; i<5; i++) {
    console.log(i); // Print numbers 0 to 4
}
```

- **while Loop:**

A 'while' loop continues to execute as long as a specified condition is true. Be cautious with the condition to avoid infinite loops.

```
let i=0;
while (i<5) {
```

```
    console.log(i); // Prints numbers 0 to 4.
    i++;
}
```

- **do-while loop:**

A 'do-while' loop is similar to a 'while' loop, but it always executes the block of code at least once before checking the condition.

- ```
let i = 0;
do {
    console.log(i); // Print numbers 0 to 4.
    i++;
} while (i < 5);
```



## 3.4. Breaking and Continuing Loop Execution

Sometimes, you need to control the flow within a loop using 'break' and 'continue' statements:

- **break:**

The 'break' statement is used to exit a loop prematurely. It immediately terminates the loop and continues with the code after the loop.
```

```
for (let i= 0; i< 10 ; i++)
  if (i == 5) {
        break; // Exit the loop when i is 5
  }
  console.log(i);
}
```


- **continue:**

The 'continue' statement is used to skip the current iteration of a loop and move to the next one.

```
for (let i=0; i<10; i++) {
  if (i === 5) {
        continue; // skip iteration when i is 5
  }
  console.log(i); // Print numbers 0 to
                  4 and 6 to 9.
}
```

Understanding control flow and loops is essential for building dynamic and responsive applications. These constructs allow you to make decisions, iterate over data, and create efficient and flexible code in JavaScript.

4. Function and Scope

In this chapter, we'll dive into the world of functions in JavaScript, exploring how to define and invoke them, work with parameters and return values, and understand the concept of scope and variable visibility.

4.1. Defining and invoking functions:

A function in JavaScript is a reusable block of code that performs a specific task when called or invoked.

Functions are essential for structuring code and making it more organised and modular.

Here's how you define and invoke a function:

- **Function Definition:**

To define a function, you use the 'function' keyword followed by the function name, a list of parameters enclosed in parantheses, and a block of code enclosed in curly braces.

```
function greet(name) {
    console.log('Hello, ${name}!');
}
```

- **Function Invocation:**

To use or call a function, you simply reference its name followed by parentheses, optionally passing values as arguments.

- `greet("Alice"); // Outputs: "Hello, Alice!"`

Functions can take zero or more parameters and may or may not return a value.

4.2. Function Parameters and Return values:

- **Parameters:**

Parameters are placeholders for values that a function experts to receive when it's called. They act as local variables within the functions scope.

- ```
  function add (a, b) {
      return a + b;
  }
  ```

In the above example 'a' and 'b' are parameters that represent the values passed to the 'add' function when it's invoked.
  ```

- **Return values:**

Functions can return values using the 'return' statement. The returned value can be used in other parts of your code.

- ```
  let result = add(5, 3); // result will be 8
  ```

The 'add' function returns the sum of 'a' and 'b' which is then assigned to the 'result' variable.

4.3. Function Expression and Arrow Functions:

In addition to traditional function declarations, JavaScript supports function expressions and arrow functions, which offer more concise syntax and flexibility.

- **Function Expression:**

A function expression is defined by assigning an anonymous function (a function without a name) to a variable.

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

You can also use named function expressions for better debugging.

```
const divide = function divideNumbers
                (a, b) {
    return a / b;
};
```

- **Arrow Functions:**

Arrow Functions provide a more concise way to write functions, especially for simple one-liners. They use the '=>' syntax.

```
const square = (x) => x * x;
```

Arrow functions are often used for short, simple functions and have a more compact syntax compared to traditional function expressions.

4.4. Scope and Variable Visibility:

- Scope in JavaScript refers to the context in which variables are declared and accessed.

Understanding scope is crucial because it determines where variables are accessible throughout your entire program.

- **Global Scope:**

Variables declared outside of any function have global scope and are accessible throughout your entire program.

- ```
  const globalVar = 42;
  function logGlobalVar() {
      console.log(globalVar); // Accessible within
                              the function
  }
  ```


- **Local Scope (function scope):**

Variables declared within a function have local scope and are only accessible within that function.

