**JavaScript BootCamp Notes**

Syllabus….

**Module 1: Introduction to Programming/Coding**

1.1 Understanding Programming Concepts

1.2 Importance of Algorithms

1.3 Syntax and Structure of a Program

1.4 Development Environment Setup

**Module 2: Fundamentals of JavaScript**

2.1 Introduction to JavaScript

2.2 Variables and Data Types

2.3 Declaring Variables in JS (let, const, var)

2.4 Working with Strings and Numbers

2.5 Operators in JavaScript

**Module 3: Control Flow in JavaScript**

3.1 Conditional Statements in JavaScript

3.2 Looping Statements in JavaScript

**Module 4: Functions in JavaScript**

4.1 Different Types of Functions

4.2 Function Declaration

4.3 Function Types (Anonymous, Arrow)

4.4 Function Scope and Closure

**Module 5: In-Built Data Structures in JS**

5.1 Arrays in JavaScript

5.2 Operations and Methods on Arrays

5.3 Objects in JavaScript

5.4 Operations and Methods on Objects

**Module 6: Object-Oriented Programming (OOP) in JavaScript**

6.1 Classes and Objects

6.2 Pillars of OOP

6.3 Polymorphism

6.4 Inheritance

6.5 Encapsulation

6.6 Abstraction

6.7 'this' Keyword and Scope

**Module 7: Asynchronous JavaScript**

7.1 setTimeout and setInterval

7.2 Promises

7.3 Async/Await

7.4 Event Listeners

7.5 Fetch API

**Module 8: Document Object Model (DOM)**

8.1 Introduction to DOM

8.2 4 Pillars of DOM Manipulation

8.3 Selecting an Element

8.4 Changing HTML Content

8.5 Event Listeners

**Module 9: Simple Projects on DOM Manipulation**

9.1 Building Basic Projects to Reinforce DOM Concepts

**Module 10: Real-World Projects using JavaScript**

10.1 Application of JavaScript in Real-World Projects

10.2 Problem Solving and Project Implementation

**Module 1: Introduction to Programming/Coding**

Programming concepts form the foundation of coding and enable developers to create efficient and effective solutions.

Concepts include variables, data types, control flow, and algorithms.

Here are some key concepts:

**Variables:**

Variables are containers for storing data values.

**Example:** let age=25

**Data Types:**

Data types define the type of data a variable can hold.

Common types include string, number, boolean, and object.

**Example:**  let name = "John";

let score = 90;

let isPassed = true;

**Control Flow:**

Control flow determines the order in which code is executed.

**Example (Conditional Statement):**

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

}

**Algorithms:**

Algorithms are step-by-step procedures for solving problems.

**Example :** Algorithm: Add Two Numbers

* + 1. Start:
    2. Read the first number (num1).
    3. Read the second number (num2).
    4. Add num1 and num2.
    5. Store the result in a variable (result).
    6. Display or print the result.
    7. End.

**Syntax and Structure:**

Syntax refers to the set of rules that govern how code is written.

Proper indentation and the use of semicolons are part of the syntax and contribute to code structure.

**Example (Function Declaration):**

function add(a,b) {

console.log(a+b);

}

a=5;

b=10;

add(a,b);

**Development Environment:**

Development environment includes tools used to write, test, and debug code.

**Example (Setting Up Visual Studio Code):**

Download and install Visual Studio Code from https://code.visualstudio.com/.

Install extensions for JavaScript development.

Understanding these concepts lays the groundwork for becoming proficient in programming.

**Module 2: Fundamentals of JavaScript**

**What is JavaScript (JS)?**

Definition: JavaScript is a programming language used to make websites interactive. It's like the magic behind buttons that change colors when you click them or forms that check if you've filled them out correctly.

**Why is JavaScript Used?**

**Enhanced Websites:** It makes websites more exciting by adding moving elements, pop-ups, and dynamic features without having to reload the whole page.

**Works with Everything:** JavaScript is part of popular web development frameworks like MERN and MEAN. These frameworks use JS for both front and back ends, meaning one language can be used for the entire website.

**Nearly Everywhere:** Around 92% of websites use JavaScript in some way. Whether it's for making things pretty on the screen or handling data on the server, JS is there.

**Applications and Use Cases:**

**Web Development:** JS is the backbone of making websites look cool and work smoothly.

**Frameworks (MERN, MEAN):** Used in frameworks that simplify web development, making it easier and faster to create awesome websites.

**Frontend and Backend:** It's not just for making things look pretty; JS also works behind the scenes, managing data and making websites do what you want.

**Used by Everyone**: Many popular websites and apps use JavaScript to give you a great experience, from Facebook to Google.

JavaScript is like the handyman of the internet, making sure everything looks good and works well!

**2.2: Variables and Data Types**

**What are Variables?**

Variables are like containers that store information. They help us keep track of data in a program.

**Data Types:** Type of data stored inside a variable or container is known as DataType

**String:**

Example: "Hello, World!"

**Number:**

Example: 42

**Boolean:**

Example: true or false

**Array:**

Example: [1, 2, 3]

**Object:**

Example: { name: "John", age: 25 }

**Declaring Variables in JS (let, const, var):**

**let:**

Use let when you expect the variable to change its value.

Example:

let age = 25;

age = 26; // Valid

**const:**

Use const when the variable's value should never change.

Example:

const pi = 3.14;

// pi = 3.15; // Error: Cannot reassign a const variable

**var:**

Historically used, but let and const are preferred in modern JavaScript.

Example:

var count = 10;

count = 11; // Valid, but not recommended in modern code

**Key Takeaways:**

* Use let for variables that may change.
* Use const for variables that should stay the same.
* Avoid using var in modern JavaScript unless required for compatibility.
* Understanding different data types and how to declare variables provides the foundation for writing dynamic and flexible JavaScript code.

**2.3 Working with Strings and Numbers in JS**

**Strings:**

Strings are sequences of characters, such as words or sentences.

Example:

let greeting = "Hello, ";

let name = "John";

let message = greeting + name; // Concatenation

console.log(message); // Output: Hello, John

**Numbers:**

Definition: Numbers are used for calculations and representing numerical values.

Example:

let num1 = 10;

let num2 = 5;

let sum = num1 + num2; // Addition

console.log(sum); // Output: 15

**String Methods:**

Methods / Operations on Strings in JavaScript are as follows:

**Length:**

let text = "Hello, World!";

console.log(text.length); // Output: 13

**toUpperCase:**

let lowercase = "abc";

let uppercase = lowercase.toUpperCase(); // Convert to uppercase

console.log(uppercase); // Output: ABC

**toLowerCase:**

let uppercase = "XYZ";

let lowercase = uppercase.toLowerCase(); // Convert to lowercase

console.log(lowercase); // Output: xyz

**Number Methods:**

Methods / Operations on numbers in JavaScript are as follows:

**toFixed:**

let price = 19.99;

let formattedPrice = price.toFixed(2); // Keep two decimal places

console.log(formattedPrice); // Output: 19.99

**parseInt:**

let numberString = "42";

let actualNumber = parseInt(numberString); // Convert string to integer

console.log(actualNumber); // Output: 42

**parseFloat:**

let floatString = "3.14";

let actualFloat = parseFloat(floatString); // Convert string to float

console.log(actualFloat); // Output: 3.14

Understanding how to manipulate strings and perform basic operations with numbers is crucial for working with real-world data and creating dynamic content in JavaScript.

**2.5: Operators in JavaScript**

**What are Operators?**

Operators are symbols or keywords in a programming language that perform operations on one or more operands. These operations can include mathematical computations, logical comparisons, and more.

**Example:**

let num1 = 5;

let num2 = 3;

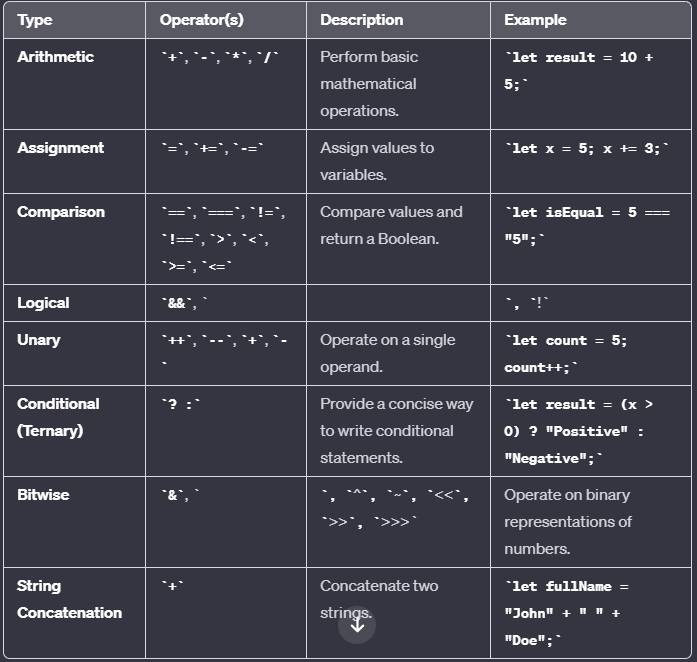
let sum = num1 + num2; // + is the operator here, performing addition