- ```
  function greet(name) {
      const message = 'Hello, ${name}!';
      // 'message' is only accessible in the
         'greet' function
      console.log(message);
  }
  ```

- **Block Scope:**

Introduced in ES6, the 'let' and 'const' keywords allow you to declare variables with block scope, limited to the block in which they are defined (e.g., inside a loop or conditional statement.).

- ```
  if (true) {
      let blockVar = "I am a block-scoped
          variable";
  }
  ```

- **Lexical Scope (Closure):**

Functions can access variables from their containing (enclosing) scope. This is known as lexical scope or closure.

- ```
  function outer() {
      const outerVar = "I'm the outer fun-
                        ction";
      function inner() {
          console.log(outerVar); // Inner function
                        can access 'outerVar'
      }
      inner();
  }
  ```

Understanding scope is crucial for avoiding variable conflicts, managing memory efficiently, and writing maintainable and bug-free code. Functions are key to organizing code into manageable pieces and controlling the scope of variables within your program.

5. Array and Array Methods:

Arrays are a fundamental data structure in JavaScript, allowing you to store and manipulate data collection.

In this chapter, we will explore how to create, manipulate arrays, access their elements, and utilize various array methods for adding, removing, and modifying elements.

Additionally, we'll cover different techniques for iterating over arrays.

5.1. Creating and Manipulating Arrays:

You can create arrays in JavaScript using square brackets '[]'. An array can hold various data types, including numbers, strings, objects, and even other arrays.

Here's how you create an array:

- `let fruits = ["apple", "banana", "cherry"];`

Arrays are zero-indexed, meaning the first element has an index of 0, the second has an index of 1, and so on.

5.2. Accessing Array Elements :

You can access array elements using square brackets and the index of the element you want to retrieve.

For Example :

- ```
  console.log(fruits[0]); // Output : "apple"
  console.log(fruits[1]); // Output : "banana"
  ```


5.3. Array Methods :

JavaScript provides a variety of methods to manipulate arrays.

Here are some commonly used ones :

- **push** : Adds one or more elements to the end of an array.

  ```
  fruits.push("grape");
  ```

- **pop** : Removes the last element from an array and returns it.

  ```
  let lastFruit = fruit.pop();
  // removes "grape" and assigns it to lastFruit.
  ```

- **shift** : Removes the first element from
  ```

an array and returns it, shifting the other elements to a lower index.


```
let firstFruit = fruits.shift();
    // Removes "apple" and assigns it to firstFruit.
```

- **unshift**: Adds one or more elements to the beginning of an array, shifting the existing elements to higher indices.

```
fruits.unshift("kiwi", "pear");
    // Adds "kiwi" and "pear" to the beginning.
```

- **slice**: Creates a new array containing a portion of the original array, defined by start and end indices (end index is exclusive).

```
let slicedFruits = fruits.slice(1, 3);
    // Creates an array with "banana" and "cherry"
```

- **splice**: Changes the contents of an array by adding or removing elements at a specified index.

```
fruits.splice(1, 1); // Removes an element at index 1 ("banana")
fruits.splice(1, 0, "blueberry"); // Adds "blueberry" at index 1.
```

5.4. Iterating over Arrays:

You can iterate over arrays in various ways, depending on your needs.

- **For Loop:**

A traditional 'for' loop allows you to iterate over an array by index.

```
for (let i = 0; i < fruits.length; i++) {
  console.log (fruits[i]);
}
```


- **forEach:**

The 'forEach' method executes a provided function once for each array element.

```
fruits.forEach (function (fruit) {
    console.log (fruit);
});
```

- **map:**

The 'map' method creates a new array by applying a function to each element of the original array.

```
let UpperCaseFruits = fruits.map (function
                      (fruit) {
```

```
    return fruit.toUpperCase();
});
```

Those methods provide powerful tools for manipulating and iterating over arrays in JavaScript.

Understanding how to use them effectively is crucial for working with collections of data in your programs.