console.log(sum); // Output: 8

**Types of Operators in JavaScript:**

There are different types of operators in JS, which are as follows:

1. **Arithmetic Operators:**
   1. Perform basic mathematical operations.
2. **Assignment Operators:**
   1. Assign values to variables.
3. **Comparison Operators:**
   1. Compare values and return a Boolean result.
4. **Logical Operators:**
   1. Perform logical operations.
5. **Unary Operators:**
   1. Operate on a single operand.
6. **Conditional (Ternary) Operator:**
   1. Provides a concise way to write conditional statements.
7. **Bitwise Operators:**
   1. Operate on binary representations of numbers.
8. **String Concatenation Operator:**
   1. Concatenate two strings.



**Module 3 : Control Flow in JavaScript**

**3.1 Conditional Statements in JavaScript**

**Conditional Statements:**

Conditional statements allow the execution of different code blocks based on whether a specified condition evaluates to true or false.

In JavaScript, there are three main types of conditional statements:

**if Statement:**

Executes a block of code if a specified condition is true.

Example:

let age = 18;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

**else-if Statement:**

Allows for the evaluation of multiple conditions in sequence.

Example:

let time = 14;

if (time < 12) {

console.log("Good morning!");

} else if (time < 18) {

console.log("Good afternoon!");

} else {

console.log("Good evening!");

}

**switch Statement:**

Evaluates an expression against multiple possible case values.

Example:

let day = "Monday";

switch (day) {

case "Monday":

console.log("It's the start of the week!");

break;

case "Friday":

console.log("It's almost the weekend!");

break;

default:

console.log("It's a regular day.");

}

These conditional statements provide the flexibility to control the flow of your program based on different conditions, making your code adaptable to various scenarios.  
  
**Key Takeaways:**

if Statement: Executes a block of code if a specified condition is true.

else-if Statement: Allows for the evaluation of multiple conditions in sequence.

switch Statement: Evaluates an expression against multiple possible case values.

**3.2 Looping Statements in JavaScript**

**Looping Statements:**

Looping statements are used to repeatedly execute a block of code as long as a specified condition is true.

In JavaScript, there are mainly two types of looping statements:

**for Loop:**

Executes a block of code a specified number of times.

**Syntax:**

for (initialization; condition; iteration) {

// Code to be repeated

}

**Example:**

for (let i = 0; i < 5; i++) {

console.log("Iteration: " + i);

}

**while Loop:**

Executes a block of code as long as a specified condition is true.

**Syntax:**

while (condition) {

// Code to be repeated

}

**Example:**

let count = 0;

while (count < 3) {

console.log("Count: " + count);

count++;

}

These looping statements allow you to create repetitive behavior in your code, making it more efficient and adaptable to different scenarios. The for loop is typically used when the number of iterations is known, while the while loop is used when the number of iterations is uncertain and depends on a condition.

**Module 4: Functions in JavaScript**

**4.1 Different Types of Functions**

**Functions:**

Functions are blocks of reusable code that perform a specific task. They make the code modular and easier to manage.

**Different Types of Functions:**

**Function Declaration:**

Declares a function with a name that can be called later in the code.

**Example:**

function greet(name) {

console.log("Hello, " + name + "!");

}

**Function Expression:**

Assigns a function to a variable, making it an anonymous function.

**Example:**

let greet = function(name) {

console.log("Hello, " + name + "!");

};

**Arrow Function:**

A concise way to write functions using the arrow (=>) syntax.

**Example:**

let greet = (name) => {

console.log("Hello, " + name + "!");

};

**Callback Function:**

A function without a name (Anonymous), often used as a callback function.

**Example:**

setTimeout(function() {

console.log("Hello!");

}, 1000);

**4.2 Function Declaration**

A function declaration defines a function with a specified name, parameters, and body. It can be called later in the code.