6. Objects and Object-oriented Programming:

In this chapter, we will explore the concept of objects in JavaScript and delve into object-oriented programming (OOP) principles, including creating objects, object literals, properties, methods, constructor functions, prototypes, inheritance, and object composition.

6.1. Creating Objects and Object Literals:

In JavaScript, an object is a composite data type that can hold multiple values as properties and methods. You can create objects using various approaches:

The simplest way to create an object is by defining it as a literal using curly braces '{}'. You can specify properties and methods within the braces.

```
let person = {
    firstName : "John",
    lastName : "Doe",
    sayHello : function(){
```

```
    console.log("Hello," + this.firstName +
                " " + this.lastName);
  }
};
```


You can use constructor functions to create multiple objects with the same structure. Constructor functions are like blueprints for creating objects.

```
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
  this.sayHello = function(){
    console.log("Hello," + this.firstName
      + " " + this.lastName);
  };
}

let John = new Person("John", "Doe");
let Jane = new Person("Jane", "Smith");
```

You can also create objects with a prototype using 'Object.create()'. This method allows you to define a prototype object and create new objects and create new objects based on it.

```
let personPrototype = {
    sayHello : function () {
    console.log ("Hello," + this.firstName +
        " " + this.lastName);
  }
};

let John = Object.create (personPrototype);
John.firstName = "John";
John.lastName = "Doe";
```


6.2. Object Properties and Methods:

Objects consist of properties (data values) and methods (functions). You can access and modify these properties and methods using dot notation or bracket notation.

```
let person = {
    firstName : "John",
    lastName : "Doe",
    age : 30,
    sayHello : function () {
  console.log ("Hello," + this.firstName +
        " " + this.lastName);
  }
};

console.log (person.firstName); // Accessing
                                a property.
```

```
person.age = 31; // Modifying a property
person["lastName"] = "Smith";
    //Another way to modify a property
person.sayHello(); // Calling a method.
```

6.3. Constructor Functions and Prototypes:

Constructor functions are a way to create objects with a shared structure and behaviour. They are typically named with an initial uppercase letter to distinguish them from regular functions.

```
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
}
```



```
Person.prototype.sayHello = function(){
  console.log("Hello, " + this.firstName +
    " " + this.lastName);
}

let John = new Person("John", "Doe");
let Jane = new Person("Jane", "Smith");
```

- In this example, the 'Person' constructor creates instances of the 'Person' object with the shared methods defined in the 'Person.prototype'. This approach saves memory and promotes code reusability.

6.4. Inheritance and Object Composition:

Inheritance is a key concept in OOP. It allows you to create new objects based on existing ones, inheriting their properties and methods. In JavaScript, inheritance is achieved using prototypes.

```
function Animal(name) {
    this.name = name;
}
```



```
Animal.prototype.speak = function() {
    console.log(this.name + "make a sound.");
};

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.bark = function() {
    console.log(this.name + "barks!");
};

let myDog = new Dog("Buddy", "Golden Retriever");
```

```
myDog.speak(); // Inherited from Animal
myDog.bark(); // Defined in dog
```

In this example, the 'Dog' constructor inherits properties and methods from the 'Animal' constructor using prototype chaining.

Object composition is another OOP concept where you create objects by combining or composing multiple objects, rather than inheriting from a single parent object.

```
function canFly(obj) {
  obj.fly = function() {
    console.log("Flying!");
  };
}
```



```
function canSwim(obj) {
  obj.swim = function() {
    console.log("Swimming!");
  };
}

let duck = {};
canFly(duck);
canSwim(duck);
```

```
duck.fly(); // Output: "Flying!"
duck.swim(); // Output: "Swimming!"
```

This approach allows you to create objects with specific behaviors by adding "mixins" or "traits" to them.

Understanding objects, constructors, prototypes, inheritence, and composition is essential for writing clean and maintainable code in JavaScript, especially when building complex applications.
These principles help you organize your code and make it more modular and extensible.

7. Working with Document Object Model

The Document Object Model (DOM) is a crucial concept in web development, as it allows JavaScript to interact with and manipulate web pages dynamically.