Example:

function greet(name) {

console.log("Hello, " + name + "!");

}

greet("John"); // Output: Hello, John!

**Key Takeaways:**

* **Function Declaration:** Declares a named function that can be called later in the code.
* **Function Expression:** Assigns a function to a variable.
* **Arrow Function:** A concise way to write functions using the arrow (=>) syntax.
* **Anonymous Function:** A function without a name, often used as a callback function.
* Understanding the different types of functions provides flexibility in how you structure and use code in JavaScript.

**Function Scope and Closure**

**Function Scope:**

Variables declared inside a function are local to that function and cannot be accessed outside of it.

Example:

function exampleFunction() {

let localVar = "I am a local variable";

console.log(localVar);

}

exampleFunction();

// console.log(localVar); // Error: localVar is not defined

**Closure:**

A function bundled with its lexical environment, allowing it to retain access to variables from the scope in which it was created.

Example:

function outerFunction() {

let outerVar = "I am from outer function";

function innerFunction() {

console.log(outerVar);

}

return innerFunction;

}

let closureFunction = outerFunction();

closureFunction(); // Output: I am from outer function

**Key Takeaways:**

* **Anonymous Function:** A function without a name, often used for short-term tasks or as a callback.
* **Arrow Function:** A concise way to write functions using the arrow (=>) syntax.
* **Function Scope:** Variables declared inside a function are local to that function.
* **Closure:** A function bundled with its lexical environment, allowing it to retain access to variables from the scope in which it was created.
* Understanding different function types and the concepts of function scope and closure is crucial for writing maintainable and efficient JavaScript code.

**Module 5: In-Built Data Structures in JS**

**5.1 Arrays in JavaScript**

**Arrays:**

Arrays are ordered lists that can hold multiple values of different data types.

**Example:**

let fruits = ["Apple", "Banana", "Orange", ”Grapes”, ”Melon”, ”Mango” , “Strawberry”]

**Operations on Arrays:**

Different types of operations which you can perform on Arrays are as follows:

**Accessing Elements:**

console.log(fruits[0]); // Output: Apple

**Adding Elements:**

fruits.push("Grapes");

**Removing Elements:**

fruits.pop();

**Length of Array:**

console.log(fruits.length); // Output: 7

**5.2 Objects in JavaScript**

**Objects:**

Objects are collections of key-value pairs, where each key is a string and each value can be any data type.

**Example:**

let person = {

name: "John",

age: 25,

city: "New York"

};

**Operations on Objects:**

let person = {

name: "Adnan",

age: 30,

gender : “Male”

};

**Accessing Properties:**

console.log(person.age); // Output: 25

**Modifying Properties:**

person.age = 30;

**Adding Properties:**

person.gender = "Male";

**Deleting Properties:**

delete person.city;

**Key Takeaways:**

**Arrays:** Ordered lists that can hold multiple values of different data types.

**Common operations**: Accessing, Adding, Removing, Length.

**Objects:** Collections of key-value pairs where keys are strings and values can be any data type.

**Common operations:** Accessing, Modifying, Adding, Deleting.

Understanding these in-built data structures in JavaScript (arrays and objects) is essential for managing and organizing data effectively in your programs.

**Array Methods:**

**forEach:**

Executes a provided function once for each array element.

**Example:**

let fruits = ["Apple", "Banana", "Orange"];

fruits.forEach(function(fruit) {

console.log(fruit);

});

**map:**

Creates a new array with the results of calling a provided function on every element in the array.

**Example:**

let numbers = [1, 2, 3];

let squaredNumbers = numbers.map(function(number) {

return number \* number;

});

**filter:**

Creates a new array with all elements that pass the test implemented by the provided function.

**Example:**

let numbers = [1, 2, 3, 4, 5];

let evenNumbers = numbers.filter(function(number) {

return number % 2 === 0;

});

**push:**

Adds one or more elements to the end of an array.

**Example:**

let fruits = ["Apple", "Banana"];

fruits.push("Orange");

**pop:**

Removes the last element from an array and returns that element.

**Example:**

let fruits = ["Apple", "Banana", "Orange"];

let removedFruit = fruits.pop();

These array methods are fundamental for performing various operations on arrays, such as iteration, transformation, and manipulation.

**Object Methods:**

**Object.keys:**

Returns an array of a given object's own enumerable property names.

**Example:**

let person = { name: "John", age: 25, gender: "Male" };

let keys = Object.keys(person);

**Object.values:**

Returns an array of a given object's own enumerable property values.

**Example:**

let person = { name: "John", age: 25, gender: "Male" };

let values = Object.values(person);

**Object.entries:**

Returns an array of a given object's own enumerable property [key, value] pairs.

**Example:**

let person = { name: "John", age: 25, gender: "Male" };

let entries = Object.entries(person);

**Object.assign:**

Copies the values of all enumerable own properties from one or more source objects to a target object.

**Example:**

let target = { a: 1, b: 2 };

let source = { b: 3, c: 4 };

Object.assign(target, source);

**Object.freeze:**

Freezes an object, preventing new properties from being added and existing properties from being removed or changed.

**Example:**

let frozenObject = Object.freeze({ key: "value" });

**Object.seal:**

Seals an object, preventing new properties from being added and marking all existing properties as non-configurable.

**Example:**

let sealedObject = Object.seal({ key: "value" });

These object methods are essential for working with and manipulating objects in JavaScript, providing functionality for accessing keys, values, entries, and performing operations like assignment, freezing, and sealing.

**Module 6: Object-Oriented Programming (OOP) in JavaScript**

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into reusable, self-contained units known as objects. Each object represents an instance of a class, which serves as a blueprint defining the attributes (data) and behaviors (methods) that the objects of the class will possess.

**Classes and Objects:**

In Object-Oriented Programming, a class is a blueprint for creating objects, and objects are instances of a class. A class defines the properties (attributes) and behaviors (methods) that its objects will have.

**Creating a Class:**

class Car {

constructor(make, model) {

this.make = make;

this.model = model;

}

startEngine() {

console.log("Engine started.");

}

drive() {

console.log("Car is moving.");

}

}

**// Creating an object of the Car class**

let myCar = new Car("Toyota", "Camry");

**4 Pillars of OOP:**

**Encapsulation:**

Encapsulation is the bundling of data (attributes) and methods that operate on the data into a single unit (class).

**Example:**

class BankAccount {

constructor(balance) {

this.balance = balance;

}

deposit(amount) {

this.balance += amount;

}

withdraw(amount) {

if (amount <= this.balance) {

this.balance -= amount;

} else {

console.log("Insufficient funds.");

}

}

}

**Inheritance:**

Inheritance allows a class (subclass/derived class) to inherit properties and behaviors from another class (superclass/base class).

**Example:**

class Animal {

constructor(name) {

this.name = name;

}

makeSound() {

console.log("Some generic animal sound.");

}

}

class Dog extends Animal {

makeSound() {

console.log("Woof! Woof!");

}

}

**Polymorphism:**

Definition: Polymorphism allows objects to be treated as instances of their parent class, providing a way to use a single interface for different types of objects.

**Example:**

function makeAnimalSound(animal) {

animal.makeSound();

}

let genericAnimal = new Animal("Generic");

let dog = new Dog("Buddy");

makeAnimalSound(genericAnimal); // Output: Some generic animal sound.

makeAnimalSound(dog); // Output: Woof! Woof!

**Key Takeaways:**

Classes and Objects: Classes define the blueprint, and objects are instances of the class.

Encapsulation: Bundling data and methods into a single unit (class).

Inheritance: Subclasses inherit properties and behaviors from a superclass.

Polymorphism: Objects can be treated as instances of their parent class, allowing for flexibility in using different types of objects.

Understanding these OOP concepts enhances code organization, reusability, and maintainability in JavaScript.

**6.7 'this' Keyword and Scope in OOP**

**'this' Keyword:** In object-oriented programming, the 'this' keyword refers to the current instance of the object within which it is used. It allows access to the object's properties and methods from within the object.

**Scope in OOP:**

**Scope:** Scope in OOP refers to the context in which variables and functions are declared and accessed. Understanding 'this' and scope is crucial for determining which object is being referred to within a method or function.

**Example:**

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

greet() {

console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");

}

}

const person1 = new Person("John", 25);

const person2 = new Person("Alice", 30);

person1.greet(); // Output: Hello, my name is John and I am 25 years old.

person2.greet(); // Output: Hello, my name is Alice and I am 30 years old.

**Explanation:**

In the 'greet' method, 'this' refers to the instance of the 'Person' object that called the method. It allows access to the specific 'name' and 'age' properties of that instance.