In this chapter, we will explore what the DOM is, how to access and Modify HTML elements using JavaScript, how to change element styles and attributes, and how to handle events with event listeners.

7.1. Introduction to the DOM:

The Document Object Model (DOM) is a crucial concept in web development and a programming ~~language~~ interface provided by web browsers to represent the structure and content of a web page as a tree-like structure of objects. Each elements in an HTML document, such as headings, paragraphs, images, forms and more is represented as a node in this tree.

The DOM allows JavaScript to:

- Access and retrieve elements from an

HTML docment.

- Modify the content, structure, and style of element.
- Respond to user interactions and events on the page.

It provides a structured way to interact with and manipulate web pages, making them dynamic and responsive.

7.2. Accessing and Manipulating HTML elements with JavaScript:

JavaScript provides several methods and properties to access and manipulate HTML elements in the DOM:

- **getElementById:**

This method allows us to retrieve an element with a specific 'id' attribute.

```
let element = document.getElementById
("myElement");
```

- **getElementsByClassName:**

You can select elements with a specific class using this method, and it returns a collection of elements.

```
let elements = document.getElementsByClassName("myClass");
```

- **getElementsByTagName :**

This method returns a collection of elements with specific HTML tag name.

```
let elements = document.getElementsByTagname("p");
```

- **querySelector and querySelectorAll :**

These methods allow you to select elements using CSS-style selectors. 'querySelector' returns the first matching element, while 'querySelectorAll' returns a collection of all matching elements.


```
let element = document.querySelector("#myElement");
let elements = document.querySelectorAll(".myClass");
```

Once you've selected an element, you can manipulate it by changing its content, attributes, or styles.

```
let element = document.getElementById("myElement");
```

```
element.innerHTML = "New Content";
                    // Change the inner HTML
element.setAttribute("class", "newClass");
                    // Change the class attribute
element.style.color = "red"; // Change the
                                text color
```


7.3. Modify Element Styles and Attributes:

You can modify the styles and attributes of HTML elements using JavaScript. The 'style' property allows you to access and change CSS properties of an element directly.

For example:

```
let element = document.getElementById
                ("myElement");
element.style.color = "blue";
element.style.backgroundColor = "yellow";
element.style.fontSize = "16px";
```

You can also modify element attributes using the 'setAttribute' method:

```
let element = document.getElementById
                ("myElement");
element.setAttribute("class", "newClass");
element.setAttribute("src", "new-image-
                .jpg");
```

7.4. Handling Events with Event Listeners:

Event listeners allows you to respond to user interactions (eg; clicks, keyboard inputs, mouse movements) and execute JavaScript code in response. You can attach event listeners to HTML elements to listen for specific events and trigger functions when those events occur.


```
let button = document.getElementById("my-
                Button");
button.addEventListener("click", function()
  {
    alert("Button clicked!");
});
```

In the above example, a click event listener is attached to a button element. When the button is clicked, the specified function is executed, displaying an alert.

Common DOM events include click, mouseover, mouseout, keydown, submit, and many more.
Event listeners make web pages interactive and enable you to create

dynamic user experiences.

Understanding how to work with the DOM is essential for web developers, as it forms the foundation for building interactive and responsive web applications. It allows you to access, modify, and respond to user interactions on web pages using JavaScript.

8. Debugging and Error Handling:

Debugging and error handling are crucial skills for web developers.

In this chapter, we will delve into understanding JavaScript errors, error-handling techniques using try-catch blocks, debugging methods such as console logging and breakpoints, and handling asynchronous errors with promises.

8.1. Understanding JavaScript errors:

JavaScript errors can occur during the execution of your code. These errors can be categorized into several types, including:

- **Syntax Errors:**

These occur when you write code that does not follow the correct syntax rules of JavaScript.

For example: missing closing parentheses, brackets, or semicolons can lead to syntax errors.

- ```
  if (x > 5 {  // Syntax error, missing closing parenthesis
      console.log("x is greater than 5");
  }
  ```

- **Runtime Errors** (Exceptions):

These occur during code execution when something goes wrong, such as trying to access a property of an undefined variable or dividing by zero.

- ```
  let undefinedVar;
  console.log(undefinedVar.property);
  // Runtime error, undefinedVar is not defined.
  ```


- **Logic Errors**:

These are not technical errors but mistakes in the logic of your code. Your code runs without throwing errors, but it produces incorrect results.

- ```
  function add(a, b) {
      return a - b; // logic error, should be a + b
  }
  ```

Understanding the type and cause of errors is essential for effective debugging and troubleshooting.
  ```

8.2 Error handling with try-catch Blocks:

The 'try-catch' block is a JavaScript construct for handling and managing errors gracefully. It allows you to attempt a block of code that might throw an error and catch and handle that error if it occurs.

```
try {
  // code that might throw an error
} catch (error) {
  // code to handle the error
}
```


Here's an example:

```
try {
  let undefinedVar;
  console.log(undefinedVar.property);
} catch (error) {
  console.error("An error occured:",
                error.message);
}
```

8.3. Debugging Techniques: Console.log, Breakpoints, Browser Dev Tools:

Debugging is the process of identifying and fixing errors in your code.

JavaScript prod provides several debugging techniques to help you find and resolve issues:

- **Console Logging:**

You can use 'console.log()' statements to ouput values, variables, and messages to the browser's console. This helps you inspect the state of your code at different points in its execution.

- `console.log("Value of x:", x);`

- **Breakpoints:**

Modern web browsers come with developer tools that allow you to set breakpoints in your code. Breakpoints pause the execution of your code at specific lines, allowing you to inspect variables and step through your code one line at a time.

- **Browser Dev Tools:**

Browser developer tools provide a range of debugging features, including a console for logging, a debugger for

setting breakpoints and stepping through code, and tools for inspecting the DOM and network requests.

8.4. Handling Asynchronous Errors with Promises:

When working with asynchronous operations in JavaScript, errors can occur in those operations. Promises provide a mechanism for handling asynchronous errors using the '.catch()' method. When a promise rejects (an error occurs), the '.catch()' block is executed.


```
fetch("https://example.com/api/data")
  .then(response => {
    if (!response.ok) {
      throw new Error("Network response
          was not ok");
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.log("An error occured:",
        error.message);
  });
```

In this example, the '.catch()' block handles errors that might occur during the network request or JSON parsing.

Understanding debugging techniques and error handling is essential for building robust and reliable web applications. These skills enable you to identify and fix issues in your code, leading to better software quality and a smoother user experience.

9. Working with Dates and Times:

Working with dates and times is a common task in web development, as it enables us to display time sensitive information, schedule events, and perform various date related calculations.

In this chapter, we will explore how to create, manipulate, format and display dates in JavaScript.

9.1 Creating and manipulating Date objects:

In JavaScript, we can work with dates using the date object, which provides methods for creating and manipulating dates.

To create a new Date object, we can use the new Date() constructor.

```
var currentDate = new Date();
console.log(currentDate);
                // output the current date
                    and time
```

We can also create specific dates by passing year, month, day, hr,

minute, second and millisecond values to the Date Constructor.

```
var specificDate = new Date(2023, 6,
                   15, 12, 30, 0, 0);
console.log(specificDate);

// output Sun Jul 15 2023 12:30:00
```


To manipulate dates, we can use various methods such as 'setFullyear(), setMonth(), setDate(), setHours(), setMinute(), setSecond(), and SetMillisecond().

```
var date = new date();
date.setFullYear(2023);
date.setMonth(7);
date.setDate(23);
console.log(date);

//output : wed Dec 05 2023 09:00:00
```

9.2 Formatting and displaying dates:

To display data in a more readable format, we can use various methods to extract specific date and components and format them as needed.

```
var date = new Date();
var year = date.getFullYear();
var month = date.getMonth() + 1;
// Adding 1 to match the actual month
var day = date.getDate();
var hours = date.getHours();
var minutes = date.getMinutes();
var seconds = date.getSeconds();
console.log(year + '-' + month + '-' +
    day + '-' + hours + ':' + minutes
    + ':' + seconds);
// output: Current date and time in
    'yyyy-mm-dd' 'HH:mm:ss'
    format
```


Alternatively, we can use 'toLocalString()' method to get a localised date and time representator.