**Use Case:**

Understanding 'this' and scope is essential for accessing and manipulating object properties within methods, ensuring that the correct data is used for each instance of the object. It plays a crucial role in building reusable and dynamic object-oriented code.

**Module 7: Asynchronous JavaScript**

**Asynchronous JavaScript:**

Asynchronous JavaScript allows tasks to be executed without waiting for the previous tasks to complete. It enables non-blocking code execution, allowing the program to continue working on other tasks while waiting for certain operations to finish.

Example:

**setTimeout:**

console.log("Start");

setTimeout(function() {

console.log("Delayed message");

}, 2000);

console.log("End");

**Explanation:**

* The console.log("Start") is executed first.
* The setTimeout function schedules the execution of its callback function after a delay of 2000 milliseconds (2 seconds).
* While waiting for the delay, the program continues to execute the next line, console.log("End").
* After the delay, the callback function inside setTimeout is executed, printing "Delayed message" to the console.

This example demonstrates how asynchronous JavaScript allows certain operations (like timeouts) to be executed without blocking the entire program, enabling smoother and more responsive applications.

**7.1 setTimeout and setInterval**

**setTimeout:** Executes a function or a piece of code after a specified delay.

**setInterval:** Repeatedly executes a function or a piece of code at defined intervals.

Syntax:

**// setTimeout**

setTimeout(function() {

// Code to be executed after a delay

}, delayInMilliseconds);

**// setInterval**

setInterval(function() {

// Code to be executed at each interval

}, intervalInMilliseconds);

**Example:**

**// setTimeout**

console.log("Start");

setTimeout(function() {

console.log("Delayed message");

}, 2000);

console.log("End");

**Example:**

// setInterval

let counter = 0;

let intervalId = setInterval(function() {

console.log("Counter: " + counter);

counter++;

if (counter === 5) {

clearInterval(intervalId);

}

}, 1000);

**Use Case:**

**setTimeout:** Useful for delaying specific actions or events in a program.

**setInterval:** Useful for executing repetitive tasks or updates at regular intervals.

**7.2 Promises**

**Promises:**

Objects representing the eventual completion or failure of an asynchronous operation, and its resulting value.

**Syntax:**

const myPromise = new Promise(function(resolve, reject) {

// Asynchronous operation or task

if (/\* Operation is successful \*/) {

resolve("Success result");

} else {

reject("Error message");

}

});

myPromise.then(function(result) {

// Code to handle successful result

}).catch(function(error) {

// Code to handle error

});

**Example:**

const myPromise = new Promise(function(resolve, reject) {

setTimeout(function() {

const success = true;

if (success) {

resolve("Promise resolved successfully");

} else {

reject("Promise failed");

}

}, 2000);

});

myPromise.then(function(result) {

console.log(result);

}).catch(function(error) {

console.log(error);

});

**Use Case:**

Promises are commonly used for handling asynchronous operations such as fetching data from a server or reading a file.

**7.3 Async/Await**

A syntactic sugar for working with promises, making asynchronous code look and behave more like synchronous code.

**Syntax:**

async function myAsyncFunction() {

try {

const result = await myPromiseFunction();

console.log(result);

} catch (error) {

console.log(error);

}

}

**Example:**

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log(data);

} catch (error) {

console.log("Error fetching data:", error);

}

}

fetchData();

**Use Case:**

Async/Await simplifies the syntax for working with promises, making it easier to read and maintain asynchronous code.

**7.4 Event Listeners**

Functions that wait for a specific event to occur and then execute a specified piece of code.

**Syntax:**

const myElement = document.getElementById('myElement');

myElement.addEventListener('click', function() {

// Code to be executed when the element is clicked

});

**Example:**

<button id="myButton">Click me</button>

<script>

const myButton = document.getElementById('myButton');

myButton.addEventListener('click', function() {

console.log("Button clicked!");

});

</script>

**Use Case:**

Event listeners are used to handle user interactions, such as clicks, mouseovers, or keyboard inputs.

**7.5 Fetch API**

A modern, asynchronous API for making HTTP requests in the browser.

**Syntax:**

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.log("Error:", error));

Example:

fetch('https://jsonplaceholder.typicode.com/todos/1')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.log("Error:", error));

**Use Case:**

The Fetch API is used to make HTTP requests to fetch data from a server or send data to a server asynchronously. It's commonly used in web applications to interact with APIs.