```
console.log(date.toLocalString());
// output: localised date and
    time representation.
```

9.3. Performing Date calculations and comparisons:

JavaScript provides several methods to perform date calculations and comparisons. We can use methods like getTime() to get the timestamp of a date and

perform mathematical operations on dates.

```
var date1 = new Date(2023, 6, 15);
var date2 = new Date(2023, 8, 20);

var differenceInMilliseconds = date2.getTime() - date1.getTime();
var differenceInDays = differenceInMilliseconds / (1000 * 60 * 60 * 24);
console.log(differenceInDays);

// Difference in days between the 2 dates
```


We can also compare dates using comparison operators like (<, >, =, >=).

9.4 Working with Time Zones:

Dealing with time zones can be challenging, as different locations have different time effects. JavaScript Date object uses the host system's time zone by default, but we can work with different time zones using libraries like Moment.js or the built-in 'toLocalString()' method with appropriate options.

```
var date = new Date();
console.log(date.toLocaleString('en-US',
  { timeZone: 'America/NewYork' })
);

// Output: Date and time in New York
           time zone.
```

10. Introduction to JavaScript Frameworks and Libraries

JavaScript frameworks and Libraries are essential tools for web developers, as they provide a structured and efficient way to build complex and interactive web applications.

In this chapter, we will explore popular JavaScript frameworks, introduce libraries, discuss their role in building interactive web applications, and examine the pros and cons of using JavaScript frameworks.

10.1 Overview of popular JavaScript frame works:

JavaScript frameworks are comprehensive tools that provide a complete structure and set of functionalities for building web applications.

Some of the most popular JavaScript frameworks include React, Angular and vue.

- React:

Developed and maintained by facebook React is a declarative compoment-

based JavaScript library for building user interfaces. It allows developers to create reusable UI components, making it easier to manage complex applications.

- **Angular:**
Developed and maintained by Google. Angular is a comprehensive JavaScript framework that provides a complete solution for building large scale applications. It offers powerful features such as data binding, dependency injection and routing.

- **Vue.js:**
Vue is a progressive and user-friendly JavaScript framework that allows developers to build interactive user interface. It provides a gentle learning curve and can be integrated incrementally into existing projects.

10.2. Introduction to Libraries (JQuery, Lodash):

JavaScript libraries on the other hand are collections of pre-written code that simplify and speed up development tasks. They are focused on providing specific functionalities,

making them lightweight and easy to use.

- **JQuery:**
JQuery is one of the most popular JavaScript libraries. It simplifies DOM manipulation, event handling, and AJAX calls, making it easier to work with web page elements and interact with servers.

- **Lodash:**
Lodash is a utility library that provides a wide range of helper functions to work with arrays, objects, strings and more. It improves code efficiency and readability by offering methods for common data manipulation tasks.

10.3. Using frameworks and Libraries for building interactive web applications:

JavaScript frameworks and libraries are essential for building interactive and dynamic web applications. They offer a structured architecture and a wealth of tools that enable developer to manage complex data, handle user interactions, and update our interface efficiently.

Using frameworks and libraries allows developers to save time and effort by leveraging pre-built solutions for common tasks. This, inturn, improves development speed and helps deliver more robust and feature rich applications.

10.4. Pros and cons of using JavaScript Frameworks:

Using JavaScript can be beneficial for many reasons:

- **Structured Architecture:**
  Frameworks provide a structured architecture that promotes organization and maintainability.

- **Reusability:**
  Components and frameworks can be reused across different parts of the application, promoting code reuse.

- **Efficiency:**
  Frameworks offer pre-optimised solutions for common tasks improving development efficiency.

However, there are also some cons to consider.

- **Learning curve:**
  Some frameworks have a steep learning curve, especially for beginners.

- **Complexity:**
  Large frameworks can introduce unnecessay complexity for smaller projects.

- **Performance:**
  Some frameworks may add overhead to the application, affecting performance